Authors
Nancy Greenberg
Priya Vennapusa

Technical Contributors and Reviewers
Howard Bradley
Laszlo Czinkoczki
Dan Gabel
Connie Daileris Green
John Hibbard
Lilian Hobbs
John Hoff
Alexander Hunold
Tamas Kerepes
Susan Kotsovolos
Herve Lejeune
Stefan Lindblad
Diana Lorentz
Howard Ostrow
Arjan Pellenkoft
Stacey Procter
Shankar Raman
Marijesus Senise
Janet Stern
Don Sullivan
Ric Van Dyke
Lachlan Williams

Publisher
Shane Mattimoe
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
12 Materialized Views and Temporary Tables
Objectives 12-2
Materialized Views 12-3
Create Materialized Views 12-4
Refresh Materialized Views 12-5
Materialized Views: Manual Refresh 12-7
Query Rewrites 12-8
Create Materialized Views: Syntax Options 12-11
Enabling and Controlling Query Rewrites 12-12
Query Rewrite Example 12-13
Dimensions: Overview 12-15
Dimensions and Hierarchies 12-16
Dimensions: Example Table 12-17
Dimensions and Hierarchies 12-18
Create Dimensions and Hierarchies 12-19
Dimensions Based on Multiple Tables 12-20
Dimensions with Multiple Hierarchies 12-21
Temporary Tables 12-22
Creating Temporary Tables 12-24
Summary 12-26

13 Alternative Storage Techniques
Objectives 13-2
Storing User Data 13-3
Index-Organized Tables 13-4
IOT Performance Characteristics 13-5
IOT Limitations 13-6
When to Use Index-Organized Tables 13-7
Creating Index-Organized Tables 13-8
IOT Row Overflow 13-9
Retrieving IOT Information 13-10
External Tables 13-11
External Tables Performance Characteristics 13-12
14 Data Warehousing Considerations

Objectives 14-2
The with Clause: Overview 14-3
The with Clause 14-4
The with Clause: Example 14-5
The with Clause: Implementation 14-6
The with Clause: Usage Notes 14-7
The with Clause: Benefits 14-8
Multitable insert statements 14-9
Multiple insert statements 14-11
Advantages of Multitable insert statements 14-12
Merge statements 14-13
Merge syntax 14-14
Example of Using the merge statement in Data Warehousing 14-15
Summary 14-16

Appendix A: Workshops

Appendix B: Diagnostic Tools Reference

Appendix C: Table Descriptions

Appendix D: Workshop Scripts

Appendix E: Practice Solutions

Index
Workshop 1: Single Table, Single Predicate

Workshop Objectives

- Learn to read SQL execution plans
- Identify the most common data access methods
- Identify when indexes are usable or not
- Learn to use SQL*Plus AUTOTRACE

Introduction

In this workshop, you will not yet investigate whether or not it is appropriate to use indexes; therefore you will not consider statistics or timings in this workshop. You will only investigate when an index is usable and learn to read SQL execution plans. Try to work together in small groups, and discuss the workshop results. Each time you load a new SQL statement, try to predict what the optimizer will do before running the statement. If you have some time left, feel free to experiment, for example, by creating additional indexes or changing the SQL statements. Take notes during the workshop as an aid for the wrap-up discussion.

Scripts Reference

<table>
<thead>
<tr>
<th>Script name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>utlxplan.sql</td>
<td>Creates the plan_table, used by EXPLAIN and AUTOTRACE</td>
</tr>
<tr>
<td>rule.sql</td>
<td>Alter session set optimizer_goal = rule</td>
</tr>
<tr>
<td>choose.sql</td>
<td>Alter session set optimizer_goal = choose</td>
</tr>
<tr>
<td>ws01_stats.sql</td>
<td>Script to run before you start this workshop</td>
</tr>
<tr>
<td>li.sql</td>
<td>Lists all indexes; accepts table name as argument; wildcards (%,__) in table names are allowed (Default: the previous tablename)</td>
</tr>
<tr>
<td>ci.sql</td>
<td>Creates an index; prompts for table name and column names</td>
</tr>
<tr>
<td>cui.sql</td>
<td>Creates a unique index; same behavior as ci.sql</td>
</tr>
<tr>
<td>aton.sql</td>
<td>Set autotrace on</td>
</tr>
<tr>
<td>atonx.sql</td>
<td>Set autotrace on explain</td>
</tr>
<tr>
<td>atto.sql</td>
<td>Set autotrace traceonly</td>
</tr>
<tr>
<td>attox.sql</td>
<td>Set autotrace traceonly explain</td>
</tr>
<tr>
<td>atoff.sql</td>
<td>Set autotrace off</td>
</tr>
</tbody>
</table>
Workshop 1: Single Table, Single Predicate (continued)

1. Run the ws01_stats script to make sure that you do not have any statistics on the tables used in this workshop. Then check the existence and column structure of the CUSTOMERS table, and run the utlxplan.sql script to create the plan table in your schema. Also list the existing indexes on the CUSTOMERS table using the li.sql script.

   SQL> @choose
   SQL> @ws01_stats
   SQL> describe customers
   SQL> @utlxplan
   SQL> @li customers

   Notes:

2. Use the attox.sql script to enable AUTOTRACE TRACEONLY EXPLAIN to suppress statement output and produce execution plans; check the settings with SHOW AUTOTRACE.

   SQL> show autotrace
   SQL> @attox
   SQL> show autotrace

   Notes:

3. Get and run queries ws01_01, ws01_01a, ws01_02, and ws01_03. First remove all indexes from the CUSTOMERS table by running the dai.sql script.

   Note: dai.sql removes all nonprimary key indexes.

   SQL> @dai
       on which table: customers
Workshop 1: Single Table, Single Predicate (continued)

```
SQL> get ws01_01
1  select cust_first_name           -- ws01_01.sql
2    ,      cust_last_name
3  from   customers
4*  where  cust_id = 1030

SQL>/
```

```
SQL> get ws01_01a
1  select cust_first_name           -- ws01_01a.sql
2    ,      cust_last_name
3  from   customers
4*  where  cust_id <> 1030

SQL>/
```

```
SQL> get ws01_02
1  select cust_first_name           -- ws01_02.sql
2    ,      cust_last_name
3  from   customers
4*  where  cust_id < 20000

SQL> /
```

```
SQL> get ws01_03
1  select cust_first_name           -- ws01_03.sql
2    ,      cust_last_name
3  from   customers
4*  where  cust_id between 70000 and 80000

SQL> /
```

The results of these three queries show that the Oracle9i optimizer can use indexes for three types of conditions:

- Equality search (ws01_01)
- Unbounded range (ws01_02)
- Bounded range (ws01_03)

The index is not used if the NOT EQUAL (<> operator is present.

Notes:
Workshop 1: Single Table, Single Predicate (continued)

4. Now create an index on the CUST_CREDIT_LIMIT column of the CUSTOMERS table using the ci.sql script and start queries ws01_04 and ws01_05.

    SQL> @ci
      
      on which table : customers
    on which column(s): cust_credit_limit
    Creating index on: customers cust_credit_limit
    Enter value for index_name: cust_credit_limit_idx
    SQL> @li customers
    SQL> @attoo
    SQL> get ws01_04
       1  select cust_id                    -- ws01_04.sql
       2    from   customers
      3*   where  cust_credit_limit*1.10 = 11000
    SQL> /
    SQL> get ws01_05
       1  select cust_id                    -- ws01_05.sql
       2    from   customers
      3*   where  cust_credit_limit = 30000/2
    SQL> /

The results show that although the CUST_CREDIT_LIMIT column is indexed, the index cannot be used if the indexed column is part of an expression in the WHERE clause. An index is only usable if the indexed column appears clean in the WHERE clause. This is an important property of indexes; before Oracle7, this was one of only a few ways to influence the Oracle optimizer.

Notes:
Workshop 1: Single Table, Single Predicate (continued)

5. Create another index on the CUST_LAST_NAME column of the CUSTOMERS table and start queries ws01_06 and ws01_07.

    SQL> @ci
    on which table    : customers
    on which column(s): cust_last_name
    Creating index on: customers cust_last_name
    Enter value for index_name: cust_last_name_idx

    SQL> @li customers

    SQL> get ws01_06
    1  select cust_id                           -- ws01_06.sql
    2    from   customers
    3  where  substr(cust_last_name,1,1) = 'S'

    SQL> /

    SQL> get ws01_07
    1  select cust_id                           -- ws01_07.sql
    2    from   customers
    3  where  cust_last_name like 'S%'

    SQL> /

These two queries are logically equivalent, but again you see that the indexed column should be clean. As soon as you apply any function, index usage is impossible.

Notes:
Workshop 1: Single Table, Single Predicate (continued)

6. Now start query ws01_08, and compare the execution plan with ws01_07. How do you explain the difference?

```
SQL> get ws01_08
1  select cust_last_name                   -- ws01_08.sql
2  from   customers
3* where cust_last_name like 'S%

SQL> /
```

Notes:

7. Start query ws01_09, and look at the LIKE search pattern:

```
SQL> get ws01_09
1  select cust_id                        -- ws01_09.sql
2  from   customers
3* where cust_last_name like '%S%

SQL> /
```

The index is unusable, because the search pattern starts with a wildcard. This makes sense.
The index is useless if you do not specify a leading part for the search.

Notes:
Workshop 1: Single Table, Single Predicate (continued)

8. This is a challenge exercise. Start query ws01_10, and try to explain why the index is not used: The indexed column is clean, and the search pattern does not start with a wildcard.

   SQL> get ws01_10
   1  select cust_last_name
   2  from   customers
   3* where cust_id like ’7%’
   SQL>

   Notes:

9. In this exercise you investigate the treatment of NULL-values and use the CUSTOMERS table for that purpose. First, run the ws01_11a.sql script to remove some values from the CUST_EMAIL column. Second, create an index on the CUST_EMAIL column of the CUSTOMERS table; this column contains many NULL-values. Then start query ws01_11b.

   SQL> @atoff
   SQL> @ws01_11a
   SQL> describe customers
   SQL> @ci
       on which table    : customers
       on which column(s): cust_email
       Creating index on:  customers cust_email
       Enter value for index_name: cust_email_idx
   SQL> @li customers
   SQL> @attox
   SQL> get ws01_11b
   1  select cust_email
   2  from   customers
   3* where cust_email is null
   SQL> /

Oracle9i: SQL Tuning Workshop A-8
Workshop 1: Single Table, Single Predicate (continued)

To explain why the index is not used in this case, you should realize that the Oracle server does not store any references to NULL-values in a B*-tree index. That is why the only way to find rows containing NULL-values is by performing a full table scan.

Notes:

10. Now suppose that you are interested in all rows that do not contain a NULL value. Start ws01_12 and compare with ws01_11b.

```
SQL> get ws01_12
  1  select cust_id                         -- ws01_12.sql
  2   from customers
  3* where cust_email is NOT null

SQL> /
```

The index is not used again, but this time for a different reason. The optimizer decision is based on selectivity considerations. You can rewrite the SQL statement and force rule-based optimization, as shown in query ws01_13.

```
SQL> get ws01_13
  1  select cust_id                         -- ws01_13.sql
  2   from customers
  3* where cust_email > 'a'

SQL> /
```

The rule-based optimizer will use the index now; for the cost-based optimizer, it depends on the statistics. Under which circumstances are queries ws01_12 and ws01_13 logically equivalent?

Notes:

The best approach to influence the cost-based optimizer is to specify a hint. Start ws01_14 and compare the results with ws01_12.
Workshop 1: Single Table, Single Predicate (continued)

SQL> @choose
SQL> get ws01_14
1 select --+ INDEX(c) -- ws01_14.sql
2    c.cust_id
3 from customers c
4* where c.cust_email is NOT null
SQL> /

Notes:

Note: If the choose.sql script is not run, the cost-based optimizer is still used because of the presence of the hint.

11. You already investigated IS NOT NULL. Now look at negations in general (using <>, !=, or NOT). Note that the distinction between normal conditions and negations is only important for the rule-based optimizer because it must assume selectivity. Because the cost-based optimizer is statistics-driven, it disregards negations, although most conditions with a negation have a bad selectivity.

SQL> @li customers
SQL> @rule
SQL> get ws01_15
1 select cust_last_name -- ws01_15.sql
2    from customers
3* where cust_credit_limit = 7000
SQL> /
SQL> get ws01_16
1 select cust_last_name -- ws01_16.sql
2    from customers
3* where cust_credit_limit != 50000
SQL> /
SQL> get ws01_17
1 select cust_last_name -- ws01_17.sql
2    from customers
3* where NOT (cust_credit_limit > 50000)
SQL> /
Workshop 1: Single Table, Single Predicate (continued)

Query ws01_15 shows that the index on the CUST_CREDIT_LIMIT column is used. Query ws01_16 shows that the index on the CUST_CREDIT_LIMIT column is not used (because of the negation). Query ws01_17 shows that a negation of an inequality will be internally translated into a positive formulation of the condition. Negations are not part of the predicate. Note that you issued an ALTER SESSION command first to force rule-based optimization.

Notes:

Workshop Solutions.

6. The difference is that ws01_08 selects the last name, so a table access is not necessary; this query only accesses the index. In ws01_07, table access is necessary to retrieve the customer ID.

8. The search pattern in ws01_10 does not start with a wildcard, but the index is still unusable. This is because the CUST_ID column is a numeric column, so the optimizer must apply an implicit data type conversion and rewrites the WHERE clause to read:

    where to_char(cust_id) like '7%'

Workshop Summary

After completing this workshop, you should have learned the following:

- Indexes are only usable if they exist and are available.
- Indexes are only usable if the corresponding column is referenced in a WHERE clause.
- Indexes are only usable if the column name appears clean in the predicate.
- Indexes are not used if the column is part of any expression, or function, or in case of implicit data type conversion.
- The LIKE operator can benefit only from indexes if the leading character of the search pattern is not a wildcard and the column contains alphanumeric data.
- If all column values to be selected are part of an index, a table access is not necessary.
- NULL-values are not stored in indexes, therefore full table scans are needed for IS NULL searches.
- You use the index hint to force the cost-based optimizer to use an index.
- You use SQL*Plus AUTOTRACE to display SQL statement execution plans.
- The index is not used if the NOT EQUAL (!=) operator is present.
Workshop 2: Sorting, Grouping, and Set Operators

Workshop Objectives

- Learn to tune sorts for ORDER BY clauses
- Identify and tune implicit sort operations caused by SELECT DISTINCT
- Learn to tune GROUP BY operations and group functions
- Learn to tune set operators (UNION, MINUS, INTERSECT)

Introduction

In this workshop, you concentrate on tuning explicit and implicit sort operations. Sorting is a common operation. If the Oracle9i Server is able to perform all sort activity in memory, then the performance is probably acceptable. However, sometimes the Oracle9i Server writes intermediate results to disk. Therefore, SQL*Plus AUTOTRACE shows two statistics: sorts (memory) and sorts (disk). Sometimes you can avoid sort operations by creating indexes, or suppress implicit sorts that are not needed for the result you want. You do not tune memory; that is a DBA task, covered in the Enterprise DBA Part 2: Performance and Tuning course.

Try to work together in small groups, and discuss the workshop results. Each time you load a new SQL statement, try to predict what the optimizer will do before running the statement. If you have some time left, feel free to experiment by creating, for example, additional indexes or changing the SQL statements. It is also recommended to take notes during the workshop as an aid for the wrap-up discussion.

Scripts Reference

<table>
<thead>
<tr>
<th>Script name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws02_stats.sql</td>
<td>Script to run before you start this workshop</td>
</tr>
<tr>
<td>li.sql</td>
<td>Lists all indexes; accepts table name as argument; wildcards (%,_) in table names are allowed (Default: the previous tablename)</td>
</tr>
<tr>
<td>ci.sql</td>
<td>Creates an index; prompts for table name and column names (Column names should be separated with commas; the index name is generated by the script.)</td>
</tr>
<tr>
<td>cbi.sql</td>
<td>Creates a bitmapped index; same behaviour as ci.sql</td>
</tr>
<tr>
<td>aton.sql</td>
<td>Set autotrace on</td>
</tr>
<tr>
<td>atonx.sql</td>
<td>Set autotrace on explain</td>
</tr>
<tr>
<td>atto.sql</td>
<td>Set autotrace traceonly</td>
</tr>
<tr>
<td>attox.sql</td>
<td>Set autotrace traceonly explain</td>
</tr>
<tr>
<td>atoff.sql</td>
<td>Set autotrace off</td>
</tr>
</tbody>
</table>
Workshop 2: Sorting, Grouping, and Set Operators (continued)

1. Run the `ws02_stats` script first, to have a clean workshop start, then drop all indexes on the `CUSTOMERS` table using the `dai.sql` script. Note that the index related to the primary key constraint cannot (and does not need to) be dropped.

   ```sql
   SQL> @ws02_stats
   SQL> @li customers
   SQL> @dai
       on which table: customers
   SQL> @li customers
   ``

   **Notes:**

2. Verify the existence of the plan table, and use the `attox.sql` script to enable AUTOTRACE `TRACEONLY EXPLAIN` to suppress SQL statement output and produce execution plans; check the settings with `SHOW AUTOTRACE`.

   ```sql
   SQL> describe plan_table
   SQL> @attox
   SQL> show autotrace
   ``

   **Notes:**

3. Get query `ws02_01`.

   ```sql
   SQL> get ws02_01
       1  select cust_first_name           -- ws02_01.sql
       2    ,      cust_last_name
       3    ,      cust_credit_limit
       4  from   customers
       5* order  by cust_credit_limit
   SQL> /
   ``

   The Oracle server must perform a sort operation. Sorting can be avoided by creating appropriate indexes, so investigate what happens when you create an index on the `CUST_CREDIT_LIMIT` column:
Workshop 2: Sorting, Grouping, and Set Operators (continued)

SQL> @ci
    on which table : customers
    on which column(s): cust_credit_limit
    Creating index on: customers cust_credit_limit
    Enter value for index_name: cust_credit_limit_idx

SQL> get ws02_01
SQL> /

The results show that although the CUST_CREDIT_LIMIT column is indexed, the index is not used by the optimizer to avoid a sort operation.

Notes:

4. Get ws02_02, and compare the results with ws02_01. Apparently, the index on the CUST_ID column is used: The rows are retrieved in sorted order by accessing the rows through the index.

SQL> get ws02_02
   1  select cust_first_name          -- ws02_02.sql
   2    , cust_last_name
   3    , cust_credit_limit
   4  from customers
   5* order by cust_id

SQL> /

Note: The presence of statistics could change the optimizer’s behaviour. If you analyzed the CUSTOMERS table, make sure to delete the statistics or force rule-based optimization.

Why do you think the index on the CUST_CREDIT_LIMIT column is not used? What is the difference between ws02_01 and ws02_02?

Notes:
Workshop 2: Sorting, Grouping, and Set Operators (continued)

5. Get ws02_03. Apparently, the WHERE clause does not matter. The index is still used to retrieve all rows in sorted order, and the WHERE clause is evaluated as a last step.

```sql
SQL> get ws02_03
1  select cust_first_name             -- ws02_03.sql
2  ,      cust_last_name
3  ,      cust_city
4  from   customers
5  where  cust_city = 'Paris'
6* order  by cust_id
SQL> /
```

Notes:

Investigate what happens if you create an additional index on the CUST_CITY column:

```sql
SQL> @ci
  on which table    : customers
  on which column(s): cust_city
  Creating index on: customers cust_city
  Enter value for index_name: cust_city_idx
SQL> @li customers
SQL> get ws02_03
SQL> /
```

This time the optimizer apparently prefers to use the index on the CUST_CITY column to reduce the number of rows that must be sorted.

Notes:
Workshop 2: Sorting, Grouping, and Set Operators (continued)

6. Now get and run queries `ws02_04`, `ws02_05`, and `ws02_06`.

```
SQL> get ws02_04
   1  select max(cust_credit_limit)       -- ws02_04.sql
     2* from   customers
SQL> /
SQL> get ws02_05
   1  select max(cust_credit_limit+1000) -- ws02_05.sql
     2* from   customers
SQL> /
SQL> get ws02_06
   1  select max(cust_credit_limit*2)    -- ws02_06.sql
     2* from   customers
SQL> /
```

This shows that an index can be useful to retrieve a maximum value (and a minimum value). If no index is available, the optimizer must scan the full table and perform a sort.

Notes:

**Note:** In `ws02_05` the index is used, although the indexed column is part of an expression (cust_credit_limit+1000). Why does `ws02_06` not show the same behavior?

7. Start query `ws02_07`. A `WHERE` clause is added and you see the result. Note that dropping the index on the `CUST_CITY` column and running `ws02_07` again shows a full table scan and a sort. The index on the `CUST_CITY` column is no longer usable. Verify the existence of the plan table, and use the `atoox.sql` script to enable `AUTOTRACE TRACEONLY EXPLAIN` to suppress SQL statement output and produce execution plans; check the settings with `SHOW AUTOTRACE`.

```
SQL> get ws02_07
   1  select max(cust_credit_limit)       -- ws02_07.sql
     2 from   customers
     3* where  cust_city = 'Paris'
SQL> /
SQL> drop index cust_city_idx;
SQL> get ws02_07
SQL> /
```

Oracle9i: SQL Tuning Workshop A-16
Workshop 2: Sorting, Grouping, and Set Operators (continued)

8. Start query ws02_08. This query shows that an implicit sort operation is needed to evaluate a `SELECT DISTINCT`.

```sql
SQL> get ws02_08
1  select distinct cust_city                -- ws02_08.sql
2* from   customers
SQL> /
```

Try to avoid using `SELECT DISTINCT` when writing SQL statements. Using the `DISTINCT` keyword unconditionally results in a sort operation. That is why creating an index will not help; it will not be used.

Notes:

9. The following queries investigate the SQL set operators. These operators unconditionally result in sort operations, regardless of the presence of indexes. Create any indexes you like to investigate this. The sorts are needed because the SQL set operators are supposed to filter duplicate rows from the result.

```sql
SQL> get ws02_09
1  select country_id from countries         -- ws02_09.sql
2  intersect
3* select country_id from customers
SQL> /
SQL> get ws02_10
1  select country_id from countries         -- ws02_10.sql
2  minus
3* select country_id from customers
SQL> /
SQL> get ws02_11
1  select country_id from countries         -- ws02_11.sql
2  union
3* select country_id from customers
SQL> /
```

Notes:
Workshop 2: Sorting, Grouping, and Set Operators (continued)

There is one exception: the UNION ALL operator does not perform a sort and does not filter duplicate rows. Use the UNION ALL operator if you are sure that there are no duplicate rows, or that duplicate rows cause no semantic problems.

```
SQL> get ws02_12
  1  select country_id from countries         -- ws02_12.sql
  2    union all
  3*   select country_id from customers
SQL> /
```

Notes:

10. Examine the GROUP BY operator. Like the SELECT DISTINCT and the set operators, a sort operation will always be part of the execution plan.

```
SQL> get ws02_13
  1  select cust_city                         -- ws02_13.sql
  2    ,      avg(cust_credit_limit)
  3    from   customers
  4*   group  by cust_city
  5   /
SQL> @ci
    on which table    : customers
    on which column(s): cust_city
    Creating index on:  customers cust_city
    Enter value for index_name: cust_city_idx
SQL> get ws02_13
SQL> /
```

Notes:
Workshop 2: Sorting, Grouping, and Set Operators (continued)

Creating an index on the CUST_CITY column does not help. Now examine the following two SQL statements, which are logically equivalent (make sure to suppress statement output, because the statements result in several rows):

```
SQL> @atto
SQL> get ws02_14
  1  select cust_city                      -- ws02_14.sql
  2  ,      avg(cust_credit_limit)
  3  from   customers
  4  where  cust_city = 'Paris'
  5* group  by cust_city
SQL> /
SQL> get ws02_15
  1  select cust_city                      -- ws02_15.sql
  2  ,      avg(cust_credit_limit)
  3  from   customers
  4  group  by cust_city
  5* having cust_city = 'Paris'
SQL> /
```

The index on the CUST_CITY column is used in ws02_14 to reduce the set of rows that must be sorted. This is not possible in ws02_15, because the HAVING clause is always evaluated after the GROUP BY clause. Note that ws02_15 is a badly formulated SQL statement. A HAVING clause usually contains a group function; COUNT, SUM, AVG, MIN, MAX. ws02_14 has fewer logical and physical reads.

Notes:
Workshop 2: Sorting, Grouping, and Set Operators (continued)

Workshop Solutions.

4. The difference between ws02_01 and ws02_02 is that the CUST_ID column is mandatory (NOT NULL) and the CUST_CREDIT_LIMIT column is not. The index on CUST_CREDIT_LIMIT cannot be guaranteed to contain an entry for each customer, because NULL-values are not stored in regular indexes.

6. In ws02_06, you do not see the same behavior as in ws02_05 for semantic reasons. The optimizer must always consider the possibility of bind variables. An execution plan must be independent of the value that the bind variable will get after optimization and before execution (remember: Parse - Bind - Execute).

\[
\text{max(cust\_credit\_limit + :x)}
\]

is always equivalent to \[
\text{max(cust\_credit\_limit ) + :x}
\]

\[
\text{max(cust\_credit\_limit * :x)}
\]

is not always equivalent to \[
\text{max(cust\_credit\_limit ) * :x}
\]

Thus the first case is no problem; the index can be used to retrieve the maximum value first. However, if the bind variable :x gets a negative value, the second rewrite is no longer valid.

Workshop Summary

After completing this workshop, you should have learned the following:

- Indexes are usable to avoid sorting. However, the optimizer must not miss any rows, which is why the index must be on a NOT NULL column. This explains the difference between ws02_01 and ws02_02. In ws02_01, the index is on a column that allows nulls. In ws02_02, the index is on a NOT NULL column.
- If any additional indexes are available to reduce the set of rows to be sorted, those paths might be more attractive. Sorts on small sets of rows usually perform well, and this approach reduces throwaway (retrieving rows that are not needed for the result set).
- Maximum and minimum values can be retrieved from indexes, with certain restrictions; for example, as long as there is no WHERE clause and no GROUP BY clause (see next comment).
- SELECT DISTINCT, GROUP BY, UNION, MINUS, and INTERSECT all result in an unconditional sort operation. This sort operation cannot be suppressed. However, it can be tuned by trying to reduce the set of rows to be sorted.
- Try to use the WHERE clause for row-level predicates instead of the HAVING clause.
Workshop 3: Joins

Workshop Objectives

- Examine join operations using a two-table join
- Identify the differences between predicates in a join
- Explain the different optimizer goals influence on the chosen join operation
- Describe join-related hints
- Find the optimal join operation for a given join
- Use Top-N SQL
- Examine hash joins
- Examine star joins

Introduction

In this workshop, you focus on simple two-table joins with and without nonjoin predicates.

Because it is important to know the number of rows processed when optimizing joins, you usually use either SQL Trace (with TKPROF). However, in this workshop you still use AUTOTRACE in SQL*Plus.

Take notes about both the optimizer’s cost estimate and the amount of logical I/O (consistent gets) for comparison purposes.

Scripts Reference

<table>
<thead>
<tr>
<th>Script name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws03_stats.sql</td>
<td>Script to run before you start this workshop</td>
</tr>
<tr>
<td>hashfalse.sql</td>
<td>Alter session set hash_join_enabled = false</td>
</tr>
<tr>
<td>hashtrue.sql</td>
<td>Alter session set hash_join_enabled = true</td>
</tr>
<tr>
<td>allrows.sql</td>
<td>Alter session set optimizer_goal = all_rows</td>
</tr>
<tr>
<td>firstrows.sql</td>
<td>Alter session set optimizer_goal = first_rows</td>
</tr>
<tr>
<td>atto.sql</td>
<td>Set autotrace traceonly</td>
</tr>
<tr>
<td>dai.sql</td>
<td>Drop all indexes (on a given table)</td>
</tr>
<tr>
<td>attox.sqlci.sql</td>
<td>Create (regular) index</td>
</tr>
<tr>
<td>atoff.sql</td>
<td>Set autotrace off</td>
</tr>
</tbody>
</table>

Instructor Note

On question 7, nested loops is still favored when the optimizer goal is set to ALL_ROWS.
Workshop 3: Joins (continued)

1. Run the `ws03_stats` script first to have a clean workshop start, then verify the following initialization parameter settings by running script `ws03_01`:

   - `hash_join_enabled = false`
   - `optimizer_mode = all_rows`

   If these parameters are not as shown above, use the appropriate scripts in SQL*Plus. The script `hashfalse` is available to change the value of the first parameter dynamically. Also, make sure to drop all indexes on the CUSTOMERS and COUNTRIES tables.

   ```sql
   SQL> @ws03_stats
   SQL> get ws03_01
   1 select name,value -- ws03_01.sql
   2 from   v$parameter
   3 where name in ('hash_join_enabled'
   4*                ,'optimizer_mode'   )
   SQL> /
   SQL> @hashfalse
   SQL> @allrows
   SQL> @dai
   on which table: countries
   SQL> @dai
   on which table: customers
   ```

Notes:
Workshop 3: Joins (continued)

2. Set AUTOTRACE to TRACEONLY and analyze the statement in ws03_02.

   SQL> @atto
   SQL> get ws03_02
   1   select c.cust_last_name -- ws03_02.sql
   2     ,  c.cust_year_of_birth
   3     ,  co.country_name
   4   from customers c
   5     ,  countries co
   6   where c.country_id = co.country_id
   SQL> /

In this case, the join operation chosen by the optimizer is normally a sort/merge join because you disabled hash joins. Is this a reasonable choice with regard to the optimizer goal (ALL_ROWS)?

Notes:

3. Change OPTIMIZER_GOAL to FIRST_ROWS and repeat your analysis.

   SQL> @firstrows
   SQL> get ws03_02
   SQL> /

   Why do you think the join operation changes to a nested loops operation?

Notes:
Workshop 3: Joins (continued)

4. Enable hash join operations by setting `HASH_JOIN_ENABLED` to TRUE, and repeat your analysis.

   ```sql
   SQL> @hashtrue
   SQL> get ws03_02
   SQL> /
   
   Why do you think the nested loops operation is still chosen?
   
   Notes:
   
   5. Change the `OPTIMIZER_GOAL` setting back to `ALL_ROWS` and repeat your analysis.

   ```sql
   SQL> @allrows
   SQL> get ws03_02
   SQL> /
   
   The optimizer chooses the hash join operation over the sort/merge join, which is reasonable. No expensive sort operations are involved.
   
   Notes:
   
   Now compare the logical I/O statistics collected so far. For your reference, on a database with a 2 KB block size, the following values were measured:

<table>
<thead>
<tr>
<th>Join operation</th>
<th>Consistent Gets</th>
<th>DB Block Gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort/Merge join</td>
<td>2369</td>
<td>14</td>
</tr>
<tr>
<td>Nested loops join</td>
<td>105542</td>
<td>5</td>
</tr>
<tr>
<td>Hash join</td>
<td>5543</td>
<td>7</td>
</tr>
</tbody>
</table>

   Oracle9i: SQL Tuning Workshop A-24
Workshop 3: Joins (continued)

6. You can use join order hints to make the optimizer choose sort/merge, nested loops, or hash join operations using the USE_MERGE, USE_NL, USE_HASH, and ORDERED hints. Also experiment with specifying arguments, such as USE_NL(C), or specify multiple hints. Use the ws03_03.sql script for this purpose.

Note: Make sure you put the table aliases in the hint.

```
SQL> get ws03_03
  1  select /*+ &hint */
  2       c.cust_last_name               -- ws03_03.sql
  3       , c.cust_year_of_birth
  4       , co.country_name
  5 from customers c
  6       , countries co
  7* where c.country_id = co.country_id

SQL> /
  Enter value for hint: ...
```

Notes:

<table>
<thead>
<tr>
<th>Join operation</th>
<th>Consistent Gets</th>
<th>DB Block Gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_MERGE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USE_NL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USE_HASH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDERED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Try USE_NL(C CO) ORDERED and be prepared: This could take several minutes.
Remember that nested loops hints (which specify the joining order) are only considered if the optimizer uses a nested loops operation.
Workshop 3: Joins (continued)

7. Analyze the SQL statement in ws03_04.sql and identify the best join operation (and join sequence, when relevant), using the same methods (USE_MERGE, USE_NL, USE_HASH) as for ws03_02.sql.

    SQL> get ws03_04
    1  select c.cust_last_name
    2    ,      c.cust_year_of_birth    -- ws03_04.sql
    3    ,      co.country_name
    4  from   customers c
    5    ,      countries co
    6  where  c.country_id = co.country_id
    7  and    co.country_region = 'Americas'
    SQL> /

Notes:
Workshop 3: Joins (continued)

8. Analyze the SQL statement in ws03_05.sql and identify the best join operation (and join sequence, when relevant) using the same methods as for ws03_03.sql.

```
SQL> get ws03_05
1  select /*+ &hint */
2       c.cust_last_name           -- ws03_05.sql
3  ,      c.cust_year_of_birth
4  ,      co.country_name
5 from   customers c
6  ,      countries co
7 where  c.country_id = co.country_id
8  and    co.country_region = 'Americas'
SQL> /
```

<table>
<thead>
<tr>
<th>Join operation</th>
<th>Consistent Gets</th>
<th>DB Block Gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_MERGE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USE_NL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USE_HASH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is an example where the merge join operation is more efficient than the other join operations, even when retrieving all rows from the query. This is caused by the significant throwaway of rows due to the non-indexed, non-join predicate.

A nested loops join with the COUNTRIES table as driving (outer) table, using the single row predicate (and CO.COUNTRY_REGION = ‘Americas’), is an effective execution plan (only one row from each row source).

If you experiment with hints, you can force a hash join with full table scans and even a nested loops join with the opposite join order (resulting in a full table scan of CUSTOMERS). This means that the rule that single row predicates should always be first in the join order can be violated when using sufficient disturbing hints.

How can the CUSTOMERS table be the outer table of the nested loops join?

Notes:
Workshop 3: Joins (continued)

9. Analyze the SQL statement in ws03_06.sql.

    SQL> get ws03_06
    1  select c.cust_last_name    -- ws03_06.sql
    2         ,      s.time_id
    3         ,      s.prod_id
    4  from   customers c,  sales s,
    5  where  c.cust_id <> s.cust_id
    6  and    s.prod_id = 2595
    7* and    s.time_id = '01-JAN-98'
    SQL> /

Which join operations can be used to execute this join?
Experiment with different join orders by using an ORDERED hint, then try a LEADING hint, and find the best choice. There is a significant difference in performance.

<table>
<thead>
<tr>
<th>Join operation</th>
<th>Consistent Gets</th>
<th>DB Block Gets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Having CUSTOMERS as an inner table results in the smallest number of rows thrown away. This is why (in this case) it is more efficient to have the largest row source as the outer table. Usually the smallest row source is the outer table in a nested loops join.

Notes:
Workshop 3: Joins (continued)

10. Analyze the SQL statement in ws03_07.sql.

```sql
SQL> get ws03_07
1  select c.cust_last_name    -- ws03_07.sql
2  ,      s.time_id
3  ,      s.prod_id
4  from   customers c, sales s
5* where  c.cust_id = s.cust_id (+)

SQL> /
```

This is an outer join statement. Try to find the optimal join operation. What happens if you use an ORDERED hint or a LEADING hint?

Notes:

11. Analyze the SQL statement in ws03_08.sql. Try to find the optimal execution plan.

```sql
SQL> get ws03_08
1  select c.cust_last_name    -- ws03_08.sql
2  ,      s.time_id
3  ,      s.prod_id
4  from   sales s, customers c
5* where  c.cust_id = s.cust_id (+)
6* and    s.prod_id = 7145

SQL> /
```

Notes:
Workshop 3: Joins (continued)

This is an outer join statement with a nonjoin predicate on the outer joined table. This nonjoin predicate is without the plus (+) sign, which will result in the outer join being disabled.

12. Analyze the SQL statement in ws03_09.sql. Try to find the optimal execution plan.

```
SQL> get ws03_09
1  select c.cust_last_name    -- ws03_09.sql
2       , s.time_id
3       , s.prod_id
4  from   sales s, customers c
5  where  c.cust_id = s.cust_id (+)
6* and    s.prod_id (+) = 7145
```

This is an outer join statement with a nonjoin predicate on the outer joined table. This nonjoin predicate is with the plus (+) sign.

Is it possible to specify a hint to force execution with a nested loops join, with SALES as outer (driving) table?

Notes:
Workshop 3: Joins (continued)

13. Analyze the SQL statement in `ws03_10.sql`.

```
SQL> get ws03_10
  1  SELECT *  -- ws03_10.sql
  2     FROM (SELECT prod_id
  3          ,        prod_name
  4          ,        prod_desc
  5          ,        prod_list_price
  6          ,        prod_min_price
  7     FROM products
  8     ORDER BY prod_min_price ASC)
  9* WHERE ROWNUM <= 10
SQL> /
```

Notes:

This is an Top-N query statement. The subquery includes the `ORDER BY` clause to ensure that the ranking is in the desired order. For results retrieving the largest values, a `DESC` parameter is needed. The outer query is used to limit the number of rows in the final result set.
Workshop 3: Joins (continued)

Additional Challenge Exercises

14. Consider the SQL statement in ws03_11.sql.

```
SQL> get ws03_11
  1  select p.prod_desc     -- ws03_11.sql
  2    ,  s.amount_sold
  3  from   products  p
  4    ,  channels  c
  5    ,  sales     s
  6  where  s.promo_id = 10
  7   and    s.channel_id = 'I'
  8   and    s.prod_id = p.prod_id
  9* and    s.channel_id   = c.channel_id
SQL> /
```

Notes:

To make this a star join, you must create an index with the following properties:

- It must be a concatenated index.
- The columns must correspond to the foreign key constraints to the smaller lookup tables (also called the dimension tables).
Workshop 3: Joins (continued)

Additional Challenge Exercises

Create an index satisfying the following requirements:

**Note:** This will take some time.

```sql
@ci
  on which table : sales
  on which column(s): channel_id, prod_id
```

Now you have a 3-table join that satisfies the conditions for a start join:

- The join uses at least 3 tables.
- The largest (fact) table has a concatenated index on the foreign key columns.
- There are no conflicting access or join methods.

Analyze the statement in `ws03_11.sql` again, and identify the special features of a star join. Then add a `STAR` hint, and rerun the statement.

**Notes:**

**Instructor Note**

Star queries are unusual, and difficult for query optimizers, because the optimal strategy requires that the smaller tables (CUSTOMERS, PRODUCTS, PROMOTIONS, and TIMES) undergo Cartesian-product joins. That is, these smaller tables are joined together despite the fact that there are no join predicates between them. In general, Cartesian-product joins are expensive and should be avoided. However, for star queries, it is more efficient to use Cartesian-product joins on DIMENSION tables than to repeatedly access the data from the FACT table.
Workshop 3: Joins (continued)

Workshop Solutions.

2. The join does not have any nonjoin predicates, so all rows from the CUSTOMERS table and about 30 percent of the rows in the COUNTRIES table are retrieved. Thus, a sort/merge seems like a reasonable choice of join operation.

3. The nested loops operation can begin to return rows as soon as the first row from the outer table that has rows from the inner table that satisfies the join condition has been retrieved. This optimizes for the first row returned as fast as possible. The sort/merge operation must retrieve all rows from both sources, sort them, and merge the two sorted lists before the first resulting row can be returned.

4. The hash join operation also retrieves at least one full partition from each row source and performs the hashing on the smallest partition before the first resulting row can be returned.

9. Only a nested loops join is possible, because this is the only join operation that allows for nonequijoins.

10. Using an ORDERED or LEADING hint will not change the results. The SALES table (outer join table) is last.

11. The SALES table can be the outer table because the optimizer recognizes the outer join being disabled, thus the statement is treated as a normal join instead.

12. The SALES table cannot act as driving table, because an outer join requires the non-outer joined table to be the driving table.

Workshop Summary

After completing this workshop, you should have learned the following:

- The optimizer goal influences the join operation.
- The join operation that executes the join influences the performance of the join.
- Join order is important for nested loops join operations.
- It is vital for optimal performance to limit the throwaway of rows retrieved but not used.
- Hash join operations are in most cases (all those in the workshop, that is) preferred for sort/merge join operations.
- Using predicates with the outer join mark (+) on the outer joined table disables the outer join functionality.
- The optimizer goal FIRST_ROWS prefers nested loops join operations, whereas ALL_ROWS prefers sort/merge or hash join operations.
- You cannot always avoid excessive throwaway of rows due to the data model.
Workshop 4: Subqueries

Workshop Objectives

- Optimize subqueries
- Explore methods of finding the optimal execution plan for correlated subqueries
- Explore and optimize antijoins
- Explore and optimize semijoins

Introduction

In this workshop, you focus on SQL statements that are not joins in the traditional way, but also reference more than one table: queries based on subqueries.

As with earlier workshops, it is important to focus on the number of rows processed, so you must use either SQL Trace (with TKPROF). You can use SQL*Plus AUTOTRACE as well, using elapsed time and logical I/O as performance indicators.

This workshop can be done entirely in the SQL*Plus environment. Try to work together in (small) groups, and discuss the workshop results. Each time you load a new SQL statement, try to predict what the optimizer will do before running the statement. If you have some time left, feel free to experiment; for example, by creating additional indexes or changing the SQL statements. Take notes during the workshop as an aid for the wrap-up discussion.

Scripts Reference

<table>
<thead>
<tr>
<th>Script name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws04_stats.sql</td>
<td>Script to run to initialize this workshop</td>
</tr>
<tr>
<td>hashfalse.sql</td>
<td>Alter session set hash_join_enabled = false</td>
</tr>
<tr>
<td>hashtrue.sql</td>
<td>Alter session set hash_join_enabled = true</td>
</tr>
<tr>
<td>allrows.sql</td>
<td>Alter session set optimizer_goal = all_rows</td>
</tr>
<tr>
<td>firstrows.sql</td>
<td>Alter session set optimizer_goal = first_rows</td>
</tr>
<tr>
<td>atto.sql</td>
<td>Set autotrace traceonly</td>
</tr>
<tr>
<td>dai.sql</td>
<td>Drop all indexes (on a given table)</td>
</tr>
<tr>
<td>attox.sqlci.sql</td>
<td>Create (regular) index</td>
</tr>
<tr>
<td>atoff.sql</td>
<td>Set autotrace off</td>
</tr>
</tbody>
</table>

Oracle9i: SQL Tuning Workshop A-35
Workshop 4: Subqueries (continued)

1. Run the `ws04_stats.sql` script and verify the following initialization parameter settings by using the `ws04_01.sql` script:

   ```sql
   hash_join_enabled = true
   optimizer_mode    = all_rows or choose
   SQL> @ws04_stats
   SQL> get ws04_01
       1 select name,value           -- ws04_stats.sql
       2 from   v$parameter
       3 where  name in ('hash_join_enabled'
       4*                 ,'optimizer_mode')
   SQL> /
   SQL> @ws04_01
   
   Notes:
   
   HASH_JOIN_ENABLED and OPTIMIZER_MODE are dynamic parameters that can be changed using the alter session command.
Workshop 4: Subqueries (continued)

2. Set AUTOTRACE to TRACE ONLY and consider the SQL statement in ws04_02.sql.

```
SQL> @dai
on which table: customers
SQL> @atto
SQL> get ws04_02
1  select c1.cust_first_name            --ws04_02
2  ,      c1.cust_last_name
3  ,      c1.cust_year_of_birth
4  from   customers c1
5  where  c1.cust_year_of_birth =
6        (select max(c2.cust_year_of_birth)
7         from   customers c2
8        where  c1.country_id = c2.country_id)

SQL> /
```

This statement retrieves the customers with the oldest birth year in every country.

This type of statement can be found in many real-life situations. What is happening in the execution plan? Are the results satisfactory?

Create an index on the COUNTRY_ID column, and measure the performance improvement.

```
SQL> @ci
on which table    : customers
on which column(s): country_id

SQL? @ws04_02
```

Is creating the index a better choice?

Notes:
Workshop 4: Subqueries (continued)

3. Use an editor to rewrite the statement in ws04_02.sql to a normal join. (Hint: Move the correlated subquery to the FROM clause or use script ws04_02a.sql.) Find the optimal execution plan (join operations, join order if applicable, and access paths) for this rewritten statement.

Notes:

What happens if you disable hash joins?

SQL> @hashfalse
SQL> @ws04_02
SQL> @ws04_02a

4. Consider the SQL statement in ws04_03.sql.

SQL> get ws04_03
1  select cust_id, cust_last_name
2  from customers c1
3  where c1.cust_credit_limit >
4    (select avg(cust_credit_limit)
5    from customers c2
6    where c1.country_id = c2.country_id
7*   group by c2.country_id)

SQL> /

This correlated subquery example processes all the rows of the outer table (C1). Each row in the outer table is checked against every row from the inner condition.

Notes:
Workshop 4: Subqueries (continued)

If you reformulate the query to use a join between two tables, one of which is built up on the spot, is performance improved? Use the ws04_04.sql script for this purpose.

```
SQL> get ws04_04
1  select cust_id, cust_last_name -- ws04_04.sql
2  from   customers c1
3  ,  (select country_id, avg(cust_credit_limit) avg_credit
4      from   customers
5      group by country_id) avgtab
6  where  c1.cust_credit_limit > avgtab.avg_credit
7* and    c1.country_id = avgtab.country_id
SQL> /
```

Notes:

Alternatively, you can use a view to construct the intermediate table AVGTAB but sometimes this view gets expanded improperly. The NO_MERGE hint can prevent this expansion of the query.

5. Analyze the SQL statement in ws04_05.sql by using SQL Trace and TKPROF.

```
SQL> @hashtrue
SQL> @atoff
SQL> alter session set timed_statistics = true;
SQL> alter session set sql_trace = true;
SQL> get ws04_05
1  select c.cust_last_name           -- ws04_05
2  from   customers c
3  where  c.country_id = 'US'
4  and    c.cust_id NOT IN (select s.cust_id
5*                          from   sales s)
SQL> /
```

This is a SELECT statement with a subquery. Because the predicate with the subquery contains a NOT IN operator, this statement is also known as an anti-join.
Workshop 4: Subqueries (continued)

Notes:

The default behavior of the Oracle server is to go through the table in the subquery for every row in the main query. By hinting to use a sort/merge or a hash operation instead (by using the MERGE_AJ or HASH_AJ hints in the subquery), there is a good chance of improving the performance of the statement. Try to find the optimal performance for this query by using ws04_06.sql to specify several hints.

```sql
SQL> @aton
SQL> get ws04_06
  1  select c.cust_last_name           -- ws04_06
  2    from   customers c
  3    where  c.country_id = 'US'
  4    and    c.cust_id NOT IN /*+ &hint */
  5          select s.cust_id
  6*          from sales s)
SQL> /
    Enter value for hint: ...
```

Notes:
Workshop 4: Subqueries (continued)

6. Analyze the SQL statement in ws04_07.sql. Make sure to disable SQL Trace and use SQL*Plus AUTOTRACE again.

```sql
SQL> alter session set sql_trace = false;
SQL> @dai
   on which table: sales
SQL> @aton
SQL> get ws04_07
1  SELECT COUNT(*)           -- ws04_07.sql
2   FROM   products p
3   WHERE  EXISTS (SELECT 'x'
4       FROM    sales s
5*      WHERE  p.prod_id = s.prod_id)
SQL> /
```

This is a select statement with a subquery. As the predicate with the subquery contains an EXISTS operator, this statement is known as a semi-join.

**Notes:**

The default behavior of the Oracle server is to go through the table in the subquery for every row in the main query.

By hinting to use a sort/merge or a hash operation instead (by using the MERGE_SJ or HASH_SJ hints in the subquery), there is only a small chance of improving the performance of the statement, because the subquery can use a good (selective) index to evaluate the EXISTS predicate.
Workshop 4: Subqueries (continued)

Try to find the optimal performance of this query.

```sql
SQL> get ws04_08
1  SELECT COUNT(*) -- ws04_08.sql
2       FROM products p
3       WHERE EXISTS (SELECT /*+ &hint */ 'x'
4       FROM sales s
5*       WHERE p.prod_id = s.prod_id)
```

SQL> /

Enter value for hint: ... 

Notes:

Create an index on the PROD_ID column in the SALES table and retest your results.

Note: This may take some time.

```sql
SQL> @ci
on which table : sales
on which column(s): prod_id
```

```sql
SQL> @ws04_08
Enter value for hint...
```

Notes:
Workshop 4: Subqueries (continued)

7. Make sure to drop all indexes on the SALES table first and analyze the SQL statement in ws04_09.sql.

```
SQL> @dai
    on which table: sales
SQL> get ws04_09
1  select /*+ &hint */ count(*)          -- ws04_09.sql
2    from   sales s
3    where  exists (select 'x'
4                from   customers c
5*               where  s.cust_id = c.cust_id)
SQL> /
```

Notes:

This is also a semi-join statement, as in the previous exercise. Try to find the optimal performance for this query by using the hints MERGE_SJ or HASH_SJ in the subquery.

This is an example of a statement where the lack of a usable index can be circumvented by using nondefault execution methods. This is important to remember when you tune statements that must run sporadically but still require acceptable performance.
Workshop 4: Subqueries (continued)

Workshop Solutions.

2. Oracle9i has made performance improvements with correlated subquery processing. Relatively few db block gets and consistent gets are required in this example. Creating an index on the COUNTRY_ID columns does result in the index being used and the number of db block gets is decreased. However, without the index, the consistent gets is lower. (In release 8i, the index resulted in better performance.)

3. See the ws04_02a.sql script. The Oracle9i optimizer creates a plan that is equivalent to the correlated subquery on ws04_02.sql. Changing the hash enabled parameter to false increases the costs and the number of db block gets.

Workshop Summary

After completing this workshop, you should have learned the following:

- In-line views can be used to help tune correlated subqueries.
- Exact numbers for the number of rows thrown away is important for tuning complex joins, so the best approach is to use SQL Trace/TKPROF.
- Be careful to define views that cannot be merged into the SELECT statement.

Note: In release 8i, the optimizer treats subqueries differently than in release 9i. Some of the results are much more significant in release 8i.
Workshop 5: Multiple Predicates

Workshop Objectives
- Tune SQL statements using the AND operator
- Tune SQL Statements using the OR and IN operators
- Set up and understand concatenated indexes
- Learn the benefits from bitmapped indexes
- Use function-based indexes

Introduction
In this workshop, the SQL statements have a compound WHERE clause, consisting of multiple predicates combined with AND and OR operators. You can tune these statements by creating indexes. In this workshop, you use hints to influence the optimizer’s behaviour. You also investigate how the optimizer can benefit from creating concatenated indexes, bitmapped indexes, and function-based indexes.

Try to work together in small groups, and discuss the workshop results. Each time you load a new SQL statement, try to predict what the optimizer will do before running the statement. If you have some time left, feel free to experiment by creating, for example, additional indexes or changing the SQL statements. Take notes during the workshop as an aid for the wrap-up discussion. For example, you could make a note about the execution plan, the associated cost, and the amount of I/O for each SQL statement.

Scripts Reference

<table>
<thead>
<tr>
<th>Script name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws05_stats.sql</td>
<td>Script to run before you start this workshop</td>
</tr>
<tr>
<td>li.sql</td>
<td>Lists all indexes; accepts table name as argument; wildcards (%,_ ) in table names are allowed (Default: the previous tablename)</td>
</tr>
<tr>
<td>cbi.sql</td>
<td>Creates a bitmapped index; same behaviour as ci.sql</td>
</tr>
<tr>
<td>dai.sql</td>
<td>Drops all indexes; prompts for a table name</td>
</tr>
<tr>
<td>index.sql</td>
<td>Queries the index names and types for a given table</td>
</tr>
<tr>
<td>ci.sql</td>
<td>Creates an index; prompts for table name and column names (Column names should be separated with commas; the index name is generated by the script.)</td>
</tr>
<tr>
<td>aton.sql</td>
<td>Set autotrace on</td>
</tr>
<tr>
<td>attox.sql</td>
<td>Set autotrace traceonly explain</td>
</tr>
<tr>
<td>atto.sql</td>
<td>Set autotrace traceonly</td>
</tr>
<tr>
<td>attoff.sql</td>
<td>Set autotrace off</td>
</tr>
<tr>
<td>atonx.sql</td>
<td>Set autotrace on explain</td>
</tr>
<tr>
<td>rewrite.sql</td>
<td>Enables query rewrite</td>
</tr>
</tbody>
</table>

Oracle9i: SQL Tuning Workshop A-45
Workshop 5: Multiple Predicates (continued)

1. Run the ws05_stats script first to have a clean workshop start. Drop all existing indexes on the CUSTOMERS table and then create indexes on the following columns: CUST_GENDER, CUST_POSTAL_CODE, and CUST_CREDIT_LIMIT. Check the indexes that are available for the SALES table.

```sql
SQL> @ws05_stats
SQL> ALTER TABLE customers drop primary key cascade;

SQL> @dai
  on which table: customers

SQL> @ci
  on which table: customers
  on which column(s): cust_gender
Creating index on: customers cust_gender
Enter value for index_name: I_CUSTOMERS_CUST_GENDER

SQL> @ci
  on which table: customers
  on which column(s): cust_postal_code
Creating index on: customers cust_postal_code
Enter value for index_name: I_CUSTOMERS_CUST_POSTAL_CODE

SQL> @ci
  on which table: customers
  on which column(s): cust_credit_limit
Creating index on: customers cust_credit_limit
Enter value for index_name: I_CUSTOMERS_CUST_CREDIT_LIMIT

SQL> @index
  Enter value for table_name: customers
```

Notes:
Workshop 5: Multiple Predicates (continued)

2. Enable AUTOTRACE, and get query ws05_01.sql. The WHERE clause contains three predicates. Execute this statement, and take notes about the indexes used, the cost of the execution plan, and the amount of I/O performed.

```
SQL> @atto
SQL> get ws05_01
1  select c.*                  -- ws05_01.sql
2  from   customers c
3  where  cust_gender   = 'M'
4  and    cust_postal_code = 40804
5* and   cust_credit_limit = 10000
SQL> / 
SQL> @atoff
```

Notes:

3. Set AUTOTRACE to explain only. Examine query ws05_02.sql. The statement contains an INDEX hint. Run this statement with different indexes and take notes about the results. Alternately examine and run ws05_02b.sql and ws05_02c.sql which use other indexes and take notes of these results.

```
SQL> @atoox
SQL> get ws05_02
1  select /*+ INDEX (r I_CUSTOMERS_CUST_CREDIT_LIMIT) */
2   c.*                  -- ws05_02.sql
3  from   customers c
4  where  cust_gender   = 'M'
5* and   cust_postal_code = 40804
6 and   cust_credit_limit = 10000
SQL> / 
```

Notes:
Workshop 5: Multiple Predicates (continued)

```
SQL> get ws05_02b
1 select /*+ INDEX (r I_CUSTOMERS_CUST_GENDER) */
2   c.*                  -- ws05_02b.sql
3  from   customers c
4  where  cust_gender   = 'M'
5  and    cust_postal_code = 40804
6 and   cust_credit_limit = 10000
SQL> /
```

Notes:

```
SQL> @ws05_02c
1 select /*+ INDEX (r I_CUSTOMERS_CUST_POSTAL_CODE) */
2   c.*                  -- ws05_02c.sql
3  from   customers c
4  where  cust_gender   = 'M'
5  and    cust_postal_code = 40804
6 and   cust_credit_limit = 10000
SQL> /
```

Notes:

```
SQL> @atoff
```

Notes:
Workshop 5: Multiple Predicates (continued)

4. Set AUTOTRACE on. Examine query ws05_03a.sql. Now the statement contains an AND_EQUAL hint. This hint accepts two or more index names, forcing the optimizer to merge those indexes. Run this script with different index names and make notes of indexes being used. Run the index script to obtain index names.

```
SQL> @index
Enter value for tab: customers

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>INDEX_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_CUSTOMERS_CUST_CREDIT_LIMIT</td>
<td>NORMAL</td>
</tr>
<tr>
<td>I_CUSTOMERS_CUST_GENDER</td>
<td>NORMAL</td>
</tr>
<tr>
<td>I_CUSTOMERS_CUST_POSTAL_CODE</td>
<td>NORMAL</td>
</tr>
</tbody>
</table>
```

SQL> @atto

```
SQL> get ws05_03
1  select /*+ AND_EQUAL (r &index_name1, &index_name2) */
2    c.*                  -- ws05_03a.sql
3  from   customers c
4  where  cust_gender   = 'M'
5* and   cust_postal_code = 40804
6* and   cust_credit_limit = 10000
SQL> /
```

Enter value for index_name: I_CUSTOMERS_CUST_CREDIT_LIMIT
Enter value for index_name: I_CUSTOMERS_CUST_GENDER

```
SQL> @atoff
```

Notes:

Try more combinations for index_name1 and index_name2. Also try to enter all three index names by entering two index names after one of the prompts.

Notes:
Workshop 5: Multiple Predicates (continued)

5. Drop the indexes again, and replace them with a single concatenated index; then run ws05_01.sql again. Also change the AUTOTRACE setting to suppress statement results.

    SQL> @dai
        on which table: customers
    SQL> @ci
        on which table : customers
        on which column(s): cust_gender, cust_credit_limit, cust_postal_code

Note: The index creation may take a while.

    SQL> @atto
    SQL> get ws05_01
       1  select c.*                  -- ws05_01.sql
       2    from   customers c
       3    where  cust_gender = 'M'
       4    and    cust_postal_code = 40804
       5*   and   cust_credit_limit = 10000
    SQL> /

This is the best approach so far. The concatenated index acts like a premerged set of indexes. This is an ideal situation because the WHERE clause contains a predicate for all three columns in the concatenated index.

Notes:
Workshop 5: Multiple Predicates (continued)

6. Now investigate what happens when not all columns of a concatenated index are present in a predicate. Get `ws05_04a.sql` and compare it with the original statement in `ws05_01.sql`.

```sql
SQL> get ws05_04a
1  select
2  c.*                  -- ws05_04a.sql
3  from   customers c
4  where  cust_gender = 'M'
5* and  cust_credit_limit = 10000
SQL> /
```

Notes:

Get `ws05_04b.sql` and compare it with the original statement in `ws05_01.sql`. Now the predicate on the `cust_credit_limit` column is removed.

```sql
SQL> get ws05_04b
1  select
2  c.*                  -- ws05_04b.sql
3  from   customers c
4  where  cust_gender = 'M'
5* and  cust_postal_code = 40804
SQL> /
```

Get `ws05_04c.sql` and compare it with the original statement in `ws05_01.sql`. Now the predicate on the `cust_gender` column is removed.

```sql
SQL> get ws05_04c
1  select
2  c.*                  -- ws05_04c.sql
3  from   customers c
4  where  cust_postal_code = 40804
5* and  cust_credit_limit = 10000
SQL> /
```

Notes:
Workshop 5: Multiple Predicates (continued)

7. You could investigate one more option: Drop all indexes on the `CUSTOMERS` table, and create three bitmapped indexes. Start `ws05_01.sql` again and edit the statement to specify an `INDEX_COMBINE` hint when needed to force bitmapped index usage. Compare the results with the concatenated index approach.

```sql
SQL> @dai
  on which table: customers

SQL> @cbi
  on which table : customers
  on which column(s): cust_gender

SQL> @cbi
  on which table : customers
  on which column(s): cust_postal_code

SQL> @cbi
  on which table : customers
  on which column(s): cust_credit_limit

SQL> @ws05_01
SQL> /
```

Notes:

The optimizer may not use the bitmap index even with a hint.

8. Make sure that you have a normal B*-tree index on the `cust_year_of_birth` column, and `cust_credit_limit` column. Now get `ws05_05.sql`. This time you see two predicates combined with an `OR` operator.

```sql
SQL> @dai
  on which table: customers

SQL> @ci
  on which table : customers
  on which column(s): cust_year_of_birth

SQL> @ci
  on which table : customers
  on which column(s): cust_credit_limit
```
Workshop 5: Multiple Predicates (continued)

SQL>@atto
SQL> get ws05_05
1  select c.*                  -- ws05_05.sql
2  from   customers c
3  where  c.cust_year_of_birth   = 1953
4* or    c.cust_credit_limit  = 10000
SQL> /

Notes:

You see that both indexes are used and combined with a CONCATENATION operator.
Investigate what happens if you drop the index on the cust_year_of_birth column:

SQL> drop index I_CUSTOMERS_CUST_YEAR_OF_BIRTH;
SQL> @ws05_05
SQL> /

This time the optimizer apparently prefers to perform a full table scan. Why?

Notes:

9. Examine and start ws05_06.sql. This creates a primary key constraint and unique index on the CUST_ID column of CUSTOMERS table. Run index.sql to check available indexes. Now get and run query ws05_08.sql.

SQL> @ws05_06.sql
SQL> @index
Enter value for table_name: customers
SQL> get ws05_08
1  select c.*                  -- ws05_08.sql
2  from   customers c
3* where  cust_id in (88340,104590,44910)
SQL> /

Oracle9i: SQL Tuning Workshop A-53
Workshop 5: Multiple Predicates (continued)

The optimizer uses the primary key index. This is possible because CUST_ID is the leading column of this index.

Notes:

10. Drop all indexes on the customers table, and create three bitmapped indexes again, then get query ws05_09.sql. This statement has a complicated WHERE clause. Bitmapped indexes are good for this type of statement; you see several bitmap operations in the execution plan.

```
SQL> @dai
    on which table: customers
SQL> @cbi
    on which table: customers
    on which column(s): cust_year_of_birth
SQL> @cbi
    on which table: customers
    on which column(s): cust_postal_code
SQL> @cbi
    on which table: customers
    on which column(s): cust_credit_limit
SQL> attox
SQL> get ws05_09
  1  select   c.*
  2  from   customers c
  3  where  (c.cust_year_of_birth = '1970' and
  4   c.cust_postal_code = 40804)
  5* and not  cust_credit_limit = 15000
SQL>/
```

Notes:
Workshop 5: Multiple Predicates (continued)

11. Drop all indexes and create a normal B*-tree index on the CUST_CREDIT_LIMIT column. Start query ws05_10.sql. Make a note about the estimated cost. The CUST_CREDIT_LIMIT column in the customers table contains skewed data (ws05_10a.sql displays how skewed the data is). Create a histogram for the CHANNEL_ID column and run ws05_10.sql again. Note the estimated cost.

```
SQL> @dai
  on which table: customers
SQL> @ci
  on which table : customers
  on which column(s): country_id
SQL>@atto
SQL> get ws05_10
  1 select s.*                  -- ws05_10.sql
  2 from   customers s
  3* where  country_id in ('US','TR')
SQL> /
SQL> @atoff
SQL> get ws05_10a
  1> select  country_id, count(*)
  2> from   customers
  3> group  by  country_id
SQL> /
SQL> analyze table customers compute statistics for table
  2> for columns country_id size 100;
SQL> @atto
SQL> get ws05_10
SQL> /
```

Note what happens when you change US to SA.

```
SQL> c/US/SA
SQL>/
```

Note the cost listed with the execution plan.

The optimizer can make this intelligent decision based on the histogram statistics.

Notes:
Workshop 5: Multiple Predicates (continued)

12. Next, investigate the benefits of fast full index scans.

```sql
SQL> @dai
  on which table: customers
SQL> @atto
SQL> get ws05_11
1  select c.cust_last_name          -- ws05_11.sql
2    ,      c.cust_first_name
3* from   customers c
SQL> /
```

```sql
SQL> @ci
  on which table    : customers
  on which column(s): cust_last_name, cust_first_name

SQL> @ws05_11
SQL> /
```

As you see, the optimizer benefits from a fast full index scan. Remember that the Oracle9i Server uses multiblock reads but does not guarantee any ordering. When you add an ORDER BY clause, you see a sort operation in the execution plan.

Notes:

13. Now examine the COUNT function. The COUNT function can benefit from bitmapped indexes by counting the number of ones in a bitmap, which is an efficient operation.

```sql
SQL> @aton
SQL> get ws05_13
1  select count(*) credit_limit
2  from   customers        -- ws05_13.sql
3* where  cust_credit_limit = 10000;
SQL> /
```

Oracle9i: SQL Tuning Workshop A-56
Workshop 5: Multiple Predicates (continued)

SQL>@dai
   on which table : customers

SQL> @cbi
   on which table : customers
   on which column: cust_credit_limit

SQL> @ws05_13
SQL> /

Notes:

The I/O is reduced considerably. If you have some time left, replace the bitmapped index with a normal one. Then you see that a normal index also improves performance; however, the bitmapped index is roughly 10 times more efficient. Note that bitmapped indexes are only considered by the CBO, so you must analyze your tables or force cost-based optimization.

14. Finally, investigate the benefits of function-based indexes. Drop all indexes on the CUSTOMERS table, analyze the table and start query ws05_14.sql.

SQL> @dai
   on which table: customers

SQL> @atto
SQL> get ws05_14
  1  select cust_id, country_id
  2  from customers -- ws05_14.sql
  3* where lower( cust_last_name) like 'gentle'
SQL> /

Notes:
Workshop 5: Multiple Predicates (continued)

Create a function-based index which utilizes the \texttt{LOWER} function on the \texttt{CUST\_LAST\_NAME} column. Rerun \texttt{ws05\_14.sql} and compare the results.

\texttt{SQL> create index lower\_cust\_last\_name\_idx on}
\texttt{2> customers(lower(cust\_last\_name));}

\texttt{SQL> @rewrite}
\texttt{SQL> @ws05\_14}
\texttt{SQL> /}

Notes:

As a last step, run the \texttt{ws05\_15.sql} script to recreate the primary key index on the \texttt{CUSTOMERS} table for the remaining workshops.

\texttt{SQL> get ws05\_15}
\texttt{1 ALTER TABLE customers}
\texttt{2* ADD CONSTRAINT customers\_pk PRIMARY KEY (cust\_id)}
\texttt{SQL> /}
Table altered.
Workshop 5: Multiple Predicates (continued)

Workshop Solutions.

1. Not all indexes are used even though the columns are referenced in the \texttt{WHERE} clause.

2. Even with hints all indexes may not be used. The same applies to exercise 3.

7. The optimizer cannot use one or more of the bitmapped indexes, even with hints. However the cost if the indexes are used is lower than a full table scan. The optimizer prefers a full table scan because it is only useful to use indexes for an \texttt{OR} predicate when both sides of the predicate are indexed; for the unindexed side, a full table scan is needed.

8. The \texttt{GETS} are less without using the index so the optimizer prefers a full table scan.

14. Function-based indexes defined on \texttt{UPPER} (column\_name) or \texttt{LOWER}(column\_name) can facilitate case-insensitive searches by using the function instead of performing a full table scan.

Workshop Summary

After completing this workshop, you should have learned the following:

- Up to five indexes can be merged to optimize predicates combined with \texttt{AND}.
- Concatenated indexes usually offer better performance because they are premerged. You do not need predicates on each column of the concatenated index; however, make sure that the leading column is specified.
- Be careful with \texttt{OR} constructs; as soon as the left or right side is unindexed, the index on the other side is useless.
- Bitmapped indexes are efficient in case of complex \texttt{WHERE} clauses with many \texttt{AND}, \texttt{NOT}, and \texttt{OR} constructs. This is based on fast internal bit-level operations.
- The \texttt{COUNT} function can benefit from bitmapped indexes by using an efficient built-in operator that counts the number of ones in a bitmap.
- Histograms enable the optimizer to better estimate execution plan costs, particularly when the data is skewed.
- Fast full index scans are an alternative to full table scans when an index contains all the columns that are needed for a query. Fast full index scans cannot be used to eliminate a sort operation. They read the entire index by using multiblock I/O.
- Function-based indexes can facilitate processing queries.
Workshop 6: Views

Workshop Objectives

- Examine the performance of views
- Examine the difference between mergeable and nonmergeable views
- Examine the properties of materialized views
- Examine the performance of materialized views

Introduction

In this workshop, you focus on queries based on a views and their impact on query performance. You examine the characteristics of mergeable views and nonmergeable views, and create a materialized view and analyze its performance.

This workshop can be done entirely in the SQL*Plus environment. Try to work together in small groups and discuss the workshop results. Each time you load a new SQL statement, try to predict what the optimizer will do before running the statement. If you have some time left, feel free to experiment; for example, by creating additional indexes or changing the SQL statements. Take notes during the workshop as an aid for the wrap-up discussion.

Scripts Reference

<table>
<thead>
<tr>
<th>Script name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws06_stats.sql</td>
<td>Script to run before you start this workshop</td>
</tr>
<tr>
<td>atoff.sql</td>
<td>Set autotrace off</td>
</tr>
<tr>
<td>atto.sql</td>
<td>Set autotrace traceonly</td>
</tr>
<tr>
<td>aton.sql</td>
<td>Set autotrace on</td>
</tr>
<tr>
<td>attox.sql</td>
<td>Set autotrace traceonly explain</td>
</tr>
<tr>
<td>atonx.sql</td>
<td>Set autotrace on explain</td>
</tr>
<tr>
<td>ci.sql</td>
<td>Creates an index; prompts for table name and column names (Column names should be separated with commas; the index name is generated by the script.)</td>
</tr>
<tr>
<td>dai.sql</td>
<td>Drops all indexes; prompts for a table name</td>
</tr>
</tbody>
</table>
Workshop 6: Views (continued)

1. Run the ws06_stats script first to have a clean workshop start. Run the SQL statement in ws06_01.sql. This script creates a view called V_SALES_DETAIL, based on an outer join.

   SQL> @ws06_stats
   SQL> get ws06_01

   1  create or replace view v_sales_detail  -- ws06_01.sql
   2    as
   3     select sa.prod_id
   4     ,      pr.prod_name
   5     ,      pr.supplier_id
   6     ,      pr.prod_status
   7     ,      sa.amount_sold
   8     ,      to_char(sa.time_id, 'YYYY-MON-DD') date_sold
   9     from   sales sa
   10     ,      products pr
   11*    where  pr.prod_id(+) = sa.prod_id

   SQL> /
   SQL> describe v_sales_detail

   Notes:

2. Drop indexes on the SALES and PRODUCTS tables. Compare the SQL statements in ws06_02.sql and ws06_03.sql.

   SQL> @dai
         on which table: sales
   SQL> @dai
         on which table: products
   SQL> set timing on
   SQL> @atto
Workshop 6: Views (continued)

SQL> get ws06_02
1  select v.prod_name                    -- ws06_02.sql
2     ,      v.prod_status
3     ,      v.date_sold
4  from   v_sales_detail v
5* where  v.prod_id = 15180
SQL> /

SQL> get ws06_03
1  select pr.prod_name                   -- ws06_03.sql
2     ,      pr.prod_status
3     ,      to_char(sa.time_id, 'YYYY-MON-DD') date_sold
4  from   sales sa
5     ,      products pr
6* where  pr.prod_id = sa.prod_id (+)
7     and  sa.prod_id = 15180
SQL> /

These two statements produce the same rows (although they may be differently sorted). Why is there a difference in the execution plans?

Notes:

3. Now compare the SQL statements in ws06_04.sql and ws06_05.sql.

SQL> get ws06_04
1  select pr.prod_name                    -- ws06_04.sql
2     ,      pr.prod_status
3     ,      to_char(sa.time_id, 'YYYY-MON-DD') date_sold
4  from   sales sa
5     ,      products pr
6* where  pr.prod_id = sa.prod_id (+)
7     and  sa.prod_id = 15180
SQL> /
Workshop 6: Views (continued)

SQL> get ws06_05
  1  select pr.prod_name                   -- ws06_05.sql
  2    , pr.prod_status
  3    , to_char(sa.time_id, 'YYYY-MON-DD') date_sold
  4  from   sales sa
  5    , products pr
  6  where pr.prod_id = sa.prod_id (+)
  7* and sa.prod_id in (4680, 15180)
SQL> /

Analyze these two statements. Only the WHERE clause has changed. Why are the execution plans different?

Notes:
Workshop 6: Views (continued)

4. Compare the SQL statements in `ws06_02.sql` and `ws06_06.sql`

```sql
SQL> get ws06_02
1  select v.prod_name                      -- ws06_02.sql
2    ,      v.prod_status
3    ,      v.date_sold
4  from   v_sales_detail v
5* where  v.prod_id = 15180
SQL> /

SQL> get ws06_06
1  select v.prod_name                      -- ws06_06.sql
2    ,      v.prod_status
3    ,      v.date_sold
4  from   v_sales_detail v
5* where  v.prod_name = 'Sunburst Dress'
SQL> /
```

Why are the execution plans different?

Notes:

5. Run the SQL statement in `ws06_07.sql`. This script creates a view called `V_CUSTYOB_1962`, based on customers born in 1962.

```sql
SQL> get ws06_07
1  create or replace view v_custyob_1962   -- ws06_07.sql
2    as select cust_id
3    ,         cust_last_name
4    ,         cust_income_level
5  from customers
6* where cust_year_of_birth = 1962
SQL> /
```
Workshop 6: Views (continued)

Consider the query in `ws06_08.sql` that accesses the view. The query selects the IDs greater than 300000 of customers who are born in 1962:

```sql
SQL> get ws06_08
1  select cust_id
2  from v_custyob_1962
3* where cust_id > 300000
SQL> /
```

Is the index on CUST_ID used?

Notes:

6. Run the SQL statement in `ws06_09.sql`. This script creates a view called `V_CUST_CREDIT_LIMIT`, based on the average credit limits per country.

```sql
SQL> get ws06_09
1  create or replace view v_avg_credit_limit – `ws06_09.sql`
2  as select country_id
3  , avg(cust_credit_limit) AVG_CREDIT
4  from customers
5* group by country_id
SQL> /
```

Consider the query in `ws06_10.sql` that accesses the view. The query selects the average credit limits for customers for a country code.

```sql
SQL> get ws06_10
1  select * -- `ws06_10.sql`
2  from v_avg_credit_limit
3* where country_id = 'US'
SQL> /
```

Notes:
Workshop 6: Views (continued)

Create an index on the COUNTRY_ID column in the CUSTOMERS table, then rerun the ws06_10.sql script and compare the results.

SQL> @dai
   on which table: customers
SQL> @ci
   on which table : customers
   on which column(s): country_id
SQL> get ws06_10
   1  select *                                 -- ws06_10.sql
   2   from v_avg_credit_limit
   3*  where country_id = 'US'
   SQL> /

Notes:

7. Run the SQL statement in ws06_11.sql. This script creates a view which contains a SELECT statement with the UNION operator. Run the ws06_12.sql script to query data from the view.

SQL> get ws06_11
   1  create or replace view v_99_00_times     -- ws06_12.sql
   2   as
   3   select *
   4   from times
   5   where calendar_year = '1999'
   6   union
   7   select *
   8   from times
   9*  where calendar_year = '2000'
   SQL> /
Workshop 6: Views (continued)

SQL> get ws06_12

1  select calendar_month_number             -- ws06_12.sql
2   ,     calendar_month_name
3   ,     calendar_quarter_desc
4   ,     calendar_quarter_number
5   from   v_99_00_times
6* where  calendar_year = '1999'

SQL> /

Notes:

Create an index on the CALENDAR_YEAR column in the TIMES table. Rerun the
ws06_12.sql script and compare the results. Is the index used? Why or why not?

SQL> @dai
on which table: times

SQL> @ci
on which table : times
   on which column(s): calendar_year

SQL> get ws06_12

1  select calendar_month_number             -- ws06_12.sql
2   ,     calendar_month_name
3   ,     calendar_quarter_desc
4   ,     calendar_quarter_number
5   from   v_99_00_times
6* where  calendar_year = '1999'

SQL> /

Notes:
Workshop 6: Views (continued)

8. Run the SQL statement in ws06_13.sql. This script creates a materialized view called FQUARTER_PSCAT_COSTS_MV.

```
SQL> get ws06_13
1  CREATE MATERIALIZED VIEW          -- ws06_13.sql
2   fquarter_pscat_costs_mv
3  ENABLE QUERY REWRITE
4  AS
5   SELECT   t.fiscal_quarter_number
6   ,        t.fiscal_quarter_desc
7   ,        t.fiscal_year
8   ,        p.prod_subcategory
9   ,        sum(c.unit_cost) AS dollars
10  FROM     costs c
11  ,        times t
12  ,        products p
13  WHERE    c.time_id = t.time_id
14  AND      c.prod_id = p.prod_id
15  GROUP BY t.fiscal_quarter_number
16  ,        t.fiscal_quarter_desc
17  ,        t.fiscal_year
18* ,        p.prod_subcategory
SQL> /
```
Workshop 6: Views (continued)

Consider the query in ws06_14.sql that accesses the materialized view. The query selects data from the materialized view for a specific city.

```
SQL> get ws06_14
  1  select *                           -- ws06_14.sql
  2    from fquarter_pscat_costs_mv
  3*   where fiscal_year = 1999
SQL> /
```

Create an index on the FISCAL_YEAR column in the materialized view, then rerun the ws06_14.sql script and compare the results.

```
SQL> @ci
  on which table    : fquarter_pscat_costs_mv
  on which column(s): fiscal_year
SQL> get ws06_14
  1  select *                           -- ws06_14.sql
  2    from fquarter_pscat_costs_mv
  3*   where fiscal_year = 1999
SQL> /
```

Notes:
Workshop 6: Views (continued)

Workshop Solutions.

2. The column PROD_ID in the V_SALES_DETAIL view does not map directly to the column PROD_ID in table SALES (because of the outer join). This means that the view cannot be merged into the SELECT, so all the rows from the row source represented by V_SALES_DETAIL are retrieved and then filtered.

3. The Oracle9i Server cannot combine outer joins with OR or IN (...) predicates. When encountering this combination of predicates, the Oracle9i Server executes the outer join as an intermediate row source (retrieving all rows), which are filtered using the OR or IN (...) predicate.

4. The execution plan is different because the WHERE clause in ws06_06.sql (PROD_NAME = 'Sunburst Dress') references a column in the V_SALES_DETAIL view that directly maps to a column in a table (PRODUCTS).

5. The optimizer transforms the query into a query that accesses the view’s base table. If there are indexes on the CUST_YEAR_OF_BIRTH column, the resulting WHERE clause makes them available.

6. The optimizer transforms this query into a query that access the view’s base table, and the index is used after it is created on the COUNTRY_ID column.

7. The index is not used because this is an example of a nonmergeable view. The view has the UNION operator and cannot transform the query into using the index on the underlying base table.

8. A full table scan on this materialized view is not expensive because the view holds only a subset of data from the tables upon which it is based. An index can be created on the materialized view to enhance performance.

Workshop Summary

After completing this workshop, you should have learned the following:

- Consider expanding the view into the statement.
- To merge the view’s query into a referencing query block in the accessing statement, the optimizer replaces the name of the view with the names of its base tables in the query block and adds the condition of the view’s query’s WHERE clause to the accessing query block’s WHERE clause.
- The optimizer can merge a view into a referencing query block when the view has one or more base tables, provided the view does not contain the following: set operators (UNION, UNION ALL, INTERSECT, MINUS), a CONNECT BY clause, a ROWNUM pseudocolumn, or aggregate functions (AVG, COUNT, MAX, MIN, SUM) in the select list. When a view contains one of the following structures, it can be merged into a referencing query block only if complex view merging is enabled.
- Cost-based optimization can use materialized views to improve query performance by automatically recognizing when a materialized view can and should be used to satisfy a request. The optimizer transparently rewrites the request to use the materialized view. Queries are then directed to the materialized view and not to the underlying detail tables or views.
Workshop 7: Alternative Storage Techniques

Workshop Objectives

- Examine the properties of an index-organized table
- Measure performance differences using index-organized tables

Introduction

In this workshop, you focus on alternative storage techniques and their impact on query performance. Changing the physical storage characteristics of tables sometimes results in significant performance improvements for certain queries. In this workshop, you analyze the performance implications of index-organized tables.

As with earlier workshops, it is important to focus on the number of rows processed, thus you must use either SQL Trace (with TKPROF).

SQL*Plus AUTOTRACE can be used as well, using elapsed time and logical I/O as performance indicators.

Try to work together in small groups and discuss the workshop results. Each time you load a new SQL statement, try to predict what the optimizer will do before running the statement. If you have some time left, feel free to experiment by creating, for example, additional indexes or changing the SQL statements. Take notes during the workshop as an aid for the wrap-up discussion.

Scripts Reference

<table>
<thead>
<tr>
<th>Script name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws07_stats.sql</td>
<td>Script to run to clean the workshop</td>
</tr>
<tr>
<td>allrows.sql</td>
<td>Alter session set optimizer_mode = all_rows</td>
</tr>
<tr>
<td>hashfalse.sql</td>
<td>Alter session set hash_join_enabled = false</td>
</tr>
<tr>
<td>hashtrue.sql</td>
<td>Alter session set hash_join_enabled = true</td>
</tr>
<tr>
<td>atto.sql</td>
<td>Set autotrace traceonly</td>
</tr>
<tr>
<td>atoff.sql</td>
<td>Set autotrace off</td>
</tr>
<tr>
<td>aton.sql</td>
<td>Set autotrace on</td>
</tr>
<tr>
<td>attox.sql</td>
<td>Set autotrace traceonly explain</td>
</tr>
<tr>
<td>ci.sql</td>
<td>Create (regular) index</td>
</tr>
</tbody>
</table>
Workshop 7: Alternative Storage Techniques (continued)

1. Run the \texttt{ws07\_stats} script first to have a clean workshop start. Using \texttt{ws07\_01.sql}, verify the following initialization parameters:

\begin{verbatim}
hash\_join\_enabled = false
optimizer\_mode    = all\_rows
\end{verbatim}

\texttt{SQL> @ws07\_stats}
\texttt{SQL> get ws07\_01}
\begin{verbatim}
1  select name, value                  -- \texttt{ws07\_01.sql}
2  from   v$parameter
3  where  name in ('hash\_join\_enabled'
4*                ,'optimizer\_mode')
SQL> /
\end{verbatim}

Notes:

2. Run the \texttt{ws07\_02.sql} script to create an index-organized table on the \texttt{PROMOTIONS} table, \texttt{PROMO\_ID} column, and populate it with data from the \texttt{PROMOTIONS} table.

\texttt{SQL> @atoff}
\texttt{SQL> get ws07\_02}
\begin{verbatim}
1  create table promotions_iot          -- \texttt{ws07\_02.sql}
2  (promo_id number primary key
3    ,  promo_name varchar2(20)
4    ,  promo_subcategory varchar2(30)
5    ,  promo_category  varchar2(30)
6    ,  promo_cost number
7    ,  promo_begin_date date
8    ,  promo_end_date date)
9* organization index
SQL> /
\end{verbatim}
Workshop 7: Alternative Storage Techniques (continued)

Analyze the execution plan for query on the PROMOTIONS_IOT table in the
ws07_03.sql script and compare it to the same query on the PROMOTIONS table in the
ws07_04.sql script.

```
SQL> @attox
SQL> get ws07_03
  1  select *                       -- ws07_03.sql
  2  from promotions
  3* where promo_id > 300
SQL> /
```

```
SQL> get ws07_04
  1  select *                       -- ws07_04.sql
  2  from promotions_iot
  3* where promo_id > 300
SQL> /
```

Notes:

3. Create an index on the PROMO_SUBCATEGORY column in the PROMOTIONS_IOT table,
then run ws07_05.sql and compare the results.

```
SQL> @ci
    on which table : promotions_iot
    on which column(s): promo_subcategory
SQL> get ws07_05
  1  select *                        -- ws07_05.sql
  2  from promotions_iot
  3* where promo_subcategory = 'online discount'
SQL> /
```

Notes:
Workshop 7: Alternative Storage Techniques (continued)

Workshop Solutions.

2. By creating the PROMOTIONS_IOT table, an index range scan rather than a full table scan is performed on the table with the PROMOTIONS table.

3. By creating a secondary index on the PROMO_SUBCATEGORY column on the index-organized table, you provide efficient access to index-organized table by using columns that are not the primary key nor a prefix of the primary key.

Workshop Summary

After completing this workshop, you should have learned the following:

- Because data rows are stored in the index, index-organized tables provide faster key-based access to table data for queries that involve exact match or range search, or both.

- The Oracle server constructs secondary indexes on index-organized tables using logical row identifiers (logical ROWIDs) that are based on the table’s primary key. This logical ROWID optionally includes a physical guess, which identifies the block location of the row. The Oracle server can use these guesses to probe directly into the leaf block of the index-organized table, bypassing the primary key search. Be aware that because rows in index-organized tables do not have permanent physical addresses, the guesses can become old when rows are moved to new blocks.
Diagnostic Tools Reference
**EXPLAIN PLAN Output Table Columns**

The `PLAN_TABLE` used by the EXPLAIN PLAN command contains the following columns:

<table>
<thead>
<tr>
<th><strong>PLAN_TABLE Column</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATEMENT_ID</td>
<td>The value of the optional STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>The date and time when the EXPLAIN PLAN statement was issued.</td>
</tr>
<tr>
<td>REMARKS</td>
<td>Any comment (of up to 80 bytes) you want to associate with each step of the explained plan. If you need to add or change a remark on any row of the <code>PLAN_TABLE</code>, then use the UPDATE statement to modify the rows of the <code>PLAN_TABLE</code>.</td>
</tr>
</tbody>
</table>
| OPERATION             | The name of the internal operation performed in this step. In the first row generated for a statement, the column contains one of the following values:  
  - DELETE STATEMENT  
  - INSERT STATEMENT  
  - SELECT STATEMENT  
  - UPDATE STATEMENT |
| OPTIONS               | A variation on the operation described in the OPERATION column. |
| OBJECT_NODE           | The name of the database link used to reference the object (a table name or view name). For local queries using parallel execution, this column describes the order in which output from operations is consumed. |
| OBJECT_OWNER          | The name of the user who owns the schema containing the table or index. |
| OBJECT_NAME           | The name of the table or index. |
| OBJECT_INSTANCE       | A number corresponding to the ordinal position of the object as it appears in the original statement. The numbering proceeds from left to right, outer to inner with respect to the original statement text. View expansion results in unpredictable numbers. |
| OBJECT_TYPE           | A modifier that provides descriptive information about the object; for example, NON-UNIQUE for indexes. |
| OPTIMIZER             | The current mode of the optimizer. |
### EXPLAIN PLAN Output Table Columns (continued)

<table>
<thead>
<tr>
<th>PLAN_TABLE Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH_COLUMNS</td>
<td>Not currently used.</td>
</tr>
<tr>
<td>ID</td>
<td>A number assigned to each step in the execution plan.</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>The ID of the next execution step that operates on the output of the ID step.</td>
</tr>
<tr>
<td>POSITION</td>
<td>For the first row of output, this indicates the optimizer’s estimated cost of executing the statement. For the other rows, it indicates the position relative to the other children of the same parent.</td>
</tr>
<tr>
<td>COST</td>
<td>The cost of the operation as estimated by the optimizer’s cost-based approach. For statements that use the rule-based approach, this column is null. Cost is not determined for table access operations. The value of this column does not have any particular unit of measurement; it is merely a weighted value used to compare costs of execution plans. The value of this column is a function of the CPU_COST and IO_COST columns.</td>
</tr>
<tr>
<td>CARDINALITY</td>
<td>The estimate by the cost-based approach of the number of rows accessed by the operation.</td>
</tr>
<tr>
<td>BYTES</td>
<td>The estimate by the cost-based approach of the number of bytes accessed by the operation.</td>
</tr>
<tr>
<td>OTHER_TAG</td>
<td>Describes the contents of the OTHER column. (See the next table for more information on the possible values for this column.)</td>
</tr>
</tbody>
</table>
| PARTITION_START   | The start partition of a range of accessed partitions. It can take one of the following values:  
  - \( n \) indicates that the start partition has been identified by the SQL compiler, and its partition number is given by \( n \).  
  - KEY indicates that the start partition will be identified at run time from partitioning key values.  
  - ROW LOCATION indicates that the start partition (same as the stop partition) will be computed at run time from the location of each record being retrieved. The record location is obtained by a user or from a global index.  
  - INVALID indicates that the range of accessed partitions is empty. |
### EXPLAIN PLAN Output Table Columns (continued)

<table>
<thead>
<tr>
<th>PLAN_TABLE Column</th>
<th>Description</th>
</tr>
</thead>
</table>
| PARTITION_STOP    | The stop partition of a range of accessed partitions. It can take one of the following values:  
• n indicates that the stop partition has been identified by the SQL compiler, and its partition number is given by n.  
• KEY indicates that the stop partition will be identified at run time from partitioning key values.  
• ROW LOCATION indicates that the stop partition (same as the start partition) will be computed at run time from the location of each record being retrieved. The record location is obtained by a user or from a global index.  
• INVALID indicates that the range of accessed partitions is empty. |
| PARTITION_ID      | The step that has computed the pair of values of the PARTITION_START and PARTITION_STOP columns. |
| OTHER             | Other information that is specific to the execution step that a user might find useful. |
| DISTRIBUTION      | Stores the method used to distribute rows from producer query servers to consumer query servers. |
| CPU_COST          | The CPU cost of the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, this column is null. The value of this column is proportional to the number of machine cycles required for the operation. |
| IO_COST           | The I/O cost of the operation as estimated by the optimizer's cost-based approach. For statements that use the rule-based approach, this column is null. The value of this column is proportional to the number of data blocks read by the operation. |
| TEMP_SPACE        | The temporary space, in bytes, used by the operation as estimated by the optimizer’s cost-based approach. For statements that use the rule-based approach, or for operations that don’t use any temporary space, this column is null. |
### OTHER_TAG Column of the PLAN_TABLE

Values of the OTHER_TAG Column of the PLAN_TABLE:

<table>
<thead>
<tr>
<th>OTHER_TAG Text</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank</td>
<td>Serial execution.</td>
</tr>
<tr>
<td>SERIAL_FROM_REMOTE (S -&gt; R)</td>
<td>Serial execution at a remote site.</td>
</tr>
<tr>
<td>SERIAL_TO_PARALLEL (S -&gt; P)</td>
<td>Serial execution; output of step is partitioned or broadcast to parallel execution servers.</td>
</tr>
<tr>
<td>PARALLEL_TO_PARALLEL (P -&gt; P)</td>
<td>Parallel execution; output of step is repartitioned to second set of parallel execution servers.</td>
</tr>
<tr>
<td>PARALLEL_TO_SERIAL (P -&gt; S)</td>
<td>Parallel execution; output of step is returned to serial &quot;query coordinator&quot; process.</td>
</tr>
<tr>
<td>PARALLEL_COMBINED_WITH_PARENT (PWP)</td>
<td>Parallel execution; output of step goes to next step in same parallel process. No interprocess communication to parent.</td>
</tr>
<tr>
<td>PARALLEL_COMBINED_WITH_CHILD (PWC)</td>
<td>Parallel execution; input of step comes from prior step in same parallel process. No interprocess communication from child.</td>
</tr>
</tbody>
</table>
DISTRIBUTION Column of the PLAN_TABLE
Values of DISTRIBUTION Column of the PLAN_TABLE:

<table>
<thead>
<tr>
<th>DISTRIBUTION Text</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTITION (ROWID)</td>
<td>Maps rows to query servers based on the partitioning of a table or index using the rowid of the row to UPDATE/DELETE.</td>
</tr>
<tr>
<td>PARTITION (KEY)</td>
<td>Maps rows to query servers based on the partitioning of a table or index using a set of columns. Used for partial partition-wise join, PARALLEL INSERT, CREATE TABLE AS SELECT of a partitioned table, and CREATE PARTITIONED GLOBAL INDEX.</td>
</tr>
<tr>
<td>HASH</td>
<td>Maps rows to query servers using a hash function on the join key. Used for PARALLEL JOIN or PARALLEL GROUP BY.</td>
</tr>
<tr>
<td>RANGE</td>
<td>Maps rows to query servers using ranges of the sort key. Used when the statement contains an ORDER BY clause.</td>
</tr>
<tr>
<td>ROUND-ROBIN</td>
<td>Randomly maps rows to query servers.</td>
</tr>
<tr>
<td>BROADCAST</td>
<td>Broadcasts the rows of the entire table to each query server. Used for a parallel join when one table is very small compared to the other.</td>
</tr>
<tr>
<td>QC (ORDER)</td>
<td>The query coordinator consumes the input in order, from the first to the last query server. Used when the statement contains an ORDER BY clause.</td>
</tr>
<tr>
<td>QC (RANDOM)</td>
<td>The query coordinator consumes the input randomly. Used when the statement does not have an ORDER BY clause.</td>
</tr>
</tbody>
</table>
**OPERATION and OPTION Values Produced by EXPLAIN PLAN**

This table lists each combination of **OPERATION** and **OPTION** produced by the EXPLAIN PLAN statement and its meaning within an execution plan:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND-EQUAL</td>
<td>CONVERSION</td>
<td>Operation accepting multiple sets of rows, returning the intersection of the sets, eliminating duplicates. Used for the single-column indexes access path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TO ROWIDS converts bitmap representations to actual rowids that can be used to access the table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FROM ROWIDS converts the rowids to a bitmap representation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• COUNT returns the number of rowids if the actual values are not needed.</td>
</tr>
<tr>
<td>INDEX</td>
<td></td>
<td>• SINGLE VALUE looks up the bitmap for a single key value in the index.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• RANGE SCAN retrieves bitmaps for a key value range.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FULL SCAN performs a full scan of a bitmap index if there is no start or stop key.</td>
</tr>
<tr>
<td>MERGE</td>
<td></td>
<td>Merges several bitmaps resulting from a range scan into one bitmap.</td>
</tr>
<tr>
<td>MINUS</td>
<td></td>
<td>Subtracts bits of one bitmap from another. Row source is used for negated predicates. Can be used only if there are nonnegated predicates yielding a bitmap from which the subtraction can take place.</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td>Computes the bitwise OR of two bitmaps.</td>
</tr>
<tr>
<td>CONNECT BY</td>
<td></td>
<td>Retrieves rows in hierarchical order for a query containing a CONNECT BY clause.</td>
</tr>
<tr>
<td>CONCATENATION</td>
<td></td>
<td>Operation accepting multiple sets of rows returning the union-all of the sets.</td>
</tr>
<tr>
<td>COUNT</td>
<td></td>
<td>Operation counting the number of rows selected from a table.</td>
</tr>
<tr>
<td>STOPKEY</td>
<td></td>
<td>Count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause.</td>
</tr>
</tbody>
</table>

*Oracle9i: SQL Tuning Workshop B-7*
<table>
<thead>
<tr>
<th>Operation</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMAIN INDEX</td>
<td></td>
<td>Retrieval of one or more rowids from a domain index. The options column contain information supplied by a user-defined domain index cost function, if any.</td>
</tr>
<tr>
<td>FILTER</td>
<td></td>
<td>Operation accepting a set of rows, eliminates some of them, and returns the rest.</td>
</tr>
<tr>
<td>FIRST ROW</td>
<td></td>
<td>Retrieval of only the first row selected by a query.</td>
</tr>
<tr>
<td>FOR UPDATE</td>
<td></td>
<td>Operation retrieving and locking the rows selected by a query containing a FOR UPDATE clause.</td>
</tr>
<tr>
<td>HASH JOIN (These are join operations.)</td>
<td></td>
<td>Operation joining two sets of rows and returning the result.</td>
</tr>
<tr>
<td>ANTI</td>
<td></td>
<td>Hash anti-join.</td>
</tr>
<tr>
<td>SEMI</td>
<td></td>
<td>Hash semi-join.</td>
</tr>
<tr>
<td>INDEX (These are access methods.)</td>
<td>UNIQUE SCAN</td>
<td>Retrieval of a single rowid from an index.</td>
</tr>
<tr>
<td>RANGE SCAN</td>
<td></td>
<td>Retrieval of one or more rowids from an index. Indexed values are scanned in ascending order.</td>
</tr>
<tr>
<td>RANGE SCAN DESCENDING</td>
<td></td>
<td>Retrieval of one or more rowids from an index. Indexed values are scanned in descending order.</td>
</tr>
<tr>
<td>INLIST ITERATOR</td>
<td></td>
<td>Iterates over the operation below it for each value in the IN-list predicate.</td>
</tr>
<tr>
<td>INTERSECTION</td>
<td></td>
<td>Operation accepting two sets of rows and returning the intersection of the sets, eliminating duplicates.</td>
</tr>
</tbody>
</table>
**OPERATION and OPTION Values Produced by EXPLAIN PLAN** (continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MERGE JOIN</td>
<td>OUTER</td>
<td>Merge join operation to perform an outer join statement.</td>
</tr>
<tr>
<td>(These are join operations.)</td>
<td>ANTI</td>
<td>Merge anti-join</td>
</tr>
<tr>
<td></td>
<td>SEMI</td>
<td>Merge semi-join</td>
</tr>
<tr>
<td></td>
<td>CONNECT BY</td>
<td>Retrieval of rows in hierarchical order for a query containing a CONNECT BY clause.</td>
</tr>
<tr>
<td></td>
<td>MINUS</td>
<td>Operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.</td>
</tr>
<tr>
<td></td>
<td>NESTED LOOPS</td>
<td>Operation accepting two sets of rows, an outer set and an inner set. Oracle compares each row of the outer set with each row of the inner set, returning rows that satisfy a condition.</td>
</tr>
<tr>
<td>(These are join operations.)</td>
<td>OUTER</td>
<td>Nested loops operation to perform an outer join statement.</td>
</tr>
<tr>
<td></td>
<td>PARTITION</td>
<td>Access one partition.</td>
</tr>
<tr>
<td></td>
<td>SINGLE</td>
<td>Access one partition.</td>
</tr>
<tr>
<td></td>
<td>ITERATOR</td>
<td>Access many partitions (a subset).</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>Access all partitions.</td>
</tr>
<tr>
<td></td>
<td>INLIST</td>
<td>Similar to iterator, but based on an IN-list predicate.</td>
</tr>
<tr>
<td></td>
<td>INVALID</td>
<td>Indicates that the partition set to be accessed is empty.</td>
</tr>
</tbody>
</table>

*INNER* operation accepting two sets of rows, an outer set and an inner set. Oracle compares each row of the outer set with each row of the inner set, returning rows that satisfy a condition.

*OUTER* operation accepting two sets of rows, each sorted by a specific value, combining each row from one set with the matching rows from the other, and returning the result.

*MERGE* joins are join operations that combine each row from one set with the matching rows from the other, and returning the result.

*MINUS* operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.

*NESTED LOOPS* (These are join operations.) Operation accepting two sets of rows, each sorted by a specific value, combining each row from one set with the matching rows from the other, and returning the result.

*PARTITION* operation accepting two sets of rows, each sorted by a specific value, combining each row from one set with the matching rows from the other, and returning the result.

*CONNECT BY* retrieval of rows in hierarchical order for a query containing a CONNECT BY clause.

*SEMI* operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.

*ANTS* operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.

*MERGE JOIN* operation accepting two sets of rows, each sorted by a specific value, combining each row from one set with the matching rows from the other, and returning the result.

*OUTER* operation accepting two sets of rows, each sorted by a specific value, combining each row from one set with the matching rows from the other, and returning the result.

*MERGE JOIN* operation accepting two sets of rows, each sorted by a specific value, combining each row from one set with the matching rows from the other, and returning the result.
## OPERATION and OPTION Values Produced by EXPLAIN PLAN (continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMOTE</td>
<td></td>
<td>Retrieval of data from a remote database.</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td></td>
<td>Operation involving accessing values of a sequence.</td>
</tr>
<tr>
<td>SORT</td>
<td>AGGREGATE</td>
<td>Retrieval of a single row that is the result of applying a group function to a group of selected rows.</td>
</tr>
<tr>
<td></td>
<td>UNIQUE</td>
<td>Operation sorting a set of rows to eliminate duplicates.</td>
</tr>
<tr>
<td></td>
<td>GROUP BY</td>
<td>Operation sorting a set of rows into groups for a query with a GROUP BY clause.</td>
</tr>
<tr>
<td></td>
<td>JOIN</td>
<td>Operation sorting a set of rows before a merge-join.</td>
</tr>
<tr>
<td></td>
<td>ORDER BY</td>
<td>Operation sorting a set of rows for a query with an ORDER BY clause.</td>
</tr>
<tr>
<td>TABLE ACCESS (These are access methods.)</td>
<td>FULL</td>
<td>Retrieval of all rows from a table.</td>
</tr>
<tr>
<td></td>
<td>SAMPLE</td>
<td>Retrieval of sampled rows from a table.</td>
</tr>
<tr>
<td></td>
<td>CLUSTER</td>
<td>Retrieval of rows from a table based on a value of an indexed cluster key.</td>
</tr>
<tr>
<td></td>
<td>HASH</td>
<td>Retrieval of rows from table based on hash cluster key value.</td>
</tr>
<tr>
<td></td>
<td>BY ROWID RANGE</td>
<td>Retrieval of rows from a table based on a rowid range.</td>
</tr>
<tr>
<td></td>
<td>SAMPLE BY ROWID RANGE</td>
<td>Retrieval of sampled rows from a table based on a rowid range.</td>
</tr>
<tr>
<td></td>
<td>BY USER ROWID</td>
<td>If the table rows are located using user-supplied rowids.</td>
</tr>
</tbody>
</table>
## OPERATION and OPTION Values Produced by EXPLAIN PLAN (continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BY INDEX ROWID</td>
<td>If the table is nonpartitioned and rows are located using indexes.</td>
</tr>
<tr>
<td></td>
<td>BY GLOBAL INDEX ROWID</td>
<td>If the table is partitioned and rows are located using only global indexes.</td>
</tr>
<tr>
<td></td>
<td>BY LOCAL INDEX ROWID</td>
<td>If the table is partitioned and rows are located using one or more local indexes and possibly some global indexes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Partition Boundaries:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The partition boundaries might have been computed by:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A previous PARTITION step, in which case the PARTITION_START and PARTITION_STOP column values replicate the values present in the PARTITION step, and the PARTITION_ID contains the ID of the PARTITION step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, INVALID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The TABLE_ACCESS or INDEX step itself, in which case the PARTITION_ID contains the ID of the step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, ROW_LOCATION (TABLE_ACCESS only), and INVALID.</td>
</tr>
<tr>
<td></td>
<td>UNION</td>
<td>Operation accepting two sets of rows and returns the union of the sets, eliminating duplicates.</td>
</tr>
<tr>
<td></td>
<td>VIEW</td>
<td>Operation performing a view’s query and then returning the resulting rows to another operation.</td>
</tr>
</tbody>
</table>
### SQL Trace Facility Initialization Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMED_STATISTICS</td>
<td>This enables and disables the collection of timed statistics, such as CPU and elapsed times, by the SQL trace facility, as well as the collection of various statistics in the dynamic performance tables. The default value of false disables timing. A value of true enables timing. Enabling timing causes extra timing calls for low-level operations. This is a dynamic parameter. It is also a session parameter.</td>
</tr>
<tr>
<td>MAX_DUMP_FILE_SIZE</td>
<td>When the SQL trace facility is enabled at the instance level, every call to the server produces a text line in a file in the operating system’s file format. The maximum size of these files (in operating system blocks) is limited by this initialization parameter. The default is 500. If you find that the trace output is truncated, then increase the value of this parameter before generating another trace file. This is a dynamic parameter. It is also a session parameter.</td>
</tr>
<tr>
<td>USER_DUMP_DEST</td>
<td>This must fully specify the destination for the trace file according to the conventions of the operating system. The default value is the default destination for system dumps on the operating system. This value can be modified with <code>ALTER SYSTEM SET USER_DUMP_DEST= newdir</code>. This is a dynamic parameter. It is also a session parameter.</td>
</tr>
</tbody>
</table>
**TKPROF Command Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename1</td>
<td></td>
<td>Specifies the input file, a trace file containing statistics produced by the SQL trace facility. This file can be either a trace file produced for a single session, or a file produced by concatenating individual trace files from multiple sessions.</td>
</tr>
<tr>
<td>filename2</td>
<td></td>
<td>Specifies the file to which TKPROF writes its formatted output.</td>
</tr>
<tr>
<td>SORT</td>
<td></td>
<td>Sorts traced SQL statements in descending order of specified sort option before listing them into the output file. If more than one option is specified, then the output is sorted in descending order by the sum of the values specified in the sort options. If you omit this parameter, then TKPROF lists statements into the output file in order of first use. Sort options are listed as follows:</td>
</tr>
<tr>
<td>PRSCNT</td>
<td></td>
<td>Number of times parsed.</td>
</tr>
<tr>
<td>PRSCPU</td>
<td></td>
<td>CPU time spent parsing.</td>
</tr>
<tr>
<td>PRSELA</td>
<td></td>
<td>Elapsed time spent parsing.</td>
</tr>
<tr>
<td>PRSDSK</td>
<td></td>
<td>Number of physical reads from disk during parse.</td>
</tr>
<tr>
<td>PRSQRY</td>
<td></td>
<td>Number of consistent mode block reads during parse.</td>
</tr>
<tr>
<td>PRSCU</td>
<td></td>
<td>Number of current mode block reads during parse.</td>
</tr>
<tr>
<td>PRSMIS</td>
<td></td>
<td>Number of library cache misses during parse.</td>
</tr>
<tr>
<td>EXECNT</td>
<td></td>
<td>Number of executes.</td>
</tr>
<tr>
<td>EXECP</td>
<td></td>
<td>CPU time spent executing.</td>
</tr>
<tr>
<td>EXEEELA</td>
<td></td>
<td>Elapsed time spent executing.</td>
</tr>
<tr>
<td>EXEDSK</td>
<td></td>
<td>Number of physical reads from disk during execute.</td>
</tr>
<tr>
<td>EXEQRY</td>
<td></td>
<td>Number of consistent mode block reads during execute.</td>
</tr>
<tr>
<td>EXECU</td>
<td></td>
<td>Number of current mode block reads during execute.</td>
</tr>
</tbody>
</table>
### TKPROF Command Arguments (continued)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEROW</td>
<td></td>
<td>Number of rows processed during execute.</td>
</tr>
<tr>
<td>EXEMIS</td>
<td></td>
<td>Number of library cache misses during execute.</td>
</tr>
<tr>
<td>FCHCNT</td>
<td></td>
<td>Number of fetches.</td>
</tr>
<tr>
<td>FCHCPU</td>
<td></td>
<td>CPU time spent fetching.</td>
</tr>
<tr>
<td>FCHELA</td>
<td></td>
<td>Elapsed time spent fetching.</td>
</tr>
<tr>
<td>FCHDSK</td>
<td></td>
<td>Number of physical reads from disk during fetch.</td>
</tr>
<tr>
<td>FCHQRY</td>
<td></td>
<td>Number of consistent mode block reads during fetch.</td>
</tr>
<tr>
<td>FCHCU</td>
<td></td>
<td>Number of current mode block reads during fetch.</td>
</tr>
<tr>
<td>FCHROW</td>
<td></td>
<td>Number of rows fetched.</td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td>Lists only the first integer sorted SQL statements from the output file. If you omit this parameter, then TKPROF lists all traced SQL statements. This parameter does not affect the optional SQL script. The SQL script always generates insert data for all traced SQL statements.</td>
</tr>
<tr>
<td>AGGREGATE</td>
<td></td>
<td>If you specify AGGREGATE = NO, then TKPROF does not aggregate multiple users of the same SQL text.</td>
</tr>
<tr>
<td>INSERT</td>
<td></td>
<td>Creates a SQL script that stores the trace file statistics in the database. TKPROF creates this script with the name filename3. This script creates a table and inserts a row of statistics for each traced SQL statement into the table.</td>
</tr>
<tr>
<td>SYS</td>
<td></td>
<td>Enables and disables the listing of SQL statements issued by the user SYS, or recursive SQL statements, into the output file. The default value of YES causes TKPROF to list these statements. The value of NO causes TKPROF to omit them. This parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements, including recursive SQL statements.</td>
</tr>
</tbody>
</table>
**TKPROF Command Arguments (continued)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE</td>
<td></td>
<td>Specifies the schema and name of the table into which TKPROF temporarily places execution plans before writing them to the output file. If the specified table already exists, then TKPROF deletes all rows in the table, uses it for the EXPLAIN PLAN statement (which writes more rows into the table), and then deletes those rows. If this table does not exist, then TKPROF creates it, uses it, and then drops it. The specified user must be able to issue INSERT, SELECT, and DELETE statements against the table. If the table does not already exist, then the user must also be able to issue CREATE TABLE and DROP TABLE statements. For the privileges to issue these statements, see the Oracle9i SQL Reference. This option allows multiple individuals to run TKPROF concurrently with the same user in the EXPLAIN value. These individuals can specify different TABLE values and avoid destructively interfering with each other’s processing on the temporary plan table. If you use the EXPLAIN parameter without the TABLE parameter, then TKPROF uses the table PROF$PLAN_TABLE in the schema of the user specified by the EXPLAIN parameter. If you use the TABLE parameter without the EXPLAIN parameter, then TKPROF ignores the TABLE parameter.</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td></td>
<td>Determines the execution plan for each SQL statement in the trace file and writes these execution plans to the output file. TKPROF determines execution plans by issuing the EXPLAIN PLAN statement after connecting to Oracle with the user and password specified in this parameter. The specified user must have CREATE SESSION system privileges. TKPROF takes longer to process a large trace file if the EXPLAIN option is used.</td>
</tr>
<tr>
<td>RECORD</td>
<td></td>
<td>Creates a SQL script with the specified filename with all of the nonrecursive SQL in the trace file. This can be used to replay the user events from the trace file.</td>
</tr>
</tbody>
</table>
Interpreting TKPROF Output

Tabular Statistics

TKPROF lists the statistics for a SQL statement returned by the SQL trace facility in rows and columns. Each row corresponds to one of three steps of SQL statement processing. The step for which each row contains statistics is identified by the value of the CALL column.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARSE</td>
<td>This step translates the SQL statement into an execution plan. This step includes checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>This step is the actual execution of the statement. For INSERT, UPDATE, and DELETE statements, this step modifies the data. For SELECT statements, the step identifies the selected rows.</td>
</tr>
<tr>
<td>FETCH</td>
<td>This step retrieves rows returned by a query. Fetches are performed only for SELECT statements.</td>
</tr>
</tbody>
</table>

The other columns of the SQL trace facility output are combined statistics for all parses, all executes, and all fetches of a statement. The sum of query and current is the total number of buffers accessed.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>Number of times a statement was parsed, executed, or fetched.</td>
</tr>
<tr>
<td>CPU</td>
<td>Total CPU time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if TIMED_STATISTICS is not turned on.</td>
</tr>
<tr>
<td>ELAPSED</td>
<td>Total elapsed time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if TIMED_STATISTICS is not turned on.</td>
</tr>
<tr>
<td>DISK</td>
<td>Total number of data blocks physically read from the datafiles 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. Usually, buffers are retrieved in consistent mode for queries.</td>
</tr>
<tr>
<td>CURRENT</td>
<td>Total number of buffers retrieved in current mode. Buffers are retrieved in current mode for statements such as INSERT, UPDATE, and DELETE.</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.)</td>
</tr>
</tbody>
</table>
Interpreting **TKPROF** Output (continued)

**Tabular Statistics (continued)**

**Note:** The row source counts are displayed when a cursor is closed. In SQL*Plus there is only one user cursor, thus each statement executed causes the previous cursor to be closed; for this reason the row source counts are displayed. PL/SQL has its own cursor handling and does not close child cursors when the parent cursor is closed. Exiting (or reconnecting) causes the counts to be displayed.

**Resolution of Statistics**

Timing statistics have a resolution of one hundredth of a second; therefore, any operation on a cursor that takes a hundredth of a second or less might not be timed accurately. Keep this in mind when interpreting statistics. In particular, be careful when interpreting the results from simple queries that execute very quickly.

**Recursive Calls**

Sometimes, in order to execute a SQL statement issued by a user, Oracle must issue additional statements. Such statements are called recursive calls or recursive SQL statements. For example, if you insert a row into a table that does not have enough space to hold that row, then Oracle 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, then **TKPROF** produces statistics for the recursive SQL statements and marks them clearly as recursive SQL statements in the output file. You can suppress the listing of Oracle internal recursive calls (for example, space management) in the output file by setting the SYS command-line parameter to **NO**. The statistics for a recursive SQL statement are included in the listing for that statement, not in the listing for the SQL statement that caused the recursive call. So, when you are calculating the total resources required to process a SQL statement, consider the statistics for that statement as well as those for recursive calls caused by that statement.

**Library Cache Misses in TKPROF**

**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. If the statement resulted in no library cache misses, then **TKPROF** does not list the statistic. In "Sample **TKPROF** Output", the statement resulted in one library cache miss for the parse step and no misses for the execute step.

**Statement Truncation in SQL Trace**

The following SQL statements are truncated to 25 characters in the SQL trace file:

- **SET ROLE**
- **GRANT**
- **ALTER USER**
- **ALTER ROLE**
- **CREATE USER**
- **CREATE ROLE**
Interpreting **TKPROF** Output (continued)

**User Issuing the SQL Statement in TKPROF**

TKPROF also lists the user ID of the user issuing each SQL statement. If the SQL trace input file contained statistics from multiple users and the statement was issued by more than one user, then TKPROF lists the ID of the last user to parse the statement. The user ID of all database users appears in the data dictionary in the column `ALL_USERS.USER_ID`.

**Execution Plan in TKPROF**

If you specify the `EXPLAIN` parameter on the TKPROF statement line, then TKPROF uses the `EXPLAIN PLAN` statement 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:** 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 I/O activity as caches in the system global area (SGA) are filled. For the purposes of tuning, ignore such trace files.

**Storing SQL Trace Facility Statistics**

You might want to keep a history of the statistics generated by the SQL trace facility for an application, and compare them over time. TKPROF can generate a SQL script that creates a table and inserts rows of statistics into it. This script contains:

- A `CREATE TABLE` statement that creates an output table named `TKPROF_TABLE`
- `INSERT` statements that add rows of statistics, one for each traced SQL statement, to the `TKPROF_TABLE`

After running TKPROF, you can run this script to store the statistics in the database.

**Generating the TKPROF Output SQL Script**

When you run TKPROF, use the `INSERT` parameter to specify the name of the generated SQL script. If you omit this parameter, then TKPROF does not generate a script.

**Editing the TKPROF Output SQL Script**

After TKPROF has created the SQL script, you might want to edit the script before running it. If you have already created an output table for previously collected statistics and you want to add new statistics to this table, then remove the `CREATE TABLE` statement from the script. The script then inserts the new rows into the existing table.

If you have created multiple output tables, perhaps to store statistics from different databases in different tables, then edit the `CREATE TABLE` and `INSERT` statements to change the name of the output table.
Interpreting **TKPROF** Output (continued)

Querying the Output Table

The following `CREATE TABLE` statement creates the `TKPROF_TABLE`:

```sql
CREATE TABLE TKPROF_TABLE
    (DATE_OF_INSERT    DATE,
    CURSOR_NUM        NUMBER,
    DEPTH             NUMBER,
    USER_ID           NUMBER,
    PARSE_CNT         NUMBER,
    PARSE_CPU         NUMBER,
    PARSE_ELAP        NUMBER,
    PARSE_DISK        NUMBER,
    PARSE_QUERY       NUMBER,
    PARSE_CURRENT     NUMBER,
    PARSE_MISS        NUMBER,
    EXE_COUNT         NUMBER,
    EXE_CPU           NUMBER,
    EXE_ELAP          NUMBER,
    EXE_DISK          NUMBER,
    EXE_QUERY         NUMBER,
    EXE_CURRENT       NUMBER,
    EXE_MISS          NUMBER,
    EXE_ROWS          NUMBER,
    FETCH_COUNT       NUMBER,
    FETCH_CPU         NUMBER,
    FETCH_ELAP        NUMBER,
    FETCH_DISK        NUMBER,
    FETCH_QUERY       NUMBER,
    FETCH_CURRENT     NUMBER,
    FETCH_ROWS        NUMBER,
    CLOCK_TICKS       NUMBER,
    SQL_STATEMENT     LONG);
```

Most output table columns correspond directly to the statistics that appear in the formatted output file. For example, the `PARSE_CNT` column value corresponds to the count statistic for the parse step in the output file.
Interpreting **TKPROF** Output (continued)

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_STATEMENT</td>
<td>This is the SQL statement for which the SQL trace facility collected the row of statistics. Because this column has datatype <strong>LONG</strong>, you cannot use it in expressions or <strong>WHERE</strong> clause conditions.</td>
</tr>
<tr>
<td>DATE_OF_INSERT</td>
<td>This is the date and time when the row was inserted into the table. This value is not exactly the same as the time the statistics were collected by the SQL trace facility.</td>
</tr>
<tr>
<td>DEPTH</td>
<td>This indicates the level of recursion at which the SQL statement was issued. For example, a value of 0 indicates that a user issued the statement. A value of 1 indicates that Oracle generated the statement as a recursive call to process a statement with a value of 0 (a statement issued by a user). A value of n indicates that Oracle generated the statement as a recursive call to process a statement with a value of n-1.</td>
</tr>
<tr>
<td>USER_ID</td>
<td>This identifies the user issuing the statement. This value also appears in the formatted output file.</td>
</tr>
<tr>
<td>CURSOR_NUM</td>
<td>Oracle uses this column value to keep track of the cursor to which each SQL statement was assigned.</td>
</tr>
</tbody>
</table>
Interpreting TKPROF Output (continued)

Interpreting EXPLAIN Execution Plans

Level and Position

Each indentation is defined as a level of subordination. Each step to be executed at a given level (that is, all steps that are indented the same) is defined as a position.

This is level 1
   This is level 2 position 1
      This is level 3 position 1 — The first operation that is processed
         This is level 3 position 2
   This is level 2 position 2
      This is level 3 position 1

Determining the Order in Which the Plan Will Be Executed

1. Find the first line that does not have anything subordinate to it. Scan down the list of operations to be executed, look for the last time that is indented from the previous line. This is your starting location. The object on that line is the driving table or the associated index or cluster. Process steps 2 – 4 until all lines have been processed.

2. Perform the operation on the current line.

3. If the next line that has not already been processed is at the same level as the current line, drop down to it and proceed to step 4. If not, then scan back up the list to find the previous line that is at the next previous level and return back to step 2. For example, while at level 3, position 2, if the next line is not position 3 then go back up to the previous level 2.

4. If there is nothing subordinate to the current line, return back to step 2, otherwise scan down the list looking for the first line that does not have anything subordinate to it before returning to step 2.

Using the above example, the order in which the lines will be processed is:

6  This is level 1
  3  This is level 2 position 1
  1  This is level 3 position 1
  2  This is level 3 position 2
  5  This is level 2 position 2
  4  This is level 3 position 1
Interpreting TKPROF Output (continued)

**Note:** Displaying the position number in front of the operation on each line of the execution plan helps in interpreting the plan, especially for long SQL statements:

```sql
2 , options
3 , object_name
4 from plan_table
5 connect by prior id = parent_id
6 start with id = 1
7 order by id;
```

Use the following expression in the `SELECT` clause to display both the level and position separated by a period:

```
lpad(' ',2*level)||level||'. '||position||' '||operation
```
Table Descriptions
Table Descriptions

Overall Description

The sample company portrayed by the Oracle9i Sample Schemas operates worldwide to fill orders for several different products. The company has several divisions:

- The Human Resources division tracks information on the company’s employees and facilities.
- The Order Entry division tracks product inventories and sales of the company’s products through various channels.
- The Product Media division maintains descriptions and detailed information on each product sold by the company.
- The Shipping division manages the shipping of products to customer.
- The Sales History division tracks business statistics to facilitate business decisions.

Each of these divisions is represented by a schema. This course concentrates on using the Sales History schema.
Table Descriptions

Sales History (SH)

The sample company does a high volume of business, so it runs business statistics reports to aid in decision support. Many of these reports are time-based and non-volatile. That is, they analyze past data trends. The company loads data into its data warehouse regularly to gather statistics for these reports. Some examples of these reports include annual, quarterly, monthly, and weekly sales figures by product and annual, quarterly, monthly, and weekly sales figures by product.

The company also runs reports on distribution channels through which its sales are delivered. When the company runs special promotions on its products, it analyzes the impact of the promotions on sales. It also analyzes sales by geographical area.
Table Descriptions
Sales History (SH)

COUNTRIES
PK COUNTRY_ID
COUNTRY NAME
COUNTRY_SUBREGION
COUNTRY_REGION

CUSTOMERS
PK CUST_ID
CUST_FIRST_NAME
CUST_LAST_NAME
CUST_GENDER
CUST_YEAR_OF_BIRTH
CUST_MARRITAL_STATUS
CUST_STREET_ADDRESS
CUST_POSTAL_CODE
CUST_CITY
CUST_STATE_PROVINCE
COUNTRY_ID
CUST_MAIN_PHONE_NUMBER
CUST_INCOME_LEVEL
CUST_CREDIT_LIMIT
CUST_EMAIL

CHANNELS
PK CHANNEL_ID
CHANNEL_DESC
CHANNEL_CLASS

TIMES
PK TIME_ID
DAY_NAME
DAY_NUMBER_IN_WEEK
DAY_NUMBER_IN_MONTH
CALENDAR_WEEK_NUMBER
FISCAL_WEEK_NUMBER
WEEK_ENDING_DAY
CALENDAR_MONTH_NUMBER
FISCAL_MONTH_NUMBER
CALENDAR_MONTH_DESC
FISCAL_MONTH_DESC
DAYS_IN_CAL_MONTH
DAYS_IN_FIS_MONTH
END_OF_CAL_MONTH
END_OF_FIS_MONTH
CALENDAR_MONTH_NAME
FISCAL_MONTH_NAME
CALENDAR_QUARTER_DESC
FISCAL_QUARTER_DESC
DAYS_IN_CAL_QUARTER
DAYS_IN_FIS_QUARTER
END_OF_CAL_QUARTER
END_OF_FIS_QUARTER
CALENDAR_QUARTER_NUMBER
FISCAL_QUARTER_NUMBER
CALENDAR_YEAR
FISCAL_YEAR
DAYS_IN_CAL_YEAR
DAYS_IN_FIS_YEAR
END_OF_CAL_YEAR
END_OF_FIS_YEAR

PROMOTIONS
PK PROMO_ID
PROMO_NAME
PROMO_SUBCATEGORY
PROMO_CATEGORY
PROMO_COST
PROMO_BEGIN_DATE
PROMO_END_DATE

SALES
FK2,3 PROD_ID
FK1,2 CUST_ID
FK4,5 PROMO_ID
FK2,4 QUANTITY_SOLD
AMOUNT_SOLD

PRODUCTS
PK PROD_ID
PROD_NAME
PROD_DESC
PROD_SUBCATEGORY
PROD_SUBCATEGORY_DESC
PROD_CATEGORY
PROD_CATEGORY_DESC
PROD_WEIGHT_CLASS
PROD_UNIT_OF_MEASURE
PROD_PACK_SIZE
SUPPLIER_ID
PROD_STATUS
PROD_LIST_PRICE
PROD_MIN_PRICE

COSTS
FK1,1 PROD_ID
FK2,2 TIME_ID
UNIT_COST
UNIT_PRICE

CAL_MONTH_SALES_MV
CAL месяц SALES_MV

CALENDAR_MONTH_DESC
DOLLARS

Oracle9i: SQL Tuning Workshop C-4
Table Descriptions
Sales History (SH) Row Counts

SQL> select count(*) from cal_month_sales_mv;
    COUNT(*)
----------
     35

SQL> select count(*) from channels;
    COUNT(*)
----------
      5

SQL> select count(*) from costs;
    COUNT(*)
----------
  787765

SQL> select count(*) from countries;
    COUNT(*)
----------
     19

SQL> select count(*) from customers;
    COUNT(*)
----------
  50000

SQL> select count(*) from fweek_pscat_sales_mv;
    COUNT(*)
----------
  149325

SQL> select count(*) from products;
    COUNT(*)
----------
   10000

SQL> select count(*) from promotions;
    COUNT(*)
----------
     501
Table Descriptions
Sales History (SH) Row Counts

SQL> select count(*) from sales;
COUNT(*)
---------
1016271

SQL> select count(*) from times;
COUNT(*)
---------
1461
Workshop Scripts
General Scripts

This first section lists the general scripts used in the workshops; the workshop queries are listed in the second section of this appendix.

Note: Some scripts work correctly only when SQL*Plus is started with the right login.sql script. The relevant part of the login.sql script is also listed in this section.

REM   script ALLROWS.SQL
REM   =====================================
alter session set optimizer_mode = all_rows
/

REM   script ATOFF.SQL
REM   =====================================
set autotrace off

REM   script ATON.SQL
REM   =====================================
set autotrace on

REM   script ATONX.SQL
REM   =====================================
set autotrace on explain

REM   script ATTO.SQL
REM   =====================================
set autotrace traceonly

REM   script ATTOX.SQL
REM   =====================================
set autotrace traceonly explain

REM   script CHOOSE.SQL
REM   =====================================
alter session set optimizer_mode=choose;

REM   script FIRSTROWS.SQL
REM   =====================================
alter session set optimizer_mode=first_rows
/
General Scripts (continued)

REM    Oracle9i SQL Tuning Workshop
REM    script CBI.SQL (create bitmap index)
REM    prompts for input
REM    ==============================================================
accept TABLE_NAME  prompt "    on which table : ",&TABLE_NAME
accept COLUMN_NAME prompt "    on which column: ",&COLUMN_NAME
set    termout   off
store  set saved_settings replace
set    heading off feedback off verify off
set    autotrace off termout on

select 'Creating bitmap index on: '
, '&&TABLE_NAME'
, '&&COLUMN_NAME'
FROM DUAL
/
create bitmap index &INDEX_NAME on &TABLE_NAME(&COLUMN_NAME)
--LOCAL NOLOGGING COMPUTE STATISTICS
/

@saved_settings
set    termout on
undef  INDEX_NAME
undef  TABLE_NAME
undef  COLUMN_NAME

REM    script HASHFALSE.SQL
REM    ==============================================================
alter session set hash_join_enabled=false
/

REM    script HASHTURE.SQL
REM    ==============================================================
alter session set hash_join_enabled=true
/
General Scripts (continued)

REM    Oracle9i SQL Tuning Workshop
REM    script CI.SQL (create index)
REM    prompts for input
REM    ========================================================
accept TABLE_NAME  prompt "     on which table    : "
accept COLUMN_NAME prompt "     on which column(s): "
set    termout off
store  set saved_settings replace
set    heading off feedback off autotrace off
set    verify  off termout  on

select 'Creating index on: ' , '&&TABLE_NAME' , '&&COLUMN_NAME'
FROM DUAL 
/
create index &INDEX_NAME on &TABLE_NAME(&COLUMN_NAME)
/

@saved_settings
set    termout on
undef   INDEX_NAME
undef   TABLE_NAME
undef   COLUMN_NAME
General Scripts (continued)

REM Oracle9i SQL Tuning Workshop
REM script CUI.SQL (create unique index)
REM prompts for input
REM
accept TABLE_NAME  prompt " on which table : "
accept COLUMN_NAME prompt " on which column(s): "
set termout off
store set saved_settings replace
set heading off feedback off verify off
set autotrace off termout on

select 'creating unique index'
,      substr('ui_&TABLE_NAME._' ||
        translate(replace('&COLUMN_NAME', ' ', '')
        ,',','_')
        ,1, 30) dummy
,      '...
from    dual
/
create unique index &INDEX_NAME on &TABLE_NAME(&COLUMN_NAME)
/

@saved_settings
set termout on
undef INDEX_NAME
undef TABLE_NAME
undef COLUMN_NAME
General Scripts (continued)

REM script LOGIN.SQL for Workshops
REM ===============================================================
col dummy           new_value INDEX_NAME
col name            format a20
col value           format a20
col segment_name    format a20
col table_name      format a20
col column_name     format a15
col index_name      format a20
col index_type      format a10
col constraint_name format a15
col plan_plus_exp   format a90
col num_distinct    format 999999
set numwidth 6
set linesize 110
set pause "[Enter]..." pause off
set echo      off
set feedback  off
set verify    off
set tab       off
set feedback  on

REM script RP.SQL (read plan_table)
REM displays plan_table in nested format,
REM and deletes all plan_table rows
REM ===============================================================
select  lpad(' ',2*(level-1))
        ||operation  ||' '
        ||options    ||' '
        ||object_name||' '
        ||decode(id,0,'Cost = '||position) as "Query Plan"
from    plan_table
start   with id = 0
connect by prior id = parent_id
/
delete from plan_table
/

Oracle9i: SQL Tuning Workshop D - 6
General Scripts (continued)

REM    script DAI.SQL (drop all indexes)
REM    prompts for a table name; % is appended
REM    =======================================
accept TABLE_NAME  prompt "     on which table: "
set    termout off
store  set saved_settings replace
set    heading off verify off autotrace off feedback off
spool doit.sql
select 'DROP INDEX '||ui.index_name||';'
from   user_indexes ui
where  table_name like upper('&TABLE_NAME.%')
/
spool  off
set    termout on
@doit
@saved_settings
undef TABLE_NAME
set    termout on

REM    script LI.SQL (list indexes)
REM    usage: @li [table_name]
REM    wildcards in table_name allowed,
REM    and a '% is appended by default
REM    =====================================
set      termout  off
store    set saved_settings replace
set      verify   off autotrace off
set      feedback off termout on
break    on table_name skip 1 on index_type
prompt
prompt   indexes on table &&1.%:
select   ui.table_name,
        decode(ui.index_type
                ,'NORMAL', ui.uniqueness
                ,ui.index_type) as index_type,
        ui.index_name
from     user_indexes  ui
where    ui.table_name like upper('&1.%')
order by ui.table_name
        ,ui.uniqueness desc
/
@saved_settings
set termout on
Workshop 1

```sql
select cust_first_name, cust_last_name from customers where cust_id = 1030 /
```

```sql
select cust_first_name, cust_last_name from customers where cust_id <> 1030 /
```

```sql
select cust_first_name, cust_last_name from customers where cust_id < 20000 /
```

```sql
select cust_first_name, cust_last_name from customers where cust_id between 70000 and 80000 /
```

```sql
select cust_id from customers where cust_credit_limit*1.10 = 11000 /
```

```sql
select cust_id from customers where cust_credit_limit = 30000/2 /
```

```sql
select cust_id from customers where substr(cust_last_name,1,1) = 'S' /
```

```sql
select cust_id from customers where cust_last_name like 'S%' /
```

```sql
select cust_last_name from customers where cust_last_name like 'S%' /
```
Workshop 1 (continued)

```sql
select cust_id                                   -- ws01_09.sql
from   customers
where  cust_last_name like '%S%'
/

select cust_last_name                            -- ws01_10.sql
from   customers
where  cust_id like '7%'
/

update customers                                -- ws01_11a.sql
set cust_email = null
where cust_main_phone_number like '_7%'
or cust_main_phone_number like '2%'
or cust_main_phone_number like '__3%'
or cust_main_phone_number like '___6%'
/

select cust_email                               -- ws01_11b.sql
from   customers
where  cust_email is null
/

select cust_id                                   -- ws01_12.sql
from   customers
where  cust_email is NOT null
/

select cust_id                                   -- ws01_13.sql
from   customers
where  cust_email > 'a'
/

select --+ INDEX(c)                              -- ws01_14.sql
  c.cust_id
from   customers c
where  c.cust_email is NOT null
/

select cust_last_name                            -- ws01_15.sql
from   customers
where  cust_credit_limit = 7000
/
```
Workshop 1 (continued)

```sql
select cust_last_name
from customers
where cust_credit_limit != 50000
/
select cust_last_name
from customers
where NOT (cust_credit_limit > 50000)
/

REM Oracle9i SQL Tuning Workshop
REM script: ws01_stats.sql for Workshop 1
REM =====================================
set echo on
set autotrace off
analyze table customers delete statistics
/
analyze table products delete statistics
/
@hashfalse
set timing off
alter session set sql_trace = false
/
alter session set timed_statistics = false
/
set echo off
```
Workshop 2

```sql
select cust_first_name , cust_last_name , cust_credit_limit
from customers
order by cust_credit_limit
/

select cust_first_name , cust_last_name , cust_credit_limit
from customers
order by cust_id
/

select cust_first_name , cust_last_name , cust_credit_limit
from customers
where cust_city = 'Paris'
order by cust_id
/

select max(cust_credit_limit)
from customers
/

select max(cust_credit_limit+1000)
from customers
/

select max(cust_credit_limit*2)
from customers
/

select max(cust_credit_limit)
from customers
where cust_city = 'Paris'
/

select distinct cust_city
from customers
/

select country_id from countries
intersect
select country_id from customers
/```

Oracle9i: SQL Tuning Workshop D - 11
Workshop 2 (continued)

```sql
select country_id from countries                 -- ws02_10.sql
minus
select country_id from customers
/

select country_id from countries                 -- ws02_11.sql
union
select country_id from customers
/

select country_id from countries                 -- ws02_12.sql
union all
select country_id from customers
/

select cust_city                                 -- ws02_13.sql
,      avg(cust_credit_limit)
from   customers
group  by cust_city
/

select cust_city                                 -- ws02_14.sql
,      avg(cust_credit_limit)
from   customers
where  cust_city = 'Paris'
group  by cust_city
/

select cust_city                                 -- ws02_15.sql
,      avg(cust_credit_limit)
from   customers
group  by cust_city
having cust_city = 'Paris'
/
```
REM Oracle9i SQL Tuning Workshop
REM script: ws02_stats.sql for Workshop 2
REM =====================================
set echo on
set autotrace off
analyze table customers delete statistics;
analyze table countries delete statistics;
@hashfalse
set timing off
alter session set sql_trace = false;
alter session set timed_statistics = false;
set echo off
Workshop 3

```sql
select name, value                                -- ws03_01.sql
from   v$parameter
where  name in ('hash_join_enabled'
                , 'optimizer_mode'
               )
/

select c.cust_last_name, c.cust_year_of_birth    -- ws03_02.sql
    , co.country_name
from   customers c
    , countries co
where  c.country_id = co.country_id
/

select /*+ &hint */
    c.cust_last_name                          -- ws03_03.sql
    , c.cust_year_of_birth
    , co.country_name
from   customers c
    , countries co
where  c.country_id = co.country_id
/

select c.cust_last_name
    , c.cust_year_of_birth                      -- ws03_04.sql
    , co.country_name
from   customers c
    , countries co
where  c.country_id = co.country_id
    and co.country_region = 'Americas'
/

select /*+ &hint */
    c.cust_last_name                          -- ws03_05.sql
    , c.cust_year_of_birth
    , co.country_name
from   customers c
    , countries co
where  c.country_id = co.country_id
    and co.country_region = 'Americas'
/```
Workshop 3 (continued)

```
select c.cust_last_name                          -- ws03_06.sql
  , s.time_id
  , s.prod_id
from customers c, sales s
where c.cust_id <> s.cust_id
and    s.prod_id = 2595
and    s.time_id = '01-JAN-98'
/

select c.cust_last_name                          -- ws03_07.sql
  , s.time_id
  , s.prod_id
from customers c, sales s
where c.cust_id = s.cust_id (+)
/

select c.cust_last_name                          -- ws03_08.sql
  , s.time_id
  , s.prod_id
from sales s, customers c
where c.cust_id = s.cust_id (+)
and    s.prod_id = 7145
/

select c.cust_last_name                          -- ws03_09.sql
  , s.time_id
  , s.prod_id
from sales s, customers c
where c.cust_id = s.cust_id (+)
and    s.prod_id (+) = 7145
/

SELECT *                                         -- ws03_10.sql
FROM (SELECT prod_id
      , prod_name
      , prod_desc
      , prod_list_price
      , prod_min_price
      FROM products
      ORDER BY prod_min_price ASC)
WHERE ROWNUM <= 10
/
```
Workshop 3 (continued)

```sql
select p.prod_desc, s.amount_sold
from products p, channels c, sales s
where s.promo_id = 10
and s.channel_id = 'I'
and s.prod_id = p.prod_id
and s.channel_id = c.channel_id
/
```

REM Oracle9i SQL Tuning Workshop
REM script: ws03_stats.sql for Workshop 3
REM =-----------------------------------------
set echo on
set autotrace off
analyze table customers delete statistics /
analyze table countries delete statistics /
@hashfalse
alter session set sql_trace = false /
alter session set timed_statistics = false /
@allrows
set verify off
set echo off
WORKSHOP 4

select name, value                              -- ws04_01.sql
from v$parameter
where lower(name) in ('hash_join_enabled'
                      ,'optimizer_mode'
                      ,'always_anti_join'
                      ,'always_semi_join')
/

select c1.cust_first_name                      -- ws04_02.sql
          , c1.cust_last_name
          , c1.cust_year_of_birth
from customers c1
where c1.cust_year_of_birth =
     (select max(c2.cust_year_of_birth)
      from customers c2
      where c1.country_id = c2.country_id)
/

select c1.cust_first_name                      -- ws04_02.sql
          , c1.cust_last_name
          , c1.cust_year_of_birth
from customers c1
          , (select country_id
              , max(cust_year_of_birth) max_old
              from customers
              group by country_id) c2
where c1.cust_year_of_birth = c2.max_old
and c1.country_id = c2.country_id
/

select cust_id, cust_last_name                  -- ws04_03.sql
from customers c1
where c1.cust_credit_limit >
     (select avg(cust_credit_limit)
      from customers c2
      where c1.country_id = c2.country_id
      group by c2.country_id)
/

select cust_id, cust_last_name                  -- ws04_04.sql
from customers c1
          , (select country_id, avg(cust_credit_limit) avg_credit
              from customers
              group by country_id) avgtab
where c1.cust_credit_limit > avgtab.avg_credit
and c1.country_id = avgtab.country_id
/
select c.cust_last_name                          -- ws04_05.sql
from   customers c
where  c.country_id = 'US'
and    c.cust_id NOT IN (select s.cust_id
                        from sales s)
/

select c.cust_last_name                          -- ws04_06.sql
from   customers c
where  c.country_id = 'US'
and    c.cust_id NOT IN (/*+ &hint */
                        select s.cust_id
                        from sales s)
/

select count(*)                                  -- ws04_07.sql
from   products p
where  exists (select 'x'
                from sales s
                where p.prod_id = s.prod_id)
/

select count(*)                                  -- ws04_08.sql
from   products p
where  exists (select /*+ &hint */ 'x'
                from sales s
                where p.prod_id = s.prod_id)
/

select count(*)                                  -- ws04_09.sql
from   sales s
where  exists (select 'x'
                from customers c
                where s.cust_id = c.cust_id)
/
Workshop 4 (continued)

REM Oracle9i SQL Tuning Workshop
REM script: ws04_stats.sql for Workshop 4
REM =====================================
set echo on
set autotrace off
@hashtrue
set timing off
alter session set sql_trace = false
/
alter session set timed_statistics = false
/
alter session set always_anti_join = nested_loops
/
alter session set always_semi_join = nested_loops
/
analyze table customers compute statistics
/
analyze table products compute statistics
/
@allrows
set echo off
Workshop 5

```sql
select c.* -- ws05_01.sql
from customers c
where cust_gender = 'M'
and cust_postal_code = 40804
and cust_credit_limit = 10000
/

select /*+ INDEX (r I_CUSTOMERS_CUST_CREDIT_LIMIT) */ c.* -- ws05_02.sql
from customers c
where cust_gender = 'M'
and cust_postal_code = 40804
and cust_credit_limit = 10000
/

select /*+ INDEX (r I_CUSTOMERS_CUST_CREDIT_LIMIT) */ c.* -- ws05_02a.sql
from customers c
where cust_gender = 'M'
and cust_postal_code = 40804
and cust_credit_limit = 10000
/

select /*+ INDEX (r I_CUSTOMERS_CUST_GENDER) */ c.* -- ws05_02b.sql
from customers c
where cust_gender = 'M'
and cust_postal_code = 40804
and cust_credit_limit = 10000
/

select /*+ INDEX (r I_CUSTOMERS_CUST_POSTAL_CODE) */ c.* -- ws05_02c.sql
from customers c
where cust_gender = 'M'
and cust_postal_code = 40804
and cust_credit_limit = 10000
/

select /*+ AND_EQUAL (r &index_name1, &index_name2) */ c.* -- ws05_03.sql
from customers c
where cust_gender = 'M'
and cust_postal_code = 40804
and cust_credit_limit = 10000
/
```
Workshop 5 (continued)

```sql
select s.*                                      -- ws05_04.sql
from   sales s
where  s.prod_id   = 23975
and    s.time_id  = '25-Nov-00'
/

select
c.*                                             -- ws05_04a.sql
from   customers c
where  cust_gender   = 'M'
and    cust_credit_limit = 10000
/

select
c.*                                             -- ws05_04b.sql
from   customers c
where  cust_gender   = 'M'
and    cust_postal_code = 40804
/

select
c.*                                             -- ws05_04c.sql
from   customers c
where   cust_postal_code = 40804
and    cust_credit_limit = 10000
/

select c.*                                       -- ws05_05.sql
from   customers c
where  c.cust_year_of_birth   = 1953
or    c.cust_credit_limit  = 10000
/

alter table customers                            -- ws05_06.sql
add constraint pk_cust_id primary key( cust_id)
/

select s.*                                       -- ws05_07.sql
from   sales s
where  s.prod_id   = 23975
or     s.time_id  = '25-Nov-00'
/

select c.*                                       -- ws05_08.sql
from   customers c
where  cust_id in (88340,104590,44910)
/```
Workshop 5 (continued)

```sql
select /*+ INDEX_COMBINE (r) */
  c.*                                     -- ws05_09.sql
from     customers c
where  (c.cust_year_of_birth = '1970'
  and c.cust_postal_code  = 40804 )
  and not  cust_credit_limit = 15000
/

select c.*                                     -- ws05_10.sql
from   customers c
where  c.country_id in ('US', 'TR')
/

select country_id, count(*)                    -- ws05_10a.sql
from    customers
group by  country_id
/

select c.cust_last_name                          -- ws05_11.sql
    ,      c.cust_first_name
from   customers c
/

select count(*) credit_limit
from   customers                                 -- ws05_13.sql
where  cust_credit_limit = 10000
/

select cust_id, country_id
from customers                                   -- ws05_14.sql
where lower( cust_last_name) like 'gentle'
/

alter table customers                            -- ws05_15.sql
add constraint customers_pk primary key (cust_id)
/

REM Oracle9i SQL Tuning Workshop
REM script: ws05_stats.sql for Workshop 5
REM ==============================================================
set autotrace off
set echo on
@hashfalse
set timing off
alter session set sql_trace = false;
alter session set timed_statistics = false;
set echo off
```

Oracle9i: SQL Tuning Workshop D - 22
create or replace view v_sales_detail            -- ws06_01.sql
as
    select sa.prod_id
    ,   pr.prod_name
    ,   pr.supplier_id
    ,   pr.prod_status
    ,   sa.amount_sold
    ,   to_char(sa.time_id, 'YYYY-MON-DD') date_sold
from   sales sa
    ,   products pr
where  pr.prod_id(+) = sa.prod_id
/

select v.prod_name                               -- ws06_02.sql
    ,   v.prod_status
    ,   v.date_sold
from   v_sales_detail v
where  v.prod_id = 15180
/

select pr.prod_name                              -- ws06_03.sql
    ,   pr.prod_status
    ,   to_char(sa.time_id, 'YYYY-MON-DD') date_sold
from   sales sa
    ,   products pr
where  pr.prod_id = sa.prod_id (+)
and sa.prod_id = 15180
/

select pr.prod_name                              -- ws06_04.sql
    ,   pr.prod_status
    ,   to_char(sa.time_id, 'YYYY-MON-DD') date_sold
from   sales sa
    ,   products pr
where  pr.prod_id = sa.prod_id (+)
and sa.prod_id = 15180
/
Workshop 6 (continued)

```sql
select pr.prod_name, pr.prod_status, to_char(sa.time_id, 'YYYY-MON-DD') date_sold
from sales sa, products pr
where pr.prod_id = sa.prod_id (+)
and sa.prod_id in (4680, 15180)
/

select v.prod_name, v.prod_status, v.date_sold
from v_sales_detail v
where v.prod_name = 'Sunburst Dress'
/

create or replace view v_custyob_1962 as select cust_id, cust_last_name, cust_income_level
from customers
where cust_year_of_birth = 1962
/

select cust_id
from v_custyob_1962
where cust_id > 300000
/

create or replace view v_avg_credit_limit as select country_id, avg(cust_credit_limit) AVG_CREDIT
from customers
group by country_id
/

select *
from v_avg_credit_limit
where country_id = 'US'
/
```

Oracle9i: SQL Tuning Workshop D - 24
create or replace view v_99_00_times             -- ws06_11.sql
as
select *
from times
where calendar_year = '1999'
union
select *
from times
where calendar_year = '2000'
/

select calendar_month_number                     -- ws06_12.sql
,      calendar_month_name
,      calendar_quarter_desc
,      calendar_quarter_number
from   v_99_00_times
where  calendar_year = '1999'
/

CREATE MATERIALIZED VIEW                         -- ws06_13.sql
fquarter_pscat_costs_mv
ENABLE QUERY REWRITE
AS
SELECT   t.fiscal_quarter_number
,        t.fiscal_quarter_desc
,        t.fiscal_year
,        p.prod_subcategory
,        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
GROUP BY t.fiscal_quarter_number
,        t.fiscal_quarter_desc
,        t.fiscal_year
,        p.prod_subcategory
/

select *                                         -- ws06_14.sql
from fquarter_pscat_costs_mv
where fiscal_year = 1999
/
Workshop 6 (continued)

REM Oracle9i SQL Tuning Workshop
REM script: ws06_stats.sql for Workshop 6
REM =====================================
set echo on
set autotrace off
@hashfalse
set timing off
alter session set sql_trace = false
/
alter session set timed_statistics = false
/
alter session set optimizer_mode = CHOOSE
/
set echo off
Workshop 7

```sql
select name, value                        -- ws07_01.sql
from v$parameter
where name in ('hash_join_enabled'
               ,'optimizer_mode'
               ,'always_anti_join'
               ,'always_semi_join')
/

cREATE TABLE promotions_iot               -- ws07_02.sql
(promo_id number PRIMARY KEY
 , promo_name varchar2(20)
 , promo_subcategory varchar2(30)
 , promo_category varchar2(30)
 , promo_cost number
 , promo_begin_date date
 , promo_end_date date)
orGANIZATION INDEX
/

select *                                 -- ws07_03.sql
from promotions
where promo_id > 300
/

select *                                 -- ws07_04.sql
from promotions_iot
where promo_id > 300
/

select *                                 -- ws07_05.sql
from promotions_iot
where promo_subcategory = 'online discount'
/
```
Workshop 7

REM Oracle9i SQL Tuning Workshop
REM script: ws07_stats.sql for Workshop 7
REM =====================================
set echo on
set autotrace off
analyze table products compute statistics /
analyze table sales compute statistics /
@hashfalse
set timing off
alter session set sqltrace = false /
alter session set timed_statistics = false /
set echo off
col name format a30
col value format a30
Practice 2 Solutions

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.

\[
\text{SQL}> \text{show arraysize}
\]

\[
\text{arraysize 15}
\]

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

\[
\text{SQL}> \text{select channel_id, channel_desc}
\]
\[
2 \text{ from channels;}
\]

\[
\text{SQL}> \text{select sql_text, sorts}
\]
\[
2 \text{ from v$sqlarea}
\]
\[
3 \text{ where command_type = 3}
\]
\[
4 \text{ and lower(sql_text) like '%channels%'};
\]

\text{No formal solution.}

3. Modify the query on the \text{CHANNELS} table and add a sort criterion, then view the information generated by this statement in the \text{V$SQLAREA} \text{view}.

\[
\text{SQL}> \text{select channel_id, channel_desc}
\]
\[
2 \text{ from channels}
\]
\[
3 \text{ order by channel_id;}
\]

\[
\text{SQL}> \text{select sql_text, sorts}
\]
\[
2 \text{ from v$sqlarea}
\]
\[
3 \text{ where command_type = 3}
\]
\[
4 \text{ and lower(sql_text) like '%channels%'};
\]

\text{No formal solution.}

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

\[
\text{SQL}> \text{insert into channels}
\]
\[
2 \text{ values('W', 'Wholesale', 'Direct');}
\]

\[
\text{SQL}> \text{select sql_text, sorts}
\]
\[
2 \text{ from v$sqlarea}
\]
\[
3 \text{ where command_type in (2,3)}
\]
\[
4 \text{ and lower(sql_text) like '%channels%'};
\]

\text{SQL> rollback;}

\text{No formal solution.}

5. Use the \text{V$LIBRARYCACHE} \text{view} to check the amount of SQL caching.

\[
\text{SQL}> \text{select gethitratio, pinhitratio}
\]
\[
2 \text{ from v$librarycache}
\]
\[
3 \text{ where namespace = 'SQL AREA'};
\]
Practice 3 Solutions

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.

   ```sql
   SQL> desc plan_table
   ERROR:
   ORA-04043: object plan_table does not exist
   SQL> @utlxplan
   ```

   -- or, if the table exists, empty it using the TRUNCATE statement.

   ```sql
   SQL> TRUNCATE TABLE plan_table;
   Table truncated.
   ```

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

   ```sql
   SQL> alter session set optimizer_mode = RULE;
   SQL> select cust_first_name, cust_last_name
   2  from   customers
   3  where  cust_id = 100;
   SQL> explain plan for
   2  select cust_first_name, cust_last_name
   3  from   customers
   4  where  cust_id = 100;
   Explained.
   ```

3. Now query the `PLAN_TABLE` table:

   ```sql
   SQL> select * from plan_table;
   Use the `rp.sql` script to query the `PLAN_TABLE` table in a more sophisticated manner:
   ```sql
   SQL> @rp
   No formal solution.
   ```

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

   ```sql
   SQL> set autotrace traceonly explain
   SQL> select cust_first_name, cust_last_name
   2  from   customers
   3  where  cust_id = 100;
   ```

Oracle9i: SQL Tuning Workshop E - 3
Practice 3 Solutions (continued)

Execution Plan

0      SELECT STATEMENT Optimizer=RULE
1    0   TABLE ACCESS (BY INDEX ROWID) OF ‘CUSTOMERS’
2    1     INDEX (UNIQUE SCAN) OF ‘CUSTOMERS_PK’ (UNIQUE)

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

```
SQL> set autotrace traceonly
SQL> select cust_first_name, cust_last_name
2      from   customers
3      where  cust_id = 100;
```

Execution Plan

0      SELECT STATEMENT Optimizer=RULE
1    0   TABLE ACCESS (BY INDEX ROWID) OF ‘CUSTOMERS’
2    1     INDEX (UNIQUE SCAN) OF ‘CUSTOMERS_PK’ (UNIQUE)

Statistics

0  recursive calls
0  db block gets
3  consistent gets
0  physical reads
0  redo size
447  bytes sent via SQL*Net to client
425  bytes received via SQL*Net from client
2  SQL*Net roundtrips to/from client
0  sorts (memory)
0  sorts (disk)
1  rows processed

6. If you have some time left, enter your own SQL statements and experiment with EXPLAIN PLAN and AUTOTRACE.

No formal solution.
Practice 4 Solutions

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.

   SQL> column name format a30
   SQL> column value format a30
   SQL> select name, value
       2   from v$parameter
       3   where name in
           4       ('timed_statistics', 'max_dump_file_size',
           5           'user_dump_dest', 'sql_trace')
           6 /
   SQL> alter session set timed_statistics = true
       2 /
   SQL> alter session set sql_trace = true
       2 /

2. Execute the following SQL command:

   SQL> select prod_name, prod_desc, supplier_id
       2   from products
       3   where prod_id = 200;

<table>
<thead>
<tr>
<th>PROD_NAME</th>
<th>PROD_DESC</th>
<th>SUPPLIER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potpourri Skirt</td>
<td>this is the famous Potpourri Skirt in color</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>orange of size XXL</td>
<td></td>
</tr>
</tbody>
</table>

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.

   SQL> alter session set sql_trace = false
       2 /
   SQL> alter session set timed_statistics = false
       2 /
   SQL> edit <your_trace_file>.trc
Practice 4 Solutions (continued)

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

   OS> tkprof

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`

   OS> tkprof <your_trace_file>.trc run1.txt
   OS> edit run1.txt
   Or
   OS> vi run1.txt

... 

select prod_name, prod_desc, supplier_id
from products
where prod_id = 200

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fetch</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>4</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

   Misses in library cache during parse: 0
   Optimizer goal: CHOOSE
   Parsing user id: 44

   Rows   Row Source Operation
          -----------------------------------------------
          1 TABLE ACCESS BY INDEX ROWID PRODUCTS
          1 INDEX UNIQUE SCAN (object id 32166)

...

6. Finally, start TKPROF in such a way that execution plans are added to the report and recursive SQL statements are suppressed.

   OS> tkprof <your_trace_file>.trc run1.txt
   explain=<username>/<password> sys=no
Practice 5 Solutions

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

   SQL> alter session set optimizer_mode = RULE;

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> set autotrace traceonly explain
   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';

   Execution Plan
   -----------------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (FULL) OF 'CUSTOMERS'
   3    1     TABLE ACCESS (BY INDEX ROWID) OF 'COUNTRIES'
   4    3       INDEX (UNIQUE SCAN) OF 'COUNTRY_PK' (UNIQUE)

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;

   Execution Plan
   -----------------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (FULL) OF 'CUSTOMERS'

Oracle9i: SQL Tuning Workshop  E - 7
Practice 5 Solutions (continued)

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.

   SQL> set autotrace off
   SQL> alter session set optimizer_mode = CHOOSE;

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

   SQL> set autotrace traceonly explain
   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';

---

Execution Plan

0      SELECT STATEMENT Optimizer=CHOOSE (Cost=362 Card=1079 Bytes=75530)
1 0  HASH JOIN (Cost=362 Card=1079 Bytes=75530)
2 1  TABLE ACCESS (FULL) OF 'COUNTRIES' (Cost=1 Card=1 Bytes=55)
3 1  TABLE ACCESS (FULL) OF 'CUSTOMERS' (Cost=360 Card=50000 Bytes=750000)

Although the optimizer goal is different, the execution plan is still the same. The rule-based optimizer is still used because there are no table statistics available yet.
Practice 7 Solutions

1. Turn SQL*Plus AUTOTRACE off.

Compute statistics for the PRODUCTS table and view the table and column statistics with appropriate select statements.

```sql
SQL> analyze table products
2   compute statistics;

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

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

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.

```sql
SQL> select count(*), prod_status
2   from products
3   group by prod_status;
```

```
COUNT(*) PROD_STATUS
-------- ---------------------
   800 available, no stock
  8000 available, on stock
    400 not available
   300 obsolete
   500 ordered
```

Because the data is highly skewed with “available, on stock” occurring much more often than the other values, the use of a histogram is appropriate.

3. Create a histogram for the PROD_STATUS column with 20 buckets and view the histogram statistics.
Practice 7 Solutions (continued)

```sql
SQL> analyze table products compute statistics for columns prod_status size 20;
Table analyzed.

SQL> select endpoint_number, endpoint_value
2  from user_histograms
3  where table_name = 'PRODUCTS'
4  and column_name = 'PROD_STATUS';

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
     800       5.0605E+35
    8800       5.0605E+35
    9200       5.7341E+35
    9500       5.7834E+35
   10000       5.7867E+35

Note: You only get five buckets, although you asked for 20. This is because the PROD_STATUS column only has five distinct values, so Oracle can store exact statistics in five rows. Note that the ENDPOINT_NUMBER column contains a cumulative total; the ENDPOINT_VALUE contains the PROD_STATUS values, stored in a floating point format which makes them unreadable.
```

4. Calculate the selectivity of the following select statement, both before and after creating the histogram:

```sql
SQL> select * from products where prod_status = 'obsolete';
```
- Before the histogram: selectivity = 1/5 = 20%
- After the histogram: selectivity = 300/10000 = 3%

5. Identify the last analyze date and sample size for all tables in your schema.

```sql
SQL> select table_name, sample_size, last_analyzed from user_tables;
```

6. Delete the statistics for the PRODUCTS table and check the data dictionary again.

```sql
SQL> analyze table products delete statistics;
SQL> select table_name, sample_size, last_analyzed from user_tables;
```
Practice 8 Solutions

1. Delete table statistics for the PROMOTIONS table.
   
   ```sql
   SQL> analyze table promotions delete statistics;
   ```

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.
   
   ```sql
   SQL> select *
   2   from   promotions;
   ```

3. Alter your session and set the optimizer mode to CHOOSE.
   
   ```sql
   SQL> alter session set optimizer_mode = CHOOSE;
   ```

4. Analyze the select statement shown.
   
   ```sql
   SQL> set autotrace traceonly explain
   SQL> select promo_id, promo_name, promo_cost
   2   from promotions
   3   where promo_id > 450;
   ```

5. What optimization method is used?
   
   Rule optimization is used because there are no table statistics. This is indicated in the explain plan. There is no cost associated with the statement.

6. Include a hint to use a full table scan with the select statement used in step 4.
   
   ```sql
   SQL> select /*+ FULL(p) */
   2   *
   3   from promotions p
   4   where promo_id > 450;
   ```

7. View the results of the EXPLAIN PLAN command. What optimization method is used? With the PRODUCTS table, which approach is more efficient?
   
   Because the table is small, a full table scan is preferable. If table statistics were available on the PRODUCTS table, the cost-based optimizer would automatically choose a full table scan for this select statement.
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 demonstration labs directory, then create a stored outline for the following SQL statement:

   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;

   SQL> select index_name
       2  from all_ind_columns
       3  where table_name = ‘PRODUCTS’
       4  and column_name = ‘SUPPLIER_ID’
    5 /

   SQL> create or replace outline team76505 on
       2  select p.prod_id, p.prod_name, p.prod_min_price
       3  from products p
       4  where p.supplier_id = 260
    5 /

2. Check the data dictionary view USER_OUTLINES 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.

   SQL> column name format a15
   SQL> column category format a15
   SQL> column sql_text format a30
   SQL> select name, category, used, timestamp, sql_text
       2  from user_outlines
       3  where name = ‘TEAM76505’
    4 /
   SQL> column name clear
   SQL> column category clear
   SQL> column sql_text clear
Practice 10 (continued)

<table>
<thead>
<tr>
<th>NAME</th>
<th>CATEGORY</th>
<th>USED</th>
<th>TIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEAM76505</td>
<td>DEFAULT</td>
<td>UNUSED</td>
<td>14-NOV-01</td>
</tr>
</tbody>
</table>

```sql
select p.prod_id, p.prod_name, p.prod_min_price
from   products p
where  p.supplier
```

3. Check the data dictionary view `USER_OUTLINE_HINTS` to see which hints are stored with your outline.

```sql
SQL> col hint format a10
SQL>
SQL> select *
2  from user_outline_hints
3  where name = 'TEAM76505'
4  /
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>NODE</th>
<th>STAGE</th>
<th>JOIN_POS</th>
<th>HINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEAM76505</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>NO_EXPAND</td>
</tr>
<tr>
<td>TEAM76505</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>ORDERED</td>
</tr>
<tr>
<td>TEAM76505</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>NO_FACT(P)</td>
</tr>
<tr>
<td>TEAM76505</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>FULL(P)</td>
</tr>
<tr>
<td>TEAM76505</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>NOREWRITE</td>
</tr>
<tr>
<td>TEAM76505</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NOREWRITE</td>
</tr>
</tbody>
</table>

6 rows selected.

4. Issue an `ALTER SESSION` command to enable your session using the stored outline.

```sql
SQL> alter session
2  set use_stored_outlines = TRUE
3  /
```
Practice 10 (continued)

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.

```
SQL> set autotrace traceonly explain
SQL> select p.prod_id, p.prod_name, p.prod_min_price
    2  from products p
    3  where p.supplier_id = 260
    4  /
```

Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=109 Card=35 Bytes=1295)
1 0   TABLE ACCESS (FULL) OF 'PRODUCTS' (Cost=109 Card=35 Bytes=1295)

```
SQL> create index prod_supplier_idx on
    2    products(supplier_id)
    3  /
Index created.
SQL> select p.prod_id, p.prod_name, p.prod_min_price
    2  from products p
    3  where p.supplier_id = 260
    4  /
```

Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=109 Card=35 Bytes=1295)
1 0   TABLE ACCESS (FULL) OF 'PRODUCTS' (Cost=109 Card=35 Bytes=1295)

```
SQL> set autotrace off
```

Oracle9i: SQL Tuning Workshop E - 14
6. Check the **USER_OUTLINES** data dictionary view again to see whether the outline has been used.

   ```sql
   SQL> select category, used, timestamp, sql_text
   2   from user_outlines
   3   where name = 'TEAM76505'
   4   /
   CATEGORY         USED                      TIMESTAMP
   ----------------- ------------------------- ---------
   SQL_TEXT
   DEFAULT           USED                      14-NOV-01
   select p.prod_id, p.prod_name, p.prod_min_price
   from products p
   where p.supplier
   ```

7. Drop the outline, and issue the SQL statement again to see that the index will now be used.

   ```sql
   SQL> drop outline team76505
   2   /
   SQL> set autotrace traceonly explain
   SQL>
   SQL> select p.prod_id, p.prod_name, p.prod_min_price
   2   from products p
   3   where p.supplier_id = 260
   4   /
   Execution Plan
   -------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=35 Bytes=1295)
   1      0   TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (Cost=4 Card=35
   Bytes=1295)
   2      1     INDEX (RANGE SCAN) OF 'PROD_SUPPLIER_IDX' (NON-UNIQUE) (Cost=1
   Card=35)
   SQL> set autotrace off
Index

A

Access method  8-16
Access path   1-20, 5-3
Ad hoc queries  2-16
Advantage of shared SQL areas  2-2
ALL_INDEXES   7-9
ALL_ROWS      5-9
ALL_TAB_COL_STATISTICS 7-11
ALL_TABLES   7-6
ALTER INDEX 6-10
ALTER SESSION 4-6
ALTER SESSION 8-2
ALTER SYSTEM 4-6
ALTER TABLE 2-6
ANALYZE 7-2, 7-3, 7-4, 7-5, 7-8, 7-22
ANALYZE_OBJECT 7-3
ANALYZE_SCHEMA 7-3
AND_EQUAL 3-10
Anti-join  9-32
Application tuning  1-7
ARRAYSIZE 2-11
AUTOTRACE 3-2, 3-7, 3-15, 3-16, 3-17, 5-4, 5-5, 7-27, 11-18

B

B*-tree index   6-7
Table access by ROWID 11-16
B*-tree index merge 11-16
B*-tree indexes  6-4, 6-8, 6-14, 11-4
B

B*-tree level statistics 7-7
B*-tree storage 13-9
B*-tree structure 13-4
Basic access methods 6-2
Benefits of temporary table 12-2
Bind variable 2-13
Bind variables 2-10
Bind—Execute—Fetch 2-8
Bitmap indexes 11-10
Bitmap indexing 11-3
Bitmap row source 11-19
Bitmapped index 11-2, 11-4, 11-8, 11-11, 11-18, 11-22
Block sample 7-13
Bounded and unbounded ranges 7-17
Bucket 7-20
Bucket endpoint 7-22
Buckets 7-28

C

CACHE 8-14
Calculate exact statistics 7-3
Cartesian product 9-23
CASCADE 7-13
CBO cost estimate 9-24
CHOOSE 5-9
Clause 5-4
Clear out statistic 7-3
Cluster 6-28, 6-29
Cluster index 6-31
Cluster join 9-36
Cluster key 6-28, 6-32
Cluster key column 6-30
Cluster statistics 7-2
Clusters performance 6-35
Collect index statistics 7-8
Collect statistics 7-3
Column expression 6-8
Column statistics 7-10, 7-11
column_name 13-8
Composite index 6-9
Compound 5-14
COMPRESS 6-9
COMPUTE STATISTICS 7-3, 7-4
Concatenated index 9-26
Concatenated indexes 9-3
Configure SQL Trace 4-2
Consider hash cluster 6-36
Constraint 6-2, 6-12
Contention 1-9
Context area 2-6
Cost-based optimization 5-3, 5-6, 5-12, 5-14
Cost-based optimizer 1-20
CPU time 4-19
CPU utilization 5-6
CREATE index-organized tables 13-8,
CREATE external tables 13-15
Creating bitmapped indexes 11-5
CREATE DIMENSION 12-2, 12-19
C
Create histogram  7-26
CREATE INDEX  6-8, 11-5
CREATE MATERIALIZED VIEW 12-4, 12-5, 12-11
CREATE OUTLINE 10-5
CREATE TABLE 13-9
Create stored outline 10-5
Cursor 2-6

D
Data buffers 2-11
Data dictionary 7-6
Data warehouse design 9-25
Database administration 1-18
Database triggers 2-16
Database tuning 1-7
DB_FILE_MULTIBLOCK_READ_COUNT 6-14
DBMS_DDL 7-3
DBMS_MVIEW 12-6
DBMS_OLAP 12-9
DBMS_SQL 7-19
DBMS_STATS 7-2, 7-12, 7-15
DBMS_UTILITY 7-3
Default number of buckets 7-27
Deferrable constraints 6-12
DENSITY 7-21
DESC 6-8
Development cycle 1-17
Dimension 12-15, 12-16, 12-18, 12-20
Dimension table 11-21
INDEX

D

DISTINCT 9-3
Distinct values is large 7-26
Distinguish between RBO and CBO 5-2
DML operation 6-11
Domain indexes 6-4
DRIVING_SITE 9-30
DROP UNUSED 10-10
Dynamic SQL 7-19

E

Elapsed time 4-19
ENABLE/DISABLE QUERY REWRITE 12-11
Entry header 6-6
Equality condition 7-18
Equijoin 9-6
Estimate statistic 7-3, 7-4
Estimates selectivity 7-21
Execute 4-3
Executing join statements 9-2
Execution plan 2-6, 2-9, 3-5, 4-3, 9-21, 9-23, 10-3, 10-8
EXPLAIN 3-16, 4-16, 5-5
EXPLAIN PLAN 3-2, 3-3, 3-7-3-10, 7-27
EXTERNAL 13-15
External table 13-3, 13-16
External tables 13-2

F

Fact table 11-21
Fast full index scan 6-14, 8-12, 11-16
Fast rebind-execute  2-10
Fetch  2-8
Fetch phase  4-3
FIRST_ROWS(n)  5-9
Force rule-based optimization  5-8
Foreign key  6-2
Foreign key constraint  6-13
Format standard  2-17
Frequently queried column  6-9
Full table scan  6-14, 9-12
Function-based indexes  11-2, 11-24, 6-4, 11-25
G
Gather statistics  7-13
GATHER_TABLE_STATS  7-16
GETHITRATIO  2-19
GLOBAL QUERY REWRITE  12-9
Good cluster key  6-30
GROUP BY  9-3
H
Hard parse  4-12
Hash cluster  6-33
Hash clusters performance  6-33
Hash function  6-33, 6-34
Hash join  9-12, 9-13, 9-33, 9-36
Hash join operation  9-32
Hash table  9-12
HASH_AREA_SIZE  9-13
HASH_JOIN_ENABLE  9-33
Hashing algorithm  6-34
Height-balanced histogram 7-20
Hierarchical relationship 12-19
Hierarchies 12-15, 12-16, 12-21
High-performance SQL statements 1-20
Hint 4-16, 8-6, 8-9, 8-10, 8-14, 8-15, 8-16, 9-27-9-30, 12-13
Hint comment 8-7
Hint delimited 8-6
Hints 5-4, 8-2, 8-8
Histogram 7-24
Histogram bucket 7-2, 7-22
Histograms 7-2, 7-20, 7-26
Histograms storage 7-25

Increasing response time 1-11
Index 6-4, 6-12, 9-11, 13-8
Index cluster 6-30, 6-31, 6-36
Index clustering factor 7-7
Index entries 6-6
Index hint 8-12
Index statistic 6-10, 7-9, 7-13
Index type 6-2
Index-organized table 13-2, 13-4, 13-5, 13-6, 13-9, 13-10
Index-organized uses 13-7
Indexed table access 9-8
Indexes 5-14
Indexing 4-18
Indicative measure 1-4
Influence rule-based optimization  5-11
Influence the oracle  8-2
Information in the dictionary  5-4
Initialization parameter  2-5, 5-9
Initialization parameters relating to SQL trace  4-5
INSERT  14-9
Instance tuning  1-7
INTERSECT  9-3
IOT  13-4, 13-5

Join hint  8-16
Join operation  9-7
Join predicate  9-5, 9-21

Key column length-value pairs  6-6

LEADING  9-27
Least recently used  2-6
Level of conformance  1-4
Library cache  4-15
Library routine  2-16
Log  12-5
Logical read  5-6
Logical reads/write  2-11
LRU  2-6
Materialized view log  12-5
Materialized view that is enabled  12-8
Materialized views  12-2, 12-3, 12-4, 12-6, 12-7, 12-14
MAX_DUMP_FILE_SIZE  4-6
Maximum selectivity  7-17
Memory usage  2-12
MERGE  8-17, 14-13, 14-15
Merge a view  8-16
Merge syntax  14-14
Most restrictive column  6-9
Multitable INSERT  14-9
Multitiered architecture  1-17

N-way join  9-35
Nested loops join  9-24, 9-9, 9-36
Network transmission  5-6
NO_INDEX  8-11
NO_MERGE  8-17
NOCACHE  8-14
NOLOGGING  6-8
Noncorrelated subquery  9-31
Nonequijoins  9-6, 9-8, 9-28
Nonjoin  9-9
Nonjoin predicate  9-5
Nonmergeable view  8-16
Nonpopular  7-21
Nonunique  6-4
NOSORT  6-8
NULL values  6-6

Object-oriented design technique  1-17
Operating system administrator  1-18
Optimization  1-20
Optimization performance  9-2
Optimizer  5-2, 8-2, 9-5
Optimizer approaches  5-9
Optimizer behavior  5-8
Optimizer mode  8-3, 8-10
Optimizer plan stability  10-2
OPTIMIZER_FEATURES_ENABLE  8-4
OPTIMIZER_INDEX_COST_AD  8-5
OPTIMIZER_MAX_PERMUTATION  9-23
OPTIMIZER_MODE  5-9, 12-12
Optimized join  9-19
Oracle call interface  2-16
Oracle optimizer  5-3
Oracle SQL Analyzer  2-22
Oracle tools  2-13
ORDERED hint  9-27, 9-29
ORGANIZATION clause  13-8, 13-15
Outer join  9-15
O
Outer join statement 9-20
OUTLN_PK 10-9
OVERFLOW 13-8, 13-9

P
Parameters 4-16, 6-14, 9-11, 9-33, 11-21
Parse 4-3, 4-12
Parse phase 2-12
Parse tree 2-11
Parsed statement 2-6
Parsing 1-20
PCTTHRESHOLD 13-8
Performance information 4-3
Performance management 1-3, 1-4
Performance problems 1-3, 1-9
Performance specialist 1-22
PGA 2-7
Physical attribute 6-8
Physical read 2-11
PINHITRATIO 2-19
Plan stability 10-4
Plan table 3-5
Plan table column 3-7, 3-8
PLAN_TABLE 3-2, 3-3
Populate 7-21
Predicate selectivity calculations 7-2
Primary or unique key constraint 6-12
Procedure GATHER_STATS 7-16
Processes SQL statement 2-2
Quantifiable objectives 1-21
Query rewrite 12-8
QUERY_REWRITE 12-9, 12-10
QUERY_REWRITE_ENABLE 12-12
QUERY_REWRITE_INTEGRITY 12-12
Query’s predicate 7-20

Ranking schema 5-12
Read consistency 4-19
readtrace.sql 4-8
Recursive SQ 2-22
Recursive SQL statements 4-15
REFRESH 12-7
REFRESH_ALL_MVIEWS 12-7
REFRESH_DEPENDENCY 12-7
Refreshing of materialized view 12-5
Regular join 9-23
Reorganize your b*-tree indexes 6-11
Resource 1-9
Response time 1-10
Restore old statistic 7-14
Restricted 6-3
Restricted ROWID 6-6
Restrictive table 5-14
Reverse key indexes 6-4
REWRITE/NOREWRITE 12-13
R

Root block  6-5
Row source  3-10
ROWID  3-10, 6-3, 6-2, 6-3, 6-5, 6-6, 11-4, 12-5
RULE  5-9
Rule-based approach  5-13
Rule-based optimization  1-20, 5-3, 5-5, 5-11, 5-12, 5-14, 9-22

S

Schema tuning  1-7
Selectivity  7-17
Semijoins  9-32
Service level agreement  1-4
Set of initialization parameter  8-4
SGA  2-4
Shared cursors  2-3, 2-14
Shared SQL area  2-2, 2-12, 2-18
SHAREDPool_SIZE  2-5
Sick join  9-35
Single-row predicate  9-5, 9-20
Soft parse  4-12
Sort merge operation  9-10, 9-22
Sort operation  9-10
Sort parameter  9-33
Sort/merge join  9-12, 9-24
SORT_AREA_SIZE  9-11
SORT_AREA_RETAINED_SIZE 9-3
SORT_MULTIBLOCK_READ_COUNT 9-3
SQL processing phases 2-3
SQL scripts 2-16
SQL statement tuning 1-19
SQL Trace 3-17
SQL trace facility 4-3, 4-5, 4-15, 4-18
SQL Trace for an entire instance 4-7
SQL Trace tool 4-4
SQL*Plus 3-7, 5-4, 5-5, 7-27
STAT 9-27
Stabilizing execution plan 10-3
Star join 9-25
Star query 11-20
Star query optimization 9-26
Star schema 9-25
Star transformation 9-25, 11-2, 11-20, 11-21
STAR_TRANSFORMATION_ENABLE 11-21
Statement 14-15
Statement syntax 2-9
STATISTICS 3-16
Statistics 5-4, 5-8
Steps 4-4
Store procedures 2-16
S

Stored outlines 10-2, 10-3, 10-4
Supplied package to switch on SQL trace 4-7
Switch off SQL trace 4-7
System Global Area 2-4

T

TABLE ACCESS 3-10
Table statistics 7-6
Temporary table 12-22, 12-24
Throughput 1-11, 1-12
Throwaway of rows 9-34
TIMED_STATISTICS 4-5
TKPROF 3-17, 4-2, 4-9, 4-12, 4-15, 4-16, 4-19,
Top-N SQL 9-2, 9-4
Trace file 4-3, 4-9
Trace statistics 4-2
Trigger 4-19
Tune the optimizer 8-5
Tuning methodology 1-2
Tuning process 1-18

U

UNION 9-3
Unique 6-4
UPDATE_BY_CAT 10-10
Use bitmapped indexes 11-9
Use cluster 6-35
USE_HASH  9-29
USE_MERGE  9-28, 9-29
USE_STORED_OUTLINES  10-6, 10-8
USER_DUMP_DEST  4-6, 4-8
USER_INDEXES  7-9
USER_OUTLINES  10-7
USER_OUTLINE_HINTS  10-7
USER_TAB_COL_STATISTICS  7-11
USER_TABLES  7-6
utlxplan.sql  3-2, 3-3

V$LIBRARYCACHE  2-18
V$LIBRARYCACHE  2-19
V$SESSION  4-7
V$SQLAREA  2-18, 2-20

WHERE  5-4, 5-14
Width-balanced histogram  7-20
WITH clause  14-3, 14-6