Troubleshooting Oracle Performance

Christian Antognini
christian.antognini@trivadis.com
@ChrisAntognini

- Senior principal consultant, trainer and partner at Trivadis in Zurich (CH)
  - christian.antognini@trivadis.com
  - http://antognini.ch
- Focus: get the most out of Oracle Database
  - Logical and physical database design
  - Query optimizer
  - Application performance management
- Proud member of OakTable Network, Oracle ACE Director
Our company

Trivadis is a market leader in IT consulting, system integration, solution engineering and the provision of IT services focusing on Oracle and Microsoft technologies in Switzerland, Germany and Austria.

We offer our services in the following strategic business fields:

- Business Intelligence
- Application Development
- Infrastructure Engineering
- Training

Trivadis Services takes over the interacting operation of your IT systems.
With over 600 specialists and IT experts in your region

- 12 Trivadis branches and more than 600 employees
- 200 Service Level Agreements
- Over 4,000 training participants
- Research and development budget: CHF 5.0 / EUR 4 million
- Financially self-supporting and sustainably profitable
- Experience from more than 1,900 projects per year at over 800 customers
Troubleshooting Oracle Performance, Apress 2014

- Foundations
  1. Performance Problems
  2. Key Concepts

- Identification
  3. Analysis of Reproducible Problems
  4. Real-Time Analysis of Irreproducible Problems
  5. Postmortem Analysis of Irreproducible Problems

- Query Optimizer
  6. Introducing the Query Optimizer
  7. System Statistics

- Query Optimizer (cont.)
  8. Object Statistics
  9. Configuring the Query Optimizer
  10. Execution Plans

- Optimization
  11. SQL Optimization Techniques
  12. Parsing
  13. Optimizing Data Access
  14. Optimizing Joins
  15. Beyond Data Access and Join Optimization
  16. Optimizing the Physical Design
Which Versions Are Covered?

- Oracle Database 10g Release 2, up to version 10.2.0.5.0
- Oracle Database 11g Release 1, up to version 11.1.0.7.0
- Oracle Database 11g Release 2, up to version 11.2.0.4.0
- Oracle Database 12c Release 1, version 12.1.0.1.0
THANK YOU.
Troubleshooting Oracle Performance

Key Concepts
Agenda

1. Selectivity and Cardinality
2. What Is a Cursor?
3. Life Cycle of a Cursor
4. How Parsing Works
Selectivity

- It’s a value between 0 and 1.
  - It can also be represented as a percentage.
- It represents the fraction of rows filtered by an operation.
- Example: an operation that reads 120 rows and returns 18 of them has a selectivity of 0.15 (18/120).
Cardinality

- It’s the number of rows returned by an operation.

  cardinality = selectivity × number of rows

- **Caution:**
  - In some publications, the term cardinality refers to the number of distinct values stored in a particular column.
  - I never use the term cardinality in this way.
Agenda

1. Selectivity and Cardinality
2. What Is a Cursor?
3. Life Cycle of a Cursor
4. How Parsing Works
Cursors, Private SQL Areas and Shared SQL Areas (1)

- It’s a handle to a *private SQL area* with an associated *shared SQL area*.

- It’s created along with its associated structures when an application issues a parse call.

![Diagram showing the relationship between Client Memory, Server Process Memory, and SGA Library Cache with references to Handle, Private SQL Area, and Shared SQL Area.](image-url)
Cursors, Private SQL Areas and Shared SQL Areas (2)

- What does a private SQL area contain?
  - Data such as bind variable values and query execution state information.
  - It’s session specific → It’s stored in the UGA.

- What does a shared SQL area contain?
  - The parse tree and the execution plan of the associated SQL statement.
  - It’s stored in the library cache.

- The terms cursor and private/shared SQL area are used interchangeably.

- The shared SQL areas contain the so-called parent and child cursors.
Agenda

1. Selectivity and Cardinality
2. What Is a Cursor?
3. Life Cycle of a Cursor
4. How Parsing Works
Life Cycle of a Cursor

- Open cursor
- Parse cursor
- Returns data?
  - Yes: Define output variables
  - No: Uses variables?
    - Yes: Bind input variables
    - No: Execute cursor
- Returns data?
  - Yes: Fetch cursor
  - No: Close cursor
- More data?
  - Yes: Continue
  - No: End
Agenda

1. Selectivity and Cardinality
2. What Is a Cursor?
3. Life Cycle of a Cursor
4. How Parsing Works
Steps Carried Out During the Parse Phase

1. Include VPD predicates
2. Check syntax, semantics, and access rights
3. Shareable parent?
   - No: Store parent cursor in library cache
   - No: Logical Optimization
4. Shareable child?
   - No: Physical Optimization
   - No: Store child cursor in library cache
Parent Cursors

- The key information stored in a parent cursor is the text of the SQL statement on which it is based.
- Several SQL statements share the same parent cursor if their text is exactly the same.
  - The only exception is when CURSOR_SHARING<>EXACT.
- V$SQLAREA returns one row for each parent cursor.
  - The SQL_ID column identifies a parent cursor.
Child Cursors

- The key information stored in a child cursor is the execution plan and the execution environment related to it.
- Several SQL statements are able to share the same child cursor only if
  - they share the same parent cursor, and
  - their execution environments are compatible.
- V$SQL_SHARED_CURSOR shows why a child cursor was created.
- V$SQL returns one row for each child cursor.
  - The SQL_ID and CHILD_NUMBER columns identify a parent cursor.
Bind Variables

- Bind variables impact applications in two ways:
  - They make programming either easier or more difficult.
  - They introduce both an pros and cons.

- Pros
  - Sharing of cursors in the library cache → Avoid hard parses.

- Cons
  - Crucial information is hidden from the query optimizer.
Bind Variable Graduation

- The type *and* the size of bind variables matter for sharing a child cursor.
- To relax the limitation related to the size, *bind variable graduation* is used to part bind variables into four groups:
  - Up to 32 bytes
  - 33 – 128 bytes
  - 129 – 2,000 bytes
  - More than 2,000 bytes
- Without this feature programming languages with character strings of variable length (e.g. Java) would require an excessive number of child cursors.
- V$SQL_BIND_METADATA shows the maximum length of the group, not of the bind variable itself.
Bind Variable Peeking

- To address the disadvantage related to bind variables, *bind variable peeking* was introduced in 9i.
- The query optimizer is able to peek at the value of bind variables.
- Cons: the generated execution plan depends on the values provided by the first execution.
Bind Variables – Best Practices

- There is no reason for *not* using bind variables with SQL statements without a WHERE clause.
- Bind variables should be avoided at all costs whenever histograms provide important information to the query optimizer.
- Whenever little data is processed, the parsing time might be close to or even higher than the execution time.
  ➔ Using bind variables is usually a good thing.
- Whenever a lot of data is processed, the parsing time is usually several orders of magnitude less than the execution time.
  ➔ Bind variables should not be used.
Feedback-Based Optimization (11g)

- The SQL engine is able to give feedback to the query optimizer.
- The query optimizer uses the feedback to improve its estimations.
- The communication occurs through the library cache.
  - Feedback is not persisted (yet).
- Feedback is disabled if STATISTICS_LEVEL is set to BASIC at the system level.
- Two features use this technique:
  - Adaptive cursor sharing
  - Cardinality feedback
Adaptive Cursor Sharing

- Was introduced to address the problem introduced by bind variable peeking.
- The SQL engine gives feedback in case of a significant variation in row source cardinalities.
- Simply put, the feedback has three main consequences:
  - The suboptimal child cursor is invalidated.
  - The cursor is made *bind-aware*.
  - A new child cursor is generated.
- By default new parent cursors are created not bind-aware.
  - Hints: (NO_)BIND_AWARE (as of 11.1.0.7 only)
- V$SQL_CS_* provide detailed runtime information.
Cardinality Feedback

- The query optimizer can take advantage of actual cardinalities obtained by a previous execution.
  - Not used for every SQL statement.
  - Only performed for the first execution of a given child cursor.
- Simply put, the feedback has two main consequences:
  - The suboptimal child cursor is invalidated.
  - A new child cursor is generated.
- As a result, when you rerun a statement, you might get a different execution plan.
Core Messages

- Use bind variables only when it makes sense.
THANK YOU.
Agenda

1. Fundamentals
2. Architecture
3. Query Transformations
Questions that Needs Answers

- Which is the optimal access path to extract data from each table referenced in the SQL statement?
- Which are the optimal join methods and join orders through which the data of the referenced tables will be processed?
- When should aggregations and/or sorts be processed during SQL statement execution?
- Is it beneficial to use parallel processing?
Search Space

- It consists of all potential execution plans.
- Even a simple query has many execution plans.

```
SELECT *
FROM t1 JOIN t2 ON t1.id = t2.t1_id
WHERE t1.n = 1 AND t2.n = 2
```

- The query optimizer estimates the cost of a number of execution plans and picks the one with the lowest cost.
- The query optimizer explorer only a subset of the search space.
The Query Optimizer Considers a Number of Inputs

SQL Statement

- Data Dictionary
- System Statistics
- Object Statistics
- Constraints
- Physical Design
- SQL Controls

Query Optimizer

- Runtime
  - Execution Environment
  - Bind Variables
  - Dynamic Sampling
  - Cardinality Feedback

Execution Plan
Agenda

1. Fundamentals
2. Architecture
3. Query Transformations
Architecture of the SQL Engine
Logical Optimizer

- New and semantically equivalent SQL statements are produced by applying different query transformation techniques.
- The purpose is to select the best combination of query transformations.
- The search space is increased.
- More execution plans can be considered.
Physical Optimizer

- Several execution plans for each SQL statement resulting from the logical optimization are generated.
- Every one of them is passed to the cost estimator to let it calculate a cost.
- The execution plan with the lowest cost is selected.
- Simply put, the physical optimizer explores the search space to find the most efficient execution plan.
Agenda

1. Fundamentals
2. Architecture
3. Query Transformations
Query Transformations (1)

- Heuristic-based query transformations
  - They are applied when specific conditions are met.
  - They’re expected to lead to better execution plans in most situations.

- Cost-based query transformations
  - They are applied when, according to the cost computed by the cost estimator, they lead to execution plans with lower costs than the original statement.
Query Transformations (2)

- Count Transformation
- Common Sub-Expression Elimination
- Or Expansion
- Simple View Merging
- Complex View Merging
- Select List Pruning
- Filter Push Down
- Join Predicate Push Down
- Predicate Move Around
- Distinct Placement
- Distinct Elimination
- Group-by Placement
Query Transformations (3)

- Order-By Elimination
- Subquery Unnesting
- Subquery Coalescing
- Subquery Removal using Window Functions
- Join Elimination
- Join Factorization
- Outer Join to Inner Join
- Full Outer Join
- Table Expansion
- Set to Join Conversion
- Star Transformation
- Query Rewrite with Materialized Views
Count Transformation

- The purpose of *count transformation* is to transform `count(column)` expressions to `count(*)`.

```
SELECT count(n2)
FROM t
```

```
SELECT count(*)
FROM t
```

- Count transformation also transforms `count(1)` expressions to `count(*)`.

```
SELECT count(1)
FROM t
```

```
SELECT count(*)
FROM t
```
Common Sub-Expression Elimination

- The purpose of *common sub-expression elimination* is to remove duplicate predicates and thereby avoid processing the same operation several times.

```sql
SELECT * FROM t
WHERE (n1 = 1 AND n2 = 2)
OR (n1 = 1)
```

```sql
SELECT * FROM t
WHERE n1 = 1
```
Or Expansion

- The purpose of or expansion is to transform a query with a WHERE clause containing disjunctive predicates into a compound query that uses one or several UNION ALL set operators.

```sql
SELECT pad
FROM t
WHERE n1 = 1
OR n2 = 2

SELECT pad
FROM t
WHERE n1 = 1
UNION ALL
SELECT pad
FROM t
WHERE n2 = 2 AND lnnvl(n1 = 1)
```
Simple View Merging

- The purpose of *view merging* is to reduce the number of query blocks due to views and inline views by merging several of them together.
- Simple view merging is used for merging plain, select-project-join query blocks.

```sql
SELECT * FROM (SELECT t1.* FROM t1, t2 WHERE t1.id = t2.t1_id) t12, (SELECT * FROM t3 WHERE id > 6) t3 WHERE t12.id = t3.t1_id

SELECT t1.*, t3.* FROM t1, t2, t3 WHERE t1.id = t3.t1_id AND t1.id = t2.t1_id AND t3.id > 6
```
Complex View Merging

- *Complex view merging* is used for merging query blocks that contain aggregations.

```sql
SELECT t1.id, t1.n, t1.pad, t2.sum_n
FROM t1,
     (SELECT n, sum(n) AS sum_n
      FROM t2
      GROUP BY n) t2
WHERE t1.n = t2.n
```

```sql
SELECT t1.id, t1.n, t1.pad, sum(n) AS sum_n
FROM t1, t2
WHERE t1.n = t2.n
GROUP BY t1.id, t1.n, t1.pad, t1.rowid, t2.n
```
Select List Pruning

- The purpose of select list pruning is to remove unnecessary columns or expressions from the SELECT clause of subqueries, inline views, or regular views.

```
SELECT n1 FROM (SELECT n1, n2, n3 FROM t)
```

```
SELECT n1 FROM (SELECT n1 FROM t)
```
Filter Push Down

- The purpose of *filter push down* is to push restrictions inside views or inline views that can’t be merged.

```sql
SELECT *
FROM (SELECT *
      FROM t1
      UNION
      SELECT *
      FROM t2)
WHERE id = 1
```

```sql
SELECT *
FROM (SELECT *
      FROM t1
      WHERE id = 1
      UNION
      SELECT *
      FROM t2
      WHERE id = 1)
```
Join Predicate Push Down

- The purpose of *join predicate push down* is to push join conditions inside views or inline views that can’t be merged.

```sql
SELECT *
FROM t1, (SELECT *
           FROM t2
           UNION
           SELECT *
           FROM t3) t23
WHERE t1.id = t23.id
```

```sql
SELECT *
FROM t1,
     (SELECT *
      FROM t2
      WHERE t2.id = t1.id
      UNION
      SELECT *
      FROM t3
      WHERE t3.id = t1.id) t23
```

This is not a valid SQL statement
Predicate Move Around

- The purpose of *predicate move around* is to pull up, move across, and push down restrictions inside views or inline views that can’t be merged.

```sql
SELECT t1.pad, t2.pad
FROM (SELECT DISTINCT n, pad
      FROM t1
      WHERE n = 1) t1,
    (SELECT DISTINCT n, pad
     FROM t2) t2
WHERE t1.n = t2.n
```

```sql
SELECT t1.pad, t2.pad
FROM (SELECT DISTINCT n, pad
      FROM t1
      WHERE n = 1) t1,
    (SELECT DISTINCT n, pad
     FROM t2
     WHERE n = 1) t2
WHERE t1.n = t2.n
```
Distinct Placement (11.2+)

- The purpose of *distinct placement* is to eliminate duplicates as soon as possible.

```sql
SELECT DISTINCT t1.n, t2.n
FROM t1, t2
WHERE t1.id = t2.t1_id
```

```sql
SELECT DISTINCT t1.n, vw_dtp.n
FROM t1,
   (SELECT DISTINCT t2.t1_id, t2.n
    FROM t2) vw_dtp
WHERE t1.id = vw_dtp.t1_id
```
Distinct Elimination (10.2.0.4+)

- The purpose of *distinct elimination* is to remove DISTINCT operators that aren’t required to guarantee that the result set doesn’t contain duplicates.

```
SELECT DISTINCT id, n
FROM t
```

```
SELECT id, n
FROM t
```
Group-by Placement (11.1+)

- The purpose of *group-by placement* is basically the same as that of *distinct placement*.
- The only obvious difference is the types of queries to which they’re applied.

```
SELECT t1.n, t2.n, count(*)
FROM t1, t2
WHERE t1.id = t2.t1_id
GROUP BY t1.n, t2.n
```

```
SELECT t1.n, vw_gb.n,
       sum(vw_gb.cnt)
FROM t1,
     (SELECT t2.t1_id, t2.n,
          count(*) AS cnt
       FROM t2
       GROUP BY t2.t1_id,
               t2.n) vw_gb
WHERE t1.id = vw_gb.t1_id
GROUP BY t1.n, vw_gb.n
```
Order-By Elimination

- The purpose of order-by elimination is to remove superfluous ORDER BY clauses from subqueries, inline views, and regular views.

```
SELECT n2, count(*)
FROM (SELECT n1, n2
     FROM t
     ORDER BY n1)
GROUP BY n2
```

```
SELECT n2, count(*)
FROM (SELECT n1, n2
     FROM t)
GROUP BY n2
```
Subquery Unnesting

- The purpose of subquery unnesting is to inject semi-, anti-join and scalar subqueries into the FROM clause of the containing query block.

```
SELECT * 
FROM t1 
WHERE EXISTS 
  (SELECT 1 FROM t2 
   WHERE t2.id = t1.id 
   AND t2.pad IS NOT NULL)
```

```
SELECT * 
FROM t1, 
  (SELECT id FROM t2 
   WHERE pad IS NOT NULL) sq 
WHERE t1.id = sq.id
```

s= is not a valid SQL operator

```
SELECT t1.* 
FROM t1, t2 
WHERE t1.id = t2.id 
AND t2.pad IS NOT NULL
```
Subquery Coalescing (11.2+)

- The purpose of *subquery coalescing* is to combine equivalent semi- and anti-join subqueries into a single query block.

```
SELECT *  
FROM t1  
WHERE EXISTS  
  (SELECT 1  
   FROM t2  
   WHERE t2.id = t1.id  
   AND t2.n > 10)  
OR EXISTS  
  (SELECT 1  
   FROM t2  
   WHERE t2.id = t1.id  
   AND t2.n < 100)
```

```
SELECT *  
FROM t1  
WHERE EXISTS  
  (SELECT 1  
   FROM t2  
   WHERE t2.id = t1.id  
   AND (t2.n > 10 OR  
   t2.n < 100))
```
Subquery Removal using Window Functions

- The purpose of subquery removal using window functions is to replace subqueries containing aggregate functions with window functions.

```
SELECT t1.id, t1.n, t2.id, t2.n
FROM t1, t2
WHERE t1.id = t2.t1_id
AND t2.n = (SELECT max(n) FROM t2 WHERE t2.t1_id = t1.id)
```

```
SELECT t1_id, t1_n, t2_id, t2_n
FROM (SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id,
      t2.n AS t2_n,
      CASE t2.n
          WHEN max(t2.n) OVER (PARTITION BY t2.t1_id) THEN 1
          ELSE 0
      END AS max
FROM t2, t1
WHERE t1.id = t2.t1_id) vw_wif
WHERE max IS NOT NULL
```
Join Elimination

- The purpose of *join elimination* is to remove redundant joins.

```sql
CREATE VIEW v AS
SELECT t1.id AS t1_id, t1.n AS t1_n, t2.id AS t2_id, t2.n AS t2_n
FROM t1, t2
WHERE t1.id = t2.t1_id

SELECT t2_id, t2_n
FROM v

SELECT t2.id AS t2_id,
       t2.n AS t2_n
FROM t2
```
Join Factorization (11.2+)

- The purpose of join factorization is to recognize whether part of the processing of a compound query can be shared across component queries, with the goal being to avoid repetitive data accesses and joins.

```sql
SELECT * 
FROM t1, t2 
WHERE t1.id = t2.id 
AND t2.id < 10 
UNION ALL 
SELECT * 
FROM t1, t2 
WHERE t1.id = t2.id 
AND t2.id > 990
```

```sql
SELECT t1.*, vw_jf.* 
FROM t1, (SELECT * 
          FROM t2 
          WHERE id < 10 
          UNION ALL 
          SELECT * 
          FROM t2 
          WHERE id > 990) vw_jf 
WHERE t1.id = vw_jf.id
```
Outer Join to Inner Join

- The purpose of *outer join to inner join* is to convert superfluous outer joins into inner joins.

```
SELECT *  
FROM t1, t2  
WHERE t1.id = t2.t1_id(+)
AND t2.id IS NOT NULL
```

```
SELECT *  
FROM t1, t2  
WHERE t1.id = t2.t1_id
```
The purpose of *table expansion* is to enable the use of as many index scans as possible by also leveraging partially unusable indexes.

```sql
CREATE TABLE t ( 
    id NUMBER PRIMARY KEY, 
    d DATE NOT NULL, 
    n NUMBER NOT NULL, 
    pad VARCHAR2(4000) NOT NULL 
) 
PARTITION BY RANGE (d) (...)

SELECT * 
FROM (SELECT * 
    FROM t 
    WHERE n = 8 
    AND d < to_date(...) 
    UNION ALL 
    SELECT * 
    FROM t 
    WHERE n = 8 
    AND d >= to_date(...) 
    AND d < to_date(...) 
) vw_te

SELECT * 
FROM t 
WHERE n = 8
```
Set to Join Conversion (disabled)

- The purpose of *set to join conversion* is to avoid sort operations in compound queries involving INTERSECT and MINUS.

```sql
SELECT * 
FROM t1 
WHERE n > 500 
INTERSECT 
SELECT * 
FROM t2 
WHERE t2.pad LIKE 'A%'
```

```sql
SELECT DISTINCT t1.* 
FROM t1, t2 
WHERE t1.id = t2.id 
AND t1.n = t2.n 
AND t1.pad = t2.pad 
AND t1.n > 500 
AND t1.pad LIKE 'A%' 
AND t2.n > 500 
AND t2.pad LIKE 'A%'
```
Core Messages

- The query optimizer bases its work on a number of inputs.
- Many query transformations exist.
THANK YOU.
Agenda

1. What System Statistics Are Available?
2. Gathering System Statistics
3. Restoring System Statistics
4. Logging of Management Operations
5. Impact on the Query Optimizer
I/O Cost Model vs. CPU Cost Model

- The query optimizer used to base its cost estimations on the number of physical reads needed to execute SQL statements.
- This method is known as the I/O cost model.
- Main drawback: single-block reads and multiblock reads are equally costly.
  - Multiblock read operations, such as full table scans, are artificially favored.
- To address this flaw, a new costing method, known as CPU cost model, is available.
CPU Cost Model Needs Additional Information

- To use the CPU cost model *system statistics* have to be provided to the query optimizer.

- Essentially, system statistics supply the following information:
  - Performance of the I/O subsystem
  - Performance of the CPU

- There are two kinds of system statistics:
  - Noworkload statistics
  - Workload statistics

- The main difference between the two is the method used to measure the performance of the disk I/O subsystem.
Data Dictionary

- System statistics are stored in the AUX_STATS$ data dictionary table.
  - Noworkload
  - Workload
- No data dictionary view is available to externalize them.

```
SQL> SELECT pname,
        2 nvl(pval2,pval1) pval
        3 FROM sys.aux_stats$;

<table>
<thead>
<tr>
<th>PNAME</th>
<th>PVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>DSTART</td>
<td>04-04-2007 14:26</td>
</tr>
<tr>
<td>DSTOP</td>
<td>04-04-2007 14:36</td>
</tr>
<tr>
<td>FLAGS</td>
<td>1</td>
</tr>
<tr>
<td>CPUSPEEDNW</td>
<td>1617.6</td>
</tr>
<tr>
<td>IOSEEKTIM</td>
<td>10</td>
</tr>
<tr>
<td>IOTFRSPEED</td>
<td>4096</td>
</tr>
<tr>
<td>SREADTIM</td>
<td>1.3</td>
</tr>
<tr>
<td>MREADTIM</td>
<td>7.8</td>
</tr>
<tr>
<td>CPUSPEED</td>
<td>1620</td>
</tr>
<tr>
<td>MBRC</td>
<td>7</td>
</tr>
<tr>
<td>MAXTHR</td>
<td>473982976</td>
</tr>
<tr>
<td>SLAVETHR</td>
<td>1781760</td>
</tr>
</tbody>
</table>
```
Agenda

1. What System Statistics Are Available?
2. Gathering System Statistics
3. Restoring System Statistics
4. Logging of Management Operations
5. Impact on the Query Optimizer
Gathering Noworkload Statistics

- Noworkload statistics are always available.
- Noworkload statistics can be gathered on an idle system.
- A synthetic benchmark is used to measure the performance of the system.

```sql
dbms_stats.gather_system_stats(gathering_mode => 'noworkload')
```
Noworkload Statistics Stored in the Data Dictionary

- **CPUSPEEDNW**: The number of operations per second (in millions) that one CPU is able to process.
- **IOSEEKTIM**: Average time (in milliseconds) needed to locate data on the disk. The default value is 10.
- **IOTFRSPEED**: Average number of bytes per millisecond that can be transferred from the disk. The default value is 4,096.
Gathering Workload Statistics (1)

- Workload statistics are available only when explicitly gathered.
- To gather them, you cannot use an idle system.
  - The regular database load is used to measure the performance of the disk I/O subsystem.

```
dbms_stats.gather_system_stats(gathering_mode => 'start')
```

```
dbms_stats.gather_system_stats(gathering_mode => 'stop')
```
Gathering Workload Statistics (2)

- It is essential to choose the right gathering period...
Workload Statistics Stored in the Data Dictionary

- **CPUSPEED**: The number of operations per second (in millions) that one CPU is able to process.
- **SREADTIM**: Average time (in milliseconds) needed to perform a single-block read operation.
- **MREADTIM**: Average time (in milliseconds) needed to perform a multiblock read operation.
- **MBRC**: Average number of blocks read during a multiblock read operation.
- **MAXTHR**: Maximum I/O throughput (in bytes per second) for the whole system.
- **SLAVETHR**: Average I/O throughput (in bytes per second) for a parallel processing slave.
Workload Statistics Bug

- In 11.2 the system statistics gathered by DBMS_STATS are broken.
  - SREADTIM and MREADTIM are very high.

- Solution:
  - Install patch 9842771 (available for 11.2.0.1 and 11.2.0.2)
  - Install patch set 11.2.0.3
Choosing Between Noworkload Statistics and Workload Statistics

- Noworkload statistics $\rightarrow$ Simplicity
  - Simplest approach: use the default values (call DELETE_SYSTEM_STATS)

- Workload statistics $\rightarrow$ Control
  - DB_FILE_MULTIBLOCK_READ_COUNT no longer impacts costs
  - MAXTHR and SLAVETHR control costing of parallel operations
Agenda

1. What System Statistics Are Available?
2. Gathering System Statistics
3. Restoring System Statistics
4. Logging of Management Operations
5. Impact on the Query Optimizer
Restoring System Statistics

- The database engine keeps a history of all changes occurring within a retention period.
  - Default retention: 31 days

- If necessary, it’s possible to restore old statistics:

  ```sql
  dbms_stats.restore_system_stats(
    as_of_timestamp => systimestamp - INTERVAL '1' DAY
  )
  ```

- **CAUTION**: delete the current system statistics before the restore

  ```sql
  dbms_stats.delete_system_stats()
  ```
Agenda

1. What System Statistics Are Available?
2. Gathering System Statistics
3. Restoring System Statistics
4. Logging of Management Operations
5. Impact on the Query Optimizer
Logging of Management Operations

- All DBMS_STATS procedures used for managing system statistics, except for RESTORE_SYSTEM_STATS, log some information about their activities into the data dictionary.

- This information is available through
  - DBA_OPTSTAT_OPERATIONS
  - DBA_OPTSTAT_OPERATION_TASKS (12.1 only)

- Only in 12.1 DBA_OPTSTAT_OPERATIONS shows the parameters with which an operation was executed.
  - For simplicity use DBMS_STATS.REPORT_SINGLE_STATS_OPERATION function
Agenda

1. What System Statistics Are Available?
2. Gathering System Statistics
3. Restoring System Statistics
4. Logging of Management Operations
5. Impact on the Query Optimizer
CPU and I/O Costs

- The query optimizer computes two costs: I/O and CPU.
- Chapter 9 describes how I/O costs are computed.
- Very little information is available about the computation of CPU costs.
  - We can imagine that a cost in terms of CPU is associated to operations.
- The overall cost is computed with the following formula:

\[
\text{cost} \approx \text{io\_cost} + \frac{\text{cpu\_cost}}{\text{cpuspeed} \cdot \text{sreadtim} \cdot 1000}
\]

- For noworkload system statistics SREADTIM is computed as follows:

\[
\text{sreadtim} \approx \text{ioseektim} + \frac{\text{db\_block\_s} \cdot \text{ize}}{\text{iotfrspeed}}
\]
Workload Statistics Sanity Checks

- The query optimizer performs several sanity checks that could disable or partially replace workload statistics.
- When MBRC isn’t available or set to 0, workload statistics are ignored.
- When SREADTIM isn’t available or set to 0, SREADTIM and MREADTIM are recomputed.
- When MREADTIM isn’t available, or when it isn’t greater than SREADTIM, SREADTIM and MREADTIM are recomputed.

\[
mreadtim \approx ioseektim + \frac{mbrc \cdot db\_block\_size}{iotfrspeed}
\]
MAXTHR and SLAVETHR Control Costing of Parallel Operations

- Without MAXTHR and SLAVETHR, the query optimizer considers that the cost of an operation executed in parallel is inversely proportional to the degree of parallelism used to execute it.

\[
\text{parallel}_i \cdot \text{o\_cost} \approx \frac{\text{serial\_io\_cost}}{\text{dop} \cdot 0.9}
\]

- SLAVETHR can be used to increase the costs.
- MAXTHR can be used to cap the costs.
  - Database servers don’t scale infinitely!
Core Messages

- The query optimizer needs system statistics to successfully execute its task.
THANK YOU.
Agenda

1. What Object Statistics Are Available?
2. Managing Object Statistics
3. Strategies for Keeping Object Statistics Up-to-Date
Table Statistics – USER_TAB_STATISTICS

- **NUM_ROWS** is the number of rows.
- **BLOCKS** is the number of blocks below the high watermark.
- **AVG_ROW_LEN** is the average row size (in bytes).
- **EMPTY_BLOCKS** is the number of blocks above the high watermark.
  - This value is not computed by DBMS_STATS, it is set to 0.
- **AVG_SPACE** is the average free space (in bytes) at block level.
  - This value is not computed by DBMS_STATS, it is set to 0.
- **CHAIN_CNT** is the number of chained/migrated rows.
  - This value is not computed by DBMS_STATS, it is set to 0.
**Column Statistics – USER_TAB_COL_STATISTICS**

- **NUM_DISTINCT** is the number of distinct values.
- **LOW_VALUE** is the lowest value.
- **HIGH_VALUE** is the highest value.
- **DENSITY** is a decimal number between 0 and 1.
  - Values close to 0 indicate that a restriction filters out the majority of the rows.
  - Values close to 1 indicate that a restriction filters almost no rows.
- **NUM_NULLS** is the number of NULL values.
- **AVG_COL_LEN** is the average column size (in bytes).
- **HISTOGRAM** indicates whether a histogram is available.
- **NUM_BUCKETS** is the number of buckets in the histogram.
Column Statistics – Frequency Histograms (1)
Column Statistics – Frequency Histograms (2)

```sql
SQL> SELECT endpoint_value, endpoint_number, 
2       endpoint_number - lag(endpoint_number,1,0) 
3       OVER (ORDER BY endpoint_number) AS frequency 
4   FROM user_tab_histograms 
5  WHERE table_name = 'T' 
6   AND column_name = 'VAL2' 
7  ORDER BY endpoint_number;
```

<table>
<thead>
<tr>
<th>ENDPOINT_VALUE</th>
<th>ENDPOINT_NUMBER</th>
<th>FREQUENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>102</td>
<td>33</td>
<td>25</td>
</tr>
<tr>
<td>103</td>
<td>101</td>
<td>68</td>
</tr>
<tr>
<td>104</td>
<td>286</td>
<td>185</td>
</tr>
<tr>
<td>105</td>
<td>788</td>
<td>502</td>
</tr>
<tr>
<td>106</td>
<td>1000</td>
<td>212</td>
</tr>
</tbody>
</table>
Column Statistics – Height-Balanced Histograms (1)
Column Statistics – Height-Balanced Histograms (2)

```sql
SQL> SELECT endpoint_value, endpoint_number
2  FROM user_tab_histograms
3  WHERE table_name = 'T'
4  AND column_name = 'VAL2'
5  ORDER BY endpoint_number;

<table>
<thead>
<tr>
<th>ENDPOINT_VALUE</th>
<th>ENDPOINT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
</tr>
<tr>
<td>105</td>
<td>3</td>
</tr>
<tr>
<td>106</td>
<td>5</td>
</tr>
</tbody>
</table>
```
Column Statistics – Histograms in 12c

- New maximum number of buckets: 2048
- More efficient way to gather histograms (AUTO_SAMPLE_SIZE only)
- New types of histograms
  - Top-frequency histograms
  - Hybrid histograms
- Top frequency histograms and hybrid histograms are supposed to replace height-balanced histograms
- Except if ESTIMATE_PERCENT is set to an integer value, height-balanced histograms are no longer created
Column Statistics – Top-Frequency Histograms (1)

- Top frequency histograms are similar to frequency histograms.
- Only the top-n values are represented.
- The minimum and maximum value is always represented.
- Conditions:
  - Number of distinct values larger than \( n \)
  - Top-n values account for at least \( x \) percent of the rows
    - \( x = 100 - \frac{100}{n} \)
  - ESTIMATE_PERCENT must be set to AUTO_SAMPLE_SIZE
Column Statistics – Top-Frequency Histograms (2)

```sql
SQL> SELECT endpoint_value, endpoint_number,
        2     endpoint_number - lag(endpoint_number,1,0)
        3   OVER (ORDER BY endpoint_number) AS frequency
        4   FROM user_tab_histograms
        5   WHERE table_name = 'T'
        6   AND column_name = 'VAL3'
        7   ORDER BY endpoint_number;

ENDPOINT_VALUE  ENDPOINT_NUMBER  FREQUENCY
---------------  ---------------  ------
     101            1            1
     103            69           68
     104           254          185
     105           756          502
     106           968          212
```
Column Statistics – Hybrid Histograms

- Combination of height-balanced histograms with frequency histograms
- Improvements compared to height-balanced histograms
  - One value is stored in a single bucket
  - Frequency information added to the endpoint values to recognize almost popular values
Extended Statistics – Expressions

- As of 11.1 it is possible to instruct DBMS_STATS to gather statistics on expressions.
  - Called extended statistics
  - Only makes sense for expressions used in WHERE clauses
  - With them the query optimizer might improve cardinality estimations

```sql
dbms_stats.create_extended_stats(ownname => user, 
                                   tabname  => 'T', 
                                   extension => '( trunc(col))')
```

- CREATE_EXTENDED_STATS only adds a hidden virtual column
  - Since the column is hidden, it is transparent for the application
  - Statistics are gathered with the usual procedures
Extended Statistics – Column Groups

- Extended statistics are not limited to expression, they can also be created for column groups.
  - Useful to track correlated columns

```sql
DBMS_STATS.CREATE_EXTENDED_STATS(ownname => user,
                                  tabname  => 'T',
                                  extension => '(col1,col2)')
```

- The query optimizer can only take advantage of extended statistics when conditions are based on equality.
Extended Statistics – Seeding Column Groups

- As of 11.2.0.2 it is possible instruct DBMS_STATS to automatically detect which column groups should be created.

  1. Record usage of column groups for x seconds.

     ```sql
     dbms_stats.seed_col_usage(time_limit => 30);
     ```

  2. Generate a report of column groups usage (optional).

     ```sql
     dbms_stats.report_col_usage(ownname => user, tabname => NULL)
     ```

  3. Create extensions based on groups of columns seen in workload.

     ```sql
     dbms_stats.create_extended_stats(ownname => user, tabname => NULL)
     ```
Index Statistics – USER_IND_STATISTICS

- BLEVEL is the number of branch blocks to be read, including the root block, in order to access a leaf block.
- LEAF_BLOCKS is the number of leaf blocks.
- DISTINCT_KEYS is the number of distinct keys.
- NUM_ROWS is the number of keys.
- CLUSTERING_FACTOR indicates how many adjacent index entries do not refer to the same data block.
- AVG_LEAF_BLOCKS_PER_KEY is the average number of leaf blocks that store a single key.
- AVG_DATA_BLOCKS_PER_KEY is the average number of data blocks in the table referenced by a single key.
Structure of B⁺-Tree Indexes

Root Block

<28 71 ≥71

Branch Blocks

<8 19 ≥19

Leaf Blocks

6 rowid 6 rowid
8 rowid 8 rowid 10 rowid
19 rowid 19 rowid
28 rowid 28 rowid 40 rowid
41 rowid 42 rowid 43 rowid
54 rowid 56 rowid 58 rowid
71 rowid 71 rowid 71 rowid
73 rowid 75 rowid 89 rowid
Index Statistics – Clustering Factor

Index Block

<table>
<thead>
<tr>
<th>Name</th>
<th>Index Block Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>AAAEDzAABAAAKQhAAK</td>
</tr>
<tr>
<td>ALLEN</td>
<td>AAAEDzAABAAAKQhAAB</td>
</tr>
<tr>
<td>BLAKE</td>
<td>AAAEDzAABAAAKQhAAF</td>
</tr>
<tr>
<td>CLARK</td>
<td>AAAEDzAABAAAKQhAAG</td>
</tr>
<tr>
<td>FORD</td>
<td>AAAEDzAABAAAKQhAAM</td>
</tr>
<tr>
<td>JAMES</td>
<td>AAAEDzAABAAAKQhAAL</td>
</tr>
<tr>
<td>JONES</td>
<td>AAAEDzAABAAAKQhAAD</td>
</tr>
<tr>
<td>KING</td>
<td>AAAEDzAABAAAKQhAAI</td>
</tr>
<tr>
<td>MARTIN</td>
<td>AAAEDzAABAAAKQhAEE</td>
</tr>
<tr>
<td>MILLER</td>
<td>AAAEDzAABAAAKQhAAN</td>
</tr>
<tr>
<td>SCOTT</td>
<td>AAAEDzAABAAAKQhAAN</td>
</tr>
<tr>
<td>SMITH</td>
<td>AAAEDzAABAAAKQhAAA</td>
</tr>
</tbody>
</table>

Data Blocks

- ALLEN
- ADAMS
- FORD
- MILLER
- CLARK
- KING
- MARTIN
- BLAKE
- SMITH
- SCOTT
- JAMES
- JONES
Statistics for Partitioned Objects

- For partitioned objects, the database engine is able to handle all object statistics discussed in the previous sections at the object-level as well as at the partition and subpartition levels.

- The query optimizer uses the partition and subpartition statistics only when, during the parse phase, it can determine that a specific partition or subpartition is accessed.
Agenda

1. What Object Statistics Are Available?
2. Managing Object Statistics
3. Strategies for Keeping Object Statistics Up-to-Date
ANALYZE or DBMS_STATS?

- It used to be that object statistics were gathered with ANALYZE.
- This is no longer the case.
- As of 9.0 ANALYZE is available for purposes of backward compatibility only.
- It is recommended that you use DBMS_STATS.
Features Provided by the DBMS_STATS Package (1)

- **Data Dictionary**
  - Export
  - Import
  - Lock/Unlock
  - Restore

- **User-Defined Backup Table**
  - Export
  - Import

- **Copy via Data Movement Utility**

- **Gather/Delete/Set**

- **Get**
  - Data Dictionary
  - User-Defined Backup Table
### Features Provided by the DBMS_STATS Package (2)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Database</th>
<th>Dictionary</th>
<th>Schema</th>
<th>Table*</th>
<th>Index*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather/delete</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Lock/unlock</td>
<td></td>
<td></td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Restore</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Export/import</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Get/set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* It is possible to limit the processing to a single partition
Gathering Object Statistics with DBMS_STATS

- GATHER_DATABASE_STATS gathers object statistics for a whole database.
- GATHER_DICTIONARY_STATS gathers object statistics for the data dictionary.
- GATHER_FIXED_OBJECTS_STATS gathers object statistics for particular objects called fixed tables that are contained in the data dictionary.
- GATHER_SCHEMA_STATS gathers object statistics for a whole schema.
- GATHER_TABLE_STATS gathers object statistics for one table and, optionally, for its indexes.
- GATHER_INDEX_STATS gathers object statistics for one index.
## DBMS_STATS Parameters – Target Objects

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>D</th>
<th>S</th>
<th>T</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>ownname</td>
<td>Schema of the segment</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tabname</td>
<td>Name of the table</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>indname</td>
<td>Name of the index</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>partname</td>
<td>Name of the partition</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>granularity</td>
<td>Specifies granularity for partitioned segments</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cascade</td>
<td>Specifies if index statistics are gathered or not</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gather_sys</td>
<td>Specifies if tables owned by SYS are processed</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gather_temp</td>
<td>Specifies if temporary tables are processed</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>options</td>
<td>Specifies for which objects the statistics have to be gathered</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>objlist</td>
<td>List of objects for which the statistics have been gathered</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>obj_filter_list</td>
<td>Restrict the gathering of statistics</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DBMS_STATS Parameters – GRANULARITY

- This parameter specifies at which level statistics for partitioned objects are processed.

- AUTO
  - Object and partition statistics are gathered. Subpartition statistics are gathered only if the table is subpartitioned by list or range.

- GLOBAL

- PARTITION

- SUBPARTITION

- GLOBAL AND PARTITIONING (called DEFAULT up to 9.2)

- APPROX_GLOBAL AND PARTITION (as of 10.2.0.4)
  - Global statistics are aggregated from partition level statistics.
  - Made obsolete by incremental statistics as of 11.1.

- ALL
DBMS_STATS Parameters – OPTIONS (1)

- This parameter specifies which objects are processed.

- GATHER
  - All objects are processed.

- GATHER AUTO
  - Lets the procedure determine not only which object are to be processed but also how they are processed.
  - All parameters except OWNNAME, OBJLIST, STATTAB, STATID and STATOWN are ignored.

- GATHER STALE
  - Only objects having stale object statistics are processed.
  - Objects without object statistics are not considered stale.

- GATHER EMPTY
  - Only objects without object statistics are processed.
DBMS_STATS Parameters – OPTIONS (2)

- To recognize whether object statistics are stale, the database engine counts, for each object, the number of rows modified.

```
SQL> SELECT inserts, updates, deletes, truncated
2   FROM user_tab_modifications
3   WHERE table_name = 'T';
```

<table>
<thead>
<tr>
<th>INSERTS</th>
<th>UPDATES</th>
<th>DELETES</th>
<th>TRUNCATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>775</td>
<td>16636</td>
<td>66</td>
<td>NO</td>
</tr>
</tbody>
</table>

- Enabled by default when STATISTICS_LEVEL is set to TYPICAL.
- Object statistics are considered stale if at least 10 percent of the rows have been modified.
  - As of 11.1 the percentage can be modified.
DBMS_STATS Parameters – OBJ_FILTER_LIST

- As of 11.1 this parameter specifies to gather statistics only for objects fulfilling at least one of the filters passed as a parameter.

```sql
DECLARE
  l_filter dbms_stats.objecttab := dbms_stats.objecttab();
BEGIN
  l_filter.extend(2);

  l_filter(1).ownname := 'HR';

  l_filter(2).ownname := 'SH';
  l_filter(2).objname := 'C%';

  dbms_stats.gather_database_stats(
    obj_filter_list => l_filter,
    options => 'gather'
  );
END;
```
## DBMS_STATS Parameters – Gathering Options

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>D</th>
<th>S</th>
<th>T</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>estimate_percent</td>
<td>Percentage of rows used to estimate the statistics, NULL means 100%</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>block_sample</td>
<td>Specifies if block sampling is used</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>method_opt</td>
<td>Specifies which/how columns are processed</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>degree</td>
<td>Degree of parallelism</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>no_invalidate</td>
<td>Specifies if dependent cursors are invalidated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
DBMS_STATS Parameters – ESTIMATE_PERCENT

- This parameter specifies the minimal value used for the sampling.

```sql
SQL> execute dbms_stats.gather_schema_stats(
>    ownname     => user,
>    estimate_percent => 0.5)

SQL> SELECT table_name, sample_size, num_rows,
>        round(sample_size/num_rows*100,1) AS "%"
> FROM user_tables;

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>SAMPLE_SIZE</th>
<th>NUM_ROWS</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANNELS</td>
<td>5</td>
<td>5</td>
<td>100.0</td>
</tr>
<tr>
<td>COSTS</td>
<td>5410</td>
<td>81799</td>
<td>6.6</td>
</tr>
<tr>
<td>COUNTRIES</td>
<td>23</td>
<td>23</td>
<td>100.0</td>
</tr>
<tr>
<td>SALES</td>
<td>4602</td>
<td>920400</td>
<td>0.5</td>
</tr>
<tr>
<td>SUPPLEMENTARY_DEMOGRAPHICS</td>
<td>3739</td>
<td>4487</td>
<td>83.3</td>
</tr>
</tbody>
</table>
```

- As of 11.1, it is recommended to use DBMS_STATS.AUTO_SAMPLE_SIZE.
DBMS_STATS Parameters – METHOD_OPT (1)

- This parameter specifies not only whether histograms are gathered but also the maximum number of buckets that should be used in case a histogram is created.

- This parameter could also be used to completely disable the gathering of column statistics.

- Histograms are essential for all columns referenced in WHERE clauses that contain skewed data.
  - Even for non-indexed columns!
  - To get the most out of them, bind variables must not be used.
DBMS_STATS Parameters – METHOD_OPT (2)

- Gathering column statistics without histograms → specify NULL
- Gathering column statistics and histograms for all columns.

Gathering column statistics and histograms only for a subset of columns or for all columns, but with different values for the SIZE_CLAUSE parameter.
DBMS_STATS Parameters – METHOD_OPT (3)

- Values accepted by the SIZE_CLAUSE parameter:
  - SIZE 1..254
    - The value specifies the maximum number of buckets.
    - If size 1 is specified, no histograms are created.
  - SIZE SKEWONLY
    - Histograms are gathered only for columns with skewed data.
    - The number of buckets is determined automatically.
  - SIZE AUTO
    - Histograms are gathered only for columns with skewed data, such as skewonly, and, in addition, that have been referenced in WHERE clauses.
    - The number of buckets is determined automatically.
  - SIZE REPEAT
    - Refreshes available histograms.
### DBMS_STATS Parameters – Backup Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>D</th>
<th>S</th>
<th>T</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>stattab</td>
<td>Name of the user-defined statistic table</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>statid</td>
<td>Identifier to be associated with the statistics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>statown</td>
<td>Schema containing <em>stattab</em></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
DBMS_STATS Parameters – Default Values

- The default values for central parameters are not hard-coded but stored in the data dictionary:
  - CASCADE
  - ESTIMATE_PERCENT
  - DEGREE
  - METHOD_OPT
  - NO_INVALIDATE
  - GRANULARITY

- They can be displayed and modified with the following functions and procedures of the DBMS_STATS package:
  - 10g: GET_PARAM and SET_PARAM
  - 11g/12c: GET_PREFS, SET_TABLE_PREFS, SET_SCHEMA_PREFS, SET_DATABASE_PREFS and SET_GLOBAL_PREFS
Incremental Statistics

- Up to 10.2 global statistics are gathered by accessing all underlying partitions.

- As of 11.1 incremental statistics can be enabled:

  ```sql
  dbms_stats.set_table_prefs(ownname => user,
   tabname => 't',
   pname => 'incremental',
   pvalue => 'true')
  ```

- When enabled, a synopsis summarizes what is stored in each partition and, therefore, not all partitions have to be accessed.

- Synopses have to be created to take advantage from them!
  - Enabling incremental statistics is not enough.
  - Synopses are created when statistics are gathered.
Statistics History

- Whenever statistics are gathered, instead of simply overwriting the current statistics, a history of all changes is kept.
  - USER_TAB_STATS_HISTORY

- Statistics are kept in the history for an interval specified by a retention period.

```
SQL> SELECT dbms_stats.get_stats_history_retention() retention 
      2   FROM dual;

RETENTION
----------
     31
```

- The retention period can be changed.

```
dbms_stats.alter_stats_history_retention(retention => 14)
```

- Statistics can be restored from the history.
Pending Statistics

- Usually, as soon as the gathering of statistics is finished, the object statistics are published (that is, made available) to the query optimizer.
- As of 11.1, it is possible to separate gathering of statistics from publishing.

```sql
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 't',
                           pname   => 'publish',
                           pvalue  => 'false')
```

- Unpublished statistics are called *pending statistics*.
- The query optimizer uses pending statistics only when OPTIMIZER_USE_PENDING_STATISTICS is set to TRUE.
- Pending statistics can be published, deleted and exported.
Locking Statistics

- If you want to guarantee that statistics for a given table or schema are not changed, you can lock them.
  - (UN_)LOCK_*_STATS procedures are available in DBMS_STATS.

- When statistics on a table are locked, all object statistics are locked.

- DBMS_STATS
  - All procedures that modify statistics of an individual table will raise an error (ORA-20005).
  - All procedures that operate on multiple tables will skip locked tables.
  - Some procedures have a parameter named FORCE to override the lock.

- ALTER INDEX REBUILD
  - Without COMPUTE STATISTICS statistics are not gathered.
  - With COMPUTE STATISTICS an error (ORA-38029) is raised.

- ANALYZE raises an error (ORA-38029).
Comparing Statistics

- When (pending) object statistics are gathered it is sometimes useful to compare them with the previous ones.
- Writing SQL statements that compare different sets of object statistics is bothersome
- DBMS_STATS provides the following functions:
  - DIFF_TABLE_STATS_IN_HISTORY (as of 10.2.0.4)
  - DIFF_TABLE_STATS_IN_STATTABLE (as of 10.2.0.4)
  - DIFF_TABLE_STATS_IN_PENDING (as of 11.1)

```sql
dbms_stats.diff_table_stats_in_pending(
    ownname => 'user',
    tabname => 'T',
    time_stamp => NULL,
    pctthreshold => 10)
```
Comparing Statistics – Sample Output

STATISTICS DIFFERENCE REPORT FOR:

TABLE : T
OWNER : OPS$CHA
SOURCE A : Statistics as of 09-NOV-07 01.40.11.794565 AM +01:00
SOURCE B : Current Statistics in dictionary
PCTTHRESHOLD : 10

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
NO DIFFERENCE IN TABLE / (SUB)PARTITION STATISTICS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

COLUMN STATISTICS DIFFERENCE:

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>SRC</th>
<th>NDV</th>
<th>DENSITY</th>
<th>HIST</th>
<th>NULLS</th>
<th>LEN</th>
<th>MIN</th>
<th>MAX</th>
<th>SAMPSIZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>NO</td>
<td>0</td>
<td>2</td>
<td>80</td>
<td>80</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>.0005</td>
<td>YES</td>
<td>0</td>
<td>2</td>
<td>80</td>
<td>80</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
NO DIFFERENCE IN INDEX / (SUB)PARTITION STATISTICS
Agenda

1. What Object Statistics Are Available?
2. Managing Object Statistics
3. Strategies for Keeping Object Statistics Up-to-Date
Strategies for Keeping Object Statistics Up-to-Date

- The query optimizer needs object statistics that describe the data stored in the database.  
  ➔ When data changes, object statistics should change as well.
- It makes no sense to gather object statistics on data that never changes.  
  ➔ Take advantage of the feature that logs the number of modifications.
- The frequency of gathering depends on the application and the business needs.
- You should lock the statistics that do not have to be gathered.
- In case of jobs that load or modify lots of data, do not wait for a scheduled gathering of object statistics.
- You should take advantage of the default gathering job as much as possible.
Scheduling Statistics Gathering

- By default statistics gathering is automatically scheduled.
  - 10g: GATHER_STATS_JOB job
  - 11g/12c: the gathering of object statistics is integrated into the automated maintenance tasks
- The default configuration should be customized to meet requirements.
JOB_QUEUE_PROCESSES

- Up to 11.1 JOB_QUEUE_PROCESSES has no impact on jobs scheduled through DBMS_SCHEDULER.
- As of 11.2 this is no longer the case, e.g. setting the parameter to 0 disables the automatic statistic gathering.
- Documentation bug 10008042
  - Fixed in the current version
Core Messages

- It is recommended that you use DBMS_STATS to gather object statistics.
- The query optimizer needs object statistics to successfully execute its task.
THANK YOU.
Troubleshooting Oracle Performance

Configuring the Query Optimizer
Agenda

1. To Configure or Not to Configure
2. Configuration Roadmap
3. Query Optimizer Parameters
4. PGA Management Parameters
Correctly Configuring the Query Optimizer (1)

- The query optimizer works well, i.e. it generates good execution plans for most SQL statements.
- This is only true when
  - It’s correctly configured, and
  - the database has been designed to take advantage of all its features.
- Correctly configuring the query optimizer isn’t an easy job.

→ This is no good reason for not doing it however!
Correctly Configuring the Query Optimizer (2)

- There is no single configuration that is good for every system.
  - Each application has its own requirements.
  - Each system has its own characteristics.

- Therefore, I’m not able to provide you with the “magic configuration”...

- I can only show you how I proceed to do such a configuration.
Agenda

1. To Configure or Not to Configure
2. Configuration Roadmap
3. Query Optimizer Parameters
4. PGA Management Parameters
Configuration = Initialization Parameters + Statistics

- **System and Object Statistics**
  - **Basic**
    - OPTIMIZER_MODE
    - DB_FILE_MULTIBLOCK_READ_COUNT
    - OPTIMIZER_FEATURES_ENABLE
    - OPTIMIZER_ADAPTIVE_FEATURES
    - OPTIMIZER_ADAPTIVE_REPORTING_ONLY
    - OPTIMIZER_DYNAMIC_SAMPLING
    - OPTIMIZER_INDEX_COST_ADJ
    - OPTIMIZER_INDEX_CACHING
  - **Query transformation**
    - QUERY_REWRITE_ENABLED
    - QUERY_REWRITE_INTEGRITY
    - STAR_TRANSFORMATION_ENABLED
    - OPTIMIZER_SECURE_VIEW_MERGING

- **Memory**
  - WORKAREA_SIZE_POLICY
  - PGA_AGGREGATE_TARGET
  - PGA_AGGREGATE_LIMIT
  - HASH_AREA_SIZE
  - SORT_AREA_SIZE
  - SORT_AREA RETAINED SIZE
  - BITMAP_MERGE_AREA_SIZE

- **Miscellaneous**
  - CURSOR_SHARING
  - SKIP_UNUSABLE_INDEXES
  - PARALLEL_*
  - Many undocumented parameters...
Configuration Roadmap

1. optimizer_mode
db_file_multiblock_read_count

2. optimizer_features_enable
   optimizer_secure_view_merging
   optimizer_adaptive_features
   optimizer_adaptive_reporting_only
   query_rewrite_enabled
   query_rewrite_integrity
   star_transformation_enabled

3. gather system statistics
gather object statistics

4. workarea_size_policy = auto
5. workarea_size_policy = manual

6. hash_area_size
   sort_area_size
   sort_area_retained_size
   bitmap_merge_area_size

7. pga_aggregate_target
   pga_aggregate_limit

8. most of the execution plans are efficient
   Yes

9. optimizer_index_caching
   optimizer_index_cost_adj
   optimizer_dynamic_sampling
   adjust histograms
   define extended statistics

No
Agenda

1. To Configure or Not to Configure
2. Configuration Roadmap
3. Query Optimizer Parameters
4. PGA Management Parameters
Set the Right Parameter! (1)

- Each parameter has been introduced for a specific reason!
- Understanding how it changes the behavior of the query optimizer is essential.
- Don’t tweak the configuration randomly.
- Instead:
  - Understand the current situation.
  - Define the goal to be achieved.
  - Find out which parameter should be changed to achieve that goal.
Choose the Right Tool!

Troubleshooting Oracle Performance - Configuring the Query Optimizer
2014-04-28
Set the Right Parameter! (2)
OPTIMIZER_MODE

- This is the most important parameter. Too often it is not set.
  - The default value is ALL_ROWS.

- If fast delivery of the last row is important, ALL_ROWS should be used.
  - Typical example: reporting systems and data warehouses

- If fast delivery of the first row(s) is important, FIRST_ROWS_n should be used.
  - Where “n” is [ 1 | 10 | 100 | 1000 ] row(s)
  - Use it only when few rows of a large result set are retrieved.
  - Typical example: OLTP systems
OPTIMIZER_FEATURES_ENABLE

- Each database version introduces new features in the query optimizer.
- OPTIMIZER_FEATURES_ENABLE can be used to set which “version” of the query optimizer should be used.
  - Valid values are database versions (e.g. 10.2.0.5, 11.1.0.7, 11.2.0.3, ...).
  - V$PARAMETER_VALID_VALUES
- The default value is the current database version.
- Not all new features are enabled/disabled by this parameter!
- It could be useful to set it when an application is upgraded to a new database version, otherwise leave it at the default value.
DB_FILE_MULTIBLOCK_READ_COUNT (1)

- DB_FILE_MULTIBLOCK_READ_COUNT specifies the maximum number of blocks that the database engine reads during a multiblock operation.

- Three common situations lead to multiblock reads that are smaller than the specified value:
  - Segment headers and other blocks containing only segment metadata are read with single-block reads.
  - Physical reads (except for a special case related to direct reads performed against a tablespace using ASSM) never span several extents.
  - Blocks already in the buffer cache, except for direct reads, aren’t reread from the disk I/O subsystem.
DB_FILE_MULTIBLOCK_READ_COUNT (2)

Extent 1

<table>
<thead>
<tr>
<th>Extent 1</th>
<th>2</th>
<th>3</th>
<th>Cached</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cached</td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>18</td>
<td></td>
<td></td>
<td>20</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extent 2

<table>
<thead>
<tr>
<th>Extent 2</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Multiblock reads are a performance feature. The parameter should be set to achieve the best performance.
- A full scan with different values assists in finding the “best” value.
DB_FILE_MULTIBLOCK_READ_COUNT (4)

- When workload system statistics are not used, this parameter has a direct impact on the cost of multi-block operations.

- I/O costs with workload system statistics

\[ io\_cost \approx \frac{blocks \cdot mreadtim}{mbrc \cdot sreadtim} \]

- I/O costs with noworkload system statistics
  - MBRC: either DB_FILE_MULTIBLOCK_READ_COUNT (if explicitly set) or 8
  - SREADTIM and MREADTIM: computed as described in Chapter 7
It’s possible to let the database engine to automatically “tune” the value of DB_FILE_MULTIBLOCK_READ_COUNT
- To use this function simply don’t set it.

Unfortunately the value is adjusted with a formula like the following one:

\[
\text{DB_FILE_MULTIBLOCK_READ_COUNT} \approx \min \left( \frac{1048576}{\text{db_block_size}} \cdot \frac{\text{db_cache_size}}{\text{sessions}} \cdot \frac{\text{ead_count}}{\text{ltiblock_r}} \right)
\]

- It’s usually better to set it explicitly.
OPTIMIZER_DYNAMIC_SAMPLING (1)

- The query optimizer used to base its estimations on statistics found in the data dictionary only.
- With dynamic sampling (dynamic statistics in 12.1) some statistics can be gathered during the parse phase as well.
- The statistics gathered by dynamic sampling aren’t stored in the data dictionary.
- OPTIMIZER_DYNAMIC_SAMPLING specifies how and when dynamic sampling is used.
- As of 11.2, for parallel statements, the query optimizer automatically decides whether to use dynamic sampling and which level to use.
## OPTIMIZER_DYNAMIC_SAMPLING (2)

<table>
<thead>
<tr>
<th>Level</th>
<th>When Is Dynamic Sampling Used?</th>
<th>#Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For all tables without object statistics, if at least one table:</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>» Is part of a join (also subquery or non-mergeable view)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>» Has no index</td>
<td></td>
</tr>
<tr>
<td></td>
<td>» Has more blocks than the number of blocks used for the sampling</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>For all tables without object statistics.</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>For all tables which fulfill the level-2 criterion or for which a guess is used to estimate the selectivity of a predicate.</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>7</td>
<td>For all tables which fulfill level-3 criteria or having two or more columns referenced in the WHERE clause.</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>4096</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>All</td>
</tr>
<tr>
<td>11</td>
<td>The query optimizer decides when and how to use it (12.1)</td>
<td>Auto</td>
</tr>
</tbody>
</table>
OPTIMIZER_DYNAMIC_SAMPLING (3)

- Default values:
  - If OPTIMIZER_FEATURES_ENABLE ≥ 10.0.0: 2
  - If OPTIMIZER_FEATURES_ENABLE = 9.2.0: 1
  - If OPTIMIZER_FEATURES_ENABLE ≤ 9.0.1: 0

- Level 1 and 2 are not very useful, in fact tables and indexes should have up-to-date object statistics!
  - An exception is when temporary tables are used, in fact, usually, no statistics are available for them.

- Level 3 and up are useful for improving selectivity estimations of predicates.
  - If the query optimizer isn’t able to do correct estimations, start by setting it to 4 or higher.
  - It’s also possible to do similar corrections with SQL profiles.
I/O Cost Index Unique Scans

\[ \text{io\_cost} \approx \text{blevel} + 2 \]
I/O Cost Index Range Scans

\[ \text{io\_cost} \approx b\text{level} + \left( \text{leaf\_block} \cdot \text{s} + \text{clustering\_factor} \right) \cdot \text{selectivity} \]
OPTIMIZER_INDEX_COST_ADJ (OICA)

- This parameter is used to adjust the cost of table accesses through index scans.
- Range of values: 1..10000 (default 100)
- Correction for an index range scan:

\[
io_cost \approx (blevel + (leaf\_block \cdot s + clustering\_factor) \cdot selectivity \cdot y) \cdot \frac{OICA}{100}
\]

- Correction for an index unique scan:

\[
io_cost \approx (blevel + 2) \cdot \frac{OICA}{100}
\]

- This parameter flattens costs and makes the clustering factor less significant. Small values should be carefully specified!
OPTIMIZER_INDEX_CACHING (OIC)

- This parameter is used to specify, in percent, the expected amount of index blocks cached in the buffer cache during a nested loop join or an indexed inlist iterator.

- Range of values: 0..100 (default 0)

- Correction for an index range scan:

\[
\text{io}_\text{cost} \approx (\text{blevel} + \text{leaf}\_\text{block} \cdot s \cdot \text{selectivit} \_y) \cdot \left(1 - \frac{\text{OIC}}{100}\right) + \text{clustering}\_\text{factor} \cdot \text{selectivit} \_y
\]

- Correction for index unique scan:

\[
\text{io}_\text{cost} \approx \text{blevel} \cdot \left(1 - \frac{\text{OIC}}{100}\right) + 2
\]
OPTIMIZER_SECURE_VIEW_MERGING

- This parameter controls query transformations that might lead to security issues (e.g. view merging and predicate move around).
  - FALSE allows the query optimizer to always do query transformations.
  - TRUE allows the query optimizer to do query transformations only when doing so won’t lead to security issues.

- The default is TRUE.

- If views nor VPD are not used for security purposes, it’s better to set this parameter to FALSE.
Agenda

1. To Configure or Not to Configure
2. Configuration Roadmap
3. Query Optimizer Parameters
4. PGA Management Parameters
PGA Management

- It’s possible to choose between two methods to manage the PGA:
  - Manual: the DBA has full control over the size of the PGA.
  - Automatic: the database engine controls the size of the PGA.
- It’s recommended to use automatic PGA management.
- Only in rare cases does manual fine-tuning provide better results than automatic PGA management.
Automatic PGA Management

- To use automatic PGA management WORKAREA_SIZE_POLICY must be set to AUTO.

- PGA_AGGREGATE_TARGET specifies the total amount of memory available for all PGAs allocated by one instance.
  - 10MB - 4TB

- The value specified with PGA_AGGREGATE_TARGET parameter could also be exceeded.
  - It’s a target, not a maximum!
  - This usually happens when a value which is too small is specified.
Automatic PGA Management – Correctly Sized (1GB)

- Maximum Instance PGA
- Average Session PGA
Automatic PGA Management – Incorrectly Sized (128MB)

The graph illustrates the relationship between the number of concurrent sessions and the PGA usage for maximum instance and average session. The x-axis represents the number of concurrent sessions, while the y-axis shows the PGA usage in MB. The graph shows a trend where PGA usage increases as the number of concurrent sessions grows, indicating potential performance issues with the current PGA size.
PGA_AGGREGATE_LIMIT (12.1)

- It sets a hard limit to the amount of PGA an instance can use.
- Its default value is set to the greater of the following values:
  - 2GB
  - Twice the value of PGA_AGGREGATE_TARGET
  - 3MB times the value of PROCESSES
- When the limit is reached, the database engine terminates calls or even kills sessions.
  - ORA-04036: PGA memory used by the instance exceeds PGA_AGGREGATE_LIMIT
- To disable the feature, set it to 0.
Manual PGA Management

- To use manual PGA management WORKAREA_SIZE_POLICY must be set to MANUAL.
- Then the following parameters should be set:
  - HASH_AREA_SIZE
  - SORT_AREA_SIZE
- It is practically impossible to give advice about their value, anyway, here some general rules.
  - Usually a minimal value of 512KB to 1MB should be used.
  - For small PGAs, to take advantage of hash join, the HASH_AREA_SIZE should be at least 3-4 times the SORT_AREA_SIZE.
- Each parameter specifies the maximum amount of memory that can be used by each process for a single work area.
Core Messages

- The query optimizer works well if it is correctly configured!
- Correctly configuring the query optimizer isn’t an easy job.
  ➔ This however is not a good reason for not doing it!
- Understanding how each parameter changes the behavior of the query optimizer is essential.
THANK YOU.
Troubleshooting Oracle Performance
Agenda

1. Obtaining Execution Plans
2. Interpreting Execution Plans
3. Recognizing Inefficient Execution Plans
Obtaining Execution Plans

- Oracle provides four methods to obtain an execution plan:
  - EXPLAIN PLAN statement
  - Dynamic performance views
  - Use Real-Time SQL Monitoring
  - Automatic Workload Repository
  - Tracing facilities

- All tools displaying execution plans take advantage of one of these methods.
EXPLAIN PLAN

- **statement** specifies which SQL statement the execution plan should be provided for.
- **id** specifies a name to distinguish between several execution plans in the plan table.
- **table** specifies the name of the plan table where the information about the execution plan is inserted.
  - The default value is PLAN_TABLE.
EXPLAIN PLAN – Querying the Plan Table

- The execution plan can be obtained by running queries against the plan table.
- There is an easier and much better way to do it...

```sql
SQL> EXPLAIN PLAN FOR SELECT * FROM emp WHERE deptno = 10 ORDER BY ename;
SQL> SELECT * FROM table(dbms_xplan.display);
```

Plan hash value: 150391907

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>5</td>
<td>185</td>
<td>4 (25)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>SORT ORDER BY</td>
<td></td>
<td>5</td>
<td>185</td>
<td>4 (25)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>5</td>
<td>185</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

2 - filter("DEPTNO"=10)
EXPLAIN PLAN – Bind Variables Trap

- Using bind variables with EXPLAIN PLAN has two problems:
  - By default, bind variables are declared as VARCHAR2.
  - No bind variable peeking is used.

- Since there is no general solution for these problems, whenever bind variables are involved, the output generated by EXPLAIN PLAN is unreliable.
Dynamic Performance Views

- Four views show information about the cursors present in the library cache:
  - V$SQL_PLAN
  - V$SQL_PLAN_STATISTICS
  - V$SQL_WORKAREA
  - V$SQL_PLAN_STATISTICS_ALL

- The cursors in these views are identified by either:
  - ADDRESS, HASH_VALUE and CHILD_NUMBER
  - SQL_ID and CHILD_NUMBER (does not work in all situations)

- Runtime statistics are only available if STATISTICS_LEVEL is set to ALL or the GATHER_PLAN_STATISTICS hint is specified.
Dynamic Performance Views – Identifying Child Cursor

- Based on a session currently connected

```
SQL> SELECT status, sql_id, sql_child_number
2  FROM v$session
3  WHERE username = 'CURTIS';
```

<table>
<thead>
<tr>
<th>STATUS</th>
<th>SQL_ID</th>
<th>SQL_CHILD_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE</td>
<td>1hqjydsjbjvmwq</td>
<td>0</td>
</tr>
</tbody>
</table>

- Based on the text of the SQL statement

```
SQL> SELECT sql_id, child_number, sql_text
2  FROM v$sql
3  WHERE sql_text LIKE '%online discount%'
4  AND sql_text NOT LIKE '%v$sql%';
```

<table>
<thead>
<tr>
<th>SQL_ID</th>
<th>CHILD_NUMBER</th>
<th>SQL_TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1hqjydsjbjvmwq</td>
<td>0</td>
<td>SELECT SUM(AMOUNT_SOLD) FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID = P.PROMO_ID AND PROMO_SUBCATEGORY = 'online_discount'</td>
</tr>
</tbody>
</table>
SQL> SELECT * FROM table(dbms_xplan.display_cursor('1hqjydsjbmwq',0));

SQL_ID 1hqjydsjbmwq, child number 0

SELECT SUM(AMOUNT_SOLD) FROM SALES S, PROMOTIONS P WHERE S.PROMO_ID = P.PROMO_ID AND PROMO_SUBCATEGORY = 'online discount'

Plan hash value: 265338492

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td>517 (100)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>HASH JOIN</td>
<td></td>
<td>913K</td>
<td>26M</td>
<td>517 (4)</td>
<td>00:00:07</td>
</tr>
<tr>
<td>* 3</td>
<td>TABLE ACCESS FULL</td>
<td>PROMOTIONS</td>
<td>23</td>
<td>483</td>
<td>17 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>4</td>
<td>PARTITION RANGE ALL</td>
<td></td>
<td>918K</td>
<td>8075K</td>
<td>494 (3)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS FULL</td>
<td>SALES</td>
<td>918K</td>
<td>8075K</td>
<td>494 (3)</td>
<td>00:00:06</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

2 - access("S"."PROMO_ID"="P"."PROMO_ID")
3 - filter("PROMO_SUBCATEGORY"='online discount')
Automatic Workload Repository (AWR)

- When a snapshot is taken, AWR is able to collect execution plans.
- The execution plans stored in AWR are available through:
  - DBA_HIST_SQL_PLAN
  - DBMS_XPLAN.DISPLAY_AWR

```
SELECT * FROM table(dbms_xplan.display_awr('1hqjydsjbvmwq'));
```

- $ORACLE_HOME/rdbms/admin/awrsqrpt.sql
Tracing Facilities

- Several tracing facilities provide information about execution plans.
- Except for SQL trace, all of them are not officially supported.
- In any case, the following may turn out to be useful...
  - Event 10053
  - Event 10132
**DBMS_XPLAN Package**

- The functions of DBMS_XPLAN have several parameters aimed to customize the output.

- The most important, FORMAT, accept the following values:
  - BASIC: displays only the minimum amount of information.
  - TYPICAL: displays the most relevant information.
  - SERIAL: like typical, except that information about parallel processing is not displayed.
  - ALL: displays all available information except the outline.
  - ADVANCED: displays all available information.
Agenda

1. Obtaining Execution Plans
2. Interpreting Execution Plans
3. Recognizing Inefficient Execution Plans
Caution

- Parallel processing makes the interpretation of execution plans more difficult.
- The reason is quite simple: several operations run concurrently.
- This section, to keep the description as simple as possible, does not pretend to cover parallel processing.
Parent-Child Relationship

- An execution plan is a tree.
- Each node in the tree is an operation.
- Between operations (nodes) there is a parent-child relationship.
- When an execution plan is displayed in a textual form, the rules governing the parent-child relationship are the following:
  - A parent has one or multiple children.
  - A child has a single parent.
  - The only operation without a parent is the root of the tree.
  - The children are indented right, with respect to their parent.
  - A parent is placed before its children (ID of the parent < ID of the children).
Parent-Child Relationship – Example

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UPDATE</td>
</tr>
<tr>
<td>2</td>
<td>NESTED LOOPS</td>
</tr>
<tr>
<td>* 3</td>
<td>TABLE ACCESS FULL</td>
</tr>
<tr>
<td>* 4</td>
<td>INDEX UNIQUE SCAN</td>
</tr>
<tr>
<td>5</td>
<td>SORT AGGREGATE</td>
</tr>
<tr>
<td>6</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
</tr>
<tr>
<td>* 7</td>
<td>INDEX RANGE SCAN</td>
</tr>
<tr>
<td>8</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
</tr>
<tr>
<td>* 9</td>
<td>INDEX UNIQUE SCAN</td>
</tr>
</tbody>
</table>

__Execution Plan diagram__

```
1 5 3
3
4
6
2
8
7
9
```

__Execution Plans SQL file__

```
execution_plans.sql
```
Order of Execution

- Parent operations, to fulfill their task, require data that is provided by their child operations.

- Even though the execution starts at the root of the tree, the first operation being fully executed is one that has no child and, therefore, is a leaf of the tree.

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SORT ORDER BY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
</tr>
<tr>
<td>3</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_PK</td>
</tr>
</tbody>
</table>
Order of Execution – Basic Rules

- Parent operations call child operations.
- Child operations are entirely executed before their parent operations.
- Child operations pass data to their parent operations.
Types of Operations

- The number of possible operations is high (more than 200).

- To fully understand an execution plan, you should know what each operation it is made of does.

- For our purpose of walking through an execution plan, you need to consider only four major types of operations:
  - Stand-alone operations
  - Iterative Operations
  - Unrelated-combine operations
  - Related-combine operations
Stand-Alone operations

- All operations having at most one child are *stand-alone operations*.
- Most operations are of this type.
- The rules governing the working of these operations are the following:
  - The basic rules mentioned before.
  - A child operation is executed at most once.
### Stand-Alone Operations – Example

```sql
SELECT deptno, count(*)
FROM emp
WHERE job = 'CLERK' AND sal < 1200
GROUP BY deptno
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HASH GROUP BY</td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>* 3</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_JOB_I</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

2 - filter("SAL"<1200)
3 - access("JOB"='CLERK')
Stand-Alone Operations – COUNT STOPKEY

- COUNT STOPKEY is commonly used to execute the top-n queries.
- Its aim is to stop the processing as soon as the required number of rows has been returned to the caller.

```
SELECT *
FROM emp
WHERE rownum <= 10
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 1</td>
<td>COUNT STOPKEY</td>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

1 - filter(ROWNUM<=10)
Stand-Alone Operations – FILTER

- FILTER applies a filter when its child passes data to it. In addition, it could decide to completely avoid the execution of a child and all the dependent operations as well.

```
SELECT *
FROM emp
WHERE job = 'CLERK' AND 1 = 2
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 1</td>
<td>FILTER</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>* 3</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_JOB_I</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1 - filter(NULL IS NOT NULL)
3 - access("JOB"='CLERK')
Iterative Operations

- All operations having at most one child that can be executed more than one time are *iterative operations*.
- They can be considered like a loop in an execution plan.
- The following operations are of this type:
  - INLIST ITERATOR
  - Most of the operations having the PARTITION suffix, e.g. PARTITION RANGE ITERATOR
- The rules governing the working of these operations are the following:
  - The basic rules mentioned before.
  - A child operation may be executed several times or not executed at all.
Iterative Operations – Example

- INLIST ITERATOR is used to implement IN lists.
- The child of INLIST ITERATOR is executed one time for each distinct value in the IN list.

```
SELECT *
FROM emp
WHERE job IN ('CLERK', 'ANALYST')
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INLIST ITERATOR</td>
<td></td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>* 3</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_JOB_I</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

3 - access('"JOB"='ANALYST' OR "JOB"='CLERK')
Unrelated-Combine Operations – Definition

- All operations having multiple children that are independently executed are *unrelated-combine operations*.
- The following operations are of this type:
  - AND-EQUAL
  - BITMAP AND, BITMAP OR and BITMAP MINUS
  - CONCATENATION
  - CONNECT BY WITHOUT FILTERING
  - HASH JOIN
  - INTERSECTION
  - MERGE JOIN
  - MINUS
  - MULTI-TABLE INSERT
  - SQL MODEL
  - TEMP TABLE TRANSFORMATION
  - UNION-ALL
Unrelated-Combine Operations – Characteristics

- The rules governing the working of these operations are the following:
  - The basic rules mentioned before.
  - Children are executed sequentially.
  - Every child is executed at most once and independently from the others.
### Unrelated-Combine Operations – Example

```sql
SELECT ename FROM emp
UNION ALL
SELECT dname FROM dept
UNION ALL
SELECT '%' FROM dual
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UNION-ALL</td>
<td></td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>DEPT</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>FAST DUAL</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Related-Combine Operations – Definition

- All operations having multiple children where one of the children controls the execution of all other children are related-combine operations.

- The following operations are of this type:
  - NESTED LOOPS
  - UPDATE
  - FILTER
  - CONNECT BY WITH FILTERING
  - UNION ALL (RECURSIVE WITH)
  - BITMAP KEY ITERATION
Related-Combine Operations – Characteristics

- The rules governing the working of these operations are the following:
  - The basic rules mentioned before.
  - The child with the smallest id controls the execution of the other children.
  - Children are not executed sequentially. Instead, a kind of interleaving is performed.
  - Only the first child is executed at most once. All other children may be executed several times or not executed at all.
Related-Combine Operations – NESTED LOOPS

```sql
SELECT *
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND emp.comm IS NULL
AND dept.dname != 'SALES'
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>* 3</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPT</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>* 4</td>
<td>INDEX UNIQUE SCAN</td>
<td>DEPT_PK</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

2 - filter("EMP"."COMM" IS NULL)
3 - filter("DEPT"."DNAME"<>'SALES')
4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO" )
Related-Combine Operations – FILTER

```sql
SELECT *
FROM emp
WHERE NOT EXISTS (SELECT 0 FROM dept WHERE dept.dname = 'SALES'
    AND dept.deptno = emp.deptno)
AND NOT EXISTS (SELECT 0 FROM bonus WHERE bonus.ename = emp.ename);
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 1</td>
<td>FILTER</td>
<td></td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>* 3</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPT</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>* 4</td>
<td>INDEX UNIQUE SCAN</td>
<td>DEPT_PK</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>* 5</td>
<td>TABLE ACCESS FULL</td>
<td>BONUS</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

1 - filter( NOT EXISTS (SELECT 0 FROM "DEPT" "DEPT" WHERE 
    "DEPT"."DEPTNO"=:B1 AND "DEPT"."DNAME"='SALES')
AND NOT EXISTS (SELECT 0 FROM "BONUS" 
    "BONUS" WHERE "BONUS"."ENAME"=:B2))

3 - filter("DEPT"."DNAME"='SALES')
4 - access("DEPT"."DEPTNO"=:B1)
5 - filter("BONUS"."ENAME"=:B1)


**UPDATE emp e1**

```sql
SET sal = (SELECT avg(sal) FROM emp e2 WHERE e2.deptno = e1.deptno),
       comm = (SELECT avg(comm) FROM emp e3)
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UPDATE</td>
<td>EMP</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>* 4</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>14</td>
</tr>
</tbody>
</table>

4 - filter("E2"."DEPTNO"=:B1)
**Related-Combine Operations – CONNECT BY**

```sql
SELECT level, ename, prior ename AS manager
FROM emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 1</td>
<td>CONNECT BY WITH FILTERING</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>NESTED LOOPS</td>
<td></td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>CONNECT BY PUMP</td>
<td></td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>* 6</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_MGR_I</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>

1 - access("MGR"=PRIOR "EMPNO")
2 - filter("MGR" IS NULL)
6 - access("MGR"=PRIOR "EMPNO")
SELECT ename, (SELECT dname FROM dept WHERE dept.deptno = emp.deptno) FROM emp

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPT</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>*2</td>
<td>INDEX UNIQUE SCAN</td>
<td>DEPT_PK</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>14</td>
</tr>
</tbody>
</table>

2 - access("DEPT"."DEPTNO"=:B1)
Special Cases – Subquery in the WHERE Clause #1

```
SELECT deptno
FROM dept
WHERE deptno NOT IN (SELECT deptno FROM emp)
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>* 1</td>
<td>INDEX FULL SCAN</td>
<td>DEPT_PK</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

1 - filter( NOT EXISTS (SELECT 0 FROM "EMP" "EMP" WHERE LNNVL("DEPTNO"<>:B1))
2 - filter(LNNVL("DEPTNO"<>:B1))
Special Cases – Subquery in the WHERE Clause #2

```
SELECT *
FROM t1
WHERE n1 = 8 AND n2 IN (SELECT t2.n1
                         FROM t2, t3
                         WHERE t2.id = t3.id AND t3.n1 = 4);
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>T1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>* 2</td>
<td>INDEX RANGE SCAN</td>
<td>I1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>* 3</td>
<td>HASH JOIN</td>
<td></td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>* 4</td>
<td>TABLE ACCESS FULL</td>
<td>T3</td>
<td>13</td>
<td>1183</td>
</tr>
<tr>
<td>* 5</td>
<td>TABLE ACCESS FULL</td>
<td>T2</td>
<td>13</td>
<td>910</td>
</tr>
</tbody>
</table>
Divide and Conquer (1)

- Reading long execution plans is no different from reading short ones.
- All you need is to methodically apply the rules provided in the previous slides.
- With them, it does not matter how many lines an execution plan has. You simply proceed in the same way.
Divide and Conquer (2)

- To read an execution plan it is necessary to both decompose the execution plan into basic blocks and recognize the order of execution.
- Each combine operations (both related and unrelated) must be identified.
  → 3, 4, 5, 6, and 14
- For each child operation of each combine operation, a block is defined.
Divide and Conquer (3)
Agenda

1. Obtaining Execution Plans
2. Interpreting Execution Plans
3. Recognizing Inefficient Execution Plans
Wrong Estimations

- The query optimizer computes costs to decide which access paths, join orders and join methods should be used to get an efficient execution plan.
- If the computation of the cost is wrong, it is likely that the query optimizer picks out a sub-optimal execution plan.
- Instead of judging the cost directly, it is much easier to check the estimated number of rows returned by an operation.
  - E.g. with DBMS_XPLAN.DISPLAY_CURSOR
Wrong Estimations – Example

```sql
SQL> SELECT count(t2.col2) FROM t1 JOIN t2 USING (id) WHERE t1.col1 = 666;

SQL> SELECT *
        2   FROM table(dbms_xplan.display_cursor(NULL,NULL,'iostats last'));
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>E-Rows</th>
<th>A-Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>32</td>
<td>75808</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>T1</td>
<td>1</td>
<td>32</td>
<td>80016</td>
</tr>
<tr>
<td>* 4</td>
<td>INDEX RANGE SCAN</td>
<td>T1_COL1</td>
<td>1</td>
<td>32</td>
<td>80016</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>T2</td>
<td>80016</td>
<td>1</td>
<td>75808</td>
</tr>
<tr>
<td>* 6</td>
<td>INDEX UNIQUE SCAN</td>
<td>T2_PK</td>
<td>80016</td>
<td>1</td>
<td>75808</td>
</tr>
</tbody>
</table>

4 - access("T1"."COL1"=666)
6 - access("T1"."ID"="T2"."ID")
Core Messages

- DBMS_XPLAN is the tool of choice for extracting and formatting execution plans.
- Simple rules can be applied for interpreting and assessing execution plans.
THANK YOU.
Troubleshooting Oracle Performance

SQL Optimization Techniques
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
Choosing a SQL Optimization Technique

To choose a SQL optimization technique it is essential to answer three basic questions:

- Is the SQL statement known and static?
- Should the measures to be taken have an impact on a single SQL statement or on all SQL statements executed by a single session (or even from the whole system)?
- It is possible to change the SQL statement?
# SQL Optimization Techniques and Their Impact

<table>
<thead>
<tr>
<th>Technique</th>
<th>System</th>
<th>Session</th>
<th>Statement</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altering the access structures</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td>All releases</td>
</tr>
<tr>
<td>Altering the SQL statement</td>
<td></td>
<td></td>
<td>✔️*</td>
<td>All releases</td>
</tr>
<tr>
<td>Hints</td>
<td></td>
<td></td>
<td>✔️*</td>
<td>All releases</td>
</tr>
<tr>
<td>Altering the execution environment</td>
<td>✔️</td>
<td>✔️*</td>
<td></td>
<td>All releases</td>
</tr>
<tr>
<td>SQL profiles</td>
<td></td>
<td></td>
<td>✔️</td>
<td>All releases†</td>
</tr>
<tr>
<td>Stored outlines</td>
<td></td>
<td>✔️</td>
<td></td>
<td>All releases</td>
</tr>
<tr>
<td>SQL plan baselines</td>
<td></td>
<td>✔️</td>
<td></td>
<td>As of 11.1‡</td>
</tr>
</tbody>
</table>

* It is required to change the SQL statement to use this technique.

† The Tuning Pack and, therefore, Enterprise Edition are required.

‡ Enterprise Edition is required.
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
How It Works

- The first thing you have to do while questioning the performance of a SQL statement is to verify which access structures are in place.
  - What is the organization type of the tables involved?
  - Are materialized views containing the needed data available?
  - What indexes exist?
  - How are all these segments partitioned?

- Next you have to assess whether the available access structures are adequate to efficiently process the SQL statement you are optimizing.

- Part 4 will cover when and how the different access structures should be used in detail.
When to Use It

- Without the necessary access structures in place it may be impossible to tune a SQL statement.
- Therefore, this technique should be considered whenever the access structures can be changed.
- Unfortunately, this is not always possible...
Pitfalls and Fallacies

- Side effects should be carefully considered.
- Every altered access structure introduces both positive and negative consequences.
- It is best to determine whether the pros outweigh the cons before altering access structures.
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
How It Works (1)

- With SQL, frequently, it is possible to submit the very same request in many different ways...

1. ```sql
   SELECT deptno
   FROM dept
   WHERE deptno NOT IN (SELECT deptno FROM emp)
```

2. ```sql
   SELECT deptno
   FROM dept
   WHERE NOT EXISTS (SELECT 1
                     FROM emp
                     WHERE emp.deptno = dept.deptno)
```

3. ```sql
   SELECT deptno FROM dept
   MINUS
   SELECT deptno FROM emp
```

4. ```sql
   SELECT dept.deptno
   FROM dept, emp
   WHERE dept.deptno = emp.deptno(+) AND emp.deptno IS NULL
```
You might expect the query optimizer to provide the same execution plan in all cases. This is, however, not what happens...

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>INDEX FULL SCAN</td>
<td>DEPT_PK</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MINUS</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SORT UNIQUE NOSORT</td>
<td>DEPT_PK</td>
</tr>
<tr>
<td>3</td>
<td>INDEX FULL SCAN</td>
<td>EMP</td>
</tr>
<tr>
<td>4</td>
<td>SORT UNIQUE</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>HASH JOIN ANTI</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDEX FULL SCAN</td>
<td>DEPT_PK</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>HASH JOIN ANTI</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDEX FULL SCAN</td>
<td>DEPT_PK</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
</tr>
</tbody>
</table>
How It Works (3)

- The key point here is to realize that the very same data can be extracted by means of different SQL statements.
- Whenever you are optimizing a SQL statement, you should ask yourself whether other equivalent SQL statements exist.
- If they do, compare them carefully to assess which one provides the best performance.
When to Use It

- Whenever you are able to change the SQL statement, you should consider this technique.
- There is no reason for not doing it.
Pitfalls and Fallacies

- SQL statements are code. The first rule of writing code is to make it maintainable. In the first place this means that it should be readable and concise.

- With SQL, the simplest or most readable way of writing a SQL statement doesn’t always lead to the most efficient execution plan.

- In some situations you may be forced to give up readability and conciseness for performance.
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
How It Works – What Are Hints? (1)

- According to the Merriam-Webster OnLine dictionary, a hint is *an indirect or summary suggestion*.

- In Oracle’s parlance, hints are directives added to SQL statements to influence the query optimizer’s decisions.

- In other words, a hint is something that impels toward an action, not merely suggesting one.

- Warning: Just because a hint is a directive, it doesn't mean that the query optimizer will always use it!
How It Works – What Are Hints? (2)

- While optimizing a SQL statement, the query optimizer may have to take a lot of execution plans into account.
- In theory, it should consider all possible execution plans.
- In practice, in order to keep the optimization time reasonable, it excludes some of the execution plans *a priori*.
- Whenever you specify a hint, your goal is to reduce the number of execution plans considered by the query optimizer.
How It Works – What Are Hints? (3)

- Hints are only evaluated when they apply to a decision that the query optimizer has to take.
- For this reason, as soon as you specify a hint, you may be forced to add several of them to ensure it works.
How It Works – What Are Hints? (4)

- The important thing to understand is that you cannot tell the query optimizer:

  \[ I \text{ want a full table scan on table emp} \]
  \[ \text{so search for an execution plan containing it} \]

- However, you can tell it:

  \[ I \text{f you have to decide between a full table scan} \]
  \[ \text{and an index scan on table emp, take a full table scan} \]
How It Works – Specifying Hints

- Hints are an Oracle extension.
- In order to not jeopardize the compatibility of the SQL statements with other database engines, they are added as comments.

```sql
SELECT /*+ full(emp) */ *
FROM emp
WHERE empno = 7788
```

```sql
SELECT --- full(emp) a comment do not invalidate the hint *
FROM emp
WHERE empno = 7788
```
How It Works – Categories of Hints

- Initialization parameter hints
  - E.g. ALL_ROWS and DYNAMIC_SAMPLING
- Query transformation hints
  - E.g. ELIMINATE_JOIN and NO_QUERY_TRANSFORMATION
- Access path hints
  - E.g. FULL and INDEX
- Join hints
  - E.g. LEADING and USE_HASH
- Parallel processing hints
  - E.g. PARALLEL and PARALLEL_INDEX
- Other hints
  - E.g. APPEND and DRIVING_SITE
How It Works – Validity of Hints (1)

- Initialization parameter hints are valid for the whole SQL statement.
- All other hints are valid for a single query block only.

WITH emps AS (SELECT /*+ full(emp) */ deptno, count(*) AS cnt FROM emp GROUP BY deptno)
SELECT /*+ full(dept) */ dept.dname, emps.cnt FROM dept, emps WHERE dept.deptno = emps.deptno

- Global hints can reference objects in others query blocks.

WITH emps AS (SELECT deptno, count(*) AS cnt FROM emp GROUP BY deptno)
SELECT /*+ full(dept) full(emps.emp) */ dept.dname, emps.cnt FROM dept, emps WHERE dept.deptno = emps.deptno
How It Works – Validity of Hints (2)

- As of 10.1 hints accept a parameter specifying for which query block they are valid.
- To allow these references, not only does the query optimizer generate a query block name for each query block, but it also allows you to specify your own names with the QB_NAME hint.

```sql
WITH
    emps AS (SELECT /*+ qb_name(sq) */ deptno, count(*) AS cnt
               FROM emp
               GROUP BY deptno)
SELECT /*+ qb_name(main) full(@main dept) full(@sq emp) */
    dept.dname, emps.cnt
FROM dept, emps
WHERE dept.deptno = emps.deptno
```
When to Use It

- The purpose of hints is twofold...
- They are convenient as workarounds when the query optimizer does not manage to automatically generate an efficient execution plan.
- They are useful to do a kind of what-if analysis.
Pitfalls and Fallacies

- To achieve stability enough hints should be specified.
- No error is raised when a hint is found with invalid syntax.
- When a table is referenced in a hint, the alias should be used instead of the table name, whenever the table has an alias.
- Applications using hints are more difficult to be upgraded to a newer database version.
- Because views may be used in different contexts, specifying hints in them is usually not recommended.
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
How It Works – Session Level (1)

- Many parameters can be changed at session level with the ALTER SESSION statement.
- If specific users or modules require a particular configuration, the defaults should be changed at session level.
- A configuration table and a database trigger could be used for that purpose...
CREATE TABLE exec_env_conf (username VARCHAR2(30),
parameter VARCHAR2(80),
value VARCHAR2(512));

CREATE OR REPLACE TRIGGER execution_environment
AFTER LOGON ON DATABASE
BEGIN
FOR c IN (SELECT parameter, value
FROM exec_env_conf
WHERE username = sys_context('userenv', 'session_user'))
LOOP
EXECUTE IMMEDIATE 'ALTER SESSION SET ' ||
c.parameter || '=' || c.value;
END LOOP;
END;
/

INSERT INTO exec_env_conf
VALUES ('OPS$CHA', 'optimizer_mode', 'first_rows_10');
# How It Works – At SQL Statement Level

<table>
<thead>
<tr>
<th>Initialization Parameter</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cursor_sharing=exact</code></td>
<td><code>cursor_sharing_exact</code></td>
</tr>
<tr>
<td><code>optimizer_dynamic_sampling=x</code></td>
<td><code>dynamic_sampling(x)</code></td>
</tr>
<tr>
<td><code>optimizer_features_enable=x</code></td>
<td><code>optimizer_features_enable('x')</code></td>
</tr>
<tr>
<td><code>optimizer_features_enable not set</code></td>
<td><code>optimizer_features_enable(default)</code></td>
</tr>
<tr>
<td><code>optimizer_index_caching=x</code></td>
<td><code>opt_param('optimizer_index_caching' x)</code></td>
</tr>
<tr>
<td><code>optimizer_index_cost_adj=x</code></td>
<td><code>opt_param('optimizer_index_cost_adj' x)</code></td>
</tr>
<tr>
<td><code>optimizer_mode=all_rows</code></td>
<td><code>all_rows</code></td>
</tr>
<tr>
<td><code>optimizer_mode=first_rows</code></td>
<td><code>first_rows</code></td>
</tr>
<tr>
<td><code>optimizer_mode=first_rows_x</code></td>
<td><code>first_rows(x)</code></td>
</tr>
<tr>
<td><code>optimizer_mode=rule</code></td>
<td><code>rule</code></td>
</tr>
<tr>
<td><code>optimizer_secure_view_merging=x</code></td>
<td><code>opt_param('optimizer_secure_view_merging' 'x')</code></td>
</tr>
<tr>
<td><code>result_cache_mode=manual</code></td>
<td><code>no_result_cache</code></td>
</tr>
<tr>
<td><code>result_cache_mode=force</code></td>
<td><code>result_cache</code></td>
</tr>
<tr>
<td><code>star_transformation_enabled=x</code></td>
<td><code>opt_param('star_transformation_enabled' 'x')</code></td>
</tr>
</tbody>
</table>
How It Works – Dynamic Performance Views

- There are three dynamic performance views that provide information about the execution environment:
  - V$SYS_OPTIMIZER_ENV
  - V$SES_OPTIMIZER_ENV
  - V$SQL_OPTIMIZER_ENV

```sql
SQL> SELECT name, value 
  2   FROM v$ses_optimizer_env 
  3   WHERE sid = 161 AND isdefault = 'NO'
  4   MINUS 
  5   SELECT name, value 
  6   FROM v$sys_optimizer_env;

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor_sharing</td>
<td>force</td>
</tr>
<tr>
<td>optimizer_mode</td>
<td>first_rows_10</td>
</tr>
</tbody>
</table>
```
When to Use It

- Whenever the default configuration is not suitable for part of the application or part of the users, it is a good thing to change it.
- While changing parameters at session level should always be possible, hints can only be used when it is also possible to change the SQL statements.
Pitfalls and Fallacies

- Altering the execution environment at session level is easy when the setting can be centralized either in the database or in the application.
- If an application needs several execution environments, multiple connection pools should be used.
  - The environment should be set only once.
- Altering the execution environment at SQL statement level is subject to the same pitfalls and fallacies previously described for hints.
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
How It Works – What Are SQL Profiles?

- A SQL profile is an object containing information (hints) that helps the query optimizer to find an efficient execution plan for a specific SQL statement.
- It provides information about the execution environment, object statistics and corrections related to the estimations performed by the query optimizer.

```
Look up SQL profile based on SQL statement signature

SQL profile available?

Yes

Include SQL profile

Generate execution plan

No
```
How It Works – Life Cycle

1..7: SQL tuning
A..F: SQL execution

1. tune SQL statement
2. provide advice
3. get stats and SQL profile
4. analyze SQL statement
5. SQL profile
6. accept SQL profile
7. store SQL profile

A. execute SQL statement
B. provide execution plan
C. get stats, SQL profile and environment
D. generate execution plan
E. execution plan
F. execution
How It Works – SQL Tuning Advisor (1)

- The core interface of the SQL Tuning Advisor is available through DBMS_SQLTUNE.

```sql
DECLARE
    l_sql_id v$sql.session.prev_sql_id%TYPE;
    l_tuning_task VARCHAR2(30);
BEGIN
    SELECT prev_sql_id INTO l_sql_id
    FROM v$sql.session
    WHERE audsid = sys_context('userenv','sessionid');
    l_tuning_task := dbms_sqltune.create_tuning_task(
        sql_id => l_sql_id
    );
    dbms_sqltune.execute_tuning_task(l_tuning_task);
    dbms_output.put_line(l_tuning_task);
END;
```

- A graphical user interface is integrated in Enterprise Manager.
How It Works – SQL Tuning Advisor (2)

- The tuning task provides the output of its analysis in several data dictionary views.
- Instead of querying the views directly, which is a bit bothersome, DBMS_SQLTUNE.REPORT_TUNING_TASK can be used to generate a detailed report about the analysis.

```sql
SELECT dbms_sqltune.report_tuning_task('TASK_16467')
FROM dual
```
How It Works – Accepting

- To accept an advised SQL profile a PL/SQL call such as the following is used.

```sql
dbms_sqltune.accept_sql_profile(
    task_name    => 'TASK_3401',
    task_owner   => user,
    name         => 'opt_estimate',
    description  => NULL,
    category     => 'TEST',
    force_match  => TRUE,
    replace      => TRUE
);
```
How It Works – Altering

- **DBMS_SQLTUNE.ALTER_SQL_PROFILE** can be used to modify the following properties:
  - NAME
  - DESCRIPTION
  - CATEGORY
  - STATUS

- For example, the following call disables a SQL profile.

```sql
  dbms_sqltune.alter_sql_profile(
    name => 'first_rows',
    attribute_name => 'status',
    value => 'disabled'
  );
```
How It Works – Text Normalization

- The signature used for the selection of SQL profiles is computed on a normalized SQL statement.
  - Case insensitive
  - Independent of the used blank spaces

- If the SQL statement contains literals that change, it is likely that the signature changes as well.

- It’s possible to let remove the literals during the normalization phase. This is done by setting FORCE_MATCH to TRUE while accepting the SQL profile.
How It Works – Activating

- The activation of SQL profiles is controlled at the system and session levels by SQLTUNE_CATEGORY.
- It takes as a value either TRUE, FALSE, or the name of a category specified while accepting the SQL profile.
- For example, the following SQL statement activates the SQL profiles belonging to the TEST category at the session level.

```
ALTER SESSION SET sqltune_category = test
```
How It Works – Moving

- DBMS_SQLTUNE provides several procedures for moving SQL plan baselines between databases.

![Diagram showing data dictionary, staging table, and data movement utilities with operations like pack, unpack, remap, and copy via data movement utility.]
How It Works – Dropping

- SQL profile can be dropped from the data dictionary with DBMS_SQLTUNE.DROP_SQL_PROFILE.

```sql
dbms_sqltune.drop_sql_profile(
    name => 'first_rows',
    ignore => TRUE
);
```
How It Works – Privileges

- In 10.2 to create, alter, and drop a SQL profile, the CREATE ANY SQL PROFILE, DROP ANY SQL PROFILE and ALTER ANY SQL PROFILE system privileges are required.
- As of 11.1 these privileges are deprecated in favor of the ADMINISTER SQL MANAGEMENT OBJECT system privilege.
- No object privileges for SQL profiles exist.
- To use the SQL Tuning Advisor, the ADVISOR system privilege is required.
- End users do not require specific privileges to use SQL profiles.
When to Use It

- This technique should be considered whenever a specific SQL statement must be tuned and it is not possible to change it in the application (e.g. when adding hints is not an option).

- To use the SQL Tuning Advisor, the Tuning Pack and the Diagnostic Pack must be licensed.
  - It is not possible to use SQL profiles with the Standard Edition.
Pitfalls and Fallacies

- Two SQL statements with the same text shared the same SQL profile. This is also true even if they reference objects in different schemas.
- Whenever a SQL statement has a SQL profile and a stored outline at the same time, the query optimizer gives precedence to the stored outline instead of using the SQL profile.
- SQL profiles are not dropped when the objects they depend on are dropped.
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
A stored outline is the set of hints that are necessary to force the query optimizer to consistently generate a specific execution plan for a given SQL statement.

Stored outlines are stored in the data dictionary and the query optimizer selects them automatically.
How It Works – Creating

- There are two main methods you can use to create stored outlines.

- Automatically
  - CREATE_STORED_OUTLINES is set to TRUE or to another value specifying a category.
  - A stored outline is created for each SQL statement which is executed.

- Manually
  - By referencing a cursor in the shared pool.
  - By using the CREATE OUTLINE statement.

```
CREATE OR REPLACE OUTLINE outline_from_text
FOR CATEGORY test
ON SELECT * FROM t WHERE n = 1970
```
How It Works – Altering

- To change the name of a stored outline, you have to execute the ALTER OUTLINE statement.

```
ALTER OUTLINE sys_outline_07072614532024107
RENAME TO outline_from_sqlarea
```

- With the ALTER OUTLINE statement or DBMS_OUTLN.UPDATE_BY_CAT you are also able to change the category of stored outlines.
How It Works – Activating

- The query optimizer considers only the stored outlines that are active.
- To be active, a stored outline must meet two conditions.
  - The stored outlines must be enabled (per default they are)
    ```sql
    ALTER OUTLINE outline_from_text ENABLE
    ```
  - The category must be activated through USE_STORED_OUTLINES at the session or system level.
    ```sql
    ALTER SESSION SET use_stored_outlines = test
    ```
How It Works – Moving

- To move stored outlines, no particular feature is provided.
- You have to copy them yourself from one data dictionary to the other.
- This is easy because the data about the stored outlines is stored in three tables in the OUTLN schema.
  - OL$
  - HINTS$
  - OL$NODES
How It Works – Editing

- Stored outlines can be modified.
- It is not possible to directly modify the so-called public stored outline. A copy, called private stored outline, must be used instead.
- The whole process is a bit awkward...
How It Works – Dropping

- **DROP OUTLINE** drops a single category.
  
  ```sql
  DROP OUTLINE outline_from_text
  ```

- **DBMS_OUTLN.DROP_BY_CAT** drops a whole category.
  
  ```sql
  dbms_outln.drop_by_cat(cat => 'TEST')
  ```
How It Works – Privileges

- The system privileges required to create, alter, and drop a stored outline are CREATE ANY OUTLINE, DROP ANY OUTLINE, and ALTER ANY OUTLINE.

- No object privileges exist for stored outlines.

- By default:
  - DBMS_OUTLN is available only to users who either have the DBA or EXECUTE_CATALOG_ROLE role.
  - DBMS_OUTLN_EDIT is available to all users.

- End users do not require specific privileges to use stored outlines.
When to Use It

- You should consider using this technique whenever you are optimizing a specific SQL statement and you are not able to change it in the application (e.g. when adding hints is not an option).
- As of 11.1 stored outlines are deprecated in favor of SQL plan baselines.
Pitfalls and Fallacies

- USE_STORED_OUTLINES cannot be specified in an initialization file.
- Two SQL statements with the same text shared the same SQL profile. This is also true even if they reference objects in different schemas.
- Stored outlines are not dropped when the objects they depend on are dropped.
Agenda

1. Choosing a SQL Optimization Technique
2. Altering the Access Structures
3. Altering the SQL Statement
4. Hints
5. Altering the Execution Environment
6. SQL Profiles
7. Stored Outlines
8. SQL Plan Baselines
Availability and Scope

- SQL plan baselines are available as of 11.1.
- They substitute stored outlines. Actually, they can be considered an enhanced version of stored outlines.
- They are designed to provide stable execution plans in case of changes in the execution environment or object statistics.
- They can also be used to tune an application without modifying it. For some unknown reason, in Oracle documentation this possibility is not mentioned.
How It Works – What Are SQL Plan Baselines?

- A SQL plan baseline is an object associated with a SQL statement that is designed to influence the query optimizer while it generates execution plans.

- SQL plan baselines are used to force the query optimizer to consistently generate a specific execution plan for a given SQL statement.
How It Works – Capturing

- There are three ways to capture new SQL plan baselines.

- Automatic capture
  - OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES is set to TRUE
  - The default value is FALSE.

- Load from library cache
  - DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE

- Load from SQL tuning set
  - DBMS_SPM.LOAD_PLANS_FROM_SQLSET
  - SQL tuning sets created in 10.1 and 10.2 are supported.
How It Works – Evolving

- When the query optimizer recognizes that an execution plan different from the one forced by a SQL plan baseline might be more efficient, new non-accepted SQL plan baselines are automatically added.
- The idea is to tell you that other and possibly better execution plans exist.
- To verify whether one of these execution plans will in fact perform better than the ones generated with the help of SQL plan baselines, a so-called evolution must be attempted.
  - DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE
- Automatic evolution of SQL plan baselines is supported with the Tuning Pack.
How It Works – SPM Evolve Advisor (12.1)

- Its purpose is to execute an evolution for the nonaccepted execution plans associated to SQL plan baselines.

- To display what it did:
  - DBA_ADVISOR_EXECUTIONS (task name: SYS_AUTO_SPM_EVOLVE_TASK)
  - DBMS_SPM.REPORT_AUTO_EVOLVE_TASK function
How It Works – Displaying

- General information about the available SQL plan baselines can be displayed through DBA_SQL_PLAN_BASELINES.

- To display detailed information about them, DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE is available.
  - In 11.x, to correctly display the execution plan, the function must be able to reproduce it.
  - In 12.1 the execution plan is stored in the SQL plan baseline.
How It Works – Altering

- The following properties of a SQL plan baselines can be changed with DBMS_SPM.ALTER_SQL_PLAN_BASELINE:
  - ENABLED
  - FIXED
  - AUTOPURGE
  - PLAN_NAME
  - DESCRIPTION
How It Works – Activating

- The query optimizer uses the available SQL plan baselines only when OPTIMIZER_USE_SQL_PLAN_BASELINES is set to TRUE.
- The default value is TRUE.
- You can change it at the session and system levels.
How It Works – Moving

- DBMS_SPM provides several procedures for moving SQL plan baselines between databases.

```
PACK
UNPACK

Copy via data movement utility

Create

Data Dictionary

Staging Table

Pack

Unpack

Data Dictionary

Staging Table

Pack

Unpack

Staging Table
```
How It Works – Dropping

- SQL plan baselines can be dropped from the data dictionary with DBMS_SPM.DROP_SQL_PLAN_BASELINE.

- SQL plan baselines are automatically purged when a retention period is over.
  - The default retention period is 53 weeks.
  - Use DBA_SQL_MANAGEMENT_CONFIG to display the actual value.
  - The retention period can be changed by calling DBMS_SPM.CONFIGURE.

```sql
dbms_spm.configure(parameter_name => 'plan_retention_weeks', parameter_value => 12);
```
How It Works – Privileges

- When SQL plan baselines are automatically captured, no particular privilege is needed to create them.
- DBMS_SPM can be executed only by users with the ADMINISTER SQL MANAGEMENT OBJECT system privilege.
  - The role DBA includes it by default.
  - No object privileges exist for SQL plan baselines.
- End users do not require specific privileges to use SQL plan baselines.
When to Use It

- You should consider using this technique in two situations.
  - When changes to the execution environment are causing instability of execution plans.
  - Whenever you are optimizing a specific SQL statement and you are not able to change it in the application (e.g. when adding hints is not an option).

- SQL plan baselines are available as of 11.1 only.
- SQL plan baselines are available only with Enterprise Edition.
Pitfalls and Fallacies (1)

- Two SQL statements with the same text have the same signature. This is also true even if they reference objects in different schemas!

- SQL plan baselines are stored in the SYSAUX tablespace.
  - By default, at most 10% of the tablespace can be used for them.
  - The current value can be displayed through `DBA_SQL_MANAGEMENT_CONFIG`.

```sql
dbms_spm.configure(
    parameter_name => 'space_budget_percent',
    parameter_value => 5);
```
Pitfalls and Fallacies (2)

- Whenever a SQL statement has a SQL plan baseline and a stored outline at the same time, the query optimizer gives precedence to the stored outline instead of using the SQL plan baseline.

- SQL plan baselines are not immediately dropped when the objects they depend on are dropped.
Core Messages

- Selecting a SQL optimization technique is not always easy. Nevertheless if you understand the pros and cons of using them the choice is much easier.

- In practice our choice is limited because not all techniques can be applied in all situations.
THANK YOU.