Oracle SQL Tuning for Developers Workshop

Student Guide - Volume II

D73549GC10
Edition 1.0
October 2012
D78800

ORACLE

Authors

Sean Kim

Dimpi Rani Sarmah

Technical Contributors and Reviewers

Nancy Greenberg

Swarnapriya Shridhar

Joel Goodman

Raza Siddiqui

Veerabhadra Rao Putrevu

Glenn Austin

Amitabh Hans

Arijit Ghosh

Editors

Malavika Jinka

Raj Kumar

Susan Moxley

Anwesha Ray

Rashmi Rajagopal

Graphic Designer

Maheshwari Krishnamurthy

Publishers

Pavithran Adka

Giri Venugopal

Contents

7 Optimizer: Table and Index Access Paths

8

Optimizer: Join Operators

Objectives

After completing this lesson, you should be able to:

- Describe the SQL operators for joins
- List the possible access paths

This lesson helps you to understand the execution plans related to join operations.

Join Methods

A join:

- Defines the relationship between two row sources
- Is a method of combining data from two data sources
- Is controlled by join predicates, which define how the objects are related
- Join methods:
  - Nested loops
  - Sort-merge join

```
SELECT e.ename, d.dname
FROM dept d JOIN emp e USING (deptno)          ← Join predicate
WHERE e.job = 'ANALYST' OR e.empno = 9999;     ← Nonjoin predicate
```

```
SELECT e.ename,d.dname
FROM   emp e, dept d
WHERE  e.deptno = d.deptno AND                 ← Join predicate
       (e.job = 'ANALYST' OR e.empno = 9999);  ← Nonjoin predicate
```

ORACLE

A row source is a set of data that can be accessed in a query. It can be a table, an index, a nonmergeable view, or even the result set of a join tree consisting of many different objects.

A join predicate is a predicate in the `WHERE` clause that combines the columns of two of the tables in the join.

A nonjoin predicate is a predicate in the `WHERE` clause that references only one table.

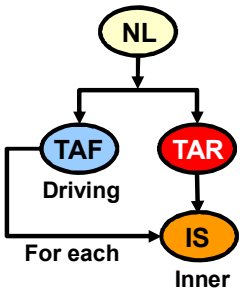
A join operation combines the output from two row sources (such as tables or views) and returns one resulting row source (data set). The optimizer supports different join methods such as the following:

- **Nested loops join:** Useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table
- **Sort-merge join:** Can be used to join rows from two independent sources. Hash joins generally perform better than sort-merge joins. On the other hand, sort-merge joins can perform better than hash joins if one or two row sources are already sorted.
- **Hash join:** Used for joining large data sets. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows. This method is best used when the smaller table fits in the available memory. The cost is then limited to a single read pass over the data for the two tables.

Note: The slide shows you the same query using both the American National Standards Institute (ANSI) and non-ANSI join syntax. The ANSI syntax is the first example.

Nested Loops Join

- Driving row source is scanned.
- Each row returned drives a lookup in inner row source.
- Joining rows are then returned.




```
select ename, e.deptno, d.deptno, d.dname
from emp e, dept d
where e.deptno = d.deptno and ename like 'A%';
-------------------------------------------------------------------------
| Id  | Operation                    |  Name       | Rows  |Cost  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     2 |    4 |
|   1 |  NESTED LOOPS                |             |     2 |    4 |
|   2 |   TABLE ACCESS FULL          | EMP         |     2 |    2 |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPT        |     1 |    1 |
|   4 |    INDEX UNIQUE SCAN         | PK_DEPT     |     1 |      |
-------------------------------------------------------------------------
   2 - filter("E"."ENAME" LIKE 'A%')
   4 - access("E"."DEPTNO"="D"."DEPTNO")
```

ORACLE

In the general form of the nested loops join, one of the two tables is defined as the outer table or the driving table. The other table is called the inner table or the right-hand side.

For each row in the outer (driving) table that matches the single table predicates, all rows in the inner table that satisfy the join predicate (matching rows) are retrieved. If an index is available, it can be used to access the inner table by row ID.

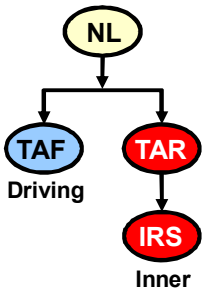
Any nonjoin predicates on the inner table are considered after this initial retrieval, unless a composite index combining both the join and the nonjoin predicate is used.

The code to emulate a nested loops join might look as follows:

```
for r1 in (select rows from EMP that match single table predicate) loop
  for r2 in (select rows from DEPT that match current row from EMP) loop
       output values from current row of EMP and current row of DEPT
    end loop
  end loop
```

The optimizer uses nested loops joins when joining a small number of rows, with a good driving condition between the two tables. You drive from the outer loop to the inner loop, so the order of tables in the execution plan is important. Therefore, you should use other join methods when two independent row sources are joined.

Nested Loops Join: Prefetching




```
select ename, e.deptno, d.deptno, d.dname
from emp e, dept d
where e.deptno = d.deptno and ename like 'A%';
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |     2 |    84 |     5
|   1 |  TABLE ACCESS BY INDEX ROWID| DEPT   |     1 |    22 |     1
|   2 |   NESTED LOOPS               |        |     2 |    84 |     5
|*  3 |    TABLE ACCESS FULL         | EMP    |     2 |    40 |     3
|*  4 |    INDEX RANGE SCAN          | IDEPT  |     1 |       |     0
-----------------------------------------------------------------------------
   3 - filter("E"."ENAME" LIKE 'A%')
   4 - access("E"."DEPTNO"="D"."DEPTNO")
```

ORACLE

Oracle 9*i*R2 introduced a mechanism called nested loops prefetching. The idea is to improve I/O utilization, and therefore response time, of index access with table lookup by batching row ID lookups into parallel block reads.

This change to the plan output is not considered a different execution plan. It does not affect the join order, join method, access method, or parallelization scheme.

This optimization is available only when the inner access path is index range scan and not if the inner access path is index unique scan.

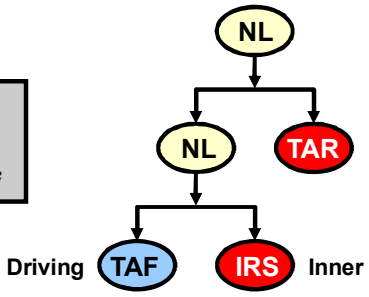
The prefetching mechanism is used by table lookup. When an index access path is chosen and the query cannot be satisfied by the index alone, the data rows indicated by the `ROWID` also must be fetched. This `ROWID` to data row access (table lookup) is improved by using data block prefetching, which involves reading an array of blocks that are pointed at by an array of qualifying `ROWID`s.

Without data block prefetching, accessing a large number of rows using a poorly clustered B*-tree index could be expensive. Each row accessed by the index would likely be in a separate data block and thus would require a separate I/O operation.

With data block prefetching, the system delays data blocks reads until multiple rows specified by the underlying index are ready to be accessed. The system then retrieves multiple data blocks simultaneously, rather than reading a single data block at a time.

Nested Loops Join: 11*g* Implementation

```
select ename, e.deptno, d.deptno, d.dname
from emp e, dept d
where e.deptno = d.deptno and ename like 'A%';
```



Explain Plan

0.299 seconds

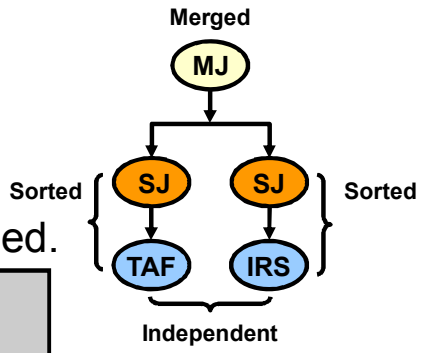
OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			4	1
NESTED LOOPS				
NESTED LOOPS			4	1
TABLE ACCESS	EMP	FULL	3	1
Filter Predicates				
ENAME LIKE 'A%'				
INDEX	PK_DEPT	UNIQUE SCAN	0	1
Access Predicates				
E.DEPTNO=D.DEPTNO				
TABLE ACCESS	DEPT	BY INDEX RO...	1	1

ORACLE

Oracle Database 11*g* introduces a new way of performing joins with `NESTED LOOPS` operators. With this `NESTED LOOPS` implementation, the system first performs a `NESTED LOOPS` join between the other table and the index. This produces a set of `ROWID`s that you can use to look up the corresponding rows from the table with the index. Instead of going to the table for each `ROWID` produced by the first `NESTED LOOPS` join, the system batches up the `ROWID`s and performs a second `NESTED LOOPS` join between the `ROWID`s and the table. This `ROWID` batching technique improves performance as the system reads each block in the inner table only once.

Sort-Merge Join

- First and second row sources are sorted by the same sort key.
- Sorted rows from both tables are merged.




```
select /*+ USE_MERGE(d e) NO_INDEX(d) */
  ename, e.deptno, d.deptno, dname
  from emp e, dept d
  where e.deptno = d.deptno and ename > 'A'
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			8	14
MERGE JOIN			8	14
SORT		JOIN	4	4
TABLE ACCESS	DEPT	FULL	3	4
SORT		JOIN	4	14
Access Predicates				
AND				
E.DEPTNO=D.DEPTNO				
SYS_OP_DESCEND(E.DEPT				
Filter Predicates				
E.DEPTNO=D.DEPTNO				
TABLE ACCESS	EMP	FULL	3	14
Filter Predicates				
ENAME>'A'				

In a sort-merge join, there is no concept of a driving table. A sort-merge join is executed as follows:

1. Get the first data set, using any access and filter predicates, and sort it on the join columns.
2. Get the second data set, using any access and filter predicates, and sort it on the join columns.
3. For each row in the first data set, find the start point in the second data set and scan until you find a row that does not join.

The merge operation combines the two sorted row sources to retrieve every pair of rows that contain matching values for the columns used in the join predicate.

If one row source has already been sorted in a previous operation (there is an index on the join column, for example), the sort-merge operation skips the sort on that row source. When you perform a merge join, you must fetch all rows from the two row sources before to return the first row to the next operation. Sorting could make this join technique expensive, especially if sorting cannot be performed in memory.

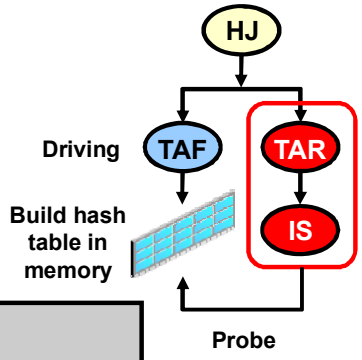
The optimizer can select a sort-merge join over a hash join for joining large amounts of data if any of the following conditions are true:

- The join condition between two tables is not an equijoin.
- Sorts are already required by previous operations.

Note: Sort-merge joins are useful when the join condition between two tables is an inequality condition (but not a nonequality), such as `<`, `<=`, `>`, or `>=`.

Hash Join

- The smallest row source is used to build a hash table.
- The second row source is hashed and checked against the hash table.




```
select /*+ USE_HASH(e d) */
  ename, e.deptno, d.deptno, dname
  from emp e, dept d
  where e.deptno = d.deptno and ename like 'A%'
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			7	1
HASH JOIN			7	1
Access Predicates				
AND				
E.DEPTNO=D.DEPTNO				
SYS_OP_DESCEND(E.DEPTNO)				
TABLE ACCESS	EMP	FULL	3	1
Filter Predicates				
ENAME LIKE 'A%'				
TABLE ACCESS	DEPT	FULL	3	4

ORACLE

To perform a hash join between two row sources, the system reads the first data set and builds an array of hash buckets in memory. A hash bucket is little more than a location that acts as the starting point for a linked list of rows from the build table. A row belongs to a hash bucket if the bucket number matches the result that the system gets by applying an internal hashing function to the join column or columns of the row.

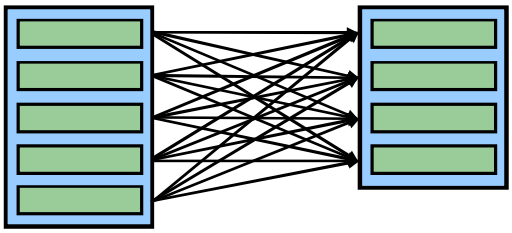
The system starts to read the second set of rows, using whatever access mechanism is most appropriate for acquiring the rows, and it uses the same hash function on the join column or columns to calculate the number of the relevant hash bucket. The system then checks to see if there are any rows in that bucket. This is known as probing the hash table.

If there are no rows in the relevant bucket, the system can immediately discard the row from the probe table.

If there are some rows in the relevant bucket, the system does an exact check on the join column or columns to see if there is a proper match. Any rows that survive the exact check can immediately be reported (or passed on to the next step in the execution plan). So, when you perform a hash join, you must fetch all rows from the smallest row source to return the first row to next operation.

Note: Hash joins are performed only for equijoins, and are most useful when joining large amounts of data.

Cartesian Join



```
select ename, e.deptno, d.deptno, dname
from emp e, dept d where ename like 'A%';
```

Explain Plan

0.295 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			6	4
MERGE JOIN		CARTESIAN	6	4
TABLE ACCESS	EMP	FULL	3	1
Filter Predicates				
ENAME LIKE 'A%'				
BUFFER		SORT	3	4
TABLE ACCESS	DEPT	FULL	3	4

ORACLE

A Cartesian join is used when one or more of the tables do not have any join conditions to any other tables in the statement. The optimizer joins every row from one data source with every row from the other data source, creating the Cartesian product of the two sets.

A Cartesian join can be seen as a nested loop with no elimination; the first row source is read, and then for every row, all the rows are returned from the other row source.

Note: Cartesian join is generally not desirable. However, it is perfectly acceptable to have one with single-row row source (guaranteed by a unique index, for example) joined to some other table.

Join Types

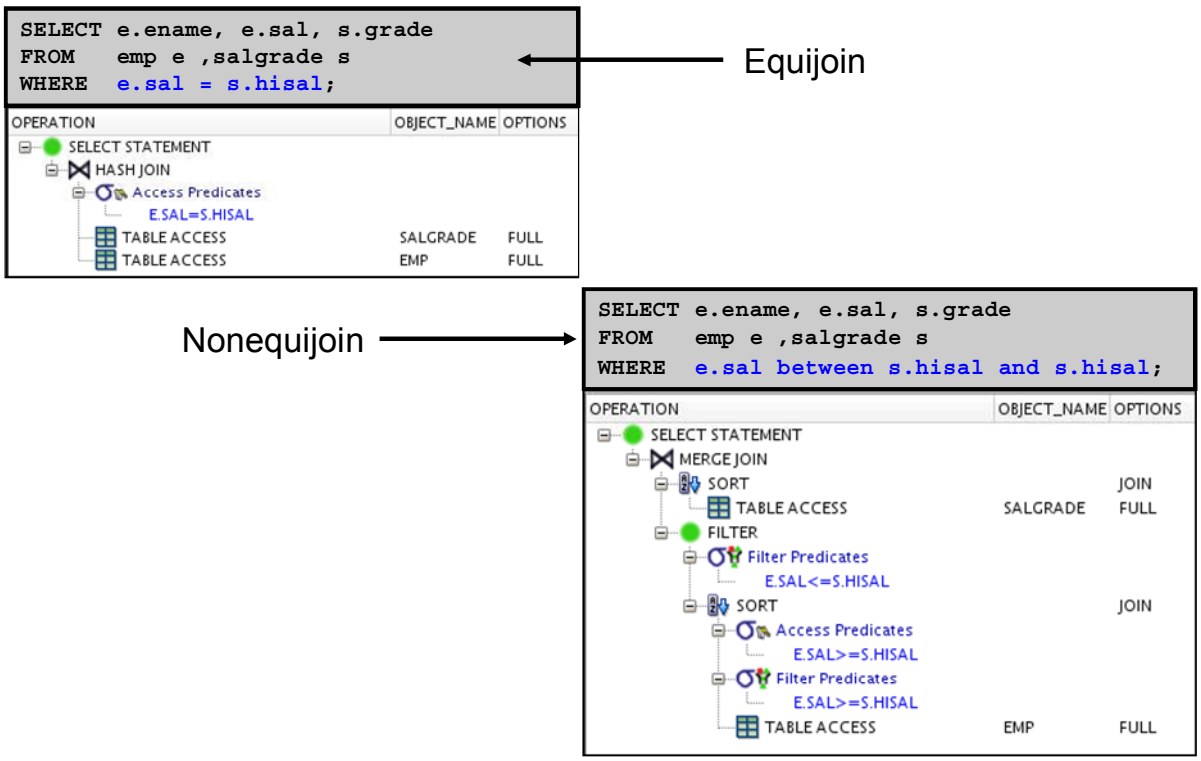
- A join operation combines the output from two row sources and returns one resulting row source.
- Join operation types include the following :
  - Join (Equijoin/Natural – Nonequijoin)
  - Outer join (Full, Left, and Right)
  - Semi join: `EXISTS` subquery
  - Anti join: `NOT IN` subquery
  - Star join (Optimization)

Join operation types include the following:

- **Join (equijoin and nonequijoin):** Returns rows that match predicate join
- **Outer join:** Returns rows that match predicate join and row when no match is found
- **Semi join:** Returns rows that match the `EXISTS` subquery. Find one match in the inner table, and then stop the search.
- **Anti join:** Returns rows with no match in the `NOT IN` subquery. Stop as soon as one match is found.
- **Star join:** This is not a join type, but just a name for an implementation of a performance optimization to better handle the fact and dimension model.

Antijoin and semijoin are considered to be join types, even though the SQL constructs that cause them are subqueries. Antijoin and semijoin are internal optimization algorithms used to flatten subquery constructs in such a way that they can be resolved in a join-like way.

Equijoins and Nonequijoins



The join condition determines whether a join is an equijoin or a nonequijoin. An equijoin is a join with a join condition containing an equality operator. When a join condition relates two tables by an operator other than equality, it is a nonequijoin.

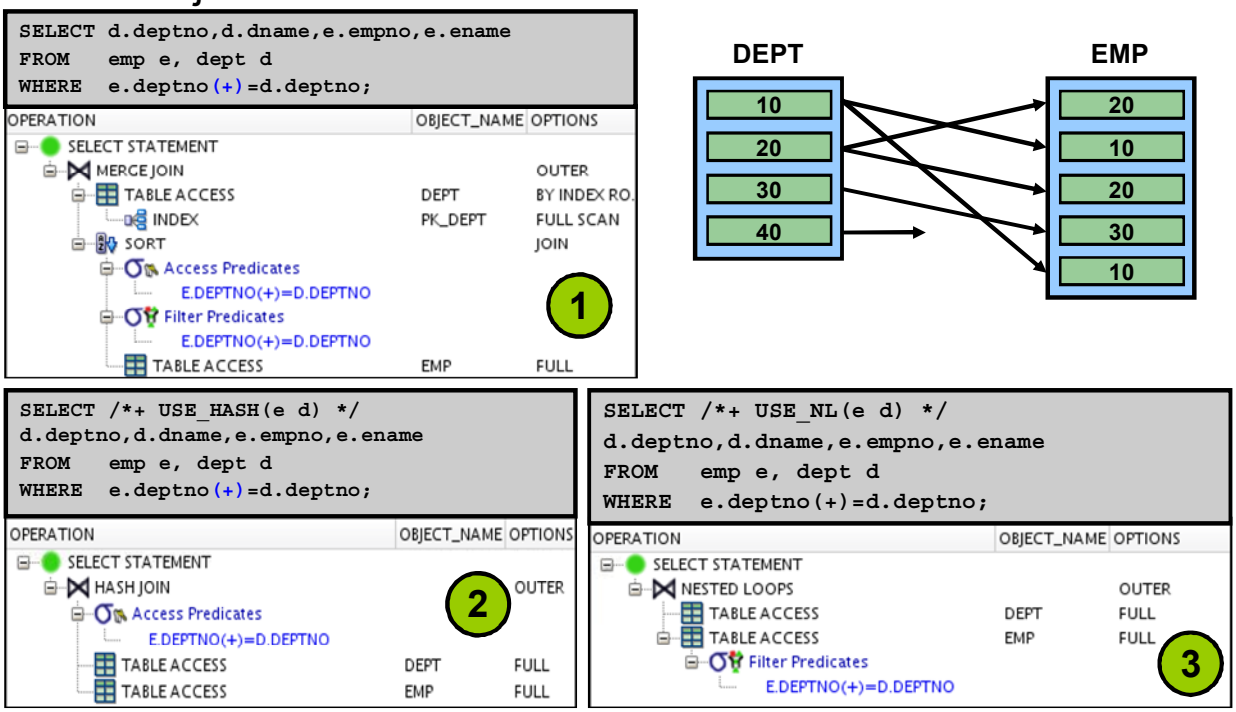
Equijoins are the most commonly used. Examples of an equijoin and a nonequijoin are shown in the slide. Nonequijoins are less frequently used.

To improve SQL efficiency, use equijoins whenever possible. Statements that perform equijoins on untransformed column values are the easiest to tune.

Note: If you have a nonequijoin, a hash join is not possible.

Outer Joins

An outer join returns a row even if no match is found.



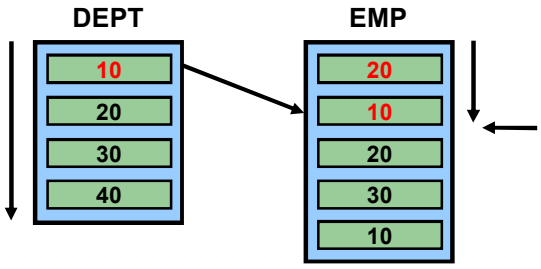
The simple join is the most commonly used join within the system. Other joins open up extra functionality, but have much more specialized uses. The outer join operator is placed on the deficient side of the query; that is, it is placed against the table that has the missing join information. Consider `EMP` and `DEPT`. There may be a department that has no employees. If `EMP` and `DEPT` are joined together, this particular department would not appear in the output because there is no row that matches the join condition for that department. By using the outer join, the missing department can be displayed.

1. **Merge Outer joins:** By default, the optimizer uses `MERGE OUTER JOIN`.
2. **Outer join with nested loops:** The left/driving table is always the table whose rows are being preserved (`DEPT` in the example). For each row from `DEPT`, look for all matching rows in `EMP`. If none is found, output `DEPT` values with null values for the `EMP` columns. If rows are found, output `DEPT` values with these `EMP` values.
3. **Hash Outer joins:** The left/outer table whose rows are being preserved is used to build the hash table, and the right/inner table is used to probe the hash table. When a match is found, the row is output and the entry in the hash table is marked as matched to a row. After the inner table is exhausted, the hash table is read over once again, and any rows that are not marked as matched are output with null values for the `EMP` columns. The system hashes the table whose rows are not being preserved, and then reads the table whose rows are being preserved, probing the hash table to see whether there was a row to join to.

Note: You can also use the ANSI syntax for full, left, and right outer joins (not shown in the slide).

Semijoins

Semijoins look only for the first match.




```
SELECT deptno, dname
FROM   dept
WHERE  EXISTS (SELECT 1 FROM emp WHERE emp.deptno=dept.deptno);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		SEMI
TABLE ACCESS	DEPT	BY INDEX RO.
INDEX	PK_DEPT	FULL SCAN
SORT		UNIQUE
Access Predicates		
EMP.DEPTNO=DEPT.DEPTNO		
Filter Predicates		
EMP.DEPTNO=DEPT.DEPTNO		
TABLE ACCESS	EMP	FULL

Semijoins return a result when you hit the first joining record. A semijoin is an internal way of transforming an `EXISTS` subquery into a join. However, you cannot see this occur anywhere.

Semijoins return rows that match an `EXISTS` subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery.

In the slide diagram, for each `DEPT` record, only the first matching `EMP` record is returned as a join result. This prevents scanning huge numbers of duplicate rows in a table when all you are interested in is if there are any matches.

When the subquery is not unnested, a similar result could be achieved by using a `FILTER` operation, scanning a row source until a match is found, and then returning it.

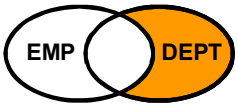
Note: A semijoin can always use a merge join. The optimizer may choose nested-loop or hash joins methods to perform semijoins as well.

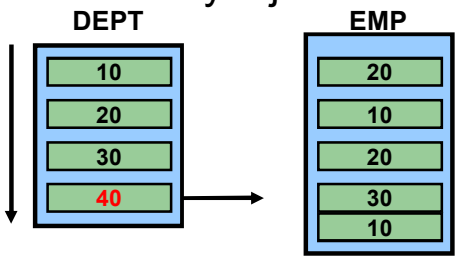
Antijoins

Reverse of what would have been returned by a join

```
SELECT deptno, dname
FROM   dept
WHERE  deptno not in
       (SELECT deptno FROM emp);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		ANTI NA
SORT		JOIN
TABLE ACCESS	DEPT	BY INDEX RO...
INDEX	PK_DEPT	FULL SCAN
SORT		UNIQUE
Access Predicates		
DEPTNO=DEPTNO		
Filter Predicates		
DEPTNO=DEPTNO		
TABLE ACCESS	EMP	FULL






```
SELECT deptno, dname FROM dept
WHERE deptno IS NOT NULL AND
deptno NOT IN
 (SELECT /*+ HASH_AJ */ deptno FROM
emp WHERE deptno IS NOT NULL);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		ANTI
Access Predicates		
DEPTNO=DEPTNO		
TABLE ACCESS	DEPT	FULL
TABLE ACCESS	EMP	FULL
Filter Predicates		
DEPTNO IS NOT NULL		

Antijoins return rows that fail to match (`NOT IN`) the subquery on the right side. For example, an antijoin can select a list of departments that do not have any employees.

The optimizer uses a merge antijoin algorithm for `NOT IN` subqueries by default. However, if the `HASH_AJ` or `NL_AJ` hints are used and various required conditions are met, the `NOT IN` uncorrelated subquery can be changed. Although antijoins are mostly transparent to the user, it is useful to know that these join types exist and could help explain unexpected performance changes between releases.

Quiz

The _____ join is used when one or more of the tables do not have any join conditions to any other tables in the statement.

a. Hash
b. Cartesian
c. Nonequijoin
d. Outer

Answer: b

Quiz

The _____ join returns a row even if no match is found.

a. Hash
b. Cartesian
c. Semi
d. Outer

Answer: d

Quiz

The _____ join looks only for the first match.

a. Hash
b. Cartesian
c. Semi
d. Outer

Answer: c

Quiz

In a hash join, the _____ row source is used to build a hash table.

a. Biggest
b. Smallest
c. Sorted
d. Unsorted

Answer: b

Summary

In this lesson, you should have learned to:

- Describe the SQL operators for joins
- List the possible access paths

Practice 8: Overview

This practice covers using different join paths for better optimization.

9 Other Optimizer Operators

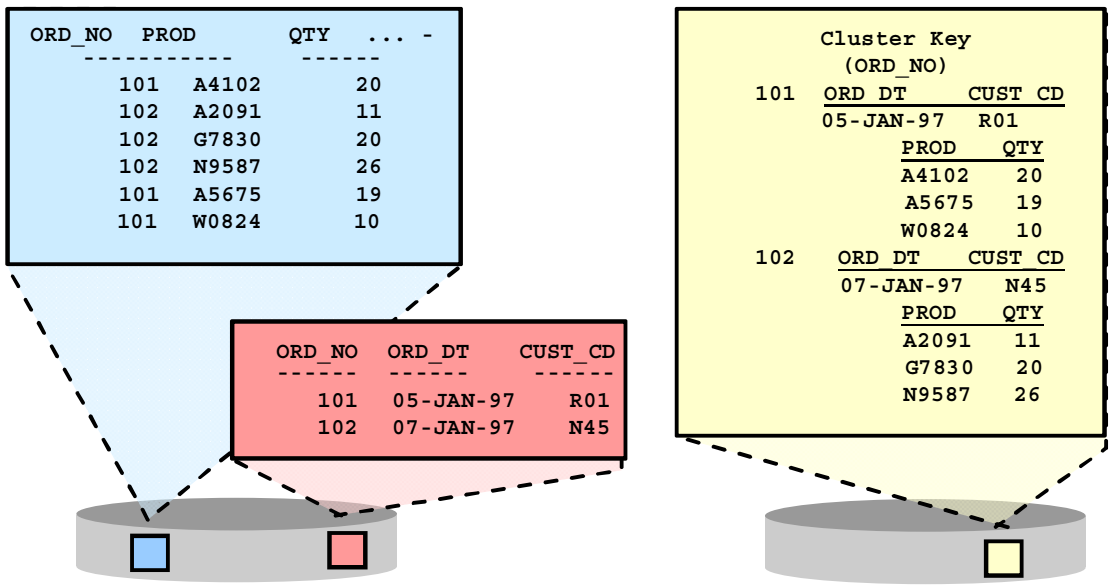
Objectives

After completing this lesson, you should be able to:

- Describe SQL operators for:
  - Clusters
  - In-List
  - Sorts
  - Filters
  - Set Operations
- Result cache operators

This lesson helps you to understand the execution plans that use common operators of other access methods.

Clusters



Unclustered ORDERS and ORDER_ITEMS tables

Clustered ORDERS and ORDER_ITEMS tables

Clusters are an optional method for storing table data. A cluster is a group of tables that share the same data blocks because they share common columns and are often used together. For example, the ORDERS and ORDER_ITEMS table share the ORDER_ID column. When you cluster the ORDERS and ORDER_ITEMS tables, the system physically stores all rows for each order from both the ORDERS and ORDER_ITEMS tables in the same data blocks.

Cluster index: A cluster index is an index defined specifically for a cluster. Such an index contains an entry for each cluster key value. To locate a row in a cluster, the cluster index is used to find the cluster key value, which points to the data block associated with that cluster key value. Therefore, the system accesses a given row with a minimum of two I/Os.

Hash clusters: Hashing is an optional way of storing table data to improve the performance of data retrieval. To use hashing, you create a hash cluster and load tables into the cluster. The system physically stores the rows of a table in a hash cluster and retrieves them according to the results of a hash function. The key of a hash cluster (just as the key of an index cluster) can be a single column or composite key. To find or store a row in a hash cluster, the system applies the hash function to the row's cluster key value; the resulting hash value corresponds to a data block in the cluster, which the system then reads or writes on behalf of the issued statement.

Note: Hash clusters are a better choice than an indexed table or index cluster when a table is queried frequently with equality queries.

When Are Clusters Useful?

- Index cluster:
  - Tables are always joined on the same keys.
  - The size of the table is not known.
  - It can be used in any type of search.
- Hash cluster:
  - Tables are always joined on the same keys.
  - Storage for all cluster keys is allocated initially.
  - It can be used in either equality (=) or nonequality (<>) searches.

- Index clusters allow row data from one or more tables that share a cluster key value to be stored in same block. You can locate these rows by using a cluster index, which has one entry per cluster key value and *not* for each row. Therefore, the index is smaller and less costly to access for finding multiple rows. The rows with the same key are in a small group of blocks. This means that in an index cluster, the clustering factor is very good and provides clustering for data from multiple tables sharing the same join key. The smaller index and smaller group of blocks reduce the cost of access by reducing block visits to the buffer cache. Index clusters are useful when the size of the tables is not known in advance (for example, creating a new table rather than converting an existing one whose size is stable), because a cluster bucket is created only after a cluster key value is used. They are also useful for all filter operations or searches. Note that full table scans do not perform well on a table in a multiple-table cluster because it has more blocks than the table would have if it were created as a heap table.
- Hash clusters allow row data from one or more tables that share a cluster key value to be stored in the same block. You can locate these rows by using a system- or user-provided hashing function or by using the cluster key value. The cluster key value method assumes that its value is evenly distributed so that row access is faster than with index clusters. Table rows with the same cluster key values hash into the same cluster buckets and can be stored in the same block or small group of blocks.

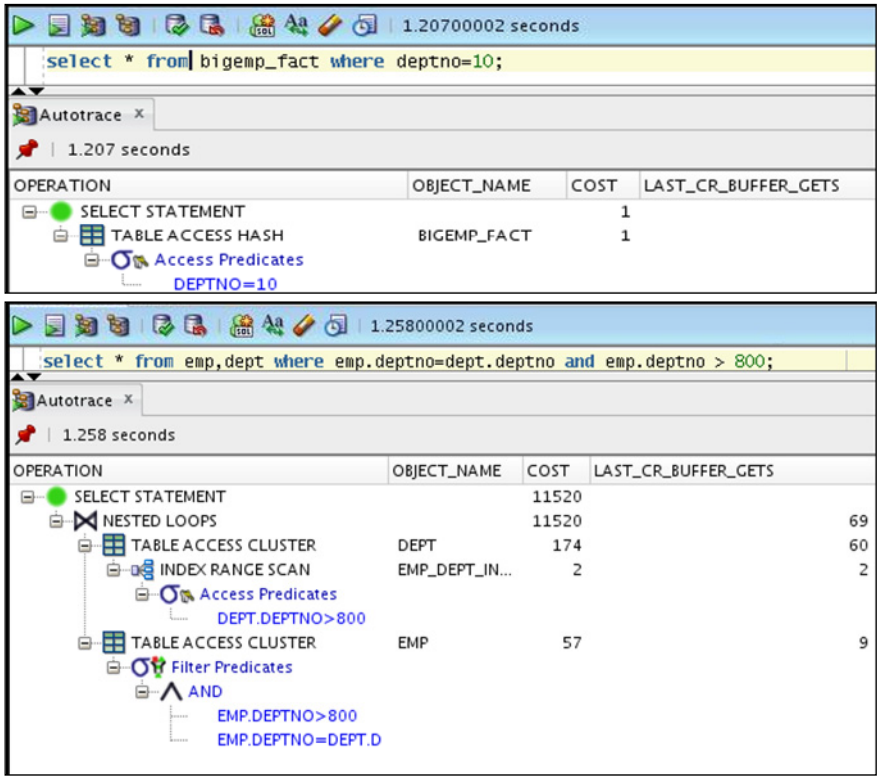
When Are Clusters Useful?

- Single-table hash cluster:
  - Fastest way to access a large table with an equality search
- Sorted hash cluster:
  - Is used only for equality search
  - Avoid sorts on batch reporting
  - Avoid overhead probe on the branch blocks of an IOT

This means that, in a hash cluster, the clustering factor is also very good and a row may be accessed by its key with one block visit only and without needing an index. Hash clusters allocate all the storage for all the hash buckets when the cluster is created, so they may waste space. They also do not perform well other than on equality searches or nonequality searches. Like index clusters if they contain multiple tables, full scans are more expensive for the same reason.

- Single-table hash clusters are similar to a hash cluster, but are optimized in the block structures for access to a single table, thereby providing the fastest possible access to a row other than by using a row ID filter. Because they have only one table, full scans, if they happen, cost as much as they would in a heap table.
- Sorted hash clusters are designed to reduce costs of accessing ordered data by using a hashing algorithm on the hash key. Accessing the first row matching the hash key may be less costly than using an IOT for a large table because it saves the cost of a B*-tree probe. All the rows that match on a particular hash key (for example, account number) are stored in the cluster in the order of the sort key or keys (for example, phone calls), thereby eliminating the need for a sort to process the order by clause. These clusters are very good for batch reporting, billing, and so on.

Cluster Access Path: Examples



The example in the slide shows you two different cluster access paths.

In the top access path example, a hash scan is used to locate rows in a hash cluster, based on a hash value. In a hash cluster, all rows with the same hash value are stored in the same data block. To perform a hash scan, the system first obtains the hash value by applying a hash function to a cluster key value specified by the statement. The system then scans the data blocks containing rows with that hash value.

The second access path example assumes that a cluster index was used to cluster both the `EMP` and `DEPT` tables. In this case, a cluster scan is used to retrieve, from a table stored in an indexed cluster, all rows that have the same cluster key value. In an indexed cluster, all rows with the same cluster key value are stored in the same data block. To perform a cluster scan, the system first obtains the `ROWID` of one of the selected rows by scanning the cluster index. The system then locates the rows based on this `ROWID`.

Note: You see examples of how to create clusters in the labs for this lesson.

Sorting Operators

- SORT operator:
  - AGGREGATE: Retrieves a single row from group function
  - UNIQUE: Removes duplicates
  - JOIN: Precedes a merge join
  - GROUP BY, ORDER BY: For these operators
- HASH operator:
  - GROUP BY: For this operator
  - UNIQUE: Equivalent to SORT UNIQUE
- If you want ordered results, *always* use ORDER BY.

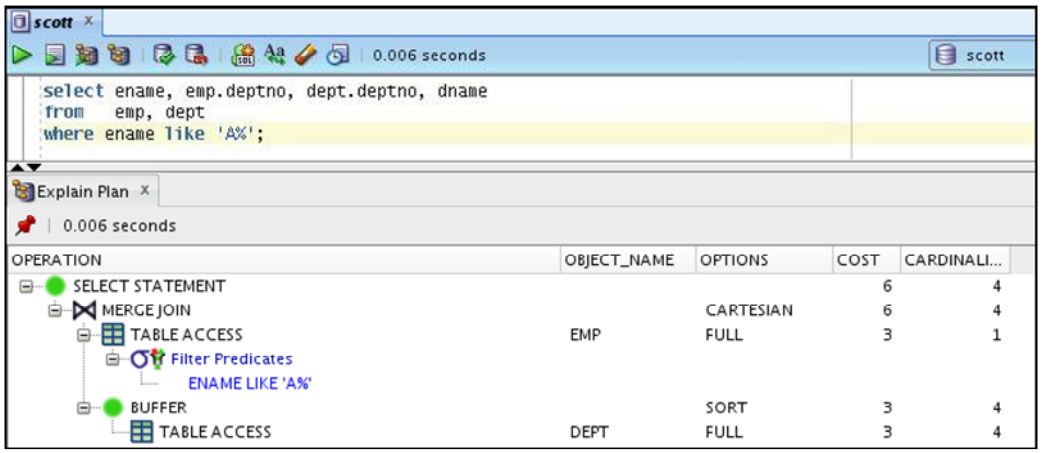
Sort operations result when users specify an operation that requires a sort. Commonly encountered operations include the following:

- SORT AGGREGATE does not involve a sort. It retrieves a single row that is the result of applying a group function to a group of selected rows. Operations such as COUNT and MIN are shown as SORT AGGREGATE.
- SORT UNIQUE sorts output rows to remove duplicates. It occurs if a user specifies a DISTINCT clause or if an operation requires unique values for the next step.
- SORT JOIN happens during a sort-merge join, if the rows need to be sorted by the join key.
- SORT GROUP BY is used when aggregates are computed for different groups in the data. The sort is required to separate the rows into different groups.
- SORT ORDER BY is required when the statement specifies an ORDER BY that cannot be satisfied by one of the indexes.
- HASH GROUP BY hashes a set of rows into groups for a query with a GROUP BY clause.
- HASH UNIQUE hashes a set of rows to remove duplicates. It occurs if a user specifies a DISTINCT clause or if an operation requires unique values for the next step. This is similar to SORT UNIQUE.

Note: Several SQL operators cause implicit sorts (or hashes since Oracle Database 10*g*, Release 2), such as `DISTINCT`, `GROUP BY`, `UNION`, `MINUS`, and `INTERSECT`. However, do not rely on these SQL operators to return ordered rows. If you want to have rows ordered, use the `ORDER BY` clause.

Buffer Sort Operator

```
SELECT ename, emp.deptno, dept.deptno, dname
FROM emp, dept
WHERE ename like 'A%';
```



ORACLE

The `BUFFER SORT` operator uses a temporary table or a sort area in memory to store intermediate data. However, the data is not necessarily sorted.

The `BUFFER SORT` operator is needed if there is an operation that needs all the input data before it can start. (See “Cartesian Join.”)

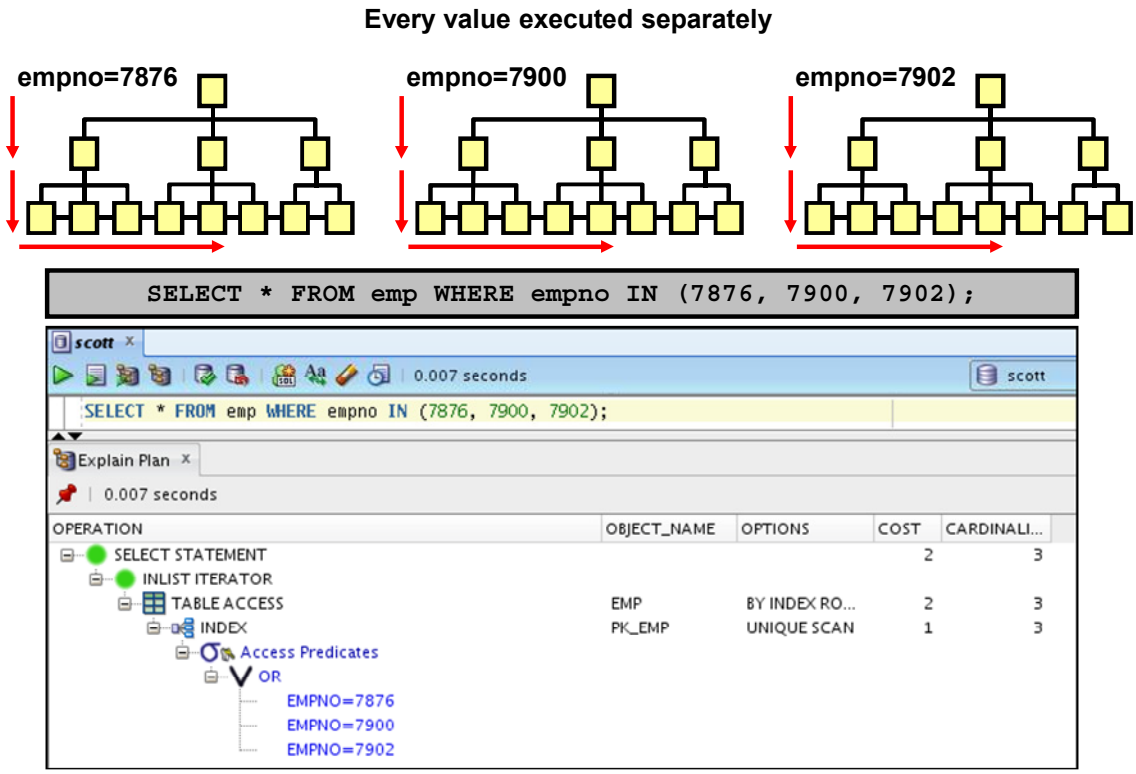
So `BUFFER SORT` uses the buffering mechanism of a traditional sort, but it does not do the sort itself. The system simply buffers the data, in the User Global Area (UGA) or Program Global Area (PGA), to avoid multiple table scans against real data blocks.

The whole sort mechanism is reused, including the swap to disk when not enough sort area memory is available, but without sorting the data.

The difference between a temporary table and a buffer sort is as follows:

- A temporary table uses System Global Area (SGA).
- A buffer sort uses UGA.

Inlist Iterator



It is used when a query contains an `IN` clause with values or multiple equality predicates on the same column linked with `OR`s.

The `INLIST ITERATOR` operator iterates over the enumerated value list, and every value is executed separately.

The execution plan is identical to the result of a statement with an equality clause instead of `IN`, except for one additional step. The extra step occurs when `INLIST ITERATOR` feeds the equality clause with unique values from the list.

You can view this operator as a `FOR LOOP` statement in PL/SQL. In the example in the slide, you iterate the index probe over three values: 7867, 7900, and 7902.

Also, it is a function that uses an index, which is scanned for each value in the list. An alternative handling is `UNION ALL` of each value or a `FILTER` of the values against all the rows; this is significantly more efficient.

The optimizer uses an `INLIST ITERATOR` when an `IN` clause is specified with values, and the optimizer finds a selective index for that column. If there are multiple `OR` clauses using the same index, the optimizer selects this operation rather than `CONCATENATION` or `UNION ALL`, because it is more efficient.

View Operator

```
create view V as select /*+ NO_MERGE */ DEPTNO, sal from emp ;
                        select * from V;
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			3	14
VIEW	V		3	14
TABLE ACCESS	EMP	FULL	3	14

```
select v.*,d.dname from (select DEPTNO, sum(sal) SUM_SAL
from emp group by deptno) v, dept d where v.deptno=d.deptno;
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			7	3
MERGE JOIN			7	3
TABLE ACCESS	DEPT	BY INDEX ...	2	4
INDEX	PK_DEPT	FULL SCAN	1	4
SORT		JOIN	5	3
Access Predicates				
V.DEPTNO=D.DEPT				
Filter Predicates				
V.DEPTNO=D.DEPT				
VIEW			4	3
HASH		GROUP BY	4	3
TABLE ACCESS	EMP	FULL	3	14

Each query produces a variable set of data in the form of a table. A view simply gives a name to this set of data.

When views are referenced in a query, the system can handle them in two ways. If a number of conditions are met, they can be merged into the main query. This means that the view text is rewritten as a join with the other tables in the query. Views can also be left as stand-alone views and selected from directly as in the case of a table. Predicates can also be pushed into or pulled out of the views as long as certain conditions are met.

When a view is not merged, you can see the `VIEW` operator. The view operation is executed separately. All rows from the view are returned, and the next operation can be done.

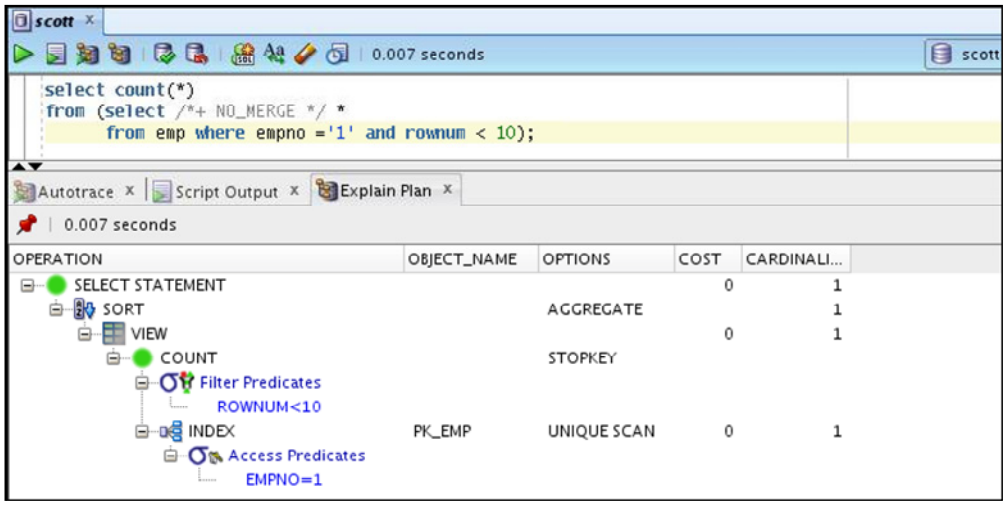
Sometimes a view cannot be merged and must be executed independently in a separate query block. In this case, you can also see the `VIEW` operator in the explain plan. The `VIEW` keyword indicates that the view is executed as a separate query block. For example, views containing `GROUP BY` functions cannot be merged.

The second example in the slide shows a nonmergeable inline view. An inline view is basically a query within the `FROM` clause of your statement.

Basically, this operator collects all rows from a query block before they can be processed by higher operations in the plan.

Count Stop Key Operator

```
SELECT count(*)
FROM (SELECT /*+ NO_MERGE */ *
      FROM emp WHERE empno ='1' and rownum < 10);
```



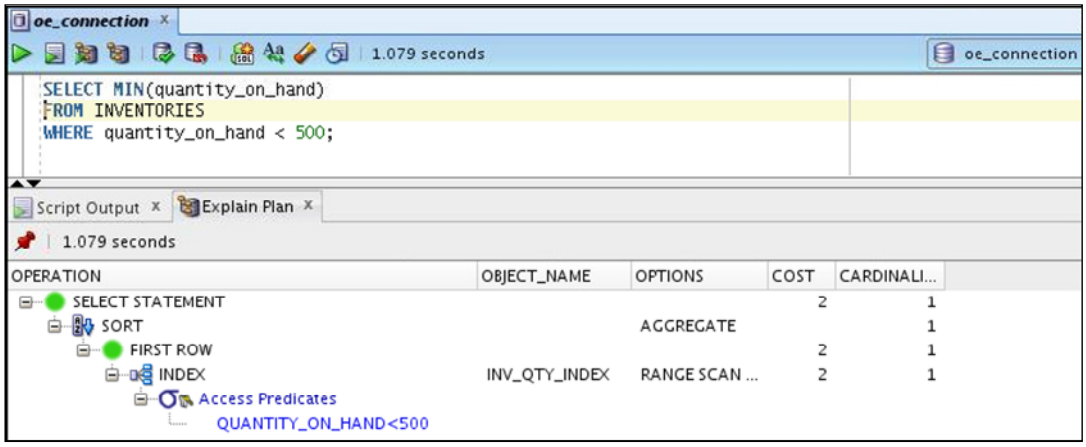
ORACLE

`COUNT STOPKEY` limits the number of rows returned. The limitation is expressed by the `ROWNUM` expression in the `WHERE` clause. It terminates the current operation when the count is reached.

Note: The cost of this operator depends on the number of occurrences of the values you try to retrieve. If the value appears very frequently in the table, the count is reached quickly. If the value is very infrequent, and there are no indexes, the system has to read most of the table's blocks before reaching the count.

Min/Max and First Row Operators

```
SELECT MIN(quantity_on_hand)
FROM INVENTORIES
WHERE quantity_on_hand < 500;
```



ORACLE

`FIRST ROW` retrieves only the first row selected by a query. It stops accessing the data after the first value is returned. This optimization was introduced in Oracle 8*i,* and it works with the index range scan and the index full scan.

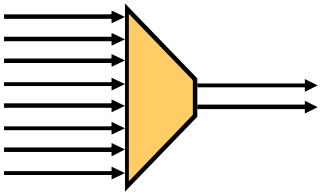
In the example in the slide, it is assumed that there is an index on the `quantity_on_hand` column.

Other N-Array Operations

- FILTER
- CONCATENATION
- UNION ALL/UNION
- INTERSECT
- MINUS

FILTER Operations

- Accepts a set of rows
- Eliminates some of them
- Returns the rest



```
SELECT  deptno, sum(sal) SUM_SAL FROM emp
 GROUP BY deptno HAVING sum(sal) > 9000;
```

SELECT STATEMENT			4	3
FILTER				
Filter Predicates				
SUM(SAL)>9000				
HASH		GROUP BY	4	3
TABLE ACCESS	EMP	FULL	3	14

1

```
SELECT deptno, dname FROM dept d WHERE NOT EXISTS
 (select 1 from emp e where e.deptno=d.deptno);
```

SELECT STATEMENT			6	1
MERGE JOIN		ANTI	6	1
TABLE ACCESS	DEPT	BY INDEX ...	2	4
INDEX	PK_DEPT	FULL SCAN	1	4
SORT		UNIQUE	4	14
Access Predicate:				
E.DEPTNO=D.D				
Filter Predicates				
E.DEPTNO=D.D				
TABLE ACCESS	EMP	FULL	3	14

2

A FILTER operation is any operation that discards rows returned by another step, but is not involved in retrieving the rows itself. All sorts of operations can be filters, including subqueries and single table predicates.

In Example 1 in the slide, FILTER applies to the groups that are created by the GROUP BY operation.

In Example 2 in the slide, FILTER is used almost in the same way as NESTED LOOPS. DEPT is accessed once, and for each row from DEPT, EMP is accessed by its index on DEPTNO. This operation is done as many times as the number of rows in DEPT.

The FILTER operation is applied, for each row, after DEPT rows are fetched. The FILTER discards rows for the inner query `(select 1 from emp e where e.deptno=d.deptno)` and returns at least one row which is TRUE.

Concatenation Operation

```
SELECT * FROM emp WHERE deptno=1 or sal=2;

-----------------------------------------------------------------------
  | Id  | Operation                     | Name      | Rows  | Bytes |
-----------------------------------------------------------------------
  |   0 | SELECT STATEMENT              |           |     8 |   696 |
  |   1 |  CONCATENATION                |           |       |       |
  |   2 |   TABLE ACCESS BY INDEX ROWID| EMP       |     4 |   348 |
  |   3 |    INDEX RANGE SCAN           | I_SAL     |     2 |       |
  |   4 |   TABLE ACCESS BY INDEX ROWID| EMP       |     4 |   348 |
  |   5 |    INDEX RANGE SCAN           | I_DEPTNO  |     2 |       |
-----------------------------------------------------------------------
        Predicate Information (identified by operation id):
        ---------------------------------------------------
                     3 - access("SAL"=2)
                   4 - filter(LNNVL("SAL"=2))
                     5 - access("DEPTNO"=1)
```

ORACLE

CONCATENATION concatenates the rows returned by two or more row sets. This works like UNION ALL and does not remove duplicate rows.

It is used with OR expansions. However, OR does not return duplicate rows, so for each component after the first, it appends a negation of the previous components (LNNVL):

```
CONCATENATION
 - BRANCH 1 - SAL=2
 - BRANCH 2 - DEPTNO = 1 AND NOT row in Branch 1
```

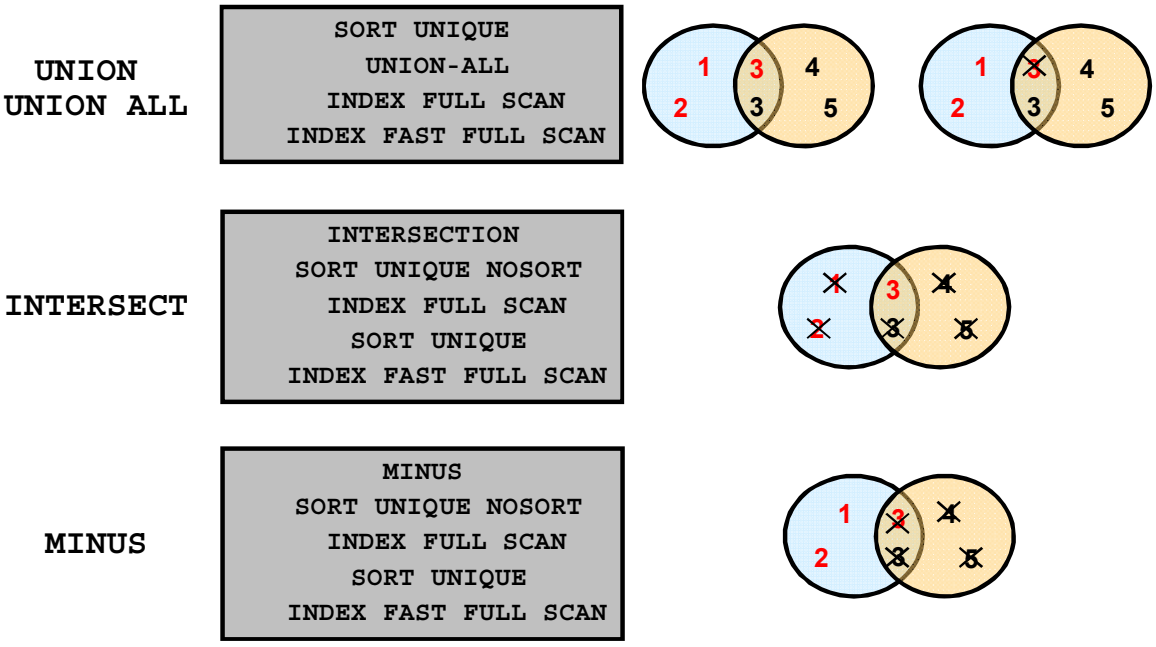
The LNNVL function is generated by the OR clause to process this negation.

The LNNVL() function returns TRUE if the predicate is NULL or FALSE.

So filter (LNNVL(SAL=2)) returns all rows for which SAL != 2 or SAL is NULL.

Note: The explain plan in the slide is from Oracle Database 11*g*, Release 1.

UNION [ALL], INTERSECT, MINUS

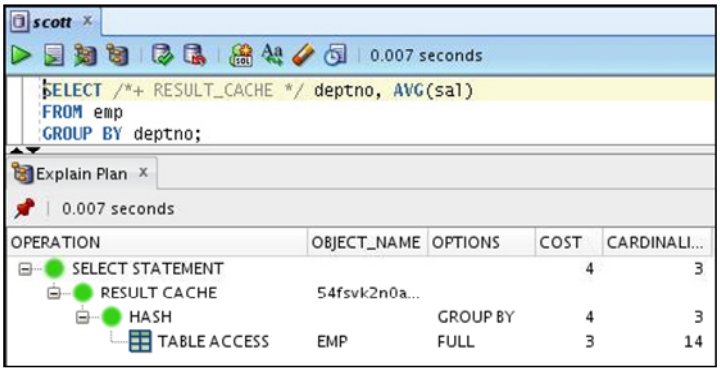


SQL handles duplicate rows with an `ALL` or `DISTINCT` modifier in different places in the language. `ALL` preserves duplicates and `DISTINCT` removes them. Here is a quick description of the possible SQL set operations:

- **`INTERSECTION`:** Operation accepting two sets of rows and returning the intersection of the sets, eliminating duplicates. Subrow sources are executed or optimized individually. This is very similar to sort-merge-join processing: Full rows are sorted and matched.
- **`MINUS`:** Operation accepting two sets of rows and returning rows appearing in the first set, but not in the second, eliminating duplicates. Subrow sources are executed or optimized individually. Similar to `INTERSECT` processing. However, instead of match-and-return, it is match-and-exclude.
- **`UNION`:** Operation accepting two sets of rows and returning the union of the sets, eliminating duplicates. Subrow sources are executed or optimized individually. Rows retrieved are concatenated and sorted to eliminate duplicate rows.
- **`UNION ALL`:** Operation accepting two sets of rows and returning the union of the sets, and not eliminating duplicates. The expensive sort operation is not necessary. Use `UNION ALL` if you know you do not have to deal with duplicates.

Result Cache Operator

```
SELECT /*+ RESULT_CACHE */ deptno, AVG(sal)
FROM emp
GROUP BY deptno;
```



ORACLE

The SQL query result cache enables explicit caching of query result sets and query fragments in database memory. A dedicated memory buffer stored in the shared pool can be used for storing and retrieving the cached results. The query results stored in this cache become invalid when data in the database objects that are accessed by the query is modified. Although the SQL query cache can be used for any query, good candidate statements are the ones that need to access a very high number of rows to return only a fraction of them. This is mostly the case for data-warehousing applications.

If you want to use the query result cache and the `RESULT_CACHE_MODE` initialization parameter is set to `MANUAL`, you must explicitly specify the `RESULT_CACHE` hint in your query. This hint introduces the `ResultCache` operator into the execution plan for the query. When you execute the query, the `ResultCache` operator looks up the result cache memory to check whether the result for the query already exists in the cache. If it exists, the result is retrieved directly out of the cache. If it does not yet exist in the cache, the query is executed, the result is returned as output, and it is also stored in the result cache memory.

If the `RESULT_CACHE_MODE` initialization parameter is set to `FORCE`, and you do not want to store the result of a query in the result cache, you must then use the `NO_RESULT_CACHE` hint in your query.

Quiz

Hash clusters are a better choice than indexed tables or index clusters when a table is queried frequently with equality queries.

a. True
b. False

Answer: a

Quiz

The _____ operator uses a temporary table to store intermediate data.

a. Buffer Sort Operator
b. Inlist
c. Min/Max
d. N-Array

Answer: a

Quiz

The following query uses the _____ operator:

```
SELECT * FROM emp WHERE empno IN (7876, 7900,
7902);
```

a. Buffer Sort Operator
b. Inlist
c. Min/Max
d. N-Array

Answer: b

Quiz

A `FILTER` operation retrieves rows returned by another statement.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned to:

- Describe SQL operators for:
  - Clusters
  - In-List
  - Sorts
  - Filters
  - Set Operations
- Result cache operators

Practice 9: Overview

This practice covers the following topics:

- Using different access paths for better optimization (case 14 to case 16)
- Using the result cache

10

Optimizer Statistics

Objectives

After completing this lesson, you should be able to:

- Describe optimizer statistics
  - Table statistics
  - Index statistics
  - Column statistics (histogram)
  - Column statistics (extended statistics)
  - System statistics
- Gather optimizer statistics
- Set statistic preferences
- Use dynamic sampling
- Manage optimizer statistics
- Discuss optimizer statistics best practices

This lesson explains why statistics are important for the query optimizer and how to gather and use optimizer statistics.

Optimizer Statistics

- Describe the database and the objects in the database
- Information used by the query optimizer to estimate:
  - Selectivity of predicates
  - Cost of each execution plan
  - Access method, join order, and join method
  - CPU and I/O costs
- Types of optimizer statistics:
  - Table statistics
  - Index statistics
  - Column statistics
  - System statistics

Optimizer statistics describe details about the database and the objects in the database. These statistics are used by the query optimizer to select the best execution plan for each SQL statement.

Because the objects in a database change constantly, statistics must be regularly updated so that they accurately describe these database objects. Statistics are maintained automatically by Oracle Database, or you can maintain the optimizer statistics manually by using the `DBMS_STATS` package.

Types of Optimizer Statistics

Most of the optimizer statistics are listed in the slide.

Starting with Oracle Database 10*g*, index statistics are automatically gathered when the index is created or rebuilt.

Note: The statistics mentioned in this slide are optimizer statistics, which are created for query optimization and are stored in the data dictionary. These statistics should not be confused with performance statistics that are visible through `V$` views.

Table Statistics (DBA_TAB_STATISTICS)

- Used to determine:
  - Table access cost
  - Join cardinality
  - Join order
- Some of the table statistics gathered are:
  - Row count (NUM_ROWS)
  - Block count (BLOCKS) *Exact*
  - Average row length (AVG_ROW_LEN)
  - Statistics status (STALE_STATS)

NUM_ROWS

This statistic is the basis for cardinality computations. Row count is especially important if the table is the driving table of a nested loops join, because it defines how many times the inner table is probed.

BLOCKS

This statistic is the number of used data blocks. Block count in combination with DB_FILE_MULTIBLOCK_READ_COUNT gives the base table access cost.

AVG_ROW_LEN

This statistic is the average length of a row in the table in bytes.

STALE_STATS

This statistic tells you if statistics are valid on the corresponding table.

Note: There are three other statistics (EMPTY_BLOCKS, AVE_ROW_LEN, and CHAIN_CNT) that are not used by the optimizer and are not gathered by the DBMS_STATS procedures. If they are required, the ANALYZE command must be used.

Index Statistics (DBA_IND_STATISTICS)

- Used to decide:
  - Full table scan versus index scan
- Statistics gathered are:
  - B*-tree level (BLEVEL) *Exact*
  - Leaf block count (LEAF_BLOCKS)
  - Clustering factor (CLUSTERING_FACTOR)
  - Distinct keys (DISTINCT_KEYS)
  - Average number of leaf blocks in which each distinct value in the index appears (AVG_LEAF_BLOCKS_PER_KEY)
  - Average number of data blocks in the table pointed to by a distinct value in the index (AVG_DATA_BLOCKS_PER_KEY)
  - Number of rows in the index (NUM_ROWS)

In general, to select an index access, the optimizer requires a predicate on the prefix of the index columns. However, in case there is no predicate and all columns referenced in the query are present in an index, the optimizer considers using a full index scan versus a full table scan.

BLEVEL

This statistic is used to calculate the cost of leaf block lookups. It indicates the depth of the index from its root block to its leaf blocks. A depth of "0" indicates that the root block and leaf block are the same.

LEAF_BLOCKS

This statistic is used to calculate the cost of a full index scan.

CLUSTERING_FACTOR

This statistic measures the order of the rows in the table based on the values of the index. If the value is near the number of blocks, the table is very well ordered. In this case, the index entries in a single leaf block tend to point to the rows in the same data blocks. If the value is near the number of rows, the table is very randomly ordered. In this case, it is unlikely that the index entries in the same leaf block point to rows in the same data blocks.

DISTINCT_KEYS

This statistic is the number of distinct indexed values. For indexes that enforce UNIQUE and PRIMARY KEY constraints, this value is the same as the number of rows in the table.

AVG_LEAF_BLOCKS_PER_KEY

This statistic is the average number of leaf blocks in which each distinct value in the index appears, rounded to the nearest integer. For indexes that enforce UNIQUE and PRIMARY KEY constraints, this value is always 1.

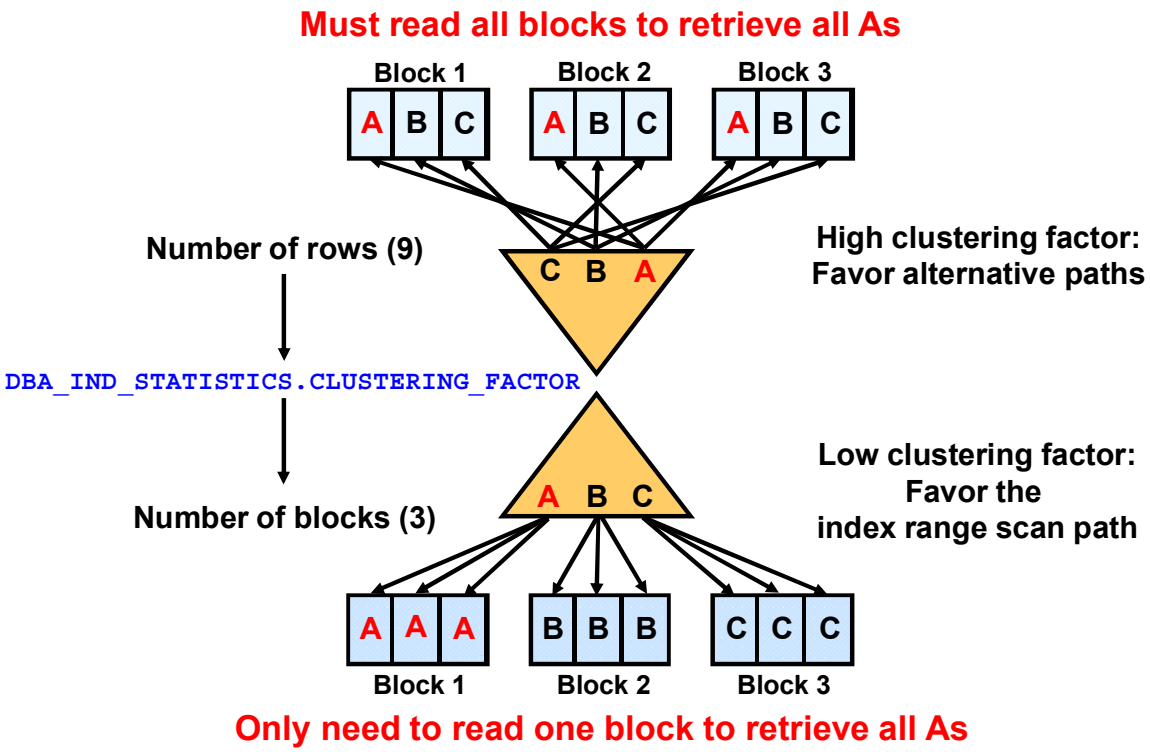
AVG_DATA_BLOCKS_PER_KEY

This statistic is the average number of data blocks in the table that are pointed to by a distinct value in the index, rounded to the nearest integer. This statistic is the average number of data blocks that contain rows with a given value for the indexed columns.

NUM_ROWS

This statistic is the number of rows in the index.

Index Clustering Factor



The system performs I/O by blocks. Therefore, the optimizer's decision to use full table scans is influenced by the percentage of blocks accessed, not rows. When an index range scan is used, each selected index entry points to a block in the table. If each entry points to a different block, the accessed rows and accessed blocks are the same. Consequently, the desired number of rows could be clustered together in a few blocks, or they could be spread out over a larger number of blocks. This is called the index clustering factor.

The cost formula of an index range scan uses the level of the B*-tree, the number of leaf blocks, the index selectivity, and the clustering factor. A clustering factor indicates that the individual rows are concentrated within fewer blocks in the table. A high clustering factor indicates that the individual rows are scattered more randomly across the blocks in the table. Therefore, a high clustering factor means that it costs more to use an index range scan to fetch rows by `ROWID` because more blocks in the table need to be visited to return the data. In real-life scenarios, it appears that the clustering factor plays an important role in determining the cost of an index range scan simply because the number of leaf blocks and the height of the B*-tree are relatively small compared to the clustering factor and table's selectivity.

Note: If you have more than one index on a table, the clustering factor for one index might be small, whereas the clustering factor for another index might be large at the same time. An attempt to reorganize the table to improve the clustering factor for one index can cause degradation of the clustering factor for the other index.

The clustering factor is computed and stored in the `CLUSTERING_FACTOR` column of the `DBA_INDEXES` view when you gather statistics on the index. The way it is computed is relatively easy. You read the index from left to right, and for each indexed entry, you add one to the clustering factor if the corresponding row is located in a different block than the one from the previous row. Based on this algorithm, the smallest possible value for the clustering factor is the number of blocks, and the highest possible value is the number of rows.

The example in the slide shows how the clustering factor can affect cost. Assume the following situation: There is a table with nine rows, there is a nonunique index on col1 for the table, the `c1` column currently stores the values A, B, and C, and the table has only three data blocks.

- **Case 1:** If the same rows in the table are arranged so that the index values are scattered across the table blocks (rather than collocated), the index clustering factor is high.
- **Case 2:** The index clustering factor is low for the rows because they are collocated in the same block for the same value.

Note: For bitmap indexes, the clustering factor is not applicable and is not used.

Column Statistics (DBA_TAB_COL_STATISTICS)

- Count of distinct values of the column (NUM_DISTINCT)
- Low value (LOW_VALUE) *Exact*
- High value (HIGH_VALUE) *Exact*
- Number of nulls (NUM_NULLS)
- Selectivity estimate for nonpopular values (DENSITY)
- Number of histogram buckets (NUM_BUCKETS)
- Type of histogram (HISTOGRAM)

NUM_DISTINCT is used in selectivity calculations; for example, 1/Number of Distinct Values.

LOW_VALUE and HIGH_VALUE: The cost-based optimizer (CBO) assumes uniform distribution of values between low and high values for all data types. These values are used to determine range selectivity.

NUM_NULLS helps with selectivity of nullable columns and the IS NULL and IS NOT NULL predicates.

DENSITY is relevant only for histograms. It is used as the selectivity estimate for nonpopular values. It can be thought of as the probability of finding one particular value in this column. The calculation depends on the histogram type.

NUM_BUCKETS is the number of buckets in histogram for the column.

HISTOGRAM indicates the existence or type of the histogram: NONE, FREQUENCY, HEIGHT BALANCED.

Column Statistics: Histograms

- The optimizer assumes uniform distributions; this may lead to suboptimal access plans in the case of data skew.
- Histograms:
  - Store additional column distribution information
  - Give better selectivity estimates in the case of nonuniform distributions
- With unlimited resources, you could store each different value and the number of rows for that value.
- This becomes unmanageable for a large number of distinct values, and a different approach is used:
  - Frequency histogram (#distinct values ≤ #buckets)
  - Height-balanced histogram (#buckets < #distinct values)
- They are stored in `DBA_TAB_HISTOGRAMS`.

Because a histogram captures the distribution of different values in a column, it yields better selectivity estimates. Having histograms on columns that contain skewed data or values with large variations in the number of duplicates helps the query optimizer generate good selectivity estimates and make better decisions regarding index usage, join orders, and join methods.

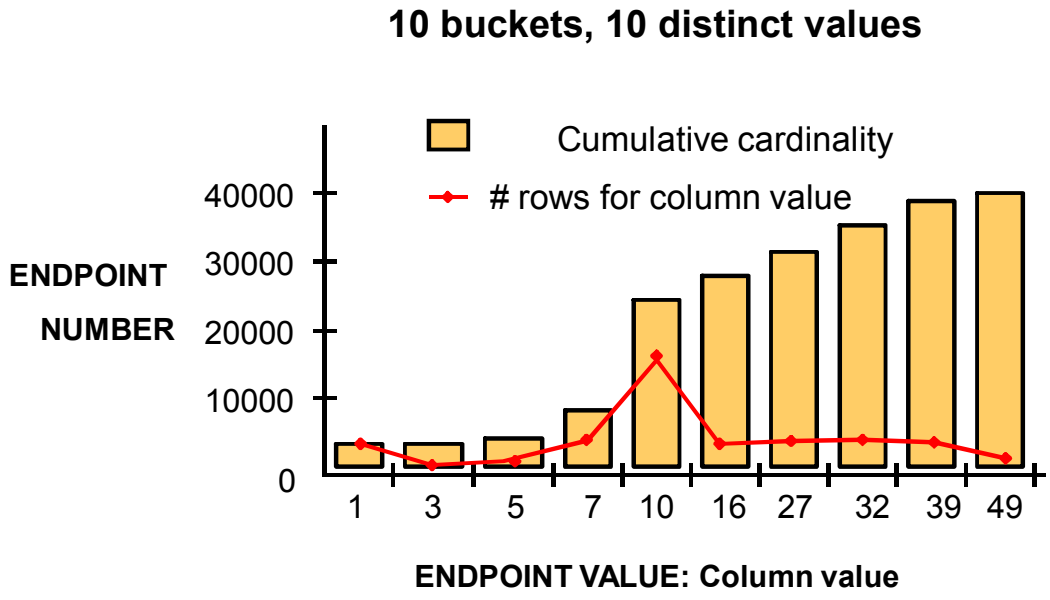
Without histograms, a uniform distribution is assumed. If a histogram is available on a column, the estimator uses it instead of the number of distinct values.

When creating histograms, Oracle Database uses two different types of histogram representations, depending on the number of distinct values found in the corresponding column. When you have a data set with less than 254 distinct values, and the number of histogram buckets is not specified, the system creates a frequency histogram. If the number of distinct values is greater than the required number of histogram buckets, the system creates a height-balanced histogram.

You can find information about histograms in these dictionary views: `DBA_TAB_HISTOGRAMS`, `DBA_PART_HISTOGRAMS`, and `DBA_SUBPART_HISTOGRAMS`.

Note: Gathering histogram statistics is the most resource-consuming operation in gathering statistics.

Frequency Histograms



Distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, 49

Number of rows: 40001

For the example in the slide, assume that you have a column that is populated with 40,001 numbers. You have only 10 distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, and 49. Value 10 is the most popular value with 16,293 occurrences.

When the requested number of buckets equals (or is greater than) the number of distinct values, you can store each different value and record exact cardinality statistics. In this case, in `DBA_TAB_HISTOGRAMS`, the `ENDPOINT_VALUE` column stores the column value and the `ENDPOINT_NUMBER` column stores the cumulative row count, including that column value, because this can avoid some calculation for range scans. The actual row counts are derived from the endpoint values if needed. The actual number of row counts is shown by the curve in the slide for clarity; only the `ENDPOINT_VALUE` and `ENDPOINT_NUMBER` columns are stored in the data dictionary.

Viewing Frequency Histograms

```
BEGIN
 DBMS_STATS.gather_table_STATS (OWNNAME=>'OE', TABNAME=>'INVENTORIES',
       METHOD_OPT => 'FOR COLUMNS SIZE 20 warehouse_id');
END;
```

```
SELECT column_name, num_distinct, num_buckets, histogram
FROM   USER_TAB_COL_STATISTICS
WHERE  table_name = 'INVENTORIES' AND
       column_name = 'WAREHOUSE_ID';

COLUMN_NAME  NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------ ------------ ----------- ---------
WAREHOUSE_ID            9           9 FREQUENCY
```

```
SELECT endpoint_number, endpoint_value
FROM   USER_HISTOGRAMS
WHERE  table_name = 'INVENTORIES' and column_name = 'WAREHOUSE_ID'
ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
36                           1
213                          2
261                          3
…
```

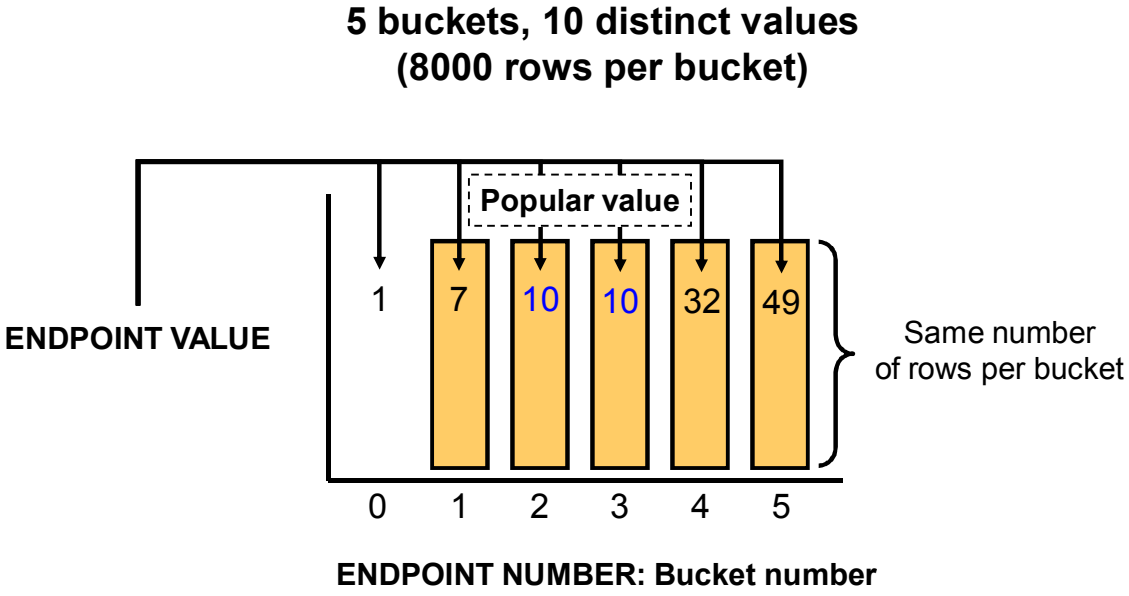

The example in the slide shows you how to view a frequency histogram. Because the number of distinct values in the WAREHOUSE_ID column of the INVENTORIES table is nine, and the number of requested buckets is 20, the system automatically creates a frequency histogram with nine buckets. You can view this information in the USER_TAB_COL_STATISTICS view.

To view the histogram itself, you can query the USER_HISTOGRAMS view. You can see both ENDPOINT_NUMBER column that corresponds to the cumulative frequency of the corresponding ENDPOINT_VALUE column, which represents, in this case, the actual value of the column data.

In this case, the warehouse_id is 1 and there are 36 rows with warehouse_id = 1. There are 177 rows with warehouse_id = 2, and the sum of rows so far (36+177) is therefore the cumulative frequency of 213.

Note: The DBMS_STATS package is covered later in the lesson.

Height-Balanced Histograms



Distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, 49

Number of rows: 40001

In a height-balanced histogram, the ordered column values are divided into bands so that each band contains approximately the same number of rows. The histogram tells you values of the endpoints of each band. For the example in the slide, assume that you have a column that is populated with 40,001 numbers. There will be 8,000 values in each band. You have only 10 distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, and 49. Value 10 is the most popular value with 16,293 occurrences. When the number of buckets is less than the number of distinct values, `ENDPOINT_NUMBER` records the bucket number and `ENDPOINT_VALUE` records the column value that corresponds to this endpoint. In the example, the number of rows per bucket is one-fifth of the total number of rows; that is 8000. Based on this assumption, value 10 appears between 8000 and 24000 times. So you are sure that value 10 is a popular value.

This type of histogram is good for equality predicates on popular value and for range predicates.

The number of rows per bucket is not recorded because it can be derived from the total number of values and the fact that all the buckets contain an equal number of values. In this example, value 10 is a popular value because it spans multiple endpoint values. To save space, the histogram does not actually store duplicated buckets. For the example in the slide, bucket 2 (with endpoint value 10) would not be recorded in `DBA_TAB_HISTOGRAMS` for that reason.

Viewing Height-Balanced Histograms

```
BEGIN
  DBMS_STATS.gather_table_STATS(OWNNAME =>'OE', TABNAME=>'INVENTORIES',
  METHOD_OPT => 'FOR COLUMNS SIZE 10 quantity_on_hand');
END;
```

```
SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HEIGHT BALANCED
```

```
SELECT endpoint_number, endpoint_value
FROM USER_HISTOGRAMS
WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
…
```


The example in the slide shows you how to view a height-balanced histogram. Because the number of distinct values in the QUANTITY_ON_HAND column of the INVENTORIES table is 237, and the number of requested buckets is 10, the system automatically creates a height-balanced histogram with 10 buckets. You can view this information in the USER_TAB_COL_STATISTICS view.

To view the histogram itself, you can query the USER_HISTOGRAMS view. You can see that ENDPOINT_NUMBER corresponds to the bucket number, and ENDPOINT_VALUE corresponds to values of the endpoint.

Note: The DBMS_STATS package is covered later in the lesson.

Best Practices: Histogram

- Histograms are useful when you have a high degree of skew in the column distribution.
- Histograms are *not* useful for:
  - Columns which do not appear in the `WHERE` or `JOIN` clauses
  - Columns with uniform distributions
  - Equality predicates with unique columns
- The maximum number of buckets is the least (254, # distinct values). If possible, frequency histograms are preferred.
- Do not use histograms unless they substantially improve performance.

Histograms are useful only when they reflect the current data distribution of a given column. The data in the column can change as long as the distribution remains constant. If the data distribution of a column changes frequently, you must recompute its histogram frequently.

Histograms are useful when you have a high degree of data skew in the columns for which you want to create histograms.

However, there is no need to create histograms for columns that do not appear in a `WHERE` clause of a SQL statement. Similarly, there is no need to create histograms for columns with uniform distribution.

In addition, for columns declared as `UNIQUE`, histograms are useless because the selectivity is obvious. Also, the maximum number of buckets is 254, which can be lower depending on the actual number of distinct column values. If possible, frequency histograms are preferred. Histograms can affect performance and should be used only when they substantially improve query plans. For uniformly distributed data, the optimizer can make fairly accurate guesses about the cost of executing a particular statement without the use of histograms.

Note: Character columns have some exceptional behavior because histogram data is stored only for the first 32 bytes of any string.

Best Practices: Histogram

- Set METHOD_OPT to FOR ALL COLUMNS AUTO.
- Use TRUNCATE instead of dropping and re-creating the same table if you need to remove all rows from a table.
- When upgrading to 11*g*, use the same histograms used initially in earlier releases.
- If incorrect cardinality / selectivity is observed in an execution plan, check to see if a histogram can resolve the problem.
- Make sure statistics for objects are collected at the highest sample size you can afford and see if the plan improves.
- In earlier releases, if a query uses binds or binds are not representative of future executions, we should not consider histograms to avoid bind peeking. In 11*g*, adaptive cursor sharing resolves bind/histogram issues.

ORACLE

When gathering statistics, histograms are specified by using the METHOD_OPT argument of the DBMS_STATS gathering procedures. Oracle recommends setting METHOD_OPT to FOR ALL COLUMNS SIZE AUTO. With this setting, Oracle Database automatically determines which columns require histograms and the number of buckets (size) of each histogram. Column usage history is collected and monitored at sys.col_usage$. This information is needed by dbms_stats to identify candidate columns on which to build histograms when METHOD_OPT is set to FOR ALL COLUMNS SIZE AUTO.

Note: When upgrading, in some cases, the effect of a histogram is adverse to the generation of a better plan (especially in the presence of bind variables combined with small or AUTO sample sizes). Again, you may want to initially set this parameter to its release value before the upgrade, and later adjust to your release default value after the upgrade.

If you need to remove all rows from a table when using DBMS_STATS, use TRUNCATE instead of dropping and re-creating the same table. When you drop a table, workload information used by the auto-histogram gathering feature and saved statistics history used by the RESTORE_*_STATS procedures are lost. Without this data, these features do not function properly.

Note: Statistics gathering is covered later in the lesson.

When upgrading to 11*g*, use the same histograms used in earlier release initially. For more information, see MOS Note 465787.1, “How to: Manage CBO Statistics During an Upgrade to 10*g* or 11*g*.”

If incorrect cardinality / selectivity is observed in an execution plan, check to see if a histogram can resolve the problem. Review the example introduced in the slide titled “Execution Plan Interpretation: Example 3” in the lesson titled “Understanding Serial Execution Plans.”

Make sure statistics for objects are collected at the highest sample size you can afford and see if the plan improves. In 11*g*, the default sample size is 100 percent.

In earlier releases, if a query uses binds or binds are not representative of future executions, we should not consider histograms to avoid bind peeking. Instead, SQL plan baselines, profiles, stored outlines, or hints are considered. In 11*g*, adaptive cursor sharing can resolve bind/histogram issues.

Column Statistics: Extended Statistics

- The optimizer poorly estimates selectivity *on Highly Correlated Column Predicates:*
  - Columns have values that are highly correlated.
  - Actual selectivity is often much lower or higher than the optimizer estimates. For example,
    ```
    WHERE cust_state_province = 'CA'
    AND country_id=52775;
    ```
- The optimizer poorly estimates *Expression on Columns:*
  - `WHERE upper(model)='MODEL'`
  - When a function is applied to a column in the `WHERE` clause, the optimizer has no way of knowing how that function affects the selectivity of the column.
- In these cases, a group of columns within a table or an expression on a column can be gathered to obtain a more accurate selectivity value.

With Oracle Database 10*g*, the query optimizer takes into account the correlation between columns when computing the selectivity of multiple predicates in the following limited cases:

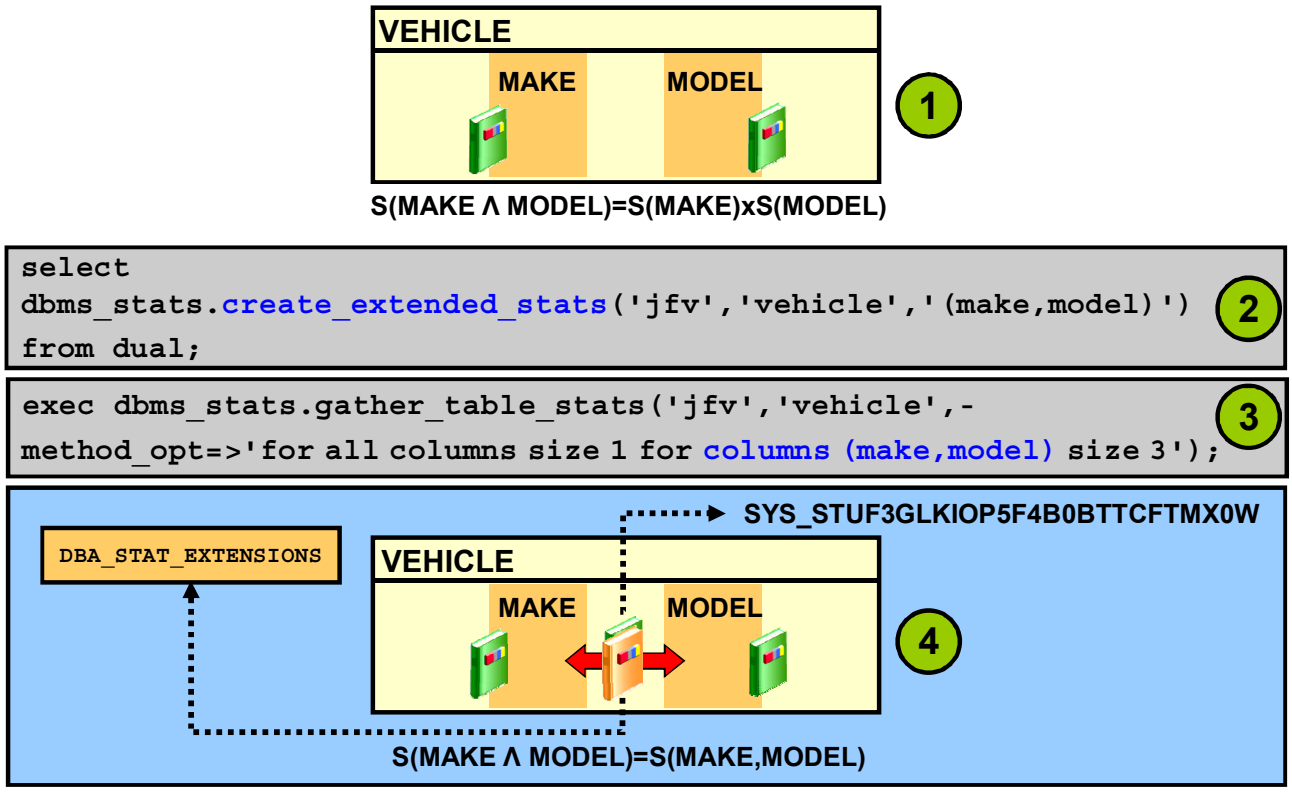
- If all columns of a conjunctive predicate match all columns of a concatenated index key, and the predicates are equalities used in equijoins, the optimizer uses the number of distinct keys (NDK) in the index for estimating selectivity, as 1/NDK.
- When `DYNAMIC_SAMPLING` is set to level `4`, the query optimizer uses dynamic sampling to estimate the selectivity of complex predicates involving several columns from the same table. However, the sample size is very small and increases parsing time. As a result, the sample is likely to be statistically inaccurate and may cause more harm than good.

In all other cases, the optimizer assumes that the values of columns used in a complex predicate are independent of each other. It estimates the selectivity of a conjunctive predicate by multiplying the selectivity of individual predicates. This approach results in underestimation of the selectivity if there is a correlation between the columns. To circumvent this issue, Oracle Database 11*g* allows you to collect, store, and use the following statistics to capture functional dependency between two or more columns (also called groups of columns): number of distinct values, number of nulls, frequency histograms, and density.

When a function is applied to a column in the `WHERE` clause of a query (function(col1)=constant), the optimizer has no way of knowing how that function affects the selectivity of the column. The optimizer assumes a static selectivity value of 1 percent. This approach almost never has the correct selectivity, and it may cause the optimizer to produce suboptimal plans. By gathering expression statistics on the expression function(col1), the optimizer obtains a more accurate selectivity value.

Oracle Database 11*g* can also gather statistics on a group of columns within a table or an expression on a column to obtain a more accurate selectivity value.

Multicolumn Statistics



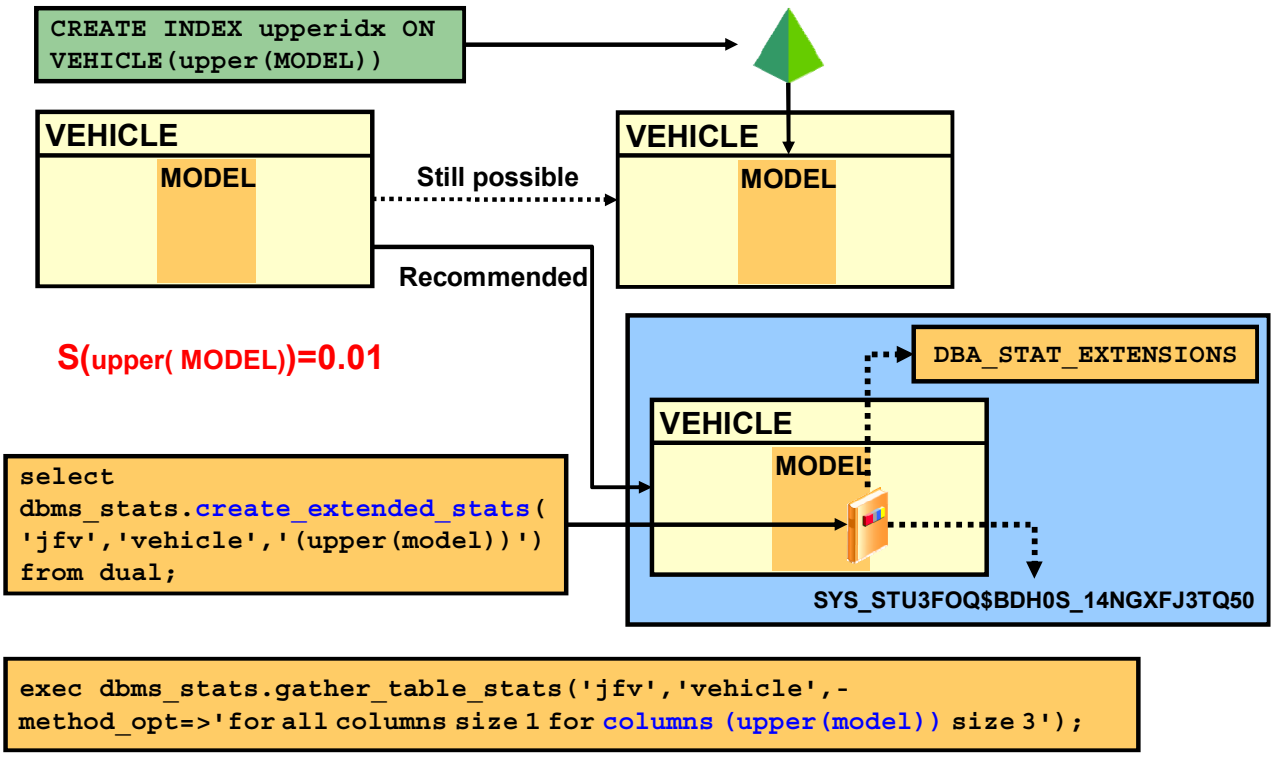
In the slide example, consider a VEHICLE table in which you store information about cars. The MAKE and MODEL columns are highly correlated, in that MODEL determines MAKE. This is a strong dependency, and both columns should be considered by the optimizer as highly correlated. You can signal that correlation to the optimizer by using the CREATE_EXTENDED_STATS function, and then compute the statistics for all columns (including the ones for the correlated groups that you created).

The optimizer uses only multicolumn statistics with equality predicates.

Note:

- The CREATE_EXTENDED_STATS function returns a virtual hidden column name, such as SYS_STUW_5RHLX443AN1ZCLPE_GLE4.
- Based on the example in the slide, the name can be determined by using the following SQL statement:
```
select
dbms_stats.show_extended_stats_name('jfv','vehicle','(make,model
)') from dual
```
- After you create the statistics extensions, you can retrieve them by using the ALL|DBA|USER_STAT_EXTENSIONS views.

Expression Statistics



Predicates involving expressions on columns are a significant issue for the query optimizer. When computing selectivity on predicates of the form *function(Column) = constant*, the optimizer assumes a static selectivity value of 1 percent. This approach almost never has the correct selectivity, and it may cause the optimizer to produce suboptimal plans.

The query optimizer has been extended to better handle such predicates in limited cases where functions preserve the data distribution characteristics of the column and thus allow the optimizer to use the columns statistics. An example of such a function is `TO_NUMBER`.

Further enhancements have been made to evaluate built-in functions during query optimization to derive better selectivity by using dynamic sampling. Finally, the optimizer collects statistics on virtual columns created to support function-based indexes.

However, these solutions are either limited to a certain class of functions or they work only for expressions used to create function-based indexes. With the expression statistics in Oracle Database 11*g*, you can use a more general solution that includes arbitrary user-defined functions and does not depend on the presence of function-based indexes. As shown in the example in the slide, this feature relies on the virtual column infrastructure to create statistics on expressions of columns.

System Statistics

- System statistics are used to estimate:
  - I/O performance and utilization
  - CPU performance and utilization
- System statistics enable the query optimizer to estimate I/O and CPU costs more accurately, enabling the query optimizer to choose a better execution plan.
- Procedures:
  - `DBMS_STATS.GATHER_SYSTEM_STATS`
  - `DBMS_STATS.SET_SYSTEM_STATS`
  - `DBMS_STATS.GET_SYSTEM_STATS`

System statistics allow the optimizer to consider a system's I/O and CPU performance and utilization. For each candidate plan, the optimizer computes estimates for I/O and CPU costs. It is important to know the system characteristics to select the most efficient plan with optimal proportion between I/O and CPU costs. System CPU and I/O characteristics depend on many factors and do not stay constant all the time.

System statistics are gathered in a user-defined time frame with the `DBMS_STATS.GATHER_SYSTEM_STATS` routine. You can also set system statistics values explicitly by using `DBMS_STATS.SET_SYSTEM_STATS`. Use `DBMS_STATS.GET_SYSTEM_STATS` to verify system statistics.

System Statistics: Example

Viewing System Statistics:

```
SELECT * FROM sys.aux_stats$;

SNAME                 PNAME               PVAL1      PVAL2
-------------------- ------------------ ---------- --------------------
SYSSTATS_INFO        STATUS                        COMPLETED
SYSSTATS_INFO        DSTART                        08-09-2001 16:40
SYSSTATS_INFO        DSTOP                         08-09-2001 16:42
SYSSTATS_INFO        FLAGS                      0
SYSSTATS_MAIN        SREADTIM                7.581
SYSSTATS_MAIN        MREADTIM               56.842
SYSSTATS_MAIN        CPUSPEED                  117
SYSSTATS_MAIN        MBRC                        9
```


The slide shows an example of the optimizer system statistics.

- **sreadtim:** Single block read time is the average time to read a single block randomly.
- **mreadtim:** Multiblock read is the average time to read a multiblock sequentially.
- **cpuspeed:** Represents workload CPU speed. CPU speed is the average number of CPU cycles in each second.
- **mbrc:** Multiblock count is the average multiblock read count sequentially.

For more information on these statistics, see the following link:

http://download.oracle.com/docs/cd/E11882_01/server.112/e16638/stats.htm#PFGRF94743 (Table 13-7 Optimizer System Statistics in the `DBMS_STAT` Package)

Best Practices: System Statistics

- System statistics must be gathered on a regular basis; this does not invalidate cached plans.
- Gathering system statistics equals analyzing system activity for a specified period of time.
- When gathering the optimizer system statistics:
  - It is highly recommended that you gather system statistics during normal workload for several hours.
  - If no real workload is available, you can also gather `NORWORKLOAD` statistics.

Using system statistics management routines, you can capture statistics in the interval of time when the system has the most common workload. For example, database applications can process online transaction processing (OLTP) transactions during the day and run online analytical processing (OLAP) reports at night. You can gather statistics for both states and activate appropriate OLTP or OLAP statistics when needed. This allows the optimizer to generate relevant costs with respect to the available system resource plans. When the system generates system statistics, it analyzes system activity in a specified period of time. Unlike the table, index, or column statistics, the system does not invalidate already parsed SQL statements when system statistics get updated. All new SQL statements are parsed using new statistics.

It is highly recommended that you gather system statistics. System statistics are gathered in a user-defined time frame with the `DBMS_STATS.GATHER_SYSTEM_STATS` routine. You can also set system statistics values explicitly by using `DBMS_STATS.SET_SYSTEM_STATS`. Use `DBMS_STATS.GET_SYSTEM_STATS` to verify system statistics.

When you use the GATHER_SYSTEM_STATS procedure, you should specify the GATHERING_MODE parameter:

- NOWORKLOAD: This is the default. This mode captures characteristics of the I/O system. Gathering may take a few minutes and depends on the size of the database. During this period the system estimates the average read seek time and transfer speed for the I/O system. This mode is suitable for all workloads. It is recommended that you run GATHER_SYSTEM_STATS ('noworkload') after you create the database and tablespaces.
- INTERVAL: Captures system activity during a specified interval. This works in combination with the interval parameter that specifies the amount of time for the capture. You should provide an interval value in minutes, after which system statistics are created or updated in the dictionary or a staging table. You can use GATHER_SYSTEM_STATS (gathering_mode=>'STOP') to stop gathering earlier than scheduled.
- START | STOP: Captures system activity during specified start and stop times, and refreshes the dictionary or a staging table with statistics for the elapsed period.

Note: Since Oracle Database 10*g*, Release 2, the system automatically gathers essential parts of system statistics at startup.

Gathering System Statistics: Automatic Collection Example

First day

```
EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS(
interval => 120,
stattab => 'mystats', statid => 'OLTP');
```

First night

```
EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS(
interval => 120,
stattab => 'mystats', statid => 'OLAP');
```

Next days

```
EXECUTE DBMS_STATS.IMPORT_SYSTEM_STATS(
stattab => 'mystats', statid => 'OLTP');
```

Next nights

```
EXECUTE DBMS_STATS.IMPORT_SYSTEM_STATS(
stattab => 'mystats', statid => 'OLAP');
```


The example in the slide shows database applications processing OLTP transactions during the day and running reports at night.

First, system statistics must be collected during the day. In this example, gathering ends after 120 minutes and is stored in the `mystats` table.

Then, system statistics are collected during the night. Gathering ends after 120 minutes and is stored in the `mystats` table.

Generally, the syntax in the slide is used to gather system statistics. Before invoking the `GATHER_SYSTEM_STATS` procedure with the `INTERVAL` parameter specified, you must activate job processes by using a command, such as `SQL> alter system set job_queue_processes = 1;`.

Note: In Oracle Database 11*g,* Release 2, the default value of `job_queue_processes` is 1000. You can also invoke the same procedure with different arguments to enable manual gathering instead of using jobs.

If appropriate, you can switch between the statistics gathered. Note that it is possible to automate this process by submitting a job to update the dictionary with appropriate statistics. During the day, a job may import the OLTP statistics for the daytime run, and during the night, another job imports the OLAP statistics for the nighttime run.

Gathering System Statistics: Manual Collection Example

- Start manual system statistics collection in the data dictionary:

```
EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS( -
gathering_mode => 'START');
```

- Generate the workload.
- End the collection of system statistics:

```
EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS( -
gathering_mode => 'STOP');
```


The example in the previous slide shows how to collect system statistics with jobs by using the internal parameter of the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure. To collect system statistics manually, another parameter of this procedure can be used, as shown in the slide.

First, you must start the system statistics collection, and then you can end the collection process at any time after you are certain that a representative workload has been generated on the instance.

The example collects system statistics and stores them directly in the data dictionary.

Gathering Statistics: Overview

- Automatic statistics gathering
  - `gather_stats_prog` automated task
- Manual statistics gathering
  - `DBMS_STATS` package
- Dynamic sampling
- When statistics are missing
- Vendor-recommended gathering

Selectivity:	
Equality	**1%**
Inequality	**5%**
Other predicates	**5%**
Table row length	**20**
# of index leaf blocks	**25**
# of distinct values	**100**
Table cardinality	**100**
Remote table cardinality	**2000**

ORACLE

Oracle Database provides several mechanisms to gather statistics. These are discussed in more detail in the subsequent slides. It is recommended that you use automatic statistics gathering for objects.

Note: When the system encounters a table with missing statistics, it dynamically gathers the necessary statistics needed by the optimizer. However, for certain types of tables (including remote tables and external tables), it does not perform dynamic sampling. In those cases and also when dynamic sampling has been disabled, the optimizer uses default values for its statistics.

Automatic Statistics Gathering

- Oracle's recommended method for collecting statistics
- Oracle Database 11*g* automates optimizer statistics collection:
  - Statistics are gathered automatically only on all database objects that have no statistics or have stale statistics (> 10% of rows modified)
  - The `gather_stats_prog` automated task is used for statistics collection and maintenance.
- Automated statistics collection:
  - Eliminates need for manual statistics collection
  - Significantly reduces the chances of poor execution plans
- The Statistic Preferences feature is available in Oracle 11*g* for some objects that require statistics collection settings that are different from the database default.

Oracle Database 11*g* automatically gathers statistics on all database objects and maintains those statistics in a regularly scheduled maintenance job. Automated statistics collection eliminates all of the manual tasks that are associated with managing the optimizer statistics, and significantly reduces the chances of getting poor execution plans because of missing or stale statistics. DML monitoring is enabled by default and is used by the Automatic Statistics Gathering feature to determine which objects have stale statistics as well as their degree of staleness. This information is used to identify candidates for statistics update.

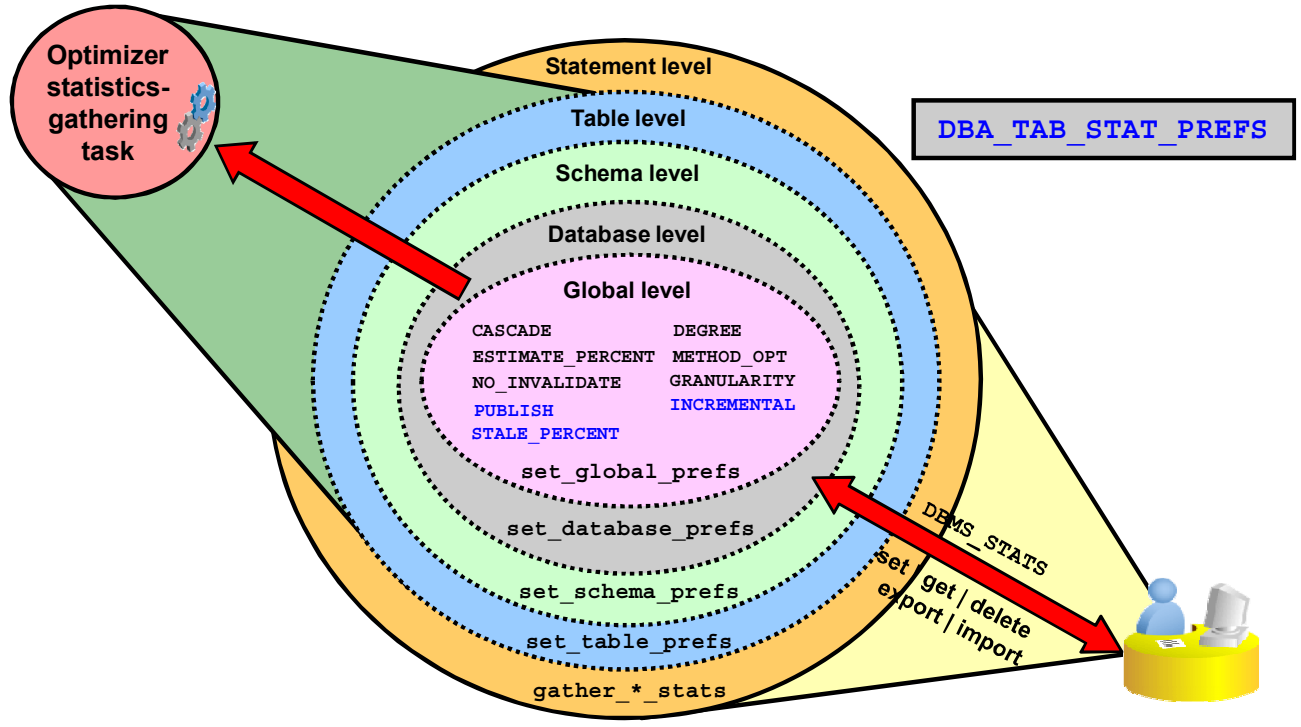
Optimizer statistics are automatically gathered with the automatic optimizer statistics-gathering task. This task does the following:

- Gathers statistics on all objects in the database that have missing or stale statistics in the predefined maintenance window
- Determines the appropriate sample size for each object
- Creates histograms as required

The task is created automatically at database creation time. System statistics are *not* gathered by the automatic optimizer statistics-gathering task.

Note: The Statistic Preferences feature is available in Oracle Database 11*g* for some objects that require statistics collection settings that differ from the database default.

Statistic Preferences: Overview




```
exec dbms_stats.set_table_prefs('SH','SALES','STALE_PERCENT','13');
```


The Automated Statistics Gathering feature was introduced in Oracle Database 10*g*, Release 1, to reduce the burden of maintaining optimizer statistics. However, there were cases where you had to disable it and run your own scripts instead. One reason was the lack of object-level control. Whenever you found a small subset of objects for which the default gather statistics options did not work well, you had to lock the statistics and analyze them separately by using your own options. For example, the feature that automatically tries to determine adequate sample size (`ESTIMATE_PERCENT=AUTO_SAMPLE_SIZE`) does not work well against columns that contain data with very high frequency skews. The only way to get around this issue is to manually specify the sample size in your own script.

The Statistic Preferences feature in Oracle Database 11*g* introduces flexibility so that you can rely more on the automated statistics-gathering feature to maintain the optimizer statistics when some objects require settings that differ from the database default.

This feature allows you to associate the statistics-gathering options that override the default behavior of the `GATHER_*_STATS` procedures and the automated Optimizer Statistics Gathering task at the object or schema level. You can use the `DBMS_STATS` package to manage the statistics-gathering options shown in the slide.

You can set, get, delete, export, and import those preferences at the table, schema, database, and global levels. Global preferences are used for tables that do not have preferences, whereas database preferences are used to set preferences on all tables. The preference values that are specified in various ways take precedence from the outer circles to the inner ones (as shown in the slide).

In the graphic in the slide, the last three highlighted options are new in Oracle Database 11*g*, Release 1:

- `CASCADE` gathers statistics on the indexes as well. Index statistics gathering is not parallelized.
- `ESTIMATE_PERCENT` is the estimated percentage of rows used to compute statistics (Null means all rows): The valid range is [0.000001,100]. Use the constant `DBMS_STATS.AUTO_SAMPLE_SIZE` to have the system determine the appropriate sample size for good statistics. This is the recommended default.
- `NO_INVALIDATE` controls the invalidation of dependent cursors of the tables for which statistics are being gathered. It does not invalidate the dependent cursors if set to `TRUE`. The procedure invalidates the dependent cursors immediately if set to `FALSE`. Use `DBMS_STATS.AUTO_INVALIDATE` to have the system decide when to invalidate dependent cursors. This is the default.
- PUBLISH is used to decide whether to publish the statistics to the dictionary or to store them in a pending area before publishing them.
- `STALE_PERCENT` is used to determine the threshold level at which an object is considered to have stale statistics. The value is a percentage of rows that were modified since the last statistics gathering. The example changes the 10 percent default to 13 percent only for `SH.SALES`.
- `DEGREE` determines the degree of parallelism used to compute statistics. The default for degree is null, which means use the table default value specified by the `DEGREE` clause in the `CREATE TABLE` or `ALTER TABLE` statement. Use the constant `DBMS_STATS.DEFAULT_DEGREE` to specify the default value based on the initialization parameters. The `AUTO_DEGREE` value automatically determines the degree of parallelism as either 1 (serial execution) or `DEFAULT_DEGREE` (the system default value based on the number of CPUs and initialization parameters), depending on the size of the object.
- `METHOD_OPT` is a SQL string used to collect histogram statistics. The default value is `FOR ALL COLUMNS SIZE AUTO`.
- `GRANULARITY` is the granularity of statistics to collect for partitioned tables.
- `INCREMENTAL` is used to gather global statistics on partitioned tables in an incremental way.

It is important to note that you can change default values for the parameters by using the `DBMS_STATS.SET_GLOBAL_PREFS` procedure.

Note: You can describe all effective statistics preference settings for all relevant tables by using the `DBA_TAB_STAT_PREFS` view.

Manual Statistics Gathering

You can use Enterprise Manager and the DBMS_STATS package to:

- Generate and manage statistics for use by the optimizer:
  - Gather/Modify
  - View/Name
  - Export/Import
  - Delete/Lock
- Gather statistics on:
  - Indexes, tables, columns, partitions
  - Object, schema, or database
- Gather statistics either serially or in parallel
- gather/set system statistics (currently not possible in EM)

Both Enterprise Manager and the DBMS_STATS package enable you to manually generate and manage statistics for the optimizer. You can use the DBMS_STATS package to gather, modify, view, export, import, lock, and delete statistics. You can also use this package to identify or name gathered statistics. You can gather statistics on indexes, tables, columns, and partitions at various granularity: object, schema, and database level.

DBMS_STATS gathers only statistics needed for optimization; it does not gather other statistics. For example, the table statistics that are gathered by DBMS_STATS include the number of rows, number of blocks currently containing data, and average row length, but not the number of chained rows, average free space, or number of unused data blocks.

Note: Do not use the COMPUTE and ESTIMATE clauses of the ANALYZE statement to collect optimizer statistics. These clauses are supported solely for backward compatibility and may be removed in a future release. The DBMS_STATS package collects a broader, more accurate set of statistics, and gathers statistics more efficiently. You may continue to use the ANALYZE statement for other purposes that are not related to the optimizer statistics collection, such as the following:

- To use the VALIDATE or LIST CHAINED ROWS clauses
- To collect information on free list blocks

When to Gather Statistics Manually

- Rely mostly on automatic statistics collection:
  - Change the frequency of automatic statistics collection to meet your needs.
  - Remember that `STATISTICS_LEVEL` should be set to `TYPICAL` or `ALL` for automatic statistics collection to work properly.
- Gather statistics manually for:
  - Objects that are volatile
  - Objects modified in batch operations (Gather statistics as part of the batch operation.)
  - External tables, system statistics, fixed objects
  - New objects (Gather statistics right after object creation.)

The automatic statistics-gathering mechanism gathers statistics on schema objects in the database for which statistics are absent or stale. It is important to determine when and how often to gather new statistics. The default gathering interval is nightly, but you can change this interval to suit your business needs. You can do so by changing the characteristics of your maintenance windows. Some cases may require manual statistics gathering. For example, the statistics on tables that are significantly modified during the day may become stale. There are typically two types of such objects:

- Volatile tables that are modified significantly during the course of the day
- Objects that are the target of large bulk loads that add 10 percent or more to the object's total size between statistics-gathering intervals

For external tables, statistics are collected manually only by using `GATHER_TABLE_STATS`. Because sampling on external tables is not supported, the `ESTIMATE_PERCENT` option should be explicitly set to null. Because data manipulation is not allowed against external tables, it is sufficient to analyze external tables when the corresponding file changes. Other areas in which statistics need to be manually gathered are the system statistics and fixed objects, such as the dynamic performance tables. These statistics are not automatically gathered.

Manual Statistics Collection: Factors

- Monitor objects for DMLs.
- Determine the correct sample sizes.
- Determine the degree of parallelism.
- Determine if histograms should be used.
- Determine the cascading effects on indexes.
- Procedures to use in `DBMS_STATS`:
  - `GATHER_INDEX_STATS`
  - `GATHER_TABLE_STATS`
  - `GATHER_SCHEMA_STATS`
  - `GATHER_DICTIONARY_STATS`
  - `GATHER_DATABASE_STATS`
  - `GATHER_SYSTEM_STATS`

When you manually gather optimizer statistics, you must pay special attention to the following factors:

- Monitoring objects for mass DML operations and gathering statistics if necessary
- Determining the correct sample sizes
- Determining the degree of parallelism to speed up queries on large objects
- Determining if histograms should be created on columns with skewed data
- Determining whether changes on objects cascade to any dependent indexes

Gathering Object Statistics: Example

```
dbms_stats.gather_table_stats
('sh'                 -- schema
,'customers'          -- table
, null                -- partition
, 20                  -- sample size(%)
, false               -- block sample?
,'for all columns'    -- column spec
, 4                   -- degree of parallelism
,'default'            -- granularity
, true );             -- cascade to indexes
```

```
dbms_stats.set_param('CASCADE',
                     'DBMS_STATS.AUTO_CASCADE');
dbms_stats.set_param('ESTIMATE_PERCENT','5');
dbms_stats.set_param('DEGREE','NULL');
```


The first example in the slide uses the DBMS_STATS package to gather statistics on the CUSTOMERS table of the SH schema. It uses some of the options discussed in the previous slides.

You can use the SET_PARAM procedure in DBMS_STATS to set default values for parameters of all DBMS_STATS procedures. The second example in the slide shows this usage. You can also use the GET_PARAM function to get the current default value of a parameter.

Note: Granularity of statistics to collect is pertinent only if the table is partitioned. This parameter determines at which level statistics should be gathered. This can be at the partition, subpartition, or table level.

Best Practices: Object Statistics

- Ensure that all objects (tables and indexes) have statistics gathered.
- Use a sample size that is large enough if feasible.
- Gather optimizer statistics during periods of low activity.
- If partitions are in use, gather global statistics if possible.
- Use Oracle Database11*g* pending statistics to verify effect of new statistics when tuning to minimize risk.
- Gather statistics after data has been loaded (>10% added), but before indexes are created.

Review the following best practices:

- Ensure that all objects have statistics gathered. An easy way to achieve this is to use the `CASCADE` parameter. Note that index statistics are automatically gathered when the index is created or rebuilt.
- Use 100 percent sample size if it is possible. In Oracle Database 11*g*, Oracle gathers optimizer statistics with a new algorithm. It is very accurate, but requires minimal time. However, for very large systems, the gathering of statistics can be a very time-consuming and resource-intensive activity. In this environment, sample sizes need to be carefully controlled to ensure that gathering is completed within an acceptable timescale and resource constraints and within the maintenance window. If the 100 percent sample size is not feasible, try using at least an estimate of 30 percent. For more information, review MOS note 44961.1, “Statistics Gathering: Frequency and Strategy Guidelines.”
- Because gathering new optimizer statistics may invalidate cursors in the shared pool, it is prudent to restrict execution of all gathering operations to periods of low activity in the database, such as the scheduled maintenance windows.

- If partitions are in use, gather global statistics if possible. Global statistics are very important, but gathering is often avoided due to the sizes involved and the length of time required for gathering. In Oracle Database 11*g*, you can use incremental global statistics, so global statistics will be gathered much faster. For more information, review MOS note 236935.1, “Global statistics – An Explanation.”
- Use Oracle Database 11*g* pending statistics to verify the effect of new statistics when tuning to minimize risk. This topic is covered later in the lesson.
- Gather statistics immediately after data has been loaded (>10 percent added), but before indexes are created.

Optimizer Dynamic Sampling: Overview

- Dynamic sampling can be done for tables and indexes:
  - Without statistics
  - Whose statistics cannot be trusted, starting with 11*g*R2 if object statistics are stale and sampling level => 4
- Used to determine more accurate statistics when estimating:
  - Table cardinality
  - Predicate selectivity
- Feature controlled by:
  - `OPTIMIZER_DYNAMIC_SAMPLING` parameter
  - `OPTIMIZER_FEATURES_ENABLE` parameter
  - `DYNAMIC_SAMPLING` hint
  - `DYNAMIC_SAMPLING_EST_CDN` hint

Dynamic sampling improves server performance by determining more accurate selectivity and cardinality estimates that allow the optimizer to produce better performing plans. For example, although it is recommended that you collect statistics on all of your tables for use by the optimizer, you may not gather statistics for your temporary tables and working tables that are used for intermediate data manipulation. In those cases, the optimizer provides a value through a simple algorithm that can lead to a suboptimal execution plan. You can use dynamic sampling to:

- Estimate single-table predicate selectivities when collected statistics cannot be used or are likely to lead to significant errors in estimation
- Estimate table cardinality for tables and relevant indexes without statistics or for tables whose statistics are too outdated to be reliable

You control dynamic sampling with the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter. The `DYNAMIC_SAMPLING` and `DYNAMIC_SAMPLING_EST_CDN` hints can be used to further control dynamic sampling.

Note: The `OPTIMIZER_FEATURES_ENABLE` initialization parameter turns off dynamic sampling if it is set to a version prior to 9.2.

Optimizer Dynamic Sampling at Work

- Sampling is done at compile time.
- If a query benefits from dynamic sampling:
  - A recursive SQL statement is executed to sample data.
  - The number of blocks sampled depends on the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter.
- During dynamic sampling, predicates are applied to the sample to determine selectivity.
- Use dynamic sampling when:
  - Sampling time is a small fraction of the execution time (like Data Warehouse, not OLTP).
  - Volatile data is used with `DELETE_*_STATS` and `LOCK_*_STATS`.
  - Correlated columns are used in the `WHERE` clause.
  - Global temporary tables are used.
  - The query is executed many times.
  - You believe a better plan can be found (during testing).

The primary performance attribute is compile time, when the system determines whether a query would benefit from dynamic sampling. If so, a recursive SQL statement is issued to scan a small random sample of the table's blocks and to apply the relevant single table predicates to estimate predicate selectivities.

Depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter, a certain number of blocks is read by the dynamic sampling query.

For a query that is normally completed quickly (less than a few seconds), you do not want to incur the cost of dynamic sampling. However, dynamic sampling can be beneficial under any of the following conditions:

- You want to look for a better plan during testing.
- The sampling time is a small fraction of total execution time for the query
- Volatile data is used with `DELETE_*_STATS` and `LOCK_*_STATS`.
- Correlated columns are used in the `WHERE` clause.
- Global temporary tables are used.
- The query is executed many times.

Note: Dynamic sampling can be applied to a subset of a single table's predicates and combined with standard selectivity estimates of predicates for which dynamic sampling is not done.

OPTIMIZER_DYNAMIC_SAMPLING

- Dynamic session or system parameter.
- Can be set to a value from "0" to "10."
- "0" turns off dynamic sampling.
- "1" samples all unanalyzed tables, if an unanalyzed table:
  - Is joined to another table or appears in a subquery or nonmergeable view
  - Has no indexes
  - Has more than 32 blocks
- "2" samples all unanalyzed tables.
- The higher the value, the more aggressive application of sampling.
- Dynamic sampling is repeatable if no update activity occurred.

You control dynamic sampling with the `OPTIMIZER_DYNAMIC_SAMPLING` parameter, which can be set to a value from "0" to "10." A value of "0" means dynamic sampling is not done.

A value of "1" means dynamic sampling is performed on all unanalyzed tables if the following criteria are met:

- There is at least one unanalyzed table in the query.
- This unanalyzed table is joined to another table or appears in a subquery or nonmergeable view.
- This unanalyzed table has no indexes.
- This unanalyzed table has more blocks than the default number of blocks that would be used for dynamic sampling of this table. This default number is 32.

The default value is "2" if `OPTIMIZER_FEATURES_ENABLE` is set to 10.0.0 or higher. At this level, the system applies dynamic sampling to all unanalyzed tables. The number of blocks sampled is two times the default number of dynamic sampling blocks (32).

Increasing the value of the parameter results in a more aggressive application of dynamic sampling, in terms of both the type of tables sampled (analyzed or unanalyzed) and the amount of I/O spent on sampling.

Note: Dynamic sampling is repeatable if no rows have been inserted, deleted, or updated in the table being sampled since the previous sample operation.

Managing Statistics: Overview (Export / Import / Lock / Restore / Publish)

- Purpose:
  - To revert to preanalyzed statistics if gathering statistics causes critical statements to perform badly
  - To test the new statistics before publishing
- Importing previously exported statistics (9*i*)
- Locking statistics on a specific table (10*g*)
- Restoring statistics archived before gathering (10*g*)
- Statistics can be pending before publishing (11*g*R2)

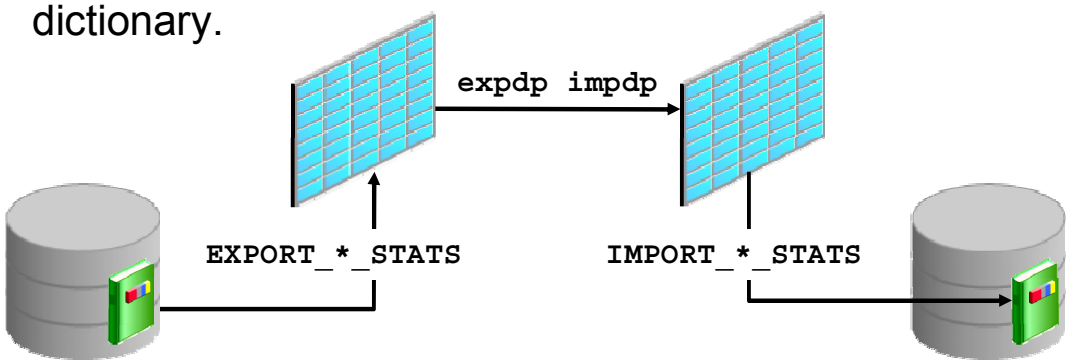
In the next slides, the following topics are covered to discuss how to manage the optimizer statistics:

- Importing previously exported statistics (9*i*)
- Locking statistics on a specific table (10*g*)
- Restoring statistics archived before gathering (10*g*)
- Statistics can be pending before publishing (11*g*R2)

Export and Import Statistics

Use DBMS_STATS procedures:

- CREATE_STAT_TABLE creates the statistics table.
- EXPORT_*_STATS moves the statistics to the statistics table.
- Use Data Pump to move the statistics table.
- IMPORT_*_STATS moves the statistics to the data dictionary.



You can export and import statistics from the data dictionary to user-owned tables, enabling you to create multiple versions of statistics for the same schema. You can also copy statistics from one database to another database. You may want to do this to copy the statistics from a production database to a scaled-down test database.

Before exporting statistics, you first need to create a table for holding the statistics. The procedure DBMS_STATS.CREATE_STAT_TABLE creates the statistics table. After table creation, you can export statistics from the data dictionary into the statistics table by using the DBMS_STATS.EXPORT_*_STATS procedures. You can then import statistics by using the DBMS_STATS.IMPORT_*_STATS procedures.

The optimizer does not use statistics stored in a user-owned table. The only statistics used by the optimizer are the statistics stored in the data dictionary. To have the optimizer use the statistics in a user-owned tables, you must import those statistics into the data dictionary by using the statistics-import procedures.

To move statistics from one database to another, you must first export the statistics on the first database, then copy the statistics table to the second database by using the Data Pump Export and Import utilities or other mechanisms, and then import the statistics into the second database.

Locking Statistics

- Prevents automatic gathering
- Is mainly used for volatile tables:
  - Lock without statistics implies dynamic sampling.

```
BEGIN
  DBMS_STATS.DELETE_TABLE_STATS('OE','ORDERS');
  DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
END;
```

  - Lock with statistics for representative values.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS('OE','ORDERS');
  DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
END;
```

- The FORCE argument overrides statistics locking.

```
SELECT stattype_locked FROM dba_tab_statistics;
```


Starting with Oracle Database 10*g*, you can lock statistics on a specified table with the LOCK_TABLE_STATS procedure of the DBMS_STATS package. You can lock statistics on a table without statistics or set them to NULL by using the DELETE_*_STATS procedures to prevent automatic statistics collection so that you can use dynamic sampling on a volatile table with no statistics. You can also lock statistics on a volatile table at a point when it is fully populated so that the table statistics are more representative of the table population.

You can also lock statistics at the schema level by using the LOCK_SCHEMA_STATS procedure. You can query the STATTYPE_LOCKED column in the {USER | ALL | DBA}_TAB_STATISTICS view to determine whether the statistics on the table are locked.

You can use the UNLOCK_TABLE_STATS procedure to unlock the statistics on a specified table.

You can set the value of the FORCE parameter to TRUE to overwrite the statistics even if they are locked. The FORCE argument is found in the following DBMS_STATS procedures: DELETE_*_STATS, IMPORT_*_STATS, RESTORE_*_STATS, and SET_*_STATS.

Note: When you lock the statistics on a table, all the dependent statistics, including table statistics, column statistics, histograms, and dependent index statistics, are considered locked.

Restoring Statistics

- Past statistics may be restored with the DBMS_STATS.RESTORE_*_STATS procedures.

```
BEGIN
 DBMS_STATS.RESTORE_TABLE_STATS(
   OWNNAME=>'OE', TABNAME=>'INVENTORIES',
   AS_OF_TIMESTAMP=>'15-JUL-10 09.28.01.597526000 AM -05:00');
END;
```

- Statistics are automatically stored:
  - With the timestamp in DBA_TAB_STATS_HISTORY
  - When collected with DBMS_STATS procedures
- Statistics are purged:
  - When the STATISTICS_LEVEL is set to TYPICAL or ALL automatically
  - After 31 days or time defined by DBMS_STATS.ALTER_STATS_HISTORY_RETENTION

Old versions of statistics are saved automatically whenever statistics in the dictionary are modified with the DBMS_STATS procedures. You can restore statistics by using the RESTORE procedures of the DBMS_STATS package. These procedures use a time stamp as an argument and restore statistics as of that time stamp. This is useful when newly-collected statistics lead to suboptimal execution plans and the administrator wants to revert to the previous set of statistics.

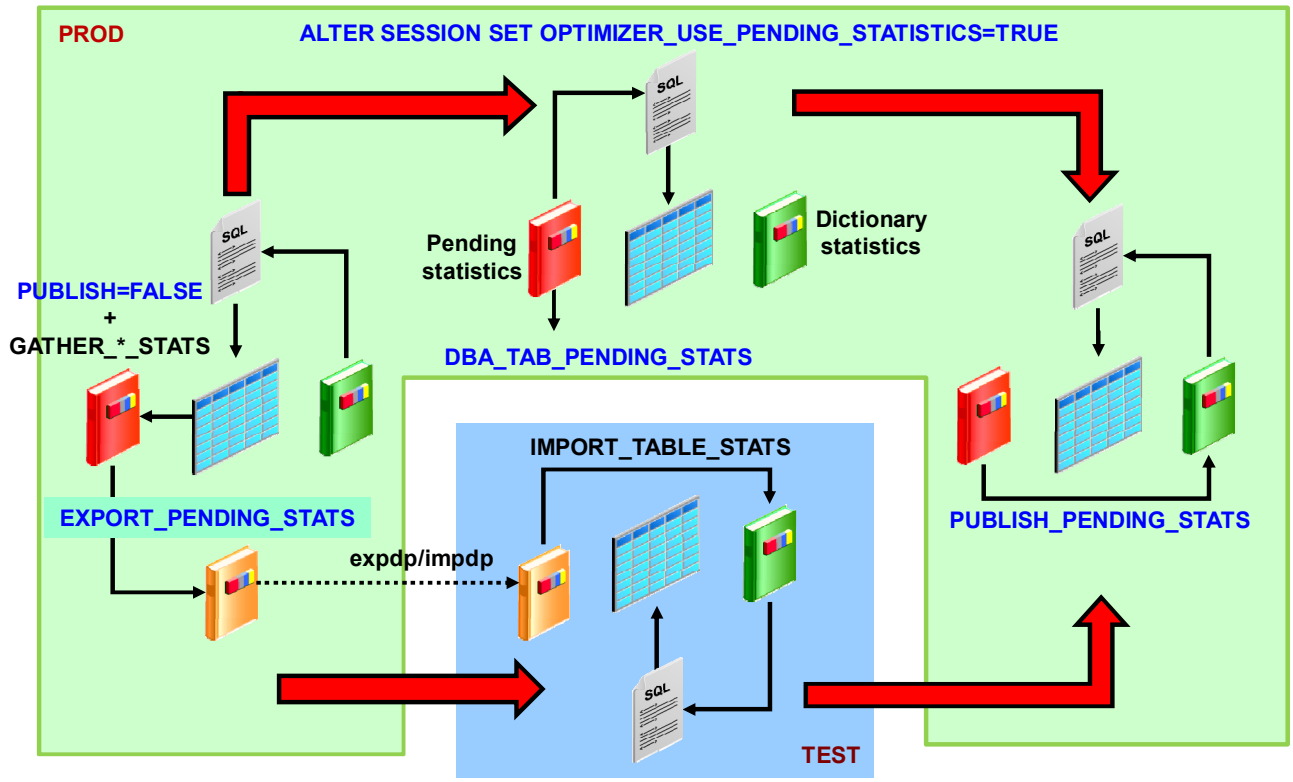
Note: The ANALYZE command does not store old statistics.

There are dictionary views that can be used to determine the time stamp for restoration of statistics. The *_TAB_STATS_HISTORY views (ALL, DBA, or USER) contain a history of table statistics modifications. For the example in the slide, the time stamp was determined by:

```
select stats_update_time from dba_tab_stats_history
where table_name = 'INVENTORIES'
```

The database purges old statistics automatically at regular intervals based on the statistics history retention setting and the time of the recent analysis of the system. You can configure retention by using the DBMS_STATS.ALTER_STATS_HISTORY_RETENTION procedure. The default value is 31 days, which means that you would be able to restore the optimizer statistics to any time in the last 31 days.

Deferred Statistics Publishing: Overview



By default, the statistics-gathering operation automatically stores the new statistics in the data dictionary each time it completes the iteration for one object (table, partition, subpartition, or index). The optimizer sees the new statistics as soon as they are written to the data dictionary, and these new statistics are called *current statistics*. Automatic publishing can be frustrating to the DBA, who is never sure of the impact of the new statistics. In addition, the statistics used by the optimizer can be inconsistent if, for example, table statistics are published before the statistics of its indexes, partitions, or subpartitions.

To avoid these potential issues, you can separate the gathering step from the publication step for optimizer statistics. There are two benefits of separating the two steps:

- Supports the statistics-gathering operation as an atomic transaction. The statistics of all tables and dependent objects (indexes, partitions, subpartitions) in a schema will be published at the same time. This means the optimizer will always have a consistent view of the statistics. If the gathering step fails during the gathering process, it will be able to resume from where it left off when it is restarted by using the `DBMS_STAT.RESUME_GATHER_STATS` procedure.
- Allows DBAs to validate the new statistics by running all or part of the workload using the newly-gathered statistics on a test system and, when satisfied with the test results, to proceed to the publishing step to make them current in the production environment

When you specify the PUBLISH to FALSE gather option, gathered statistics are stored in the pending statistics tables instead of being current. These pending statistics are accessible from a number of views: {ALL|DBA|USER}_{TAB|COL|IND|TAB_HISTGRM}_PENDING_STATS.

The slide shows two paths for testing the statistics. The lower path shows exporting the pending statistics to a test system. The upper path shows enabling the use of pending statistics (usually in a session). Both paths continue to the production system and the publication of the pending statistics.

To test the pending statistics, you have two options:

- Transfer the pending statistics to your own statistics table by using the new DBMS_STAT.EXPORT_PENDING_STATS procedure, export your statistics table to a test system where you can test the impact of the pending statistics, and then render the pending statistics current by using the DBMS_STAT.IMPORT_TABLE_STATS procedure.
- Enable session-pending statistics by altering your OPTIMIZER_USE_PENDING_STATISTICS session initialization parameter to TRUE. By default, this new initialization parameter is set to FALSE. This means that in your session, you parse SQL statements by using the current optimizer statistics. By setting it to TRUE in your session, you switch to the pending statistics instead.

When you have tested the pending statistics and are satisfied with them, you can publish them as current in your production environment by using the new DBMS_STAT.PUBLISH_PENDING_STATS procedure.

Note: For more information about the DBMS_STATS package, see the *PL/SQL Packages and Types Reference*.

Deferred Statistics Publishing: Example

```
exec dbms_stats.set_table_prefs('SH','CUSTOMERS','PUBLISH','false');  1
```

```
exec dbms_stats.gather_table_stats('SH','CUSTOMERS');  2
```

```
alter session set optimizer_use_pending_statistics = true;  3
```

Execute your workload from the same session. 4

```
exec dbms_stats.publish_pending_stats('SH','CUSTOMERS');  5
```

1. Use the `SET_TABLE_PREFS` procedure to set the `PUBLISH` option to `FALSE`. This setting prevents the next statistics-gathering operation from automatically publishing statistics as current. According to the first statement, this is true only for the `SH.CUSTOMERS` table.
2. Gather statistics for the `SH.CUSTOMERS` table in the pending area of the dictionary.
3. Test the new set of pending statistics from your session by setting the `OPTIMIZER_USE_PENDING_STATISTICS` to `TRUE`.
4. Issue queries against `SH.CUSTOMERS`.
5. If you are satisfied with the test results, use the `PUBLISH_PENDING_STATS` procedure to render the pending statistics for `SH.CUSTOMERS` current.

Note: To analyze the differences between the pending statistics and the current ones, you could export the pending statistics to your own statistics table and then use the new `DBMS_STAT.DIFF_TABLE_STATS` function.

Quiz

When there are no statistics for an object being used in a SQL statement, the optimizer uses:

a. Rule-based optimization
b. Dynamic sampling
c. Fixed values
d. Statistics gathered during the parse phase
e. Random values

Answer: b, c

Quiz

The optimizer depends on accurate statistics to produce the best execution plans. The automatic statistics-gathering task does not gather statistics on everything. Which objects require you to gather statistics manually?

a. External tables
b. Data dictionary
c. Fixed objects
d. Volatile tables
e. System statistics

Answer: a, c, e

Quiz

There is a very volatile table in the database. The size of the table changes by more than 50 percent daily. What steps are part of the procedure to force dynamic sampling?

a. Delete statistics.
b. Lock statistics.
c. Gather statistics when the table is at its largest.
d. Set `DYNAMIC_SAMPLING=9`.
e. Set `DYNAMIC_SAMPLING=0`.
f. Allow the `DYNAMIC_SAMPLING` parameter to default.

Answer: a, b, f

Summary

In this lesson, you should have learned how to:

- Describe optimizer statistics
  - Table statistics
  - Index statistics
  - Column statistics (histogram)
  - Column statistics (extended statistics)
  - System statistics
- Gather optimizer statistics
- Set statistic preferences
- Use dynamic sampling
- Manage optimizer statistics
- Discuss optimizer statistics best practices

Practice 10: Overview

This practice covers the following topics:

- Using system statistics
- Using automatic statistics gathering

11

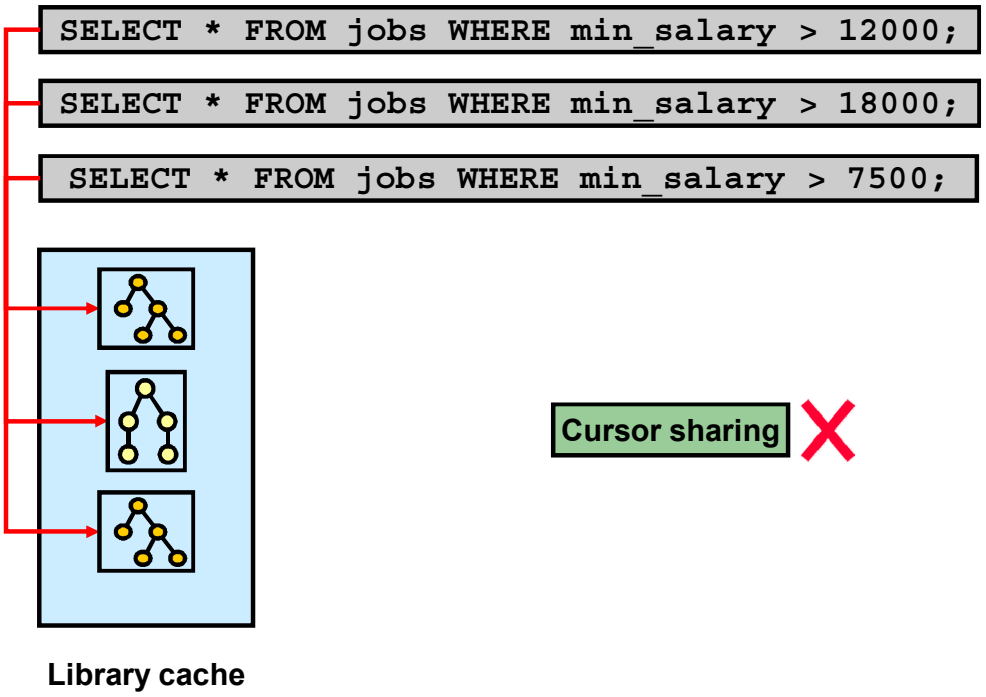
Using Bind Variables

Objectives

After completing this lesson, you should be able to:

- List the benefits of using bind variables
- Use bind peeking
- Use adaptive cursor sharing
- Describe common observations

Cursor Sharing and Different Literal Values



If your SQL statements use literal values for the `WHERE` clause conditions, there will be many versions of almost identical SQL stored in the library cache. For each different SQL statement, the optimizer must perform all the steps for processing a new SQL statement. This may also cause the library cache to fill up quickly because of all the different statements stored in it.

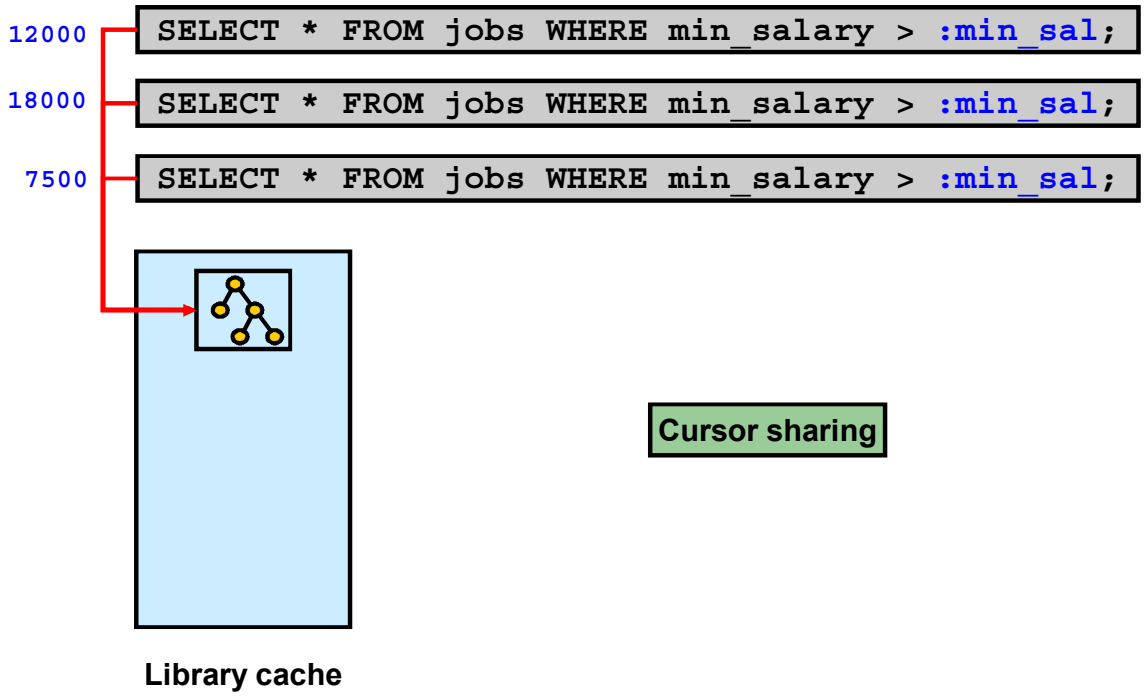
When coded this way, you are not taking advantage of cursor sharing. If the cursor is shared by using a bind variable rather than a literal variable, there will be one shared cursor, with one execution plan.

However, depending on the literal value provided, different execution plans might be generated by the optimizer. For example, there might be several `JOBS`, where `MIN_SALARY` is greater than 12000. Alternatively, there might be very few `JOBS` that have a `MIN_SALARY` greater than 18000. This difference in data distribution could justify the addition of an index so that different plans can be used depending on the value provided in the query. This is illustrated in the slide. As you can see, the first and third queries use the same execution plan, but the second query uses a different one.

From a performance perspective, it is good to have separate cursors. However, this is not very economical because you could have shared cursors for the first and last queries in this example.

Note: In the case of the example in the slide, `V$SQL.PLAN_HASH_VALUE` is identical for the first and third query.

Cursor Sharing and Bind Variables



If, instead of issuing different statements for each literal, you use a bind variable, that extra parse activity is eliminated (in theory). The elimination occurs because the optimizer recognizes that the statement is already parsed and decides to reuse the same execution plan even though you specify different bind values the next time you execute the same statement.

For the example in the slide, the bind variable is called `min_sal`. It is to be compared with the `MIN_SALARY` column of the `JOBS` table. Instead of issuing three different statements, issue a single statement that uses a bind variable. At execution time, the same execution plan is used, and the given value is substituted for the variable.

However, from a performance perspective, this is not the best situation because you get best performance two times out of three. On the other hand, this is very economical because you need only one shared cursor in the library cache to execute all three statements.

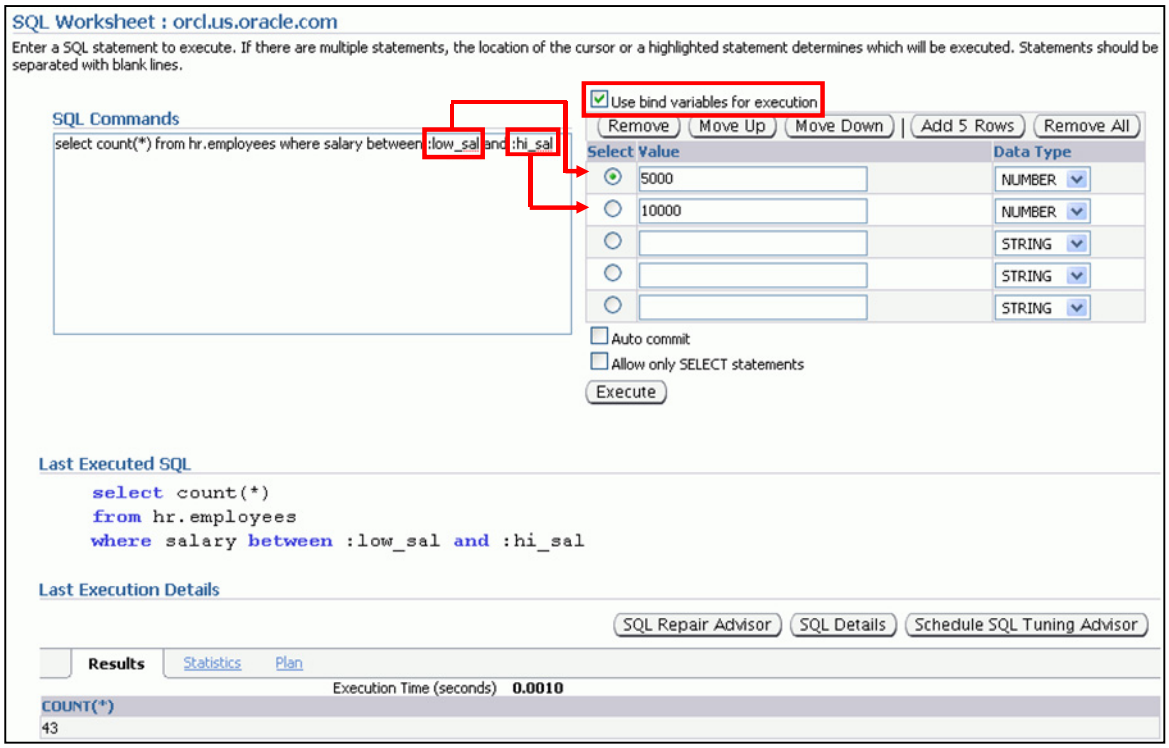
Bind Variables in SQL*Plus

```
SQL> variable job_id varchar2(10)
SQL> exec :job_id := 'SA_REP';

PL/SQL procedure successfully completed.

SQL> select count(*) from employees where job_id = :job_id;

  COUNT(*)
----------
        30

SQL> exec :job_id := 'AD_VP';

PL/SQL procedure successfully completed.

SQL> select count(*) from employees where job_id = :job_id;

  COUNT(*)
----------
         2
```

Bind variables can be used in Oracle SQL*Plus sessions. In Oracle SQL*Plus, use the `VARIABLE` command to define a bind variable. Then, you can assign values to the variable by executing an assignment statement with the `EXEC[UTE]` command. Any references to that variable from then on use the assigned value.

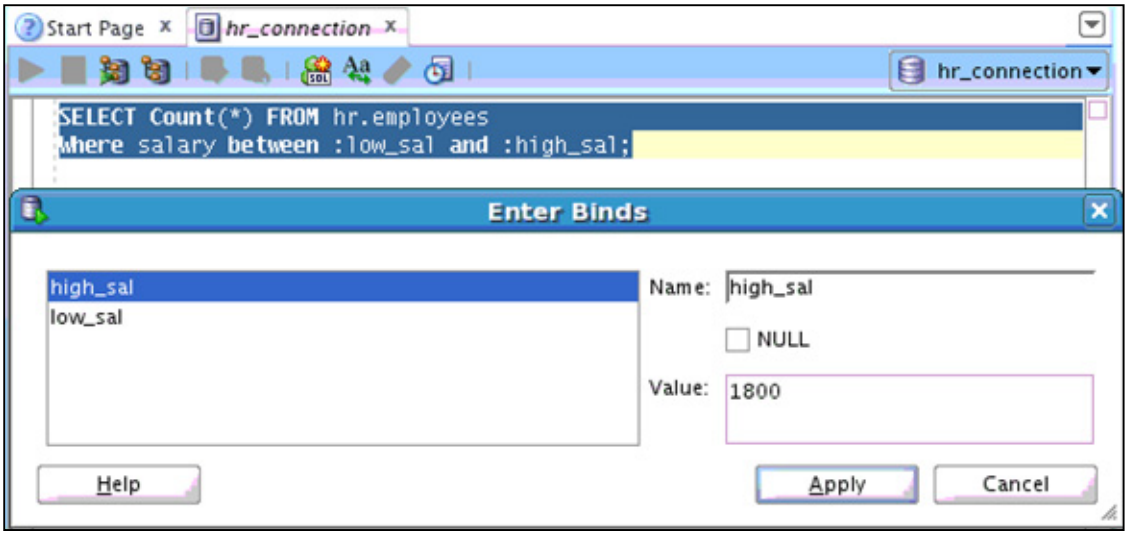
For the example in the slide, the first count is selected while `SA_REP` is assigned to the variable. The result is 30. Then, `AD_VP` is assigned to the variable, and the resulting count is 2.

Bind Variables in Enterprise Manager



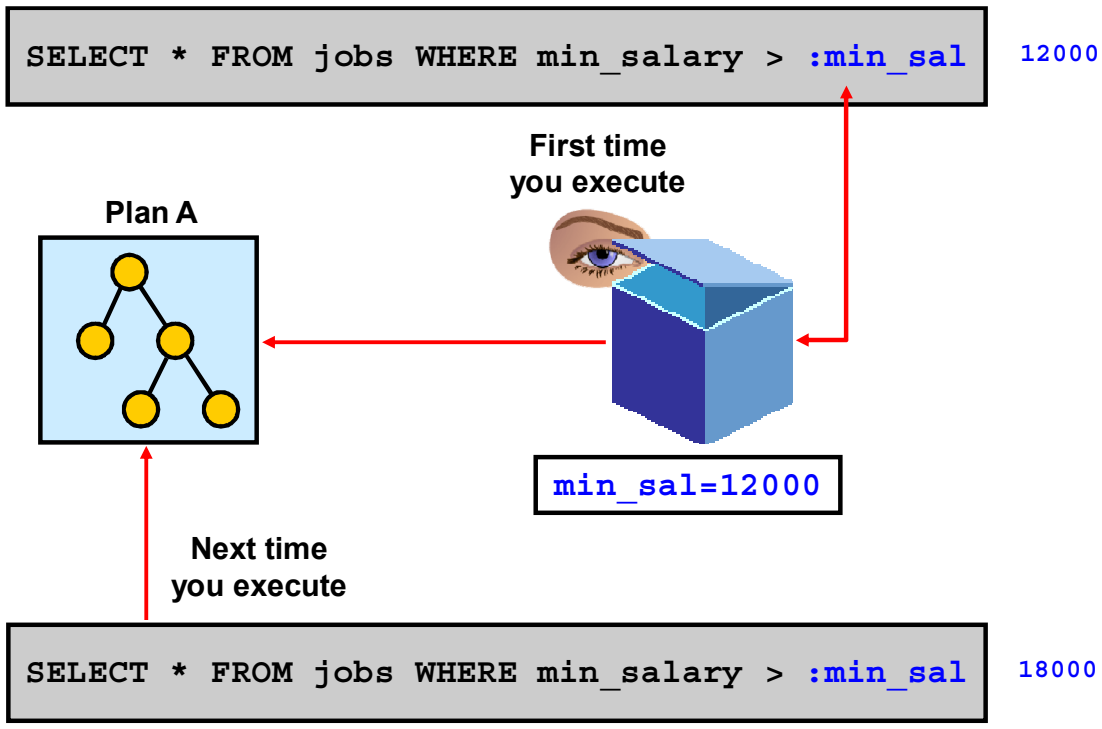
On the SQL Worksheet page of Enterprise Manager (see the SQL Worksheet link in the Related Links region on the Database Home page), you can specify that a SQL statement should use bind variables. You can do this by selecting the "Use bind variables for execution" check box. When you select that check box, you can enter bind variable values in the generated fields. Refer to these values in the SQL statement by using variable names that begin with a colon. The order in which variables are referred to defines which variable gets which value. The first variable referred to gets the first value, the second variable gets the second value, and so on. If you change the order in which variables are referenced in the statement, you may need to change the value list to match that order.

Bind Variables in Oracle SQL Developer



On the SQL Worksheet pane of SQL, you can specify that a SQL statement uses bind variables. When you execute the statement, you can enter bind variable values in the Enter Binds dialog box. Refer to these values in the SQL statement by using variable names that begin with a colon. Select each bind variable in turn to enter a value for that variable.

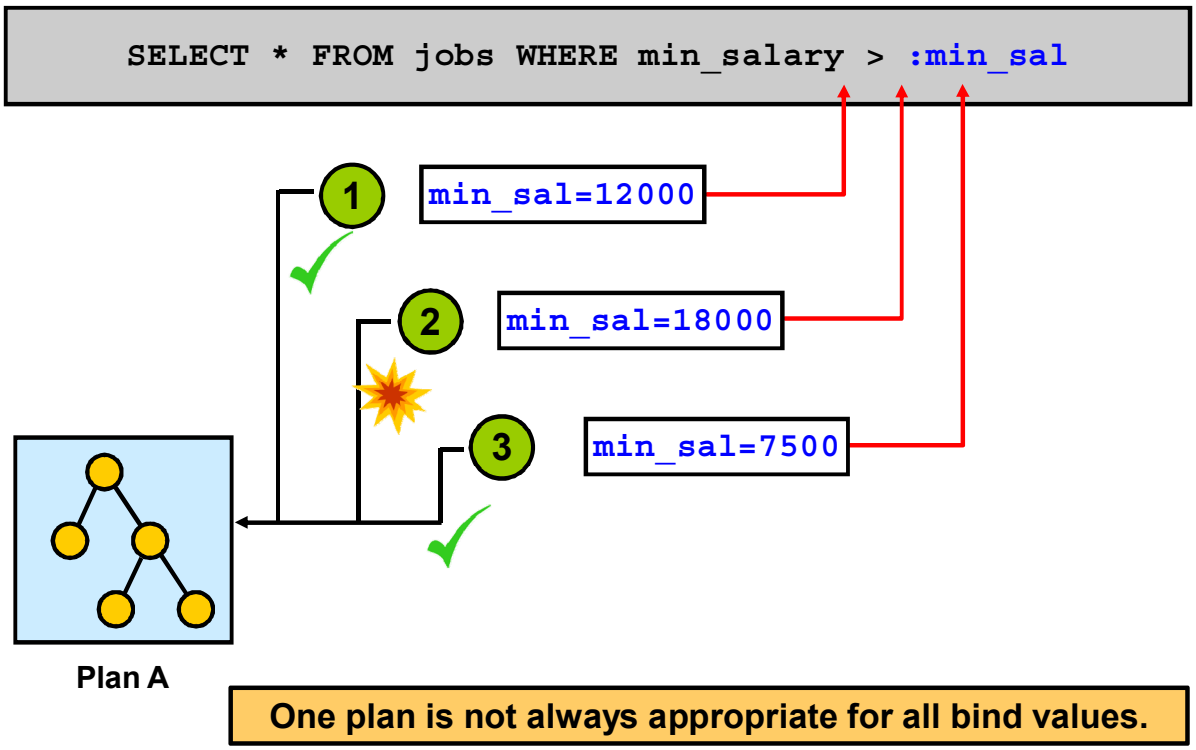
Bind Variable Peeking



When literals are used in a query, those literal values can be used by the optimizer to decide on the best plan. However, when bind variables are used, the optimizer still needs to select the best plan based on the values of the conditions in the query, but cannot readily see those values in the SQL text. That means, as a SQL statement is parsed, the system needs to be able to see the value of the bind variables to ensure that a good plan that would suit those values is selected. The optimizer does this by *peeking* at the value in the bind variable. When the SQL statement is hard parsed, the optimizer evaluates the value for each bind variable, and uses that as input in determining the best plan. After the execution is determined the first time you parsed the query, it is reused when you execute the same statement regardless of the bind values used.

This feature was introduced in Oracle9*i* Database, Release 2. Oracle Database 11*g* changes this behavior.

Bind Variable Peeking



Under some conditions, bind variable peeking can cause the optimizer to select the suboptimal plan. This occurs because the first value of the bind variable is used to determine the plan for all subsequent executions of the query. Therefore, even though subsequent executions provide different bind values, the same plan is used. It is possible that a different plan would be better for executions that have different bind variable values. An example is where the selectivity of a particular index varies extremely depending on the column value. For low selectivity, a full table scan may be faster. For high selectivity, an index range scan may be more appropriate.

As shown in the slide, plan A may be good for the first and third values of `min_sal`, but it may not be the best for the second one. Suppose there are very few `MIN_SALARY` values that are above 18000, and plan A is a full table scan. It is probable that a full table scan is not a good plan for the second execution, in that case.

Bind variables are beneficial in that they cause more cursor sharing to happen, and thus reduce parsing of SQL. But, as in this case, it is possible that they cause a suboptimal plan to be chosen for some of the bind variable values. This is a good reason for not using bind variables for decision support system (DSS) environments, where the parsing of the query is a very small percentage of the work done when submitting a query. The parsing may take fractions of a second, but the execution may take minutes or hours. To execute with a slower plan is not worth the savings gained in parse time.

Cursor Sharing Enhancements

- Oracle8*i* introduced the possibility of sharing SQL statements that differ only in literal values.
- Oracle9*i* extends this feature by limiting it only to similar statements, instead of forcing it.
- Similar: Regardless of the literal value, same execution plan

```
SQL> SELECT * FROM employees
  2  WHERE employee_id = 153;
```

- Not similar: Possible different execution plans for different literal values.

```
SQL> SELECT * FROM employees
  2  WHERE department_id = 50;
```

Oracle8*i* introduced the possibility of sharing SQL statements that differ only in literal values. Rather than developing an execution plan, each time the same statement—with a different literal value—is executed, the optimizer generates a common execution plan used for all subsequent executions of the statement.

Because only one execution plan is used instead of potential different ones, this feature should be tested against your applications before you decide whether to enable it or not. That is why Oracle9*i* extends this feature by sharing only statements considered as similar, that is, only when the optimizer has the guarantee that the execution plan is independent of the literal value used. For example, consider a query where `EMPLOYEE_ID` is the primary key:

```
SQL> SELECT * FROM employees WHERE employee_id = 153;
```

The substitution of any value would produce the same execution plan. It would, therefore, be safe for the optimizer to generate only one plan for different occurrences of the same statement executed with different literal values.

On the other hand, assume that the same `EMPLOYEES` table has a wide range of values in its `DEPARTMENT_ID` column. For example, department 50 could contain more than one-third of all employees, and department 70 could contain only one or two.

See the two queries:

```
SQL> SELECT * FROM employees WHERE department_id = 50;
SQL> SELECT * FROM employees WHERE department_id = 70;
```

Using only one execution plan for sharing the same cursor would not be safe if you have histogram statistics (and there is skew in the data) on the `DEPARTMENT_ID` column. In this case, depending on which statement was executed first, the execution plan could contain a full table (or fast full index) scan, or it could use a simple index range scan.

The CURSOR_SHARING Parameter

- CURSOR_SHARING parameter values:
  - FORCE
  - EXACT (default)
  - SIMILAR
- CURSOR_SHARING can be changed by using:
  - ALTER SYSTEM
  - ALTER SESSION
  - Initialization parameter files
- CURSOR_SHARING_EXACT hint

The value of the CURSOR_SHARING initialization parameter determines how the optimizer processes statements with bind variables:

- EXACT: Literal replacement disabled completely
- FORCE: Causes sharing for all literals
- SIMILAR: Causes sharing for safe literals only

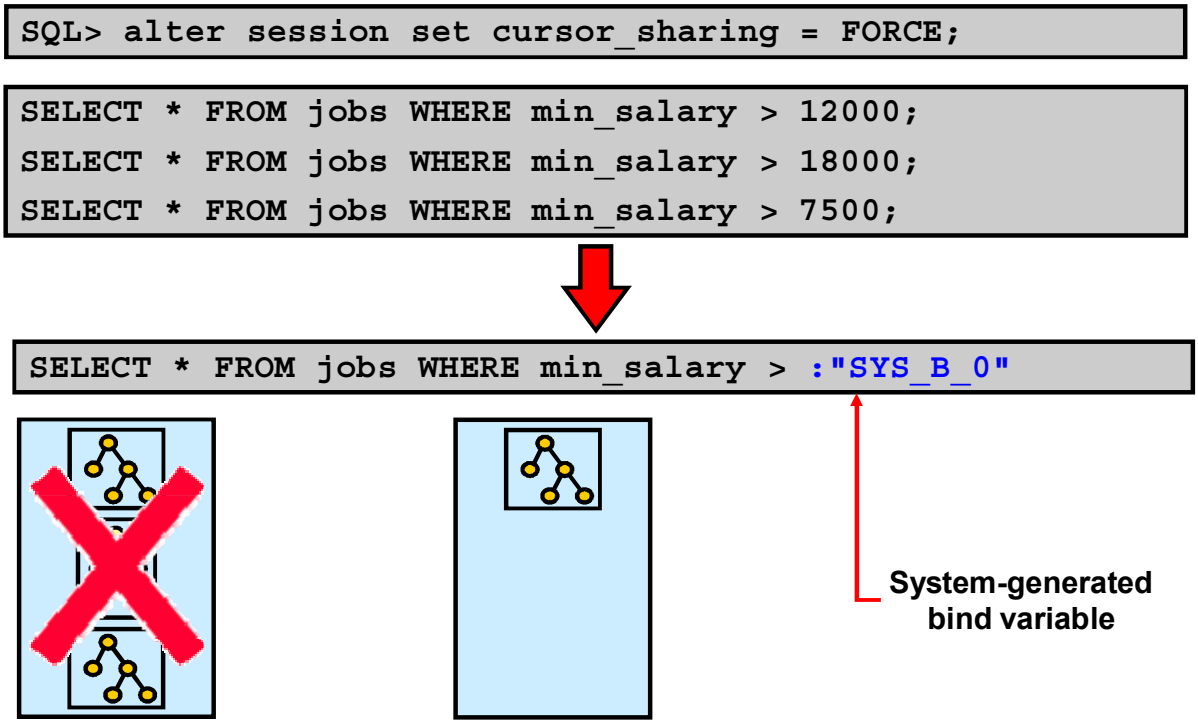
In earlier releases, you could select only the EXACT or the FORCE option. Setting the value to SIMILAR causes the optimizer to examine the statement to ensure that replacement occurs only for safe literals. It can then use information about the nature of any available index (unique or nonunique) and statistics collected on the index or underlying table, including histograms.

The value of CURSOR_SHARING in the initialization file can be overridden with an ALTER SYSTEM SET CURSOR_SHARING or an ALTER SESSION SET CURSOR_SHARING command.

The CURSOR_SHARING_EXACT hint causes the system to execute the SQL statement without any attempt to replace literals by bind variables.

Note: For more information about CURSOR_SHARING = SIMILAR, see MOS note 1169017.1, "Announcement: Deprecating the cursor_sharing = 'SIMILAR' setting."

Forcing Cursor Sharing: Example



For the example in the slide, because you forced cursor sharing with the `ALTER SESSION` command, all your queries that differ only with literal values are automatically rewritten to use the same system-generated bind variable called `SYS_B_0`. As a result, you end up with only one child cursor instead of three.

Note: Adaptive cursor sharing may also apply, and might generate a second child cursor in this case.

Adaptive Cursor Sharing: Overview

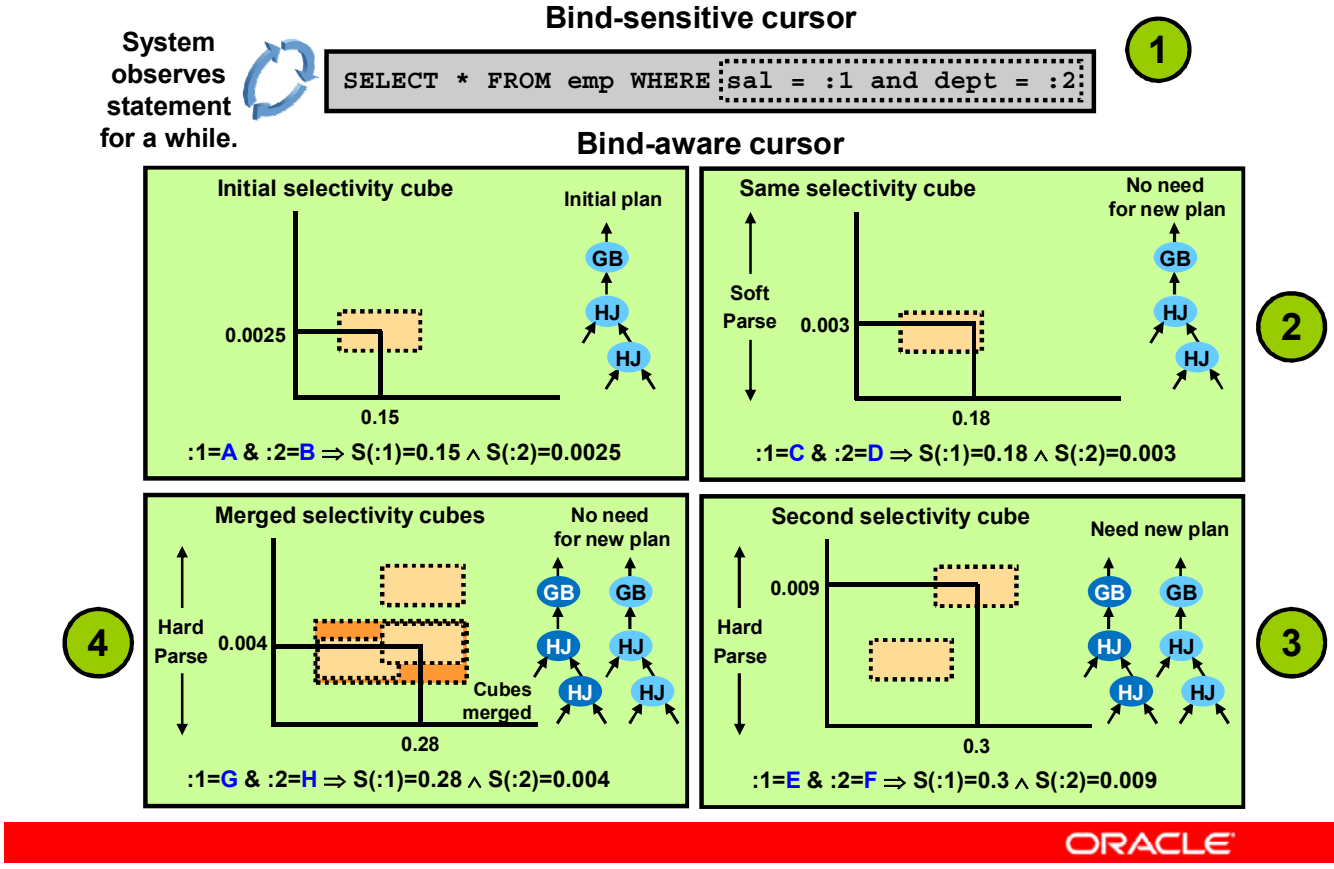
- Allows for intelligent cursor sharing for statements that use bind variables
- Is used to compromise between cursor sharing and optimization
- Has the following benefits:
  - Automatically detects when different executions would benefit from different execution plans
  - Limits the number of generated child cursors to a minimum
  - Provides an automated mechanism that cannot be turned off

ORACLE

Bind variables were designed to allow Oracle Database to share a single cursor for multiple SQL statements to reduce the amount of shared memory used to parse SQL statements. However, cursor sharing and SQL optimization are conflicting goals. Writing a SQL statement with literals provides more information for the optimizer and naturally leads to better execution plans, while increasing memory and CPU overhead caused by excessive hard parses. Oracle Database 9*i* was the first attempt to introduce a compromise solution by allowing similar SQL statements using different literal values to be shared. For statements using bind variables, Oracle Database 9*i* also introduced the concept of bind peeking. To benefit from bind peeking, it is assumed that cursor sharing is intended and that different invocations of the statement are supposed to use the same execution plan. If different invocations of the statement would significantly benefit from different execution plans, bind peeking is of no use in generating good execution plans.

To address this issue as much as possible, Oracle Database 11*g* introduces adaptive cursor sharing. This feature is a more sophisticated strategy designed not to share the cursor blindly, but generate multiple plans per SQL statement with bind variables if the benefit of using multiple execution plans outweighs the parse time and memory usage overhead. However, because the purpose of using bind variables is to share cursors in memory, a compromise must be found regarding the number of child cursors that need to be generated.

Adaptive Cursor Sharing: Architecture



When you use adaptive cursor sharing, the following steps take place in the scenario illustrated in the slide:

1. The cursor starts its life with a hard parse, as usual. If bind peeking takes place, and a histogram is used to compute selectivity of the predicate containing the bind variable, the cursor is marked as a bind-sensitive cursor. In addition, some information is stored about the predicate containing the bind variables, including the predicate selectivity. In the slide example, the predicate selectivity that would be stored is a cube centered around (0.15,0.0025). Because of the initial hard parse, an initial execution plan is determined by using the peeked binds. After the cursor is executed, the bind values and the execution statistics of the cursor are stored in that cursor.

   During the next execution of the statement, when a new set of bind values is used, the system performs a usual soft parse, and finds the matching cursor for execution. At the end of execution, execution statistics are compared with the ones currently stored in the cursor. The system then observes the pattern of the statistics over all the previous runs (see `V$SQL_CS_...` views in the slide that follows) and decides whether or not to mark the cursor as bind aware.

2. On the next soft parse of this query, if the cursor is now bind aware, bind-aware cursor matching is used. Suppose the selectivity of the predicate with the new set of bind values is now (0.18,0.003). Because selectivity is used as part of bind-aware cursor matching, and because the selectivity is within an existing cube, the statement uses the existing child cursor's execution plan to run.
3. On the next soft parse of this query, suppose that the selectivity of the predicate with the new set of bind values is now (0.3,0.009). Because that selectivity is not within an existing cube, no child cursor match is found; the system does a hard parse, which generates a new child cursor with a second execution plan in that case. In addition, the new selectivity cube is stored as part of the new child cursor. After the new child cursor executes, the system stores the bind values and execution statistics in the cursor.
4. On the next soft parse of this query, suppose the selectivity of the predicate with the new set of bind values is now (0.28,0.004). Because that selectivity is not within one of the existing cubes, the system does a hard parse. Suppose that this time, the hard parse generates the same execution plan as the first one. Because the plan is the same as the first child cursor, both child cursors are merged. That is, both cubes are merged into a new bigger cube, and one of the child cursors is deleted. The next time there is a soft parse, if the selectivity falls within the new cube, the child cursor matches.

Adaptive Cursor Sharing: Views

The following views provide information about adaptive cursor sharing usage:

V$SQL	Two new columns show whether a cursor is bind sensitive or bind aware.
V$SQL_CS_HISTOGRAM	Shows the distribution of the execution count across the execution history histogram.
V$SQL_CS_SELECTIVITY	Shows the selectivity cubes stored for every predicate containing a bind variable and whose selectivity is used in the cursor sharing checks.
V$SQL_CS_STATISTICS	Shows execution statistics of a cursor using different bind sets.

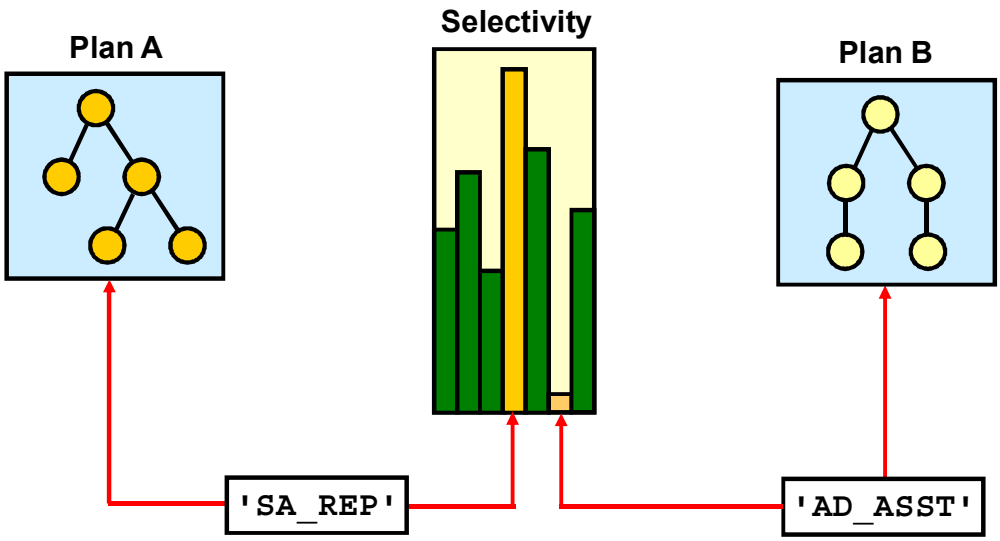
These views determine whether a query is bind aware or not and whether it is handled automatically, without any user input. However, information about what goes on is exposed through `V$` views so that you can diagnose problems, if any. New columns have been added to `V$SQL`:

- `IS_BIND_SENSITIVE`: Indicates if a cursor is bind sensitive; value `YES` | `NO`. A query for which the optimizer peeked at bind variable values when computing predicate selectivities and where a change in a bind variable value may lead to a different plan is called *bind sensitive*.
- `IS_BIND_AWARE`: Indicates if a cursor is bind aware; value `YES` | `NO`. A cursor in the cursor cache that has been marked to use bind-aware cursor sharing is called *bind aware*.
- `V$SQL_CS_HISTOGRAM`: Shows the distribution of the execution count across a three-bucket execution history histogram
- `V$SQL_CS_SELECTIVITY`: Shows the selectivity cubes or ranges stored in a cursor for every predicate containing a bind variable and whose selectivity is used in the cursor sharing checks. It contains the text of the predicates and the selectivity range low and high values.

- V$SQL_CS_STATISTICS: Adaptive cursor sharing monitors execution of a query, collects information about it for a while, and uses this information to decide whether to switch to bind-aware cursor sharing for the query. This view summarizes the information that it collects to make this decision. For a sample of executions, it keeps track of the rows processed, buffer gets, and CPU time. The PEEKED column has the value YES if the bind set was used to build the cursor; it has the value NO otherwise.

Adaptive Cursor Sharing: Example

```
SQL> variable job varchar2(6)
SQL> exec :job := 'AD_ASST'
SQL> select count(*), max(salary) from employees where
    job_id=:job;
```



Consider the data in the slide. Histogram statistics on the JOB_ID column show that the occurrence of SA_REP is many thousands of times more than that of AD_ASST. In this case, if literals were used instead of a bind variable, the query optimizer would see that the AD_ASST value occurs in less than 1 percent of the rows, whereas the SA_REP value occurs in approximately one-third of the rows. If the table has over a million rows, the execution plans are different for each of these values' queries. The AD_ASST query results in an index range scan because there are so few rows with that value. The SA_REP query results in a full table scan because so many of the rows have that value that it is more efficient to read the entire table. But, as it is, using a bind variable causes the same execution plan to be used at first for both of the values. So, even though there exist different and better plans for each of these values, they use the same plan.

After several executions of this query using a bind variable, the system considers the query bind aware, at which point it changes the plan based on the bound value. This means the best plan is used for the query, based on the bind variable value.

Interacting with Adaptive Cursor Sharing

- CURSOR_SHARING:
  - If CURSOR_SHARING <> EXACT, statements containing literals may be rewritten by using bind variables.
  - If statements are rewritten, adaptive cursor sharing may apply to them.
- SQL Plan Management (SPM):
  - If OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES is set to TRUE, only the first generated plan is used.
  - As a workaround, set this parameter to FALSE, and run your application until all plans are loaded in the cursor cache.
  - Manually load the cursor cache into the corresponding plan baseline.

ORACLE

Adaptive cursor sharing is independent of the CURSOR_SHARING parameter. The setting of this parameter determines whether literals are replaced by the system-generated bind variables. If they are, adaptive cursor sharing behaves just as it would if the user supplied binds to begin with.

When using the SPM automatic plan capture, the first plan captured for a SQL statement with bind variables is marked as the corresponding SQL plan baseline. If another plan is found for that same SQL statement (which may be the case with adaptive cursor sharing), it is added to the SQL statement's plan history and marked for verification. It will not be used immediately. Although adaptive cursor sharing has come up with a new plan based on a new set of bind values, SPM does not let it be used until the plan has been verified. Thus reverting to Oracle Database 10*g* behavior, only the plan generated based on the first set of bind values is used by all subsequent executions of the statement.

One possible workaround is to run the system for some time with automatic plan capture set to False, and after the cursor cache has been populated with all the plans for a SQL statement with bind, load the entire plan directly from the cursor cache into the corresponding SQL plan baseline. By doing this, all the plans for a single SQL statement are marked as SQL baseline plans by default. Refer to the lesson titled "SQL Plan Management" for more information.

Common Observations

Consider the following areas to resolve excessive parsing time as well:

- CPU time dominates the parse time
- Wait time dominates the parse time

```
SELECT * FROM ….

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ----------  ----------  ----------
Parse      555    100.09     300.83          0          0           0           0
Execute    555      0.42       0.78          0          0           0           0
Fetch      555     14.04      85.03        513    1448514           0       11724
------- ------  -------- ---------- ---------- ----------  ----------  ----------
total     1665    114.55     386.65        513    1448514           0       11724
```


You might also need to consider the following areas to resolve excessive parsing problems:

CPU time dominates the parse time: If high CPU usage (in general more than 50 percent of parse elapsed time) is observed during the hard parses, you would need to review the following common causes:

- **Dynamic sampling:** Dynamic sampling is performed via hint or parameters or missing statistics. Because dynamic sampling is done at parse time, depending on the level of the dynamic sampling, it may impact the parse time.
- **Many `IN`-LIST parameters /`OR` clauses:** It may take a long time to estimate a cost of a statement with many `IN`-LIST or `OR` clauses. You could use the `NO_EXPAND` hint to avoid query transformation, eventually to save the parse time.

Wait time dominates the parse time: High wait time during hard parses are usually due to contention for resources or are related to very large queries. You would need to examine the waits (“SQL*Net more data from client”) in 10046 trace for the query. One of the common causes is seen when sending a large query containing lots of text. Because it may take several round trips to be sent from the client to the server, each trip takes time, especially on slow networks.

Quiz

Which three statements are true about applications that are coded with literals rather than bind variables in the SQL statements?

a. More shared pool space is required for cursors.
b. Less shared pool space is required for cursors.
c. Histograms are used if available.
d. Histograms are not used.
e. No parsing is required for literal values.
f. Every different literal value requires parsing.

Answer: a, c, f

Quiz

The `CURSOR_SHARING` parameter should be set to _____ for systems with large tables and long-running queries, such as a data warehouse.

a. Similar
b. Force
c. Exact
d. Literal
e. True
f. False

Answer: c

Quiz

Adaptive cursor sharing can be turned off by setting the `CURSOR_SHARING` parameter to `FALSE`.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- List the benefits of using bind variables
- Use bind peeking
- Use adaptive cursor sharing
- Describe common observations

Practice 11: Overview

This practice covers the following topics:

- Using adaptive cursor sharing and bind peeking
- Using the `CURSOR_SHARING` initialization parameter

12

SQL Plan Management

Objectives

After completing this lesson, you should be able to:

- Manage SQL performance through changes
- Set up SQL Plan Management
- Set up various SQL Plan Management scenarios
- Load hinted plans into SQL Plan Management
- Migrate stored outlines to SQL plan baselines

Maintaining SQL Performance

Maintaining performance may require using SQL plan baselines.

Any number of factors that influence the optimizer can change over time. The challenge is to maintain the SQL performance levels in spite of the changes.

Optimizer statistics change for many reasons. Managing the changes to SQL performance despite the changes to statistics is the task of the DBA.

Some SQL statements on any system will stand out as high-resource consumers. It is not always the same statements. The performance of these statements must be tuned, without having to change the code. SQL profiles provide the means to control the performance of these statements.

SQL plan baselines are the key objects that SQL Plan Management (SPM) uses to prevent unverified change to SQL execution plans. When SPM is active, there will not be drastic changes in performance, even as the statistics change or as the database version changes. Until a new plan is verified to produce better performance than the current plan, it will not be considered by the optimizer. This in effect freezes the SQL plan.

SQL outlines have been used in past versions. They are still available for backward compatibility, but outlines are deprecated in favor of SPM.

SQL Plan Management: Overview

- SQL Plan Management is automatically controlled SQL plan evolution.
- Optimizer automatically manages SQL plan baselines.
  - Only known and verified plans are used.
- Plan changes are automatically verified.
  - Only comparable or better plans are subsequently used.
- The plan baseline can be seeded for critical SQL with SQL tuning set (STS) from SQL Performance Analyzer.

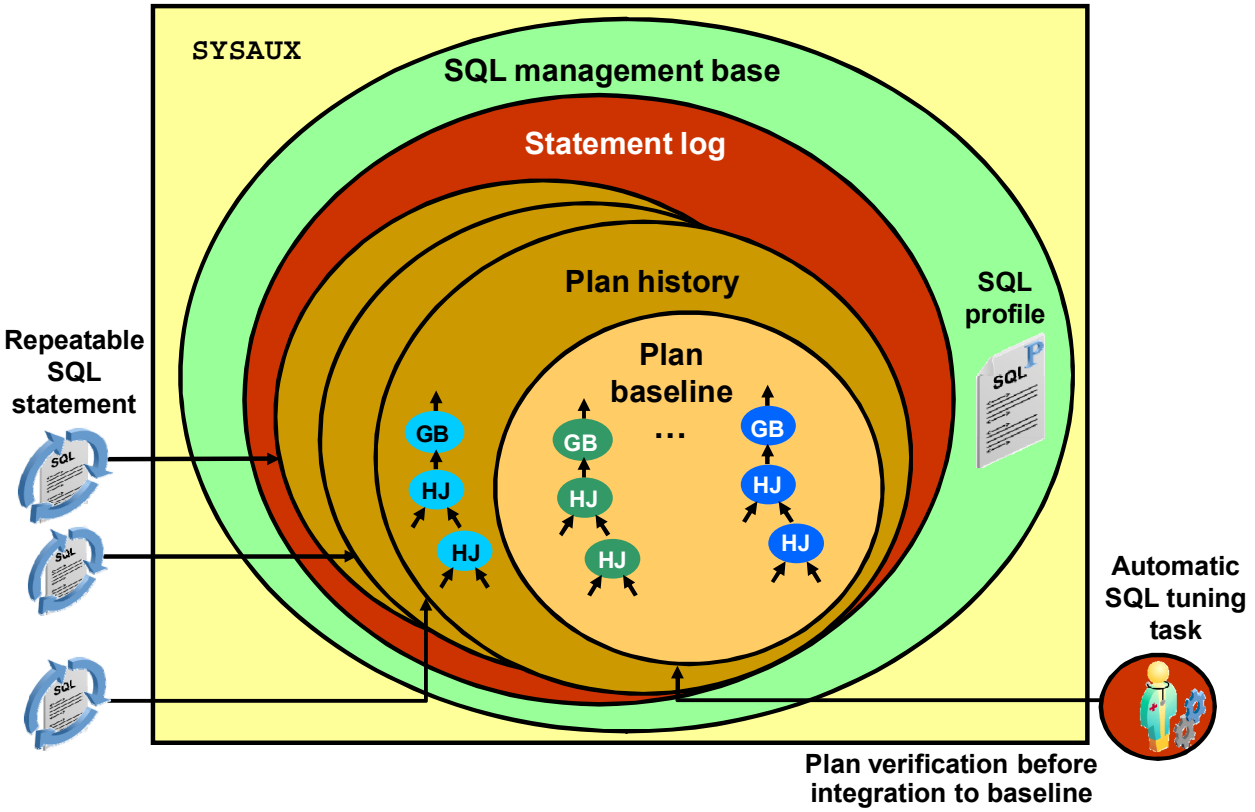
Potential performance risk occurs when the SQL execution plan changes for a SQL statement. A SQL plan change can occur due to a variety of reasons, such as optimizer version, optimizer statistics, optimizer parameters, schema definitions, system settings, and SQL profile creation.

Various plan control techniques are available in Oracle Database to address performance regressions due to plan changes. The oldest is the use of hints in the SQL code to force a specific access path. Stored outlines allowed the hints to be stored separately from the code and modified. Both techniques focused on making the plan static. SQL profiles created by the SQL Tuning Advisor allow the optimizer to collect and store additional statistics that will guide the choice of a plan; if the plan becomes inefficient, the Tuning Advisor can be invoked to produce a new profile.

SPM automatically controls SQL plan evolution by maintaining what is called “SQL plan baselines.” With this feature enabled, a newly-generated SQL plan can join a SQL plan baseline only if it has been proven that doing so will not result in performance regression. During the execution of a SQL statement, only a plan that is part of the SQL plan baseline can be used. As described later in this lesson, SQL plan baselines can be automatically loaded or they can be seeded by using SQL tuning sets. Various scenarios are covered later in this lesson.

The main benefit of the SPM feature is performance stability of the system by avoiding plan regressions. In addition, it saves the DBA time that is often spent in identifying and analyzing SQL performance regressions and finding workable solutions.

SQL Plan Baseline: Architecture



The SPM feature introduces necessary infrastructure and services in support of plan maintenance and performance verification of new plans.

For SQL statements that are executed more than once, the optimizer maintains a history of plans for individual SQL statements. The optimizer recognizes a repeatable SQL statement by maintaining a statement log. A SQL statement is recognized as repeatable when it is parsed or executed again after it has been logged. After a SQL statement is recognized as repeatable, various plans generated by the optimizer are maintained as a plan history containing relevant information (such as SQL text, outline, bind variables, and compilation environment) that is used by the optimizer to reproduce an execution plan.

The DBA may also add plans to the SQL plan baseline by manually seeding a set of SQL statements.

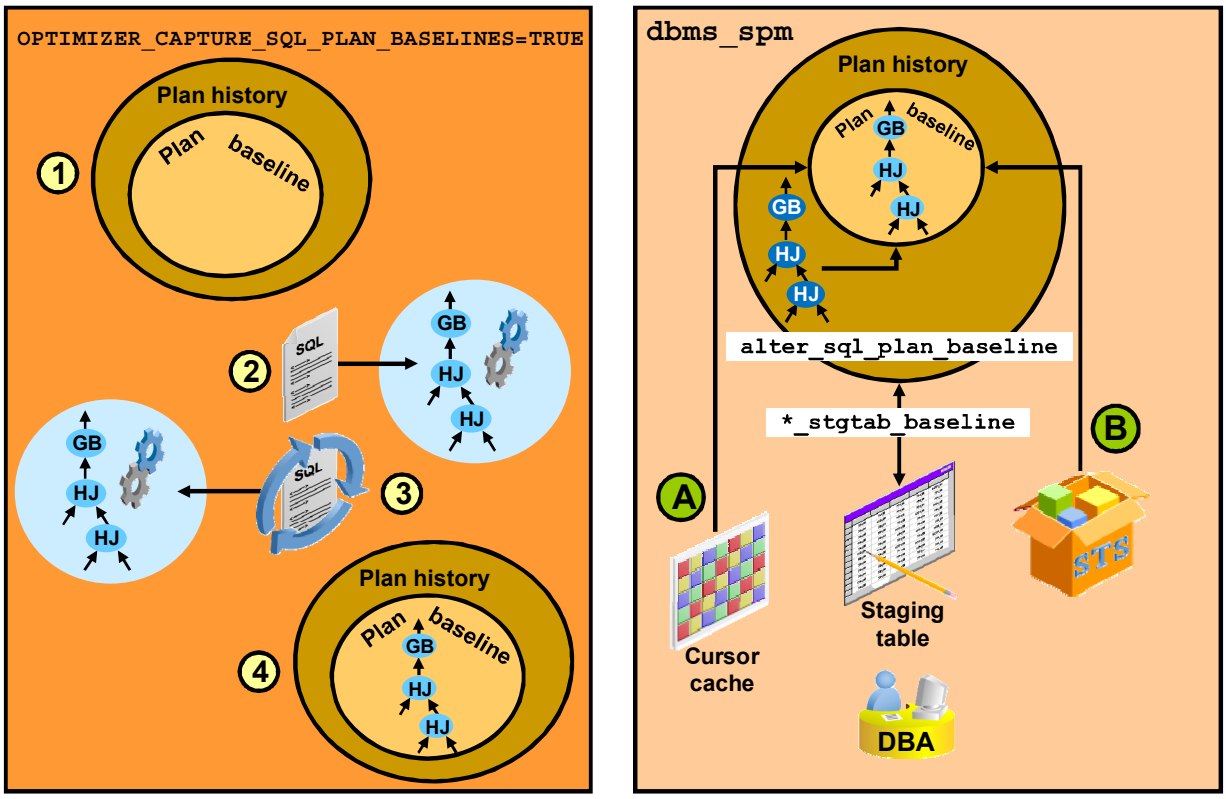
A plan history contains different plans generated by the optimizer for a SQL statement over time. However, only some of the plans in the plan history may be accepted for use. For example, a new plan generated by the optimizer is not normally used until it has been verified not to cause a performance regression. Plan verification is done by default as part of the Automatic SQL Tuning task that is running as an automated task in a maintenance window.

An Automatic SQL Tuning task targets only high-load SQL statements. For those statements, it automatically implements such actions as making a successfully-verified plan an accepted plan. A set of acceptable plans constitutes a SQL plan baseline. The very first plan generated for a SQL statement is obviously acceptable for use; therefore, it forms the original plan baseline. Any new plans subsequently found by the optimizer are part of the plan history, but they are not part of the initial plan baseline.

The statement log, plan history, and plan baselines are stored in the SQL management base (SMB), which also contains SQL profiles. The SMB is part of the database dictionary and is stored in the `SYSAUX` tablespace. The SMB has automatic space management (for example, periodic purging of unused plans). You can configure the SMB to change the plan retention policy and set space size limits.

Note: With Oracle Database 11*g*, if the database instance is up but the `SYSAUX` tablespace is `OFFLINE`, the optimizer is unable to access SQL management objects. This situation can affect performance on some of the SQL workload.

Loading SQL Plan Baselines

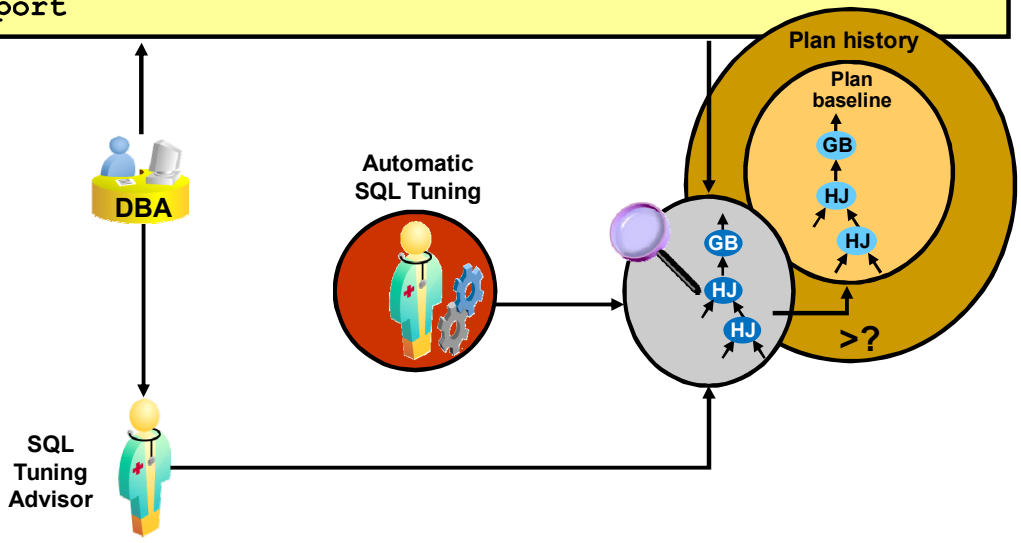


There are two ways to load SQL plan baselines.

- **Automatic Plan Capture:** To use automatic plan capture, set the `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` initialization parameter to `TRUE`. This parameter is set to `FALSE` by default. Setting it to `TRUE` turns on automatic recognition of repeatable SQL statements and automatic creation of plan history for such statements. This is illustrated in the left graphic in the slide, where the first generated SQL plan is automatically integrated into the original SQL plan baseline when it becomes a repeating SQL statement.
- **Bulk loading:** Uses the `DBMS_SPM` package, which enables you to manually manage SQL plan baselines. With procedures from this package, you can load SQL plans into a SQL plan baseline directly from the cursor cache (A) by using `LOAD_PLANS_FROM_CURSOR_CACHE` or from an existing STS (B) by using `LOAD_PLANS_FROM_SQLSET`. For a SQL statement to be loaded into a SQL plan baseline from an STS, the plan for the SQL statement needs to be stored in the STS. `DBMS_SPM.ALTER_SQL_PLAN_BASELINE` enables you to enable and disable a baseline plan and change other plan attributes. To move baselines between databases, use the `DBMS_SPM.*_STGTAB_BASELINE` procedures to create a staging table and to export and import baseline plans from a staging table. The staging table can be moved between databases by using the Data Pump Export and Import utilities.

Evolving SQL Plan Baselines

```
variable report clob

exec :report:=DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE(-
sql_handle=>'SYS_SQL_593bc74fca8e6738');

Print report
```

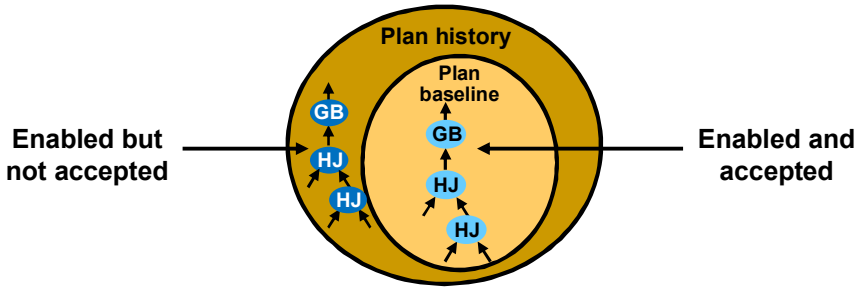


When the optimizer finds a new plan for a SQL statement, the plan is added to the plan history as a nonaccepted plan. The plan will not be accepted into the SQL plan baseline until it is verified for performance relative to the SQL plan baseline performance. Verification means a nonaccepted plan does not cause a performance regression (either manually or automatically). The verification of a nonaccepted plan consists of comparing its performance to the performance of one plan selected from the SQL plan baseline and ensuring that it delivers better performance.

There are two ways to evolve SQL plan baselines:

- **By using the `DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE` function:** An example is shown in the slide. The function returns a report that tells you whether some of the existing history plans were moved to the plan baseline. The example specifies a specific plan in the history to be tested. The function also allows verification without accepting the plan.
- **By running SQL Tuning Advisor:** SQL plan baselines can be evolved by manually or automatically tuning SQL statements with SQL Tuning Advisor. When it finds a tuned plan and verifies its performance to be better than a plan chosen from the corresponding SQL plan baseline, it makes a recommendation to accept a SQL profile. When the SQL profile is accepted, the tuned plan is added to the corresponding SQL plan baseline.

Important Baseline SQL Plan Attributes




```
select signature, sql_handle, sql_text, plan_name, origin, enabled,
       accepted, fixed, autopurge
from dba_sql_plan_baselines;
```

```
SIGNATURE  SQL_HANDLE  SQL_TEXT PLAN_NAME       ORIGIN       ENA ACC FIX AUT
---------- ----------- -------- --------------- ------------ --- --- --- ---
8.062E+18 SYS_SQL_6fe2 select.. SQL_PLAN_6zsn…   AUTO-CAPTURE YES NO  NO  YES
8.062E+18 SYS_SQL_e23f select.. SQL_PLAN_f4gy…   AUTO-CAPTURE YES YES NO  YES
…
```

```
 exec :cnt := dbms_spm.alter_sql_plan_baseline(-
              sql_handle => 'SYS_SQL_6fe28d438dfc352f', -
              plan_name       => 'SQL_PLAN_6zsnd8f6zsd9g54bc8843',-
              attribute_name  => 'ENABLED', attribute_value => 'NO');
```

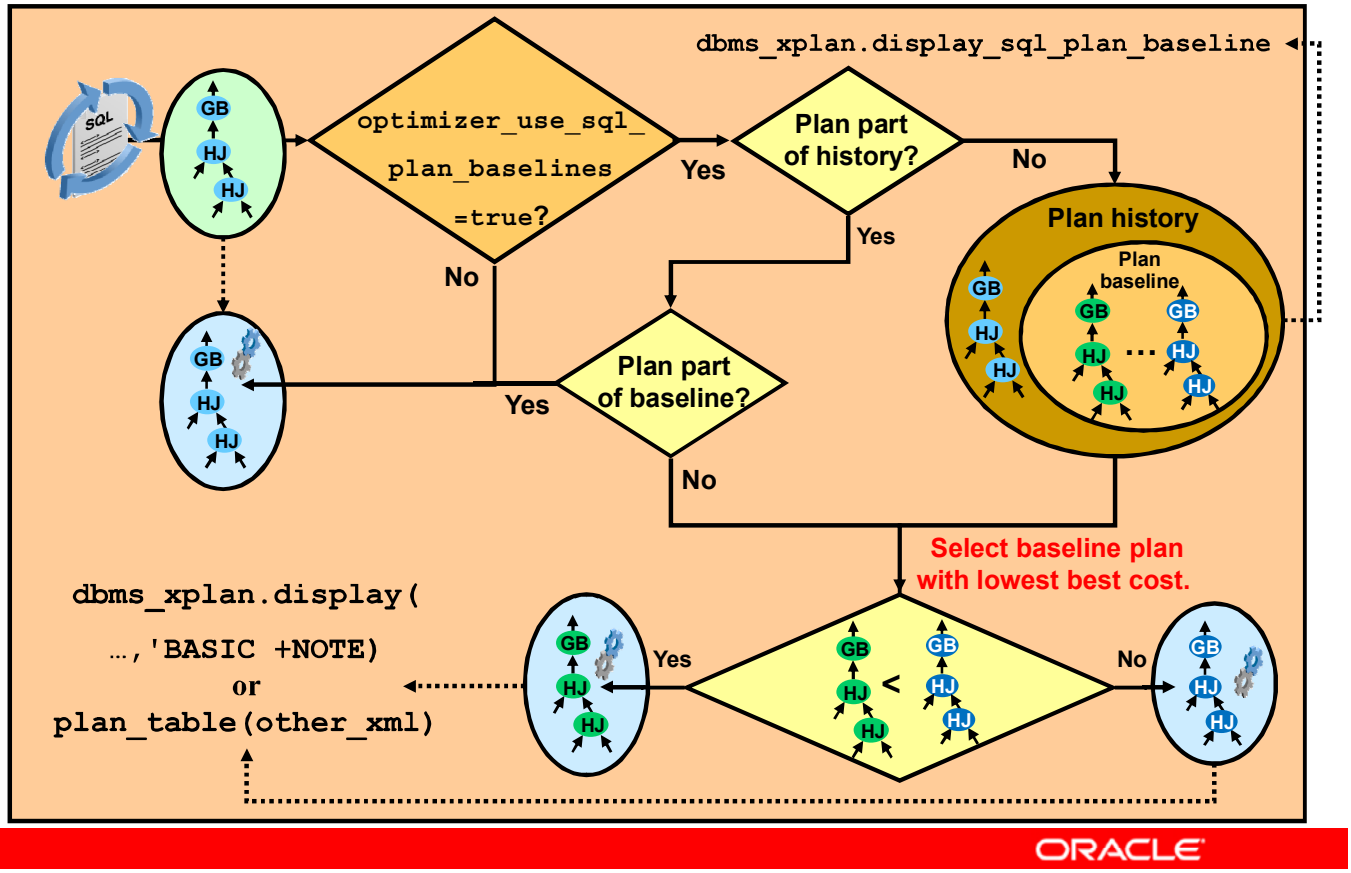

When a plan enters the plan history, it is associated with a number of important attributes:

- `SIGNATURE`, `SQL_HANDLE`, `SQL_TEXT`, and `PLAN_NAME` are important identifiers for search operations.
- ORIGIN allows you to determine whether the plan was automatically captured (`AUTO-CAPTURE`), manually evolved (`MANUAL-LOAD`), automatically evolved by SQL Tuning Advisor (`MANUAL-SQLTUNE`), or automatically evolved by Automatic SQL Tuning (`AUTO-SQLTUNE`).
- The `ENABLED` and `ACCEPTED` attributes must be set to `YES` or else the plan is not considered by the optimizer. The `ENABLED` attribute means that the plan is enabled for use by the optimizer. The `ACCEPTED` attribute means that the plan was validated as a good plan, either automatically by the system or manually when the user changes it to `ACCEPTED`. When a plan status changes to `ACCEPTED`, it will continue to be `ACCEPTED` until `DBMS_SPM.ALTER_SQL_PLAN_BASELINE()` is used to change its status. An `ACCEPTED` plan can be temporarily disabled by removing the `ENABLED` setting.

- FIXED means that the optimizer considers only those plans and not other plans. For example, if you have 10 baseline plans and three are marked FIXED, the optimizer uses only the best plan from these three, ignoring all the others. A SQL plan baseline is said to be FIXED if it contains at least one enabled fixed plan. If new plans are added to a fixed SQL plan baseline, they cannot be used until they are manually declared as FIXED. You can look at each plan's attributes by using the DBA_SQL_PLAN_BASELINES view, as shown in the slide. You can then use the DBMS_SPM.ALTER_SQL_PLAN_BASELINE function to change some of them. You can also remove plans or a complete plan history by using the DBMS_SPM.DROP_SQL_PLAN_BASELINE function. The example shown in the slide changes the ENABLED attribute of SQL_PLAN_6zsnd8f6zsd9g54bc8843 to NO.

Note: The DBA_SQL_PLAN_BASELINES view contains additional attributes that enable you to determine when each plan was last used and whether a plan should be automatically purged.

SQL Plan Selection



If you are using automatic plan capture, the first time that a SQL statement is recognized as repeatable, its best-cost plan is added to the corresponding SQL plan baseline. That plan is then used to execute the statement.

The optimizer uses a comparative plan selection policy when a plan baseline exists for a SQL statement and the `OPTIMIZER_USE_SQL_PLAN_BASELINES` initialization parameter is set to `TRUE` (default value). Each time a SQL statement is compiled, the optimizer first uses the traditional cost-based search method to build a best-cost plan. Then it tries to find a matching plan in the SQL plan baseline. If a match is found, it proceeds as usual. If no match is found, it first adds the new plan to the plan history, then calculates the cost of each accepted plan in the SQL plan baseline, and then picks the one with the lowest cost. The accepted plans are reproduced by using the outline that is stored with each of them. So the effect of having a SQL plan baseline for a SQL statement is that the optimizer always selects one of the accepted plans in that SQL plan baseline.

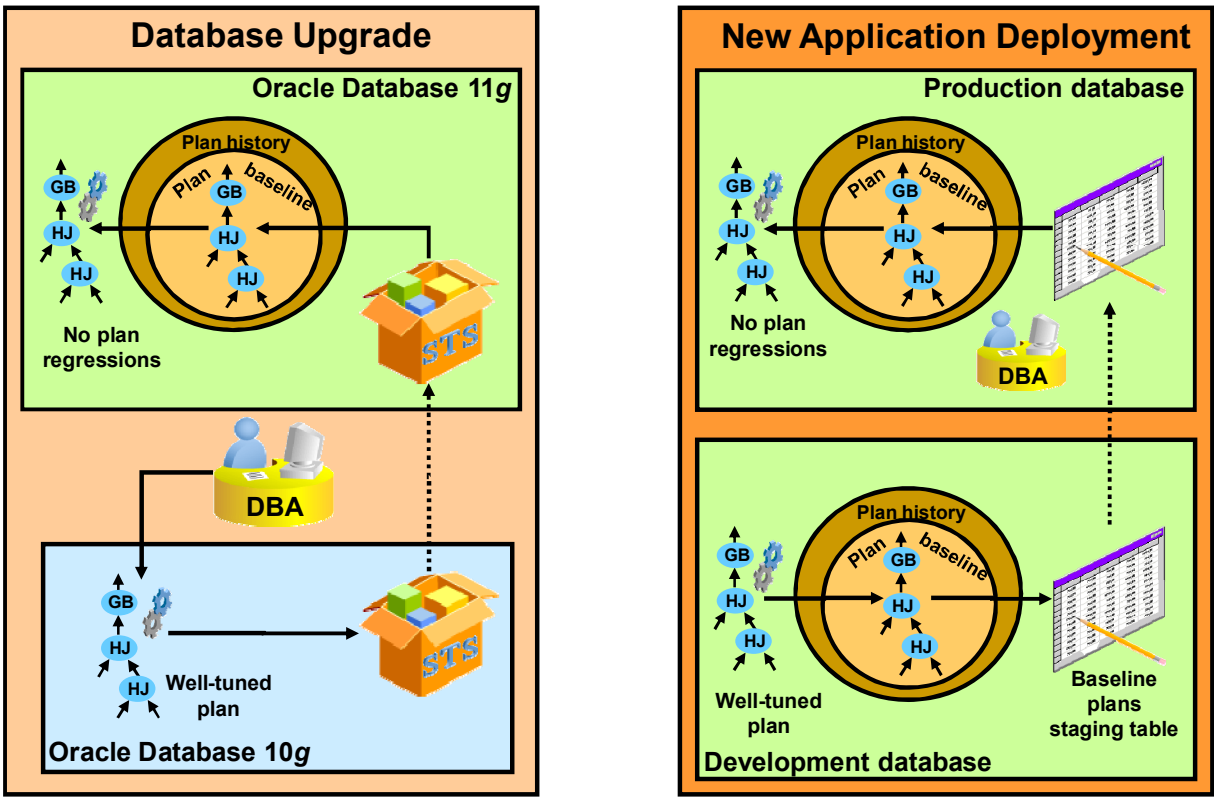
With SPM, the optimizer can produce a plan that could be either a best-cost plan or a baseline plan. This information is dumped in the `other_xml` column of the `plan_table` upon `EXPLAIN PLAN`. However, the optimizer can use only an accepted and enabled baseline plan.

In addition, you can use the new `dbms_xplain.display_sql_plan_baseline` function to display one or more execution plans for the specified `sql_handle` of a plan baseline. If `plan_name` is also specified, the corresponding execution plan is displayed.

Note: The Stored Outline feature is deprecated. To preserve backward compatibility, if a stored outline for a SQL statement is active for the user session, the statement is compiled by using the stored outline. In addition, a plan generated by the optimizer using a stored outline is not stored in the SMB even if automatic plan capture has been enabled for the session.

Stored outlines can be migrated to SQL plan baselines by using the `MIGRATE_STORED_OUTLINE` procedure from the `DBMS_SPM` package. When the migration is complete, you should disable or drop the original stored outline by using the `DROP_MIGRATED_STORED_OUTLINE` procedure of the `DBMS_SPM` package.

Possible SQL Plan Manageability Scenarios

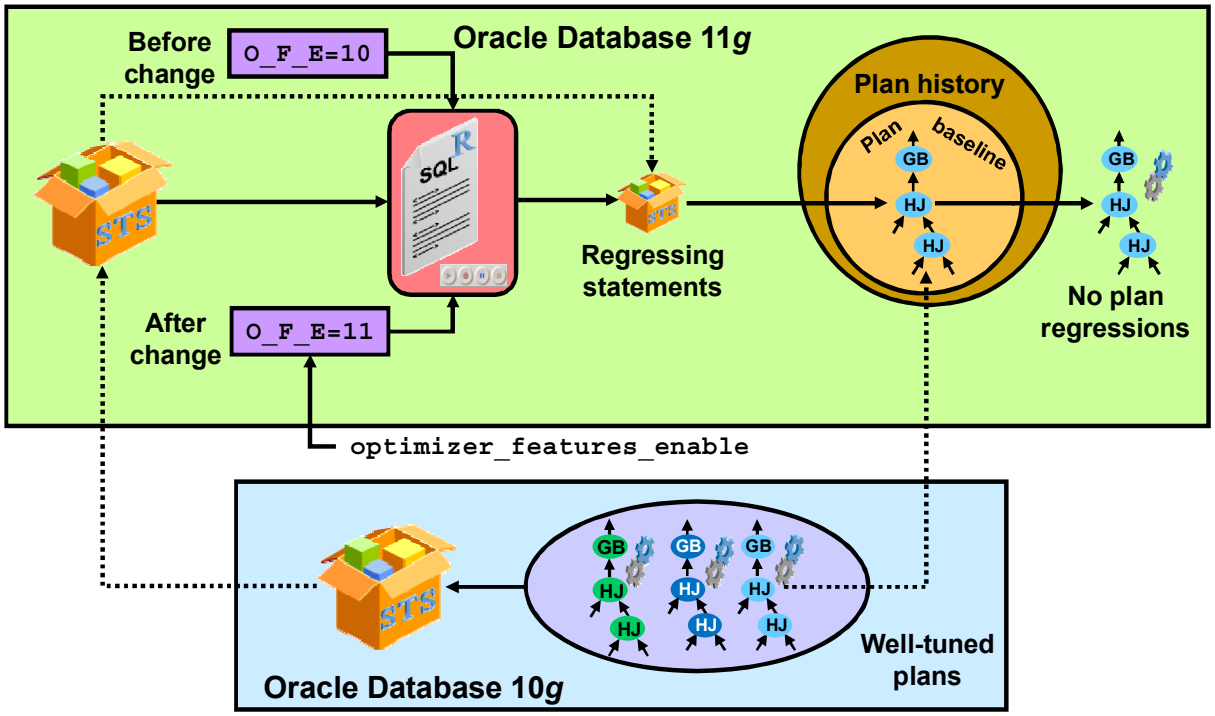


- **Database upgrade:** Bulk SQL plan loading is especially useful when the system is being upgraded from an earlier version to Oracle Database 11*g*. For this, you can capture plans for a SQL workload into an STS before the upgrade, and then load these plans from the STS into the SQL plan baseline immediately after the upgrade. This strategy can minimize plan regressions resulting from the use of the new optimizer version.
- **New application deployment:** The deployment of a new application module means the introduction of new SQL statements into the system. The software vendor can ship the application software along with the appropriate SQL plan baselines for the new SQL being introduced. Because of the plan baselines, the new SQL statements will initially run with the plans that are known to give good performance under a standard test configuration. However, if the customer system configuration is very different from the test configuration, the plan baselines can be evolved over time to produce better performance.

In both scenarios, you can use the automatic SQL plan capture after manual loading to make sure that only better plans will be used for your applications in the future.

Note: In all scenarios in this lesson, assume that `OPTIMIZER_USE_SQL_PLAN_BASELINES` is set to `TRUE`.

SQL Performance Analyzer and SQL Plan Baseline Scenario

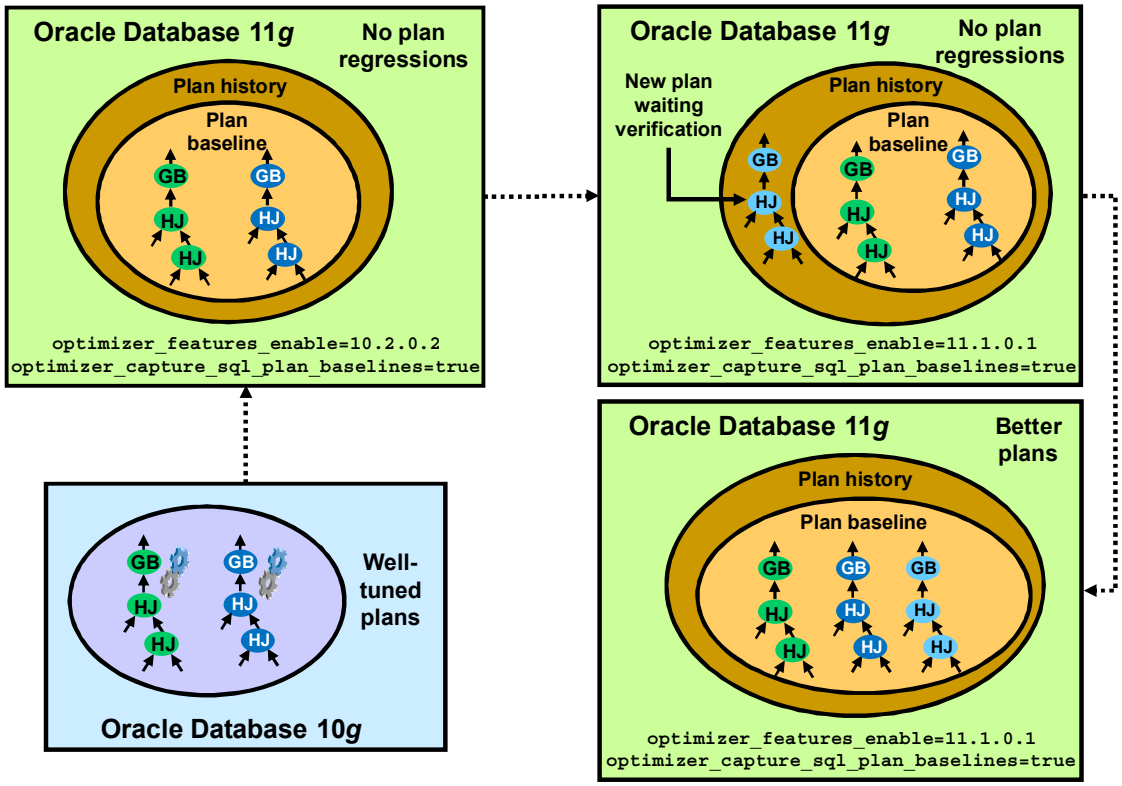


A variation of the first method described in the previous slide is through the use of SQL Performance Analyzer. You can capture pre-Oracle Database 11*g* plans in an STS and import them into Oracle Database 11*g*. Then set the `optimizer_features_enable` (`O_F_E`) initialization parameter to 10.1.0 to make the optimizer behave as if this were Oracle Database 10*g*. Next run SQL Performance Analyzer for the STS. When that is complete, set the `optimizer_features_enable` initialization parameter back to 11.2.0 and rerun SQL Performance Analyzer for the STS.

SQL Performance Analyzer produces a report that lists a SQL statement whose plan has regressed from 10*g* to 11*g*. For those SQL statements that are shown by SQL Performance Analyzer to incur performance regression due to the new optimizer version, you can capture their plans by using an STS and then load them into the SMB.

This method represents the best form of the plan-seeding process because it helps prevent performance regressions while preserving performance improvements upon database upgrade.

Loading a SQL Plan Baseline Automatically



Another upgrade scenario involves using the automatic SQL plan capture mechanism. In this case, set the initialization parameter `optimizer_features_enable` (`O_F_E`) to the pre-Oracle Database 11*g* version value for an initial period of time, such as a quarter, and execute your workload after upgrade by using the automatic SQL plan capture.

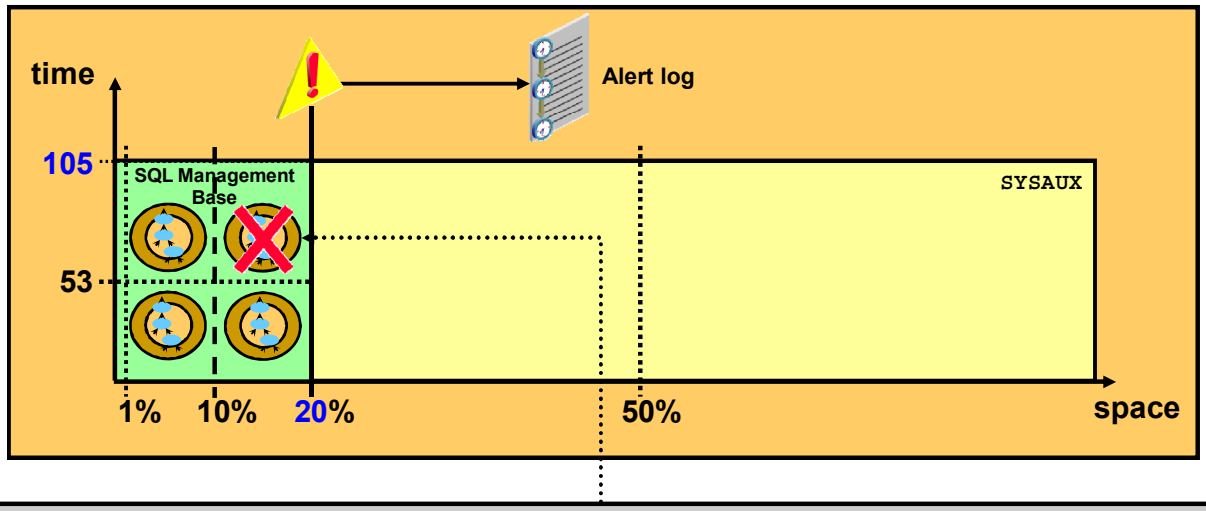
During this initial time period, because of the `O_F_E` parameter setting, the optimizer is able to reproduce pre-Oracle Database 11*g* plans for a majority of the SQL statements. Because automatic SQL plan capture is also enabled during this period, the pre-Oracle Database 11*g* plans produced by the optimizer are captured as SQL plan baselines.

When the initial time period ends, you can remove the setting of `O_F_E` to take advantage of the new optimizer version while incurring minimal or no plan regressions due to the plan baselines. Regressed plans will use the previous optimizer version; nonregressed statements will benefit from the new optimizer version.

Purging SQL Management Base Policy

```
SQL> exec dbms_spm.configure('SPACE_BUDGET_PERCENT',20);
SQL> exec dbms_spm.configure('PLAN_RETENTION_WEEKS',105);
```

DBA_SQL_MANAGEMENT_CONFIG




```
SQL> exec :cnt := dbms_spm.drop_sql_plan_baseline('SYS_SQL_37e0168b04e73efe');
```

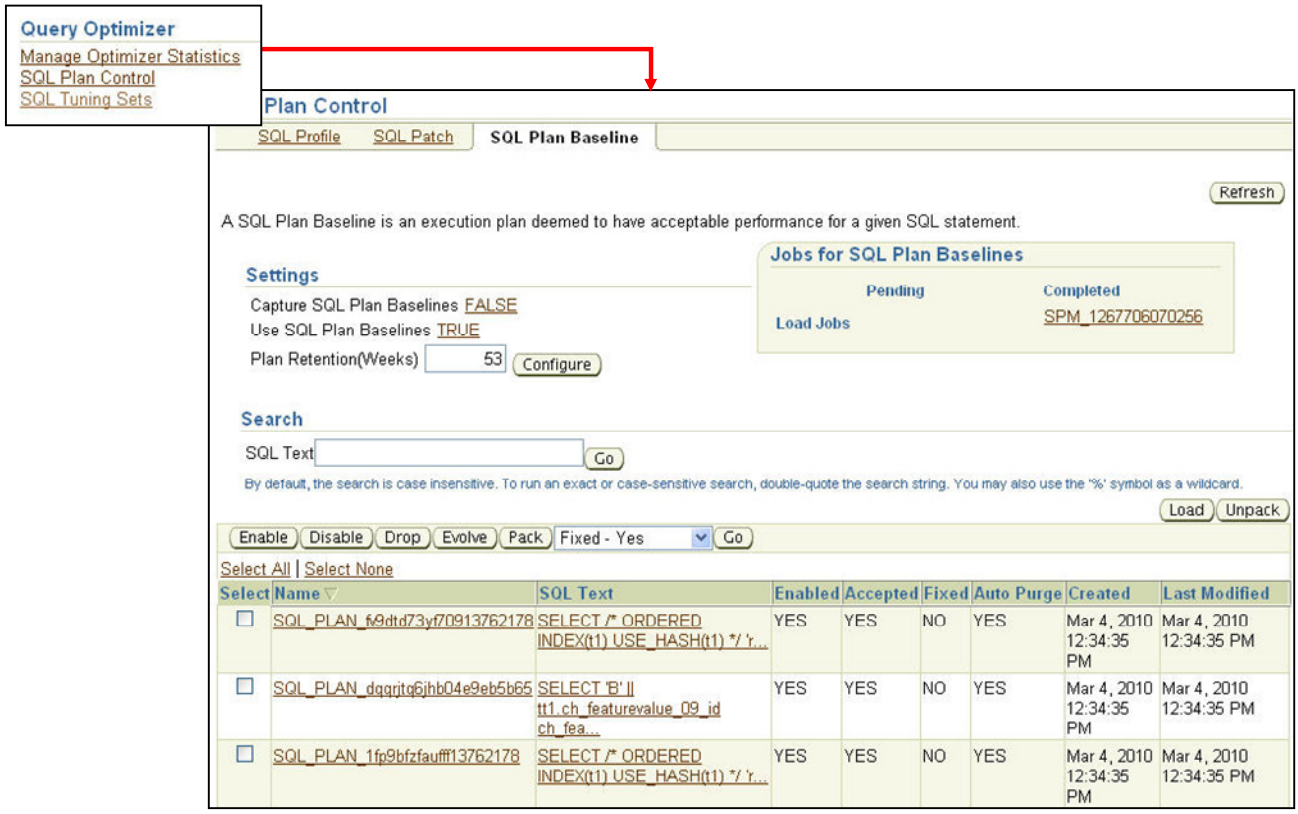

The space occupied by the SMB is checked weekly against a defined limit. A limit based on the percentage size of the SYSAUX tablespace is defined. By default, the space budget limit for the SMB is set to 10 percent of SYSAUX size. However, you can configure SMB and change the space budget to a value between 1 percent and 50 percent by using the DBMS_SPM.CONFIGURE procedure.

If SMB space exceeds the defined percentage limit, warnings are written to the alert log. Warnings are generated weekly until the SMB space limit is increased, the size of SYSAUX is increased, or the size of SMB is decreased by purging some of the SQL management objects (such as SQL plan baselines or SQL profiles).

The space management of SQL plan baselines is done proactively with a weekly purging task. The task runs as an automated task in the maintenance window. Any plan that has not been used for more than 53 weeks is purged. However, you can configure SMB and change the unused plan retention period to a value between 5 weeks and 523 weeks (a little more than 10 years). To do so, use the DBMS_SPM.CONFIGURE procedure.

You can look at the current configuration settings for the SMB by examining the DBA_SQL_MANAGEMENT_CONFIG view. In addition, you can manually purge the SMB by using the DBMS_SPM.DROP_SQL_PLAN_BASELINE function (as shown in the example in the slide).

Enterprise Manager and SQL Plan Baselines

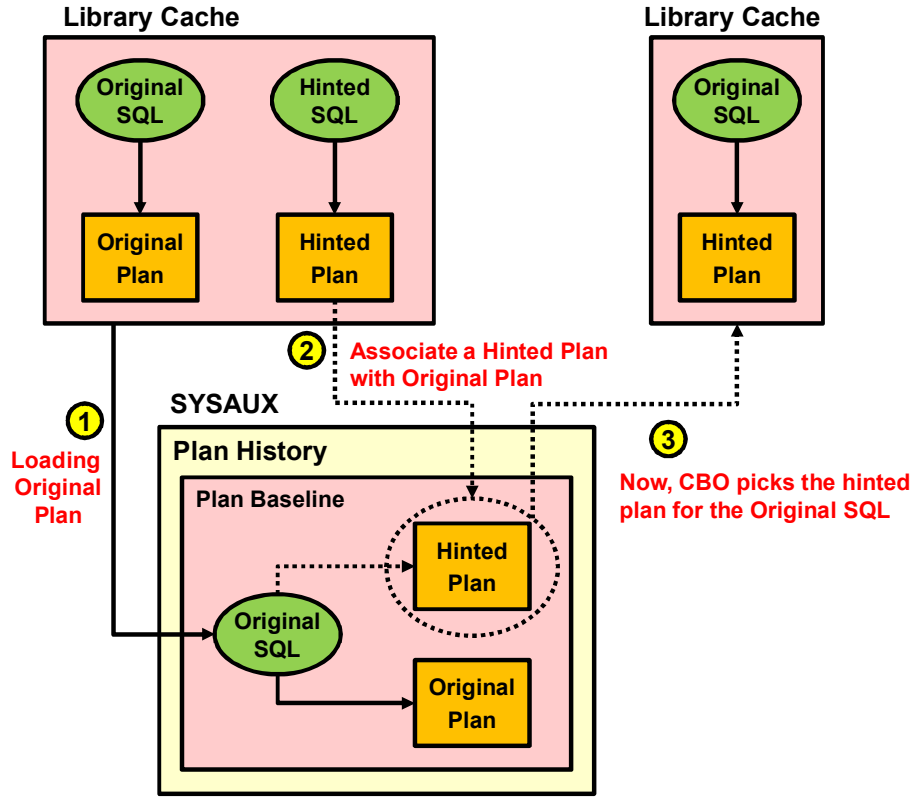


Use the SQL Plan Management page to manage SQL profiles, SQL patches, and SQL plan baselines from one location rather than from separate locations in Enterprise Manager. You can also enable, disable, drop, pack, unpack, load, and evolve selected baselines.

From this page, you can also configure the various SQL plan baseline settings.

To navigate to this page, click the Server tab, and then click the SQL Plan Control entry in the Query Optimizer section.

Loading Hinted Plans into SPM: Example



You can create SQL plan baselines for SQL statements:

- Coming from an application where the SQL statement cannot be modified
- Which need hints to run a good execution plan

To load hinted plans into SPM:

1. Capture the SQL plan baseline for the original SQL, which is the problem SQL identified.

```
var res number ;
exec :res := dbms_spm.load_plans_from_cursor_cache(sql_id =>
   '&original_sql_id',    plan_hash_value =>
   '&original_plan_hash_value' );
```

2. Add hints into the problem SQL and execute the hinted SQL.
3. Find the SQL_ID and plan_hash_value for the hinted SQL.

```
select * from table(dbms_xplan.display_cursor);
```

4. Verify that the original SQL baseline exists.

```
select sql_text, sql_handle, plan_name, enabled, accepted from
dba_sql_plan_baselines;
```

5. Associate the hinted execution plan with the original sql_handle.

```
var res number
exec :res := dbms_spm.load_plans_from_cursor_cache( -
sql_id => '&hinted_SQL_ID', -
plan_hash_value => &hinted_plan_hash_value, -
sql_handle => '&sql_handle_for_original');
```

6. Verify that the new baseline was added.

```
select sql_text, sql_handle, plan_name, enabled, accepted from
dba_sql_plan_baselines;
```

7. If the original plan capture is not needed initially, drop or disable it.

```
exec :res :=DBMS_SPM.DROP_SQL_PLAN_BASELINE
('&original_sql_handle','&original_plan_name');
```

8. Reexecute the original SQL to verify that the SQL statement is now using the SQL plan baseline.

```
select SQL_PLAN_BASELINE from V$SQL where
SQL_ID='&original_SQL_ID‘
```

Note: See MOS note 215187.1 for a `SQLTXPLAIN` script (`./utl/coe_load_sql_baseline.sql`) that automates the steps. Review the script before running it.

Using the MIGRATE_STORED_OUTLINE Functions

- Specify stored outlines to be migrated based on outline name, SQL text, or outline category, or migrate all stored outlines in the system to SQL plan baselines:

```
DBMS_SPM.MIGRATE_STORED_OUTLINE (
attribute_name IN VARCHAR2,
attribute_value IN CLOB,
fixed IN VARCHAR2 := 'NO')
RETURN CLOB;
```

- Specify one or more stored outlines to be migrated:

```
DBMS_SPM.MIGRATE_STORED_OUTLINE (
outln_list IN DBMS_SPM.NAME_LIST,
fixed IN VARCHAR2 := 'NO')
RETURN CLOB;
```


You can use the `DBMS_SPM.MIGRATE_STORED_OUTLINE` function to migrate stored outlines for one or more SQL statements to plan baselines in the SMB. Specify which stored outlines to migrate based on the outline name, SQL text, or outline category. You can also migrate all stored outlines in the system to SQL plan baselines. The parameters are as follows:

- `attribute_name`: Specifies the type of parameter used in `attribute_value` to identify the migrated stored outlines. Values (case-sensitive) are `outline_name`, `sql_text`, `category`, and `all`.
- `attribute_value`: Based on the value specified in `attribute_name`. `NULL` if `attribute_name` is `all`.
- `fixed`: Values of `NO` (default) and `YES`. Specifies the “fixed” status of the plans generated during migration. By default, plans are generated as “non-fixed” plans.

The second overload of the function is used to migrate stored outlines for one or more SQL statements to plan baselines in the SMB given one or more outline names. The parameters are as follows:

- `outln_list`: List of outline names to be migrated.
- `fixed`: Values of `NO` (default) and `YES`. Specifies the “fixed” status of the plans generated during migration. By default, plans are generated as “non-fixed” plans.

A returned `CLOB` contains a formatted report that describes the statistics during the migration.

Note: When choosing a plan for a SQL statement, stored outlines take priority over plan baselines. For more information on migrating stored outlines to SQL plan baselines, read section 15.8 in the following link:
http://docs.oracle.com/cd/E18283_01/server.112/e16638/optplanmgmt.htm#BABFCFHC

Quiz

When the `OPTIMIZER_USE_SQL_PLAN_BASELINES` parameter is set to `TRUE`, the optimizer:

a. Does not develop an execution plan; it uses an accepted plan in the baselines
b. Compares the plan that it develops with accepted plans in the baselines
c. Compares the plan that it develops with enabled plans in the baselines
d. Does not develop an execution plan; it uses enabled plans in the baselines
e. Develops plans and adds them to the baselines as verified

Answer: b

The optimizer always develops an execution plan, and then compares the plan with accepted plans in the SQL baseline. If an accepted baseline exists, the baseline is used. If the plan developed by the optimizer is different, it is stored in the plan history, but it is not part of the baseline until it is verified and marked as accepted.

Summary

In this lesson, you should have learned how to:

- Manage SQL performance through changes
- Set up SQL Plan Management
- Set up various SQL Plan Management scenarios
- Load hinted plans into SQL Plan Management
- Migrate Stored Outlines to SQL Plan Baselines

Practice 12: Overview
Using SQL Plan Management

This practice covers the use of SQL Plan Management.

13

Workshop

Objectives

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

- Identify some common problems in writing SQL statement
- Determine the common causes and observations
- Understand the different tuning approaches
- Understand the behavior of the optimizer
- Understand the effects of functions on index usage
- Tune sort operations for `ORDER BY` clause
- Tune high parse time using parse time reduction strategy
- Maintain stable execution plans over time

Overview

- This lesson is made up of nine workshops based on the SQL tuning techniques covered in the previous lessons.
- All workshops are based on problem solving.
- You have to find ways to enhance performance through several tasks in each workshop.
- You also review concepts learned in each workshop through quizzes.
- Finally, by resolving the problems, you reinforce your learning.

Workshop 1

Introduction:

- In this workshop, you have to find a workaround to enhance performance.
- You analyze a poorly-written SQL statement and perform additional tasks such as creating a function-based index, redesigning a simple table, and rewriting the SQL statement.

Workshop 2

Introduction:

- This workshop is to review the *execution steps of a SQL statement* section.
- You review the SQL statement that uses a bind variable in the indexed column.
- You analyze two execution plans of the SQL statement using the `EXPLAIN PLAN` command and V$ view.
- You should be able to understand why the two execution plans are different for the same SQL statement.

Workshop 3

Introduction:

- This workshop shows the possible usage of an index in the `ORDER BY` clause to tune a sort operation.
- You perform several tasks to be able to use the indexed `ORDER BY` column.
- Once all tasks are done, you verify if the index always produces a better plan cost.

Workshop 4

Introduction:

- In this workshop, you assume that you have identified a poorly-running SQL statement.
- You already noticed that a root cause of the issue is the table design issue. However, recreating the table is not an option at this point. The table is already in use.
- Try to tune the SQL statement by rewriting it.

Workshop 5

Introduction:

- In this workshop, you write several SQL statements that return the same output to see which SQL statement is more efficient.
- You also examine the effects of changing column order in a composite index.

Workshop 6 & 7

Introduction:

- In these workshops, you review the execution plan and several sections in an event 10053 trace file.
- You interpret the information to understand the optimizer's decision.
- There are several questions on cost model, selectivity, and cardinality.
- Finally, you learn how to use the information in the 10053 file to tune the SQL statement.

Workshop 8

Introduction:

- You have identified the slow-running SQL statement. It was run 534 times in a certain time period. You noticed that a different literal was used in each execution. This caused the system to parse the same statement 534 times with the high parse time, which is not efficient.
- You perform several tasks to tune the high parse time.
- What tuning strategy could be used?

Workshop 9

Introduction:

- Oracle has introduced several methods to maintain the stable execution plans of SQL statements such as hints, stored outlines, and SQL profile. In Oracle 11*g*, we can use a better way to manage the execution plan called SQL Plan Baseline. This feature allows the optimizer to make the right decision in choosing a verified or tested execution plan in SQL Plan Baseline even if the optimizer generates a new plan during hard parse.
- In this workshop, you learn how to use this feature to associate a hinted execution plan with a hard-coded SQL statement.

ORACLE

Summary

In this lesson, you should have learned how to:

- Identify some common problems in writing SQL statement
- Determine the common causes and observations
- Understand the different tuning approaches
- Understand the behavior of the optimizer
- Understand the effects of functions on index usage
- Tune sort operations for `ORDER BY` clause
- Tune high parse time using parse time reduction strategy
- Maintain stable execution plans over time

SQL Tuning Advisor

Objectives

After completing this lesson, you should be able to:

- Describe statement profiling
- Use SQL Tuning Advisor

Tuning SQL Statements Automatically

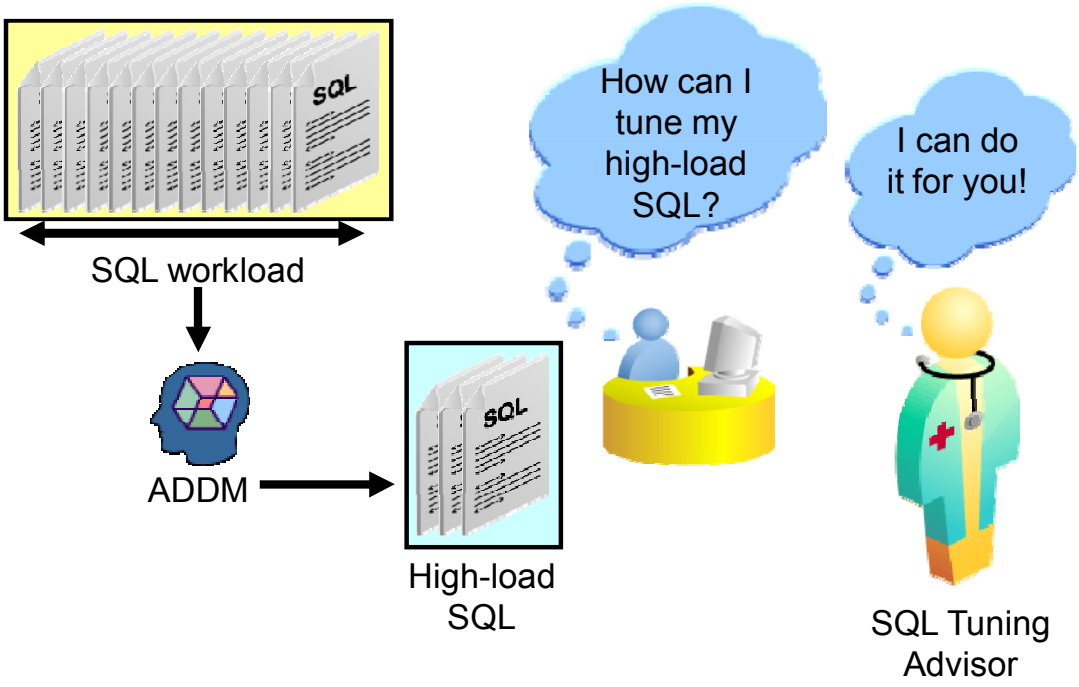
- Tuning SQL statements automatically eases the entire SQL tuning process and replaces manual SQL tuning.
- The optimizer has two modes:
  - Normal mode
  - Tuning mode or Automatic Tuning Optimizer (ATO)
- You use SQL Tuning Advisor to access the tuning mode.
- You should use tuning mode only for high-load SQL statements.

Tuning SQL statements automatically is the capability of the query optimizer to automate the entire SQL tuning process. This automatic process replaces manual SQL tuning, which is a complex, repetitive, and time-consuming function. SQL Tuning Advisor exposes the features of SQL tuning to the user. The enhanced query optimizer has two modes:

- **Normal mode:** The optimizer compiles SQL and generates an execution plan. The normal mode of the optimizer generates a reasonable execution plan for the majority of SQL statements. In the normal mode, the optimizer operates with very strict time constraints, usually a fraction of a second, during which it must find a good execution plan.
- **Tuning mode:** The optimizer performs additional analysis to check whether the execution plan produced under the normal mode can be further improved. The output of the query optimizer in the tuning mode is not an execution plan, but a series of actions, along with their rationale and expected benefit (for producing a significantly superior plan). When called under tuning mode, the optimizer is referred to as Automatic Tuning Optimizer (ATO). The tuning performed by ATO is called system SQL tuning.

  In the tuning mode, the optimizer can take several minutes to tune a single statement. ATO is meant to be used for complex and high-load SQL statements that have a nontrivial impact on the entire system.

Application Tuning Challenges

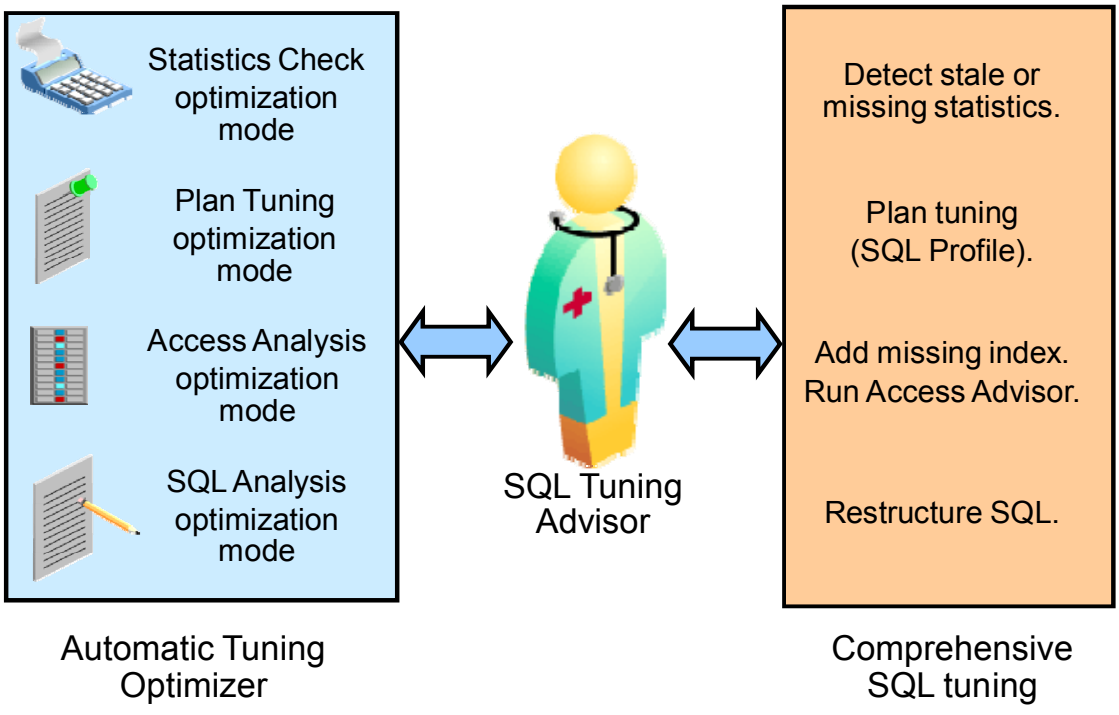


The process of identifying high-load SQL statements and tuning them is very challenging even for an expert. SQL tuning is not only one of the most critical aspects of managing the performance of a database server, but also one of the most difficult tasks to accomplish. Starting with Oracle Database 10*g*, the task of identifying high-load SQL statements has been automated by Automatic Database Diagnostic Monitor (ADDM). Even though the number of high-load SQL statements that are identified by ADDM may represent a very small percentage of the total SQL workload, the task of tuning them is still highly complex and requires a high level of expertise.

Also, the SQL tuning activity is a continuous task because the SQL workload can change relatively often when new application modules are deployed.

SQL Tuning Advisor, introduced with Oracle Database 10*g*, is designed to replace the manual tuning of SQL statements. SQL statements that consume high resources (such as CPU, I/O, and temporary space) are good candidates for SQL Tuning Advisor. The advisor receives one or more SQL statements as input and then provides advice on how to optimize the execution plan, a rationale for the advice, estimated performance benefits, and the actual command to implement the advice. You accept the advice, thereby tuning the SQL statements. With the introduction of SQL Tuning Advisor, you can now let the Oracle optimizer tune the SQL code for you.

SQL Tuning Advisor: Overview



ORACLE

SQL Tuning Advisor is primarily the driver of the tuning process. It calls ATO to perform the following four specific types of analysis:

- **Statistics Analysis:** ATO checks each query object for missing or stale statistics and makes a recommendation to gather relevant statistics. It also collects auxiliary information to supply missing statistics or correct stale statistics in case recommendations are not implemented.
- **SQL Profiling:** ATO verifies its own estimates and collects auxiliary information to remove estimation errors. It also collects auxiliary information in the form of customized optimizer settings, such as *first rows* and *all rows*, based on the past execution history of the SQL statement. It builds a SQL Profile by using the auxiliary information and makes a recommendation to create it. When a SQL Profile is created, the profile enables the query optimizer, under normal mode, to generate a well-tuned plan.
- **Access Path Analysis:** ATO explores whether a new index can be used to significantly improve access to each table in the query, and when appropriate makes recommendations to create such indexes.
- **SQL Structure Analysis:** Here, ATO tries to identify SQL statements that lend themselves to bad plans, and makes relevant suggestions to restructure them. The suggested restructuring can be syntactic as well as semantic changes to the SQL code.

Stale or Missing Object Statistics

- Object statistics are key inputs to the optimizer.
- ATO verifies object statistics for each query object.
- ATO uses dynamic sampling and generates:
  - Auxiliary object statistics to compensate for missing or stale object statistics
  - Recommendations to gather object statistics where appropriate

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(
ownname=>'SH', tabname=>'CUSTOMERS',
estimate_percent=>DBMS_STATS.AUTO_SAMPLE_SIZE);
```


The query optimizer relies on object statistics to generate execution plans. If these statistics are stale or missing, the optimizer does not have the necessary information and can generate suboptimal execution plans.

ATO checks each query object for missing or stale statistics and produces two types of outputs:

- Auxiliary information in the form of statistics for objects with no statistics, and statistic adjustment factor for objects with stale statistics
- Recommendations to gather relevant statistics for objects with stale or no statistics

For optimal results, you gather statistics when recommended and then rerun ATO. However, you may be hesitant to accept this recommendation immediately because of the impact it could have on other queries in the system.

SQL Statement Profiling

- Statement statistics are key inputs to the optimizer.
- ATO verifies statement statistics such as:
  - Predicate selectivity
  - Optimizer settings (`FIRST_ROWS` versus `ALL_ROWS`)
- ATO uses:
  - Dynamic sampling
  - Partial execution of the statement
  - Past execution history statistics of the statement
- ATO builds a profile if statistics were generated.

```
exec :profile_name :=                -
dbms_sqltune.accept_sql_profile( -
task_name =>'my_sql_tuning_task');
```

The main verification step during SQL Profiling is the verification of the query optimizer's own estimates of cost, selectivity, and cardinality for the statement that is tuned.

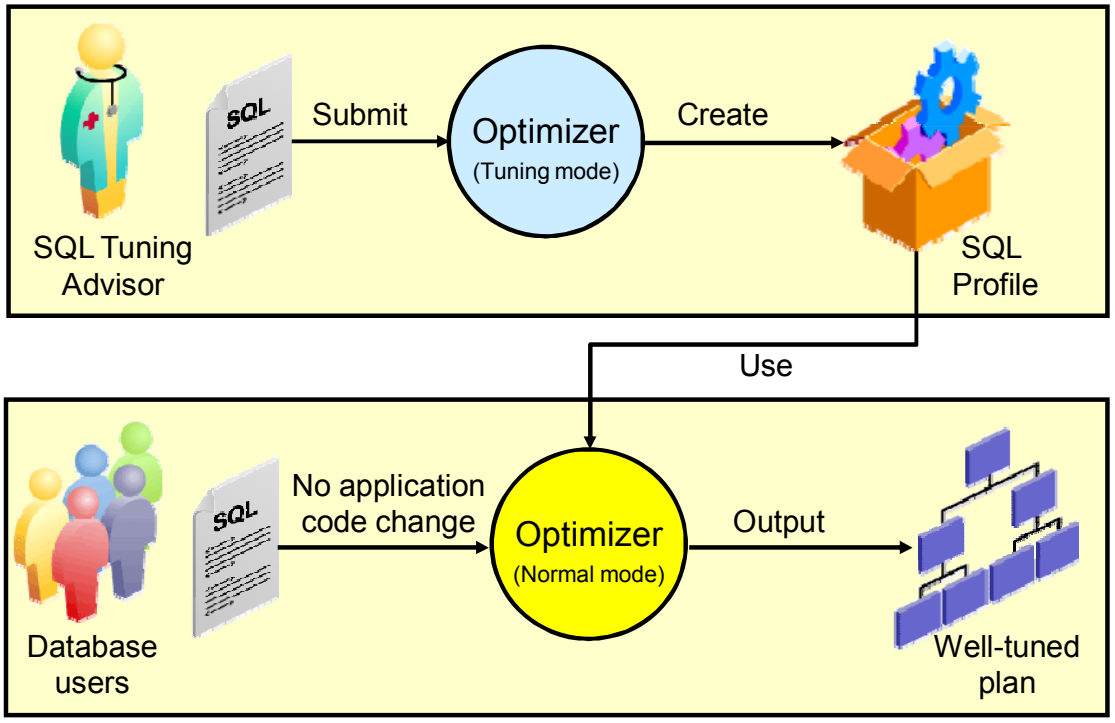
During SQL Profiling, ATO performs verification steps to validate its own estimates. The validation consists of taking a sample of data and applying appropriate predicates to the sample. The new estimate is compared to the regular estimate, and if the difference is large enough, a correction factor is applied. Another method of estimate validation involves the execution of a fragment of the SQL statement. The partial execution method is more efficient than the sampling method when the respective predicates provide efficient access paths. ATO picks the appropriate estimate validation method.

ATO also uses the past execution history of the SQL statement to determine correct settings. For example, if the execution history indicates that a SQL statement is only partially executed the majority of times, ATO uses the `FIRST_ROWS` optimization as opposed to `ALL_ROWS`.

ATO builds a SQL Profile if it has generated auxiliary information either during Statistics Analysis or during SQL Profiling. When a SQL Profile is built, it generates a user recommendation to create a SQL Profile.

In this mode, ATO can recommend the acceptance of the generated SQL Profile to activate it.

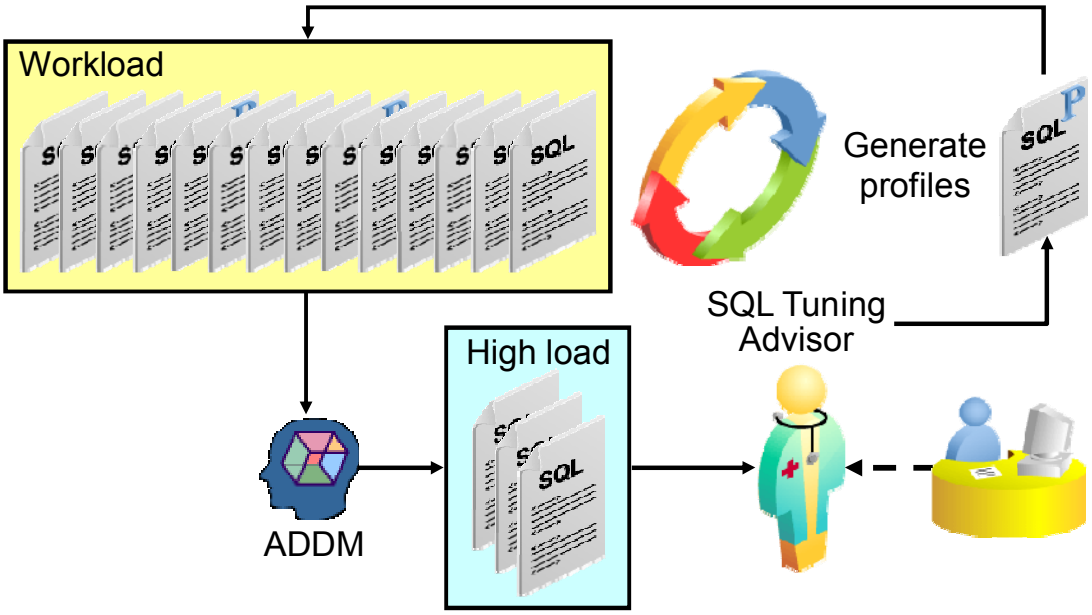
Plan Tuning Flow and SQL Profile Creation



A SQL Profile is a collection of auxiliary information that is built during automatic tuning of a SQL statement. Thus, a SQL Profile is to a SQL statement what statistics are to a table or index. After it is created, a SQL Profile is used in conjunction with the existing statistics by the query optimizer, in normal mode, to produce a well-tuned plan for the corresponding SQL statement. A SQL Profile is stored persistently in the data dictionary. However, the SQL profile information is not exposed through regular dictionary views. After creation of a SQL Profile, every time the corresponding SQL statement is compiled in normal mode, the query optimizer uses the SQL Profile to produce a well-tuned plan.

The slide shows the process flow of the creation and use of a SQL Profile. The process consists of two phases: system SQL tuning phase and regular optimization phase. During the system SQL tuning phase, you select a SQL statement for system tuning and run SQL Tuning Advisor by using either Database Control or the command-line interface. SQL Tuning Advisor invokes ATO to generate tuning recommendations, possibly with a SQL Profile. If a SQL Profile is built, you can accept it. When it is accepted, the SQL Profile is stored in the data dictionary. In the next phase, when an end user issues the same SQL statement, the query optimizer (under normal mode) uses the SQL Profile to build a well-tuned plan. The use of the SQL Profile remains completely transparent to the end user and does not require changes to the application source code.

SQL Tuning Loop

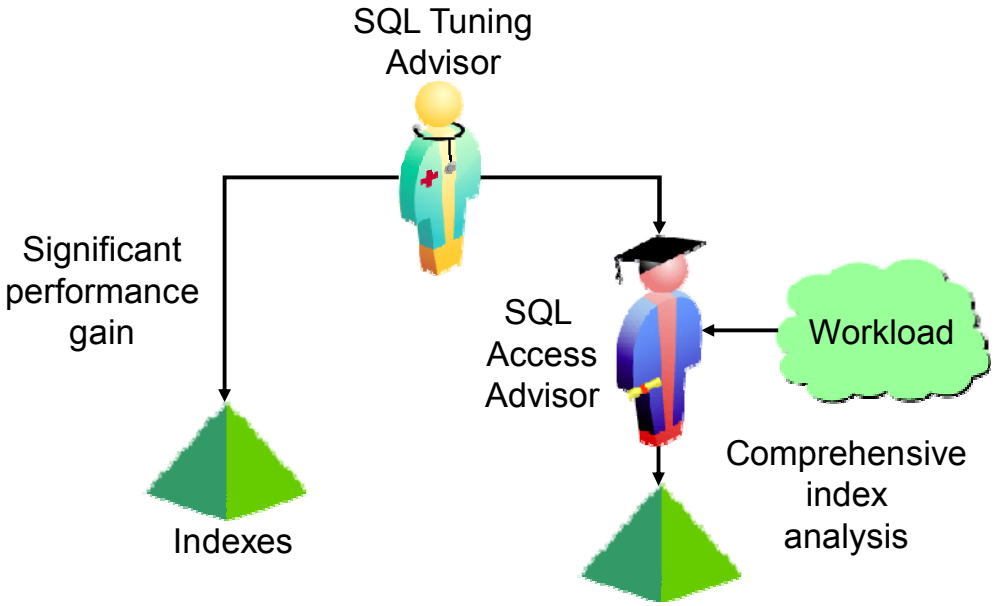


The auxiliary information contained in a SQL Profile is stored in such a way that it stays relevant after database changes, such as addition or removal of indexes, growth in the size of tables, and periodic collection of database statistics. Therefore, when a profile is created, the corresponding plan is not frozen (as when outlines are used).

However, a SQL Profile may not adapt to massive changes in the database or changes that have accumulated over a long period of time. In such cases, a new SQL Profile needs to be built to replace the old one.

For example, when a SQL Profile becomes outdated, the performance of the corresponding SQL statement may become noticeably worse. In such a case, the corresponding SQL statement may start showing up as high-load or top SQL, thus becoming again a target for system SQL Tuning. In such a situation, ADDM again captures the statement as high-load SQL. If that happens, you can decide to re-create a new profile for that statement.

Access Path Analysis




```
CREATE INDEX JFV.IDX$_00002 on JFV.TEST("C");
```

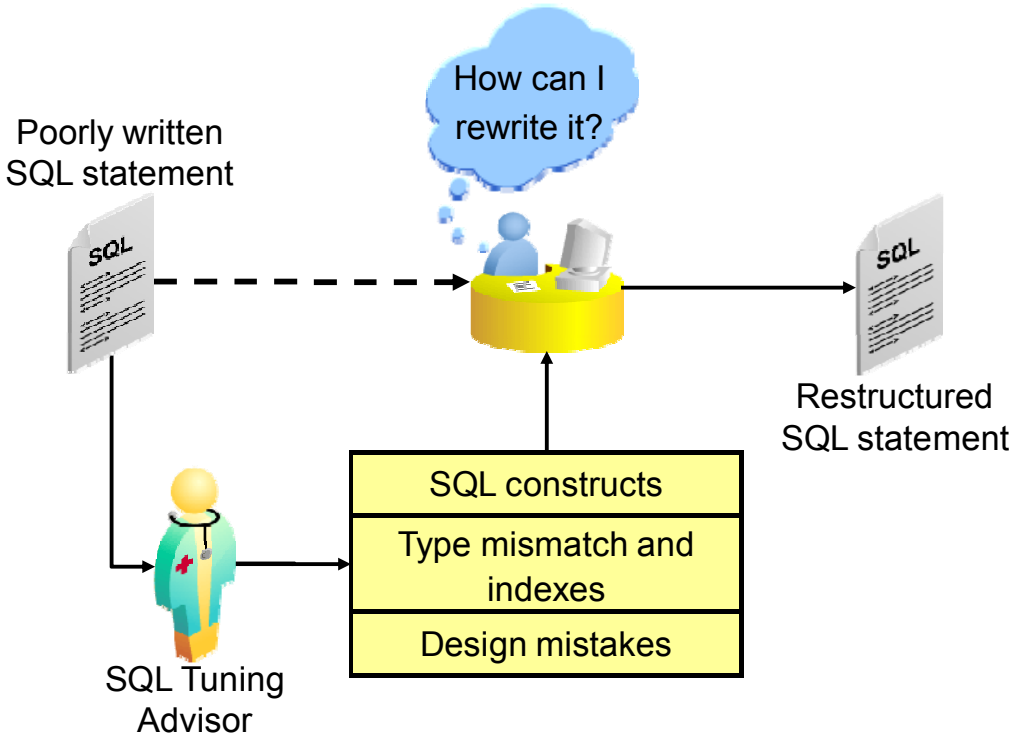

ATO also provides advice on indexes. Effective indexing is a well-known tuning technique that can significantly improve the performance of SQL statements by reducing the need for full table scans. Any index recommendations generated by ATO are specific to the SQL statement being tuned. Therefore, it provides a quick solution to the performance problem associated with a single SQL statement.

Because ATO does not perform an analysis of how its index recommendations affect the entire SQL workload, it recommends running the Access Advisor on the SQL statement along with a representative SQL workload. The Access Advisor collects advice given on each statement of a SQL workload and consolidates it into global advice for the entire SQL workload.

The Access Path Analysis can make the following recommendations:

- Create new indexes if they provide significantly superior performance.
- Run SQL Access Advisor to perform a comprehensive index analysis based on application workload.

SQL Structure Analysis



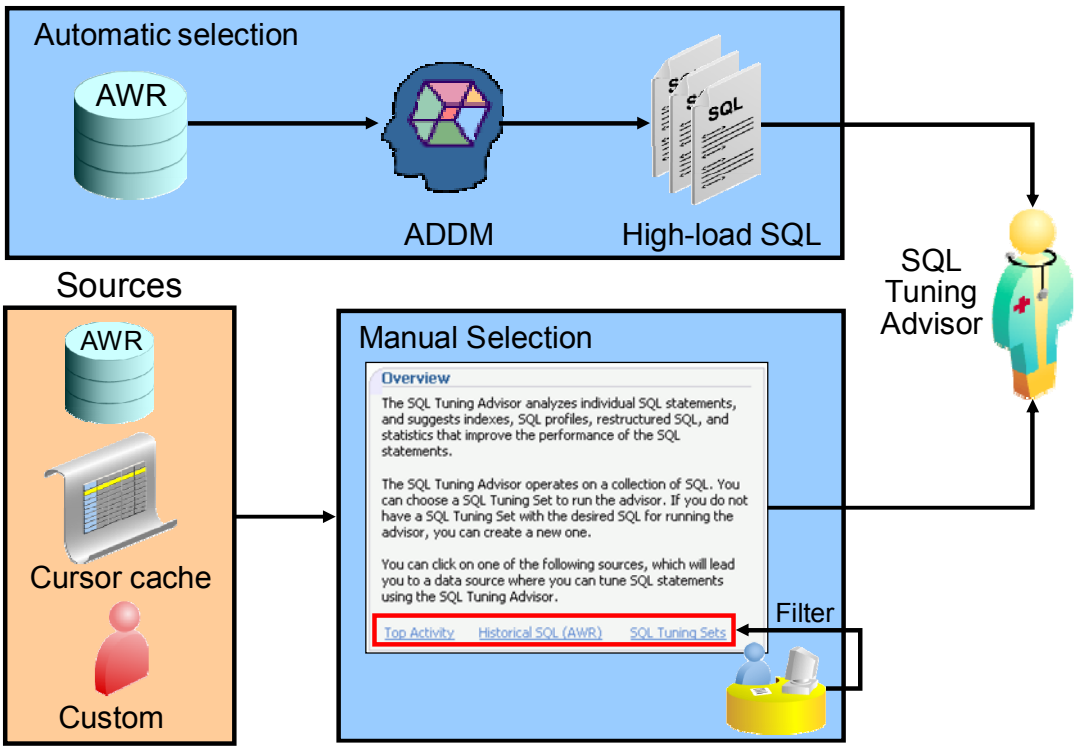
The goal of the SQL Structure Analysis is to help you identify poorly written SQL statements as well as to advise you on how to restructure them.

Certain syntax variations are known to have a negative impact on performance. In this mode, ATO evaluates statements against a set of rules, identifying less-efficient coding techniques, and providing recommendations for an alternative statement where possible. The recommendation may be very similar, but not precisely equivalent to the original query. For example, the `NOT EXISTS` and `NOT IN` constructors are similar, but not exactly the same. Therefore, you have to decide whether the recommendation is valid. For this reason, ATO does not automatically rewrite the query, but gives advice instead.

The following categories of problems are detected by the SQL Structure Analysis:

- Use of SQL constructors such as `NOT IN` instead of `NOT EXISTS`, or `UNION` instead of `UNION ALL`
- Use of predicates involving indexed columns with data-type mismatch that prevents use of the index
- Design mistakes (such as Cartesian products)

SQL Tuning Advisor: Usage Model



SQL Tuning Advisor takes one or more SQL statements as input. The input can come from different sources:

- High-load SQL statements identified by ADDM
- SQL statements that are currently in cursor cache
- SQL statements from Automatic Workload Repository (AWR): A user can select any set of SQL statements captured by AWR. This can be done using snapshots or baselines.
- Custom workload: A user can create a custom workload consisting of statements of interest to the user. These may be statements that are not in cursor cache and are not high-load SQL statements to be captured by ADDM or AWR. For such statements, a user can create a custom workload and tune it by using the advisor.

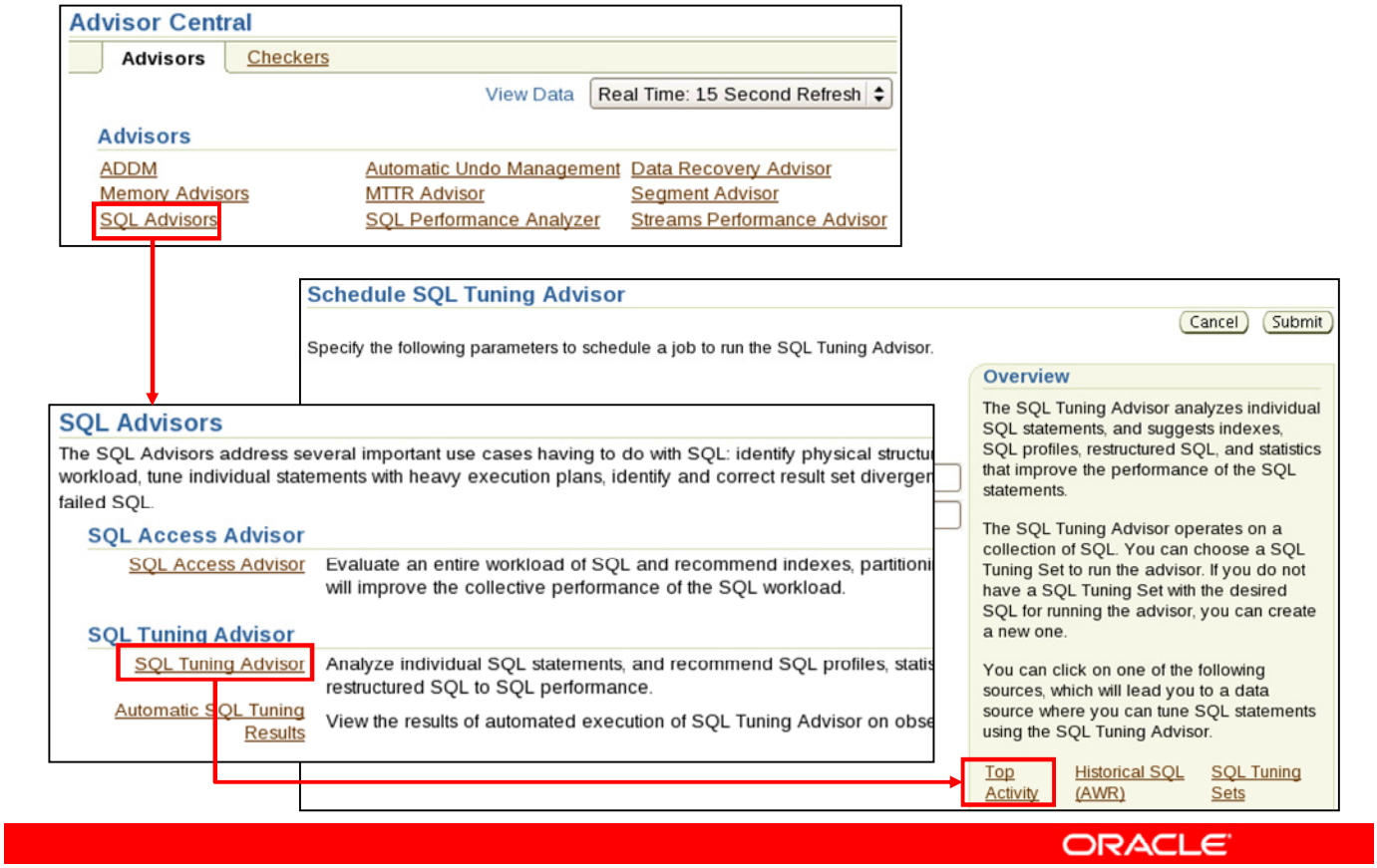
SQL statements from cursor cache, AWR, and custom workload can be filtered and ranked before they are input to SQL Tuning Advisor.

For a multistatement input, an object called SQL Tuning Set (STS) is provided. An STS stores multiple SQL statements along with their execution information:

- **Execution context:** Parsing schema name and bind values
- **Execution statistics:** Average elapsed time and execution count

Note: Another STS can be a possible source for STS creation.

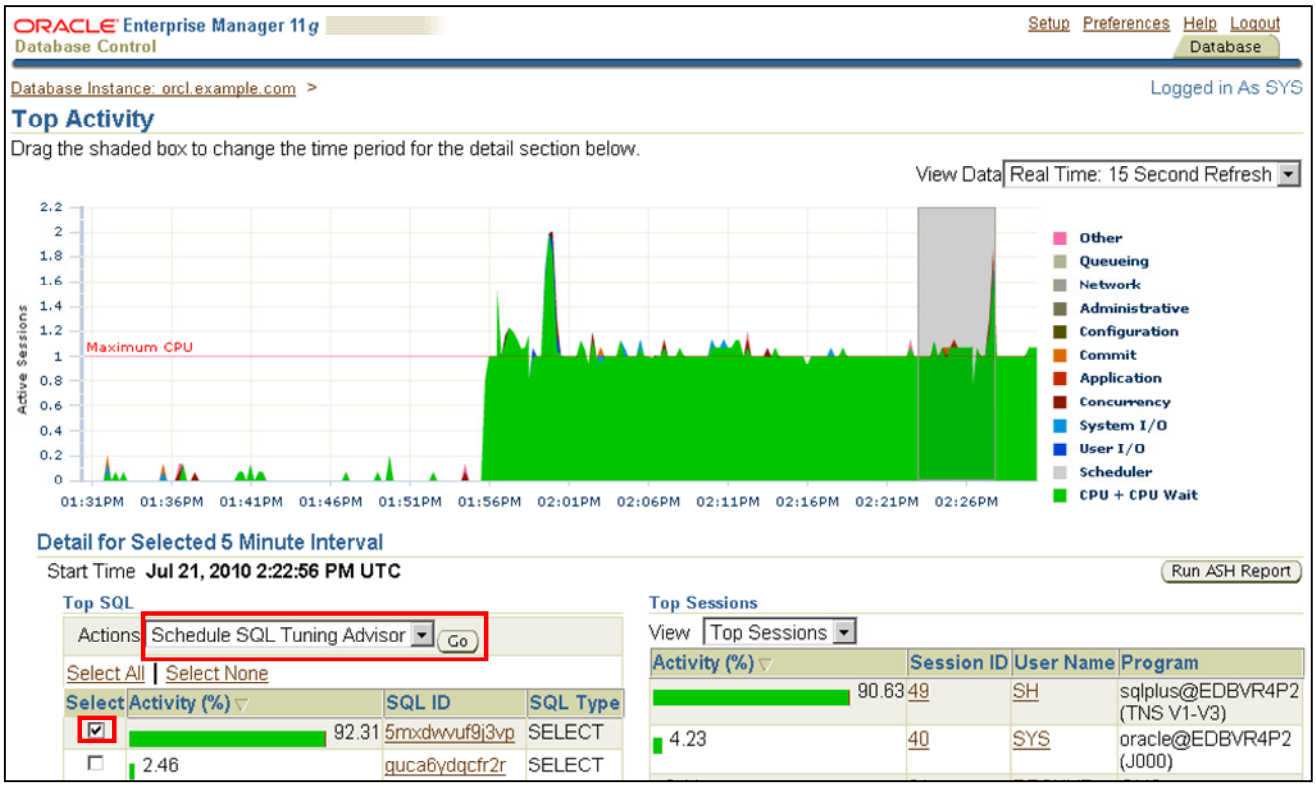
Database Control and SQL Tuning Advisor



The easiest way to access the SQL Tuning Advisor from Enterprise Manager is on the Advisor Central page. On the Home page, click the Advisor Central link located in the Related Links section to open the Advisor Central page.

On the Advisor Central page, click the SQL Advisors link. On the SQL Advisors page, click the SQL Tuning Advisor link. This takes you to the Schedule SQL Tuning Advisor page. On this page, you find links to various other pages. You click the Top Activity link to open the Top Activity page.

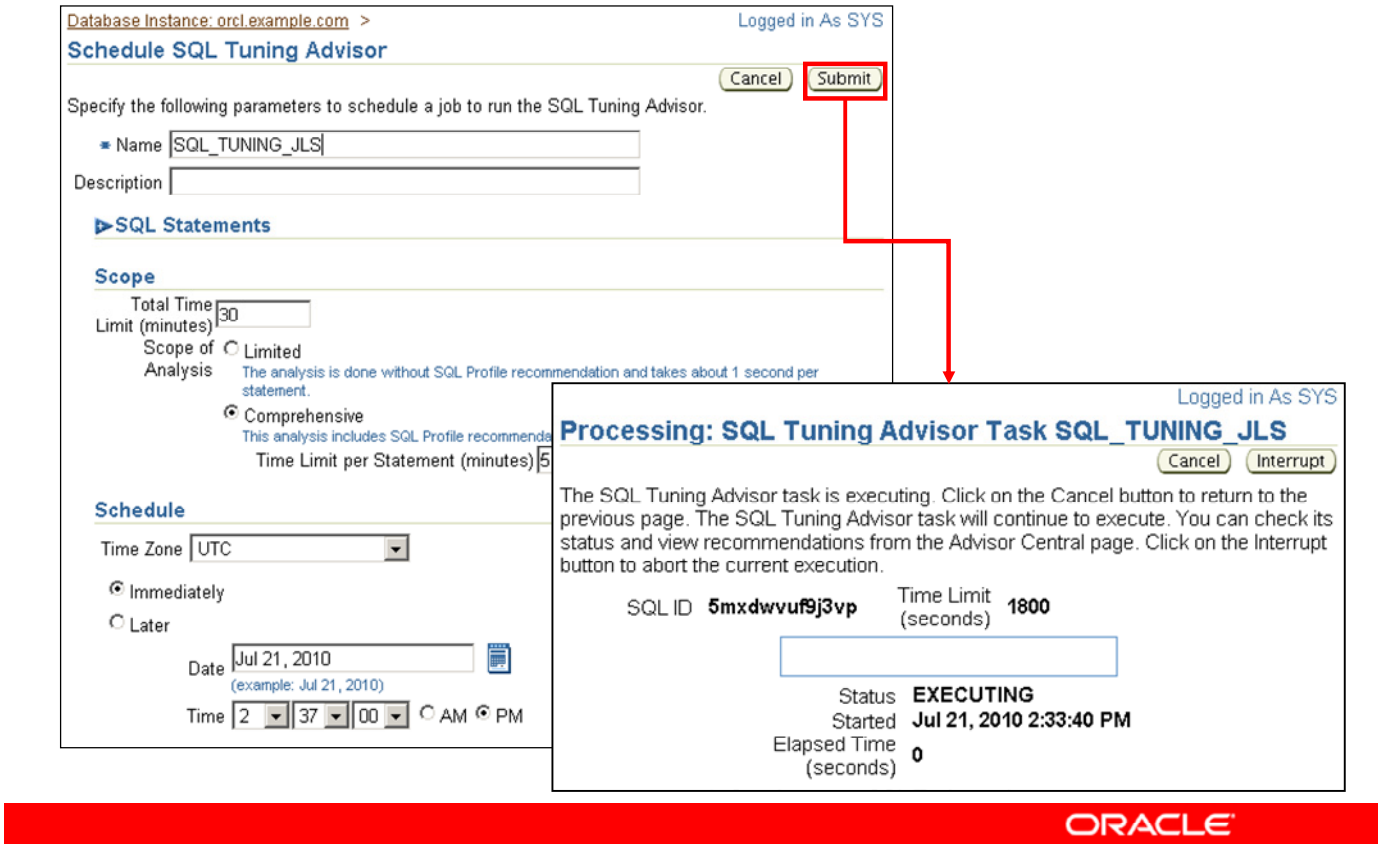
Running SQL Tuning Advisor: Example



You can use Database Control to identify the high-load or top SQL statements. There are several locations in Database Control from where SQL Tuning Advisor can be launched with the identified SQL statements or with an STS:

- **Tuning ADDM-identified SQL statements:** The ADDM Finding Details page shows high-load SQL statements identified by ADDM. Each of these high-load SQL statements is known to consume a significant proportion of one or more system resources. You can use this page to launch SQL Tuning Advisor on a selected high-load SQL statement.
- **Tuning top SQL statements:** Another SQL source is the list of top SQL statements, as shown in the slide. You can identify the list of top SQL statements by looking at their cumulative execution statistics based on a selected time window. The user can select one or more top SQL statements identified by their SQL IDs, and then click Schedule SQL Tuning Advisor.
- **Tuning a SQL Tuning Set:** It is also possible to look at various STSs created by different users. An STS could have been created from a list of top SQL statements, by selecting SQL statements from a range of snapshots created by AWR or by selecting customized SQL statements.

Schedule SQL Tuning Advisor

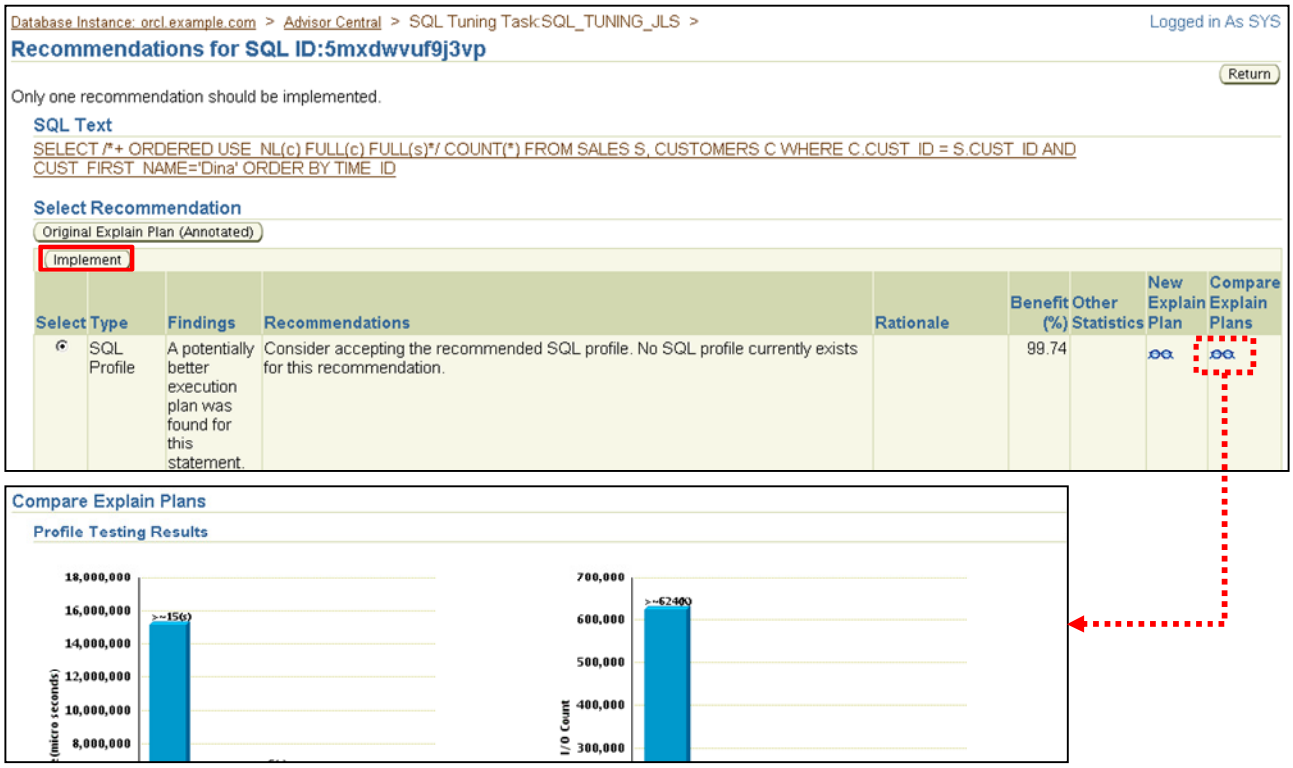


When SQL Tuning Advisor is launched, Enterprise Manager automatically creates a tuning task, provided that the user has the appropriate `ADVISOR` privilege to do so. Enterprise Manager shows the tuning task with automatic defaults on the Schedule SQL Tuning Advisor page. On this page, the user can change the automatic defaults pertaining to a tuning task.

One of the important options is to select the scope of the tuning task. If you select the Limited option, SQL Tuning Advisor produces recommendations based on statistics check, access path analysis, and SQL structure analysis. No SQL Profile recommendation is generated with Limited scope. If you select the Comprehensive option, SQL Tuning Advisor performs all of the Limited scope actions, and invokes the optimizer under SQL Profiling mode to build a SQL Profile, if applicable. With the Comprehensive option, you can also specify a time limit for the tuning task, which by default is 30 minutes. Another useful option is to run the tuning task immediately or schedule it to be run at a later time.

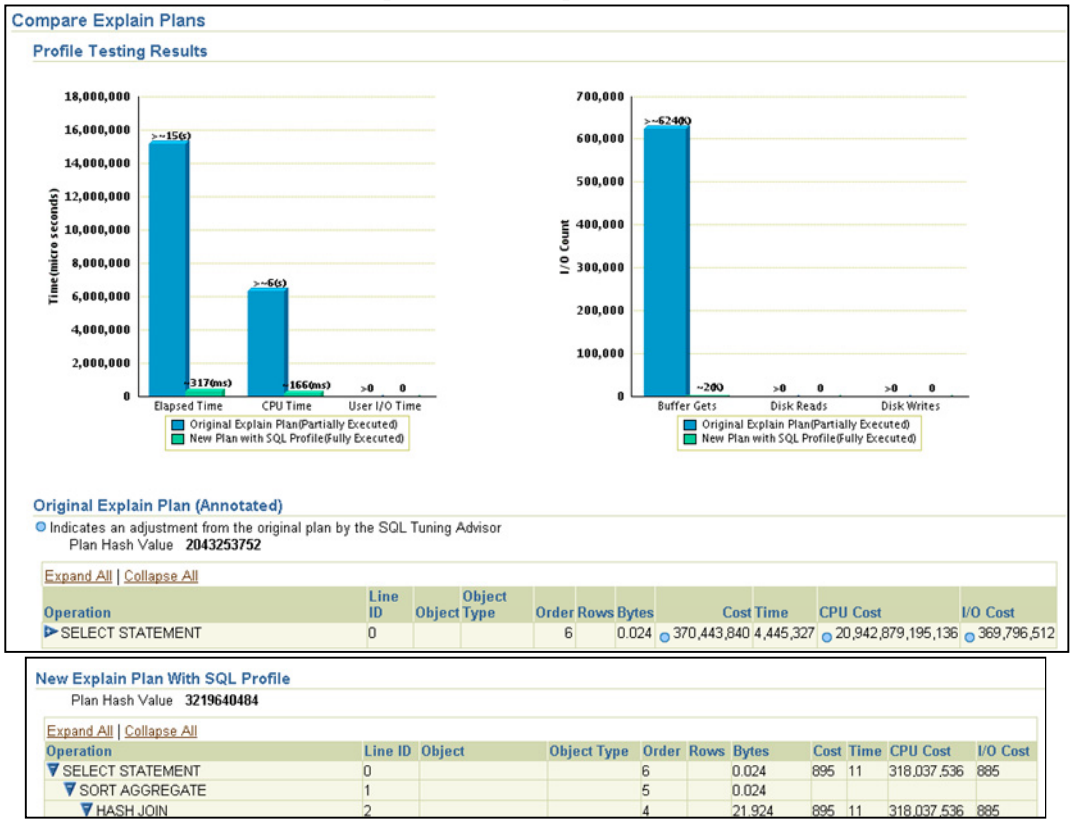
When the task is submitted, the Processing page appears. When the task is complete, the Recommendations page appears.

Implementing Recommendations



On the Recommendations page, you can view the various recommendations. For each recommendation, as shown, a SQL Profile has been created; you can implement it if you want, after you view the new plan. Click the eyeglass icon to view the Compare Explain Plan page.

Compare Explain Plan



The Compare Explain Plan page gives you the opportunity to view the projected benefits of implementing the recommendation, in this case a SQL profile. You can see the benefits graphically and in a table. Notice the Cost values for the SQL statement in the original and new explain plans. If the difference is not enough or the explain is not acceptable, you can ignore or delete the recommendation.

Quiz

SQL Tuning Advisor recommends:

a. SQL Profiles
b. Additional Indexes
c. Deleting Indexes
d. Rewriting SQL Statements
e. All of the above

Answer: a, b, d

SQL Tuning Advisor in comprehensive mode recommends all except deleting indexes. SQL Tuning Advisor focuses on one SQL statement at a time. An entire workload must be considered to determine if deleting an index will help performance.

Quiz

The SQL Profile forces the best execution plan even when the data in the table changes.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

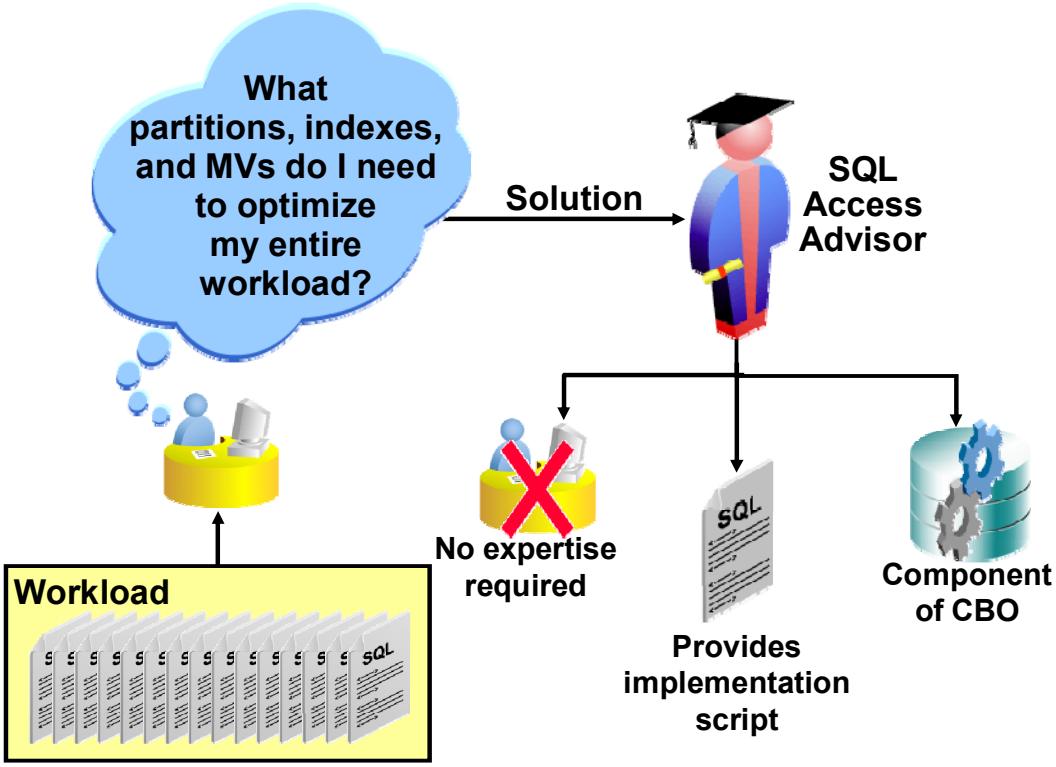
- Describe statement profiling
- Use SQL Tuning Advisor

Using SQL Access Advisor

Objectives

After completing this lesson, you should be able to use SQL Access Advisor.

SQL Access Advisor: Overview



Defining appropriate access structures to optimize SQL queries has always been a concern for the developer. As a result, there have been many papers and scripts written as well as high-end tools developed to address the matter. In addition, with the development of partitioning and materialized view (MV) technology, deciding on access structures has become even more complex.

As part of the manageability improvements in Oracle Database 10*g* and 11*g*, SQL Access Advisor has been introduced to address this critical need.

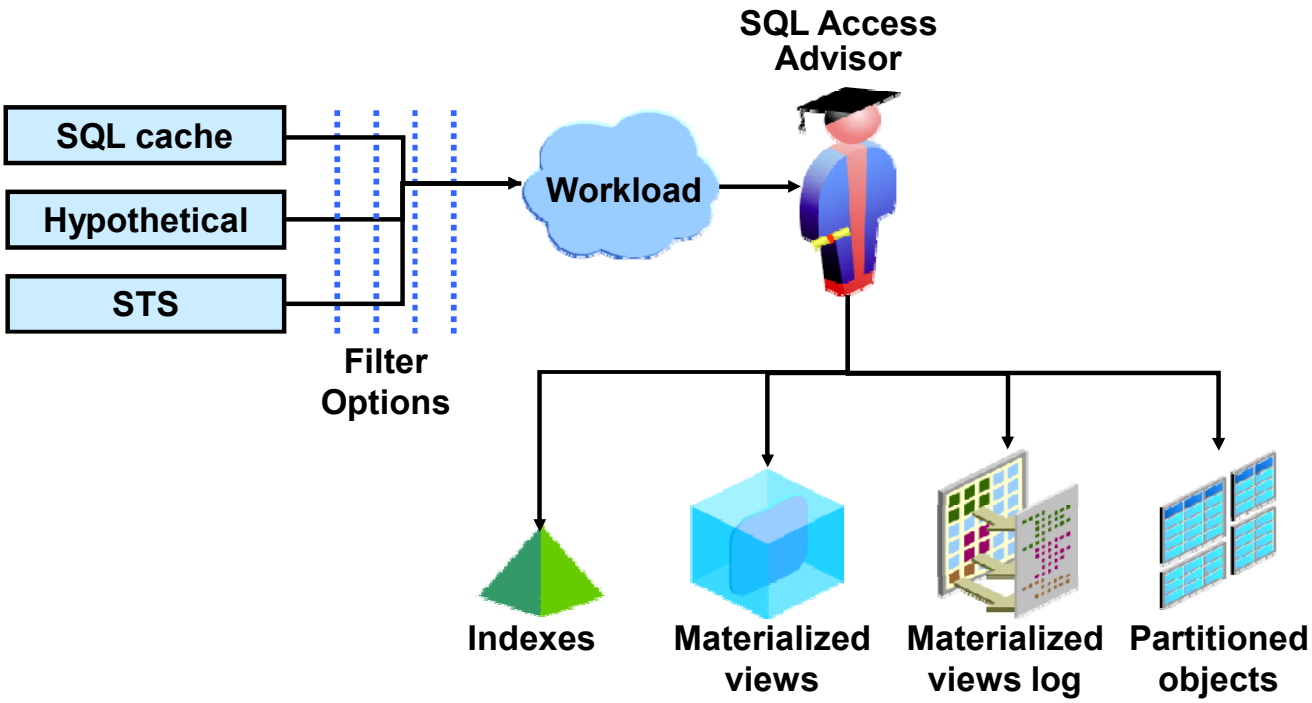
SQL Access Advisor identifies and helps resolve performance problems relating to the execution of SQL statements by recommending which indexes, materialized views, materialized view logs, or partitions to create, drop, or retain. It can be run from Database Control or from the command line by using PL/SQL procedures.

SQL Access Advisor takes an actual workload as input, or it can derive a hypothetical workload from the schema. It then recommends the access structures for a faster execution path. It provides the following advantages:

- Does not require you to have expert knowledge
- Bases decision making on rules that actually reside in the cost-based optimizer (CBO)
- Is synchronized with the optimizer and Oracle Database enhancements

- Is a single advisor covering all aspects of SQL access methods
- Provides simple, user-friendly graphical user interface (GUI) wizards
- Generates scripts for implementation of recommendations

SQL Access Advisor: Usage Model



SQL Access Advisor takes as input a workload that can be derived from multiple sources:

- SQL cache, to take the current content of `V$SQL`
- Hypothetical, to generate a likely workload from your dimensional model (This option is useful when your system is being initially designed.)
- SQL tuning sets (STS), from the workload repository

SQL Access Advisor also provides powerful workload filters that you can use to target the tuning. For example, a user can specify that the advisor should look at only the 30 most resource-intensive statements in the workload, based on optimizer cost. For the given workload, the advisor then does the following:

- Simultaneously considers index solutions, MV solutions, partition solutions, or combinations of all three
- Considers storage for creation and maintenance costs
- Does not generate drop recommendations for partial workloads
- Optimizes MVs for maximum query rewrite usage and fast refresh
- Recommends materialized view logs for fast refresh
- Recommends partitioning for tables, indexes, and materialized views
- Combines similar indexes into a single index
- Generates recommendations that support multiple workload queries

Possible Recommendations

Recommendation	Comprehensive	Limited
Add new (partitioned) index on table or materialized view.	YES	YES
Drop an unused index.	YES	NO
Modify an existing index by changing the index type.	YES	NO
Modify an existing index by adding columns at the end.	YES	YES
Add a new (partitioned) materialized view.	YES	YES
Drop an unused materialized view (log).	YES	NO
Add a new materialized view log.	YES	YES
Modify an existing materialized view log to add new columns or clauses.	YES	YES
Partition an existing unpartitioned table or index.	YES	YES

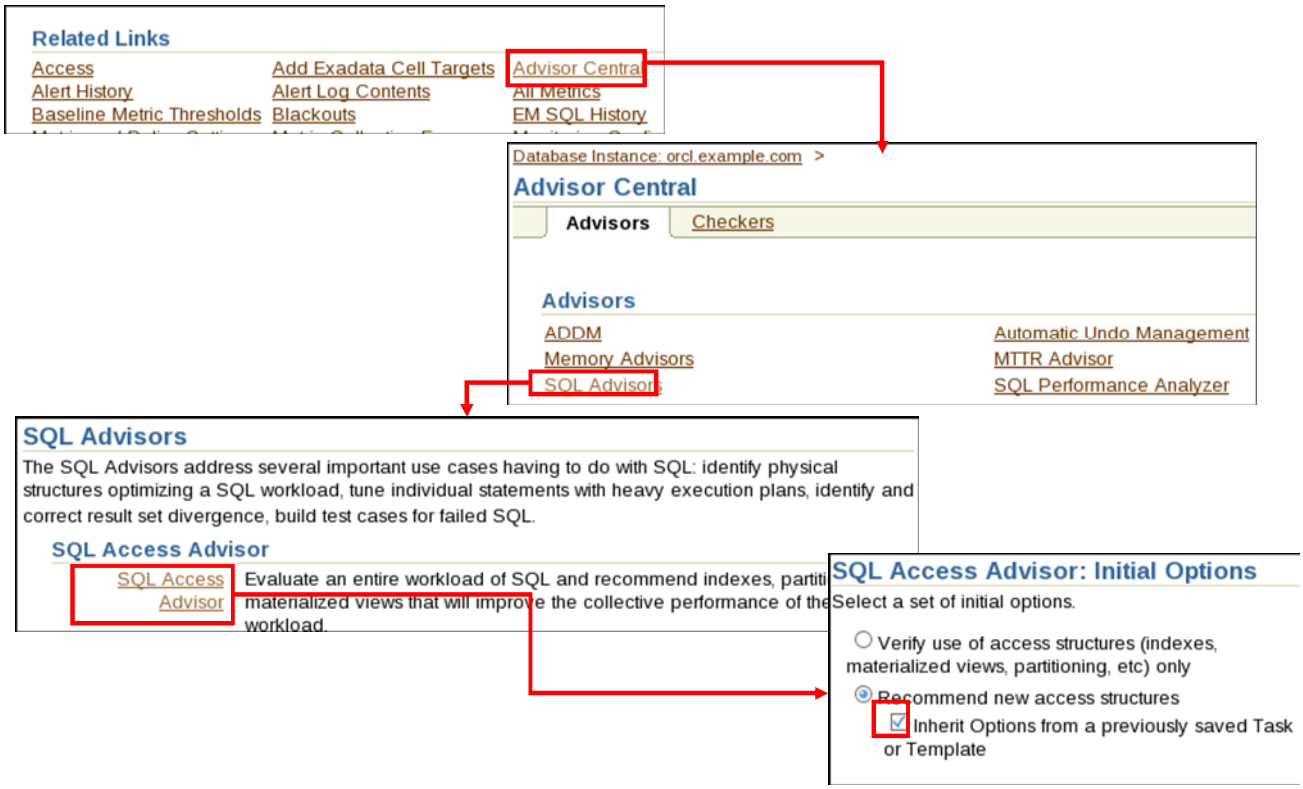
SQL Access Advisor carefully considers the overall impact of recommendations and makes recommendations by using only the known workload and supplied information. Two workload analysis approaches are available:

- **Comprehensive:** SQL Access Advisor addresses all aspects of tuning partitions, MVs, indexes, and MV logs. It assumes that the workload contains a complete and representative set of application SQL statements.
- **Limited:** Unlike the comprehensive workload approach, a limited workload approach assumes that the workload contains only problematic SQL statements. Thus, advice is sought for improving the performance of a portion of an application environment.

When comprehensive workload analysis is chosen, SQL Access Advisor forms a better set of global tuning adjustments, but the effect may be a longer analysis time. As shown in the slide, the chosen workload approach determines the type of recommendations made by the advisor.

Note: Partition recommendations can work only on those tables that have at least 10,000 rows and on workloads that have some predicates and joins on columns of the `NUMBER` or `DATE` type. Partitioning recommendations can be generated only on these types of columns. In addition, partitioning recommendations can be generated only for single-column interval and hash partitions. Interval partitioning recommendations can be output as range syntax, but interval is the default. Hash partitioning is done to leverage only partition-wise joins.

SQL Access Advisor Session: Initial Options



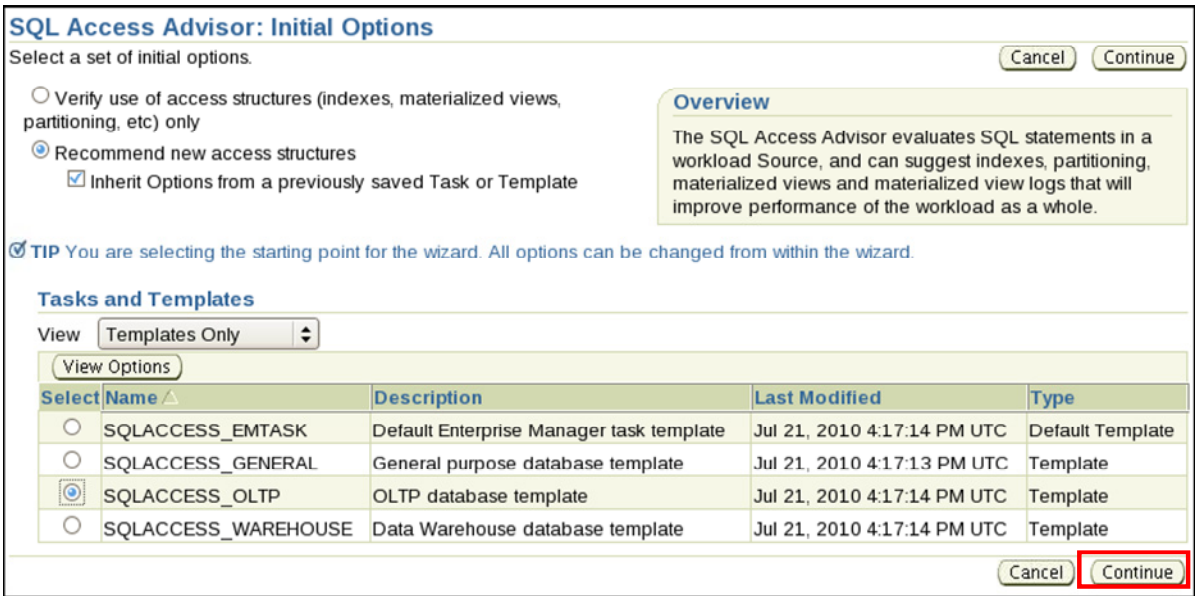
The next few slides describe a typical SQL Access Advisor session. You can access SQL Access Advisor by clicking the Advisor Central link on the Database Home page or through individual alerts or performance pages that may include a link to facilitate solving a performance problem. SQL Access Advisor consists of several steps where you supply the SQL statements to tune and the types of access methods that you want to use.

On the SQL Access Advisor: Initial Options page, you can select a template or task from which to populate default options before starting the wizard.

Note: SQL Access Advisor may be interrupted while generating recommendations, thereby allowing the results to be reviewed.

For general information about using SQL Access Advisor, see the “Overview of the SQL Access Advisor” section in the lesson titled “SQL Access Advisor” of the *Oracle Data Warehousing Guide*.

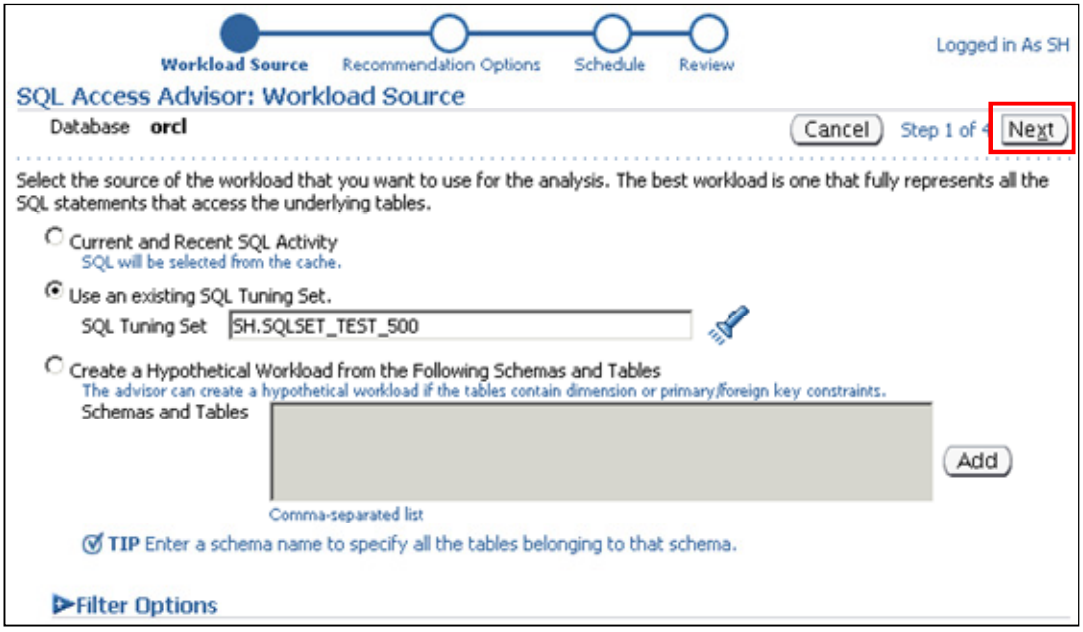
SQL Access Advisor Session: Initial Options



If you select the “Inherit Options from a previously saved Task or Template” option on the SQL Access Advisor: Initial Options page, you can select an existing task or an existing template to inherit the SQL Access Advisor options. By default, the SQLACCESS_EMTASK template is used.

You can view the various options defined by a task or a template by selecting the corresponding object and clicking View Options.

SQL Access Advisor: Workload Source



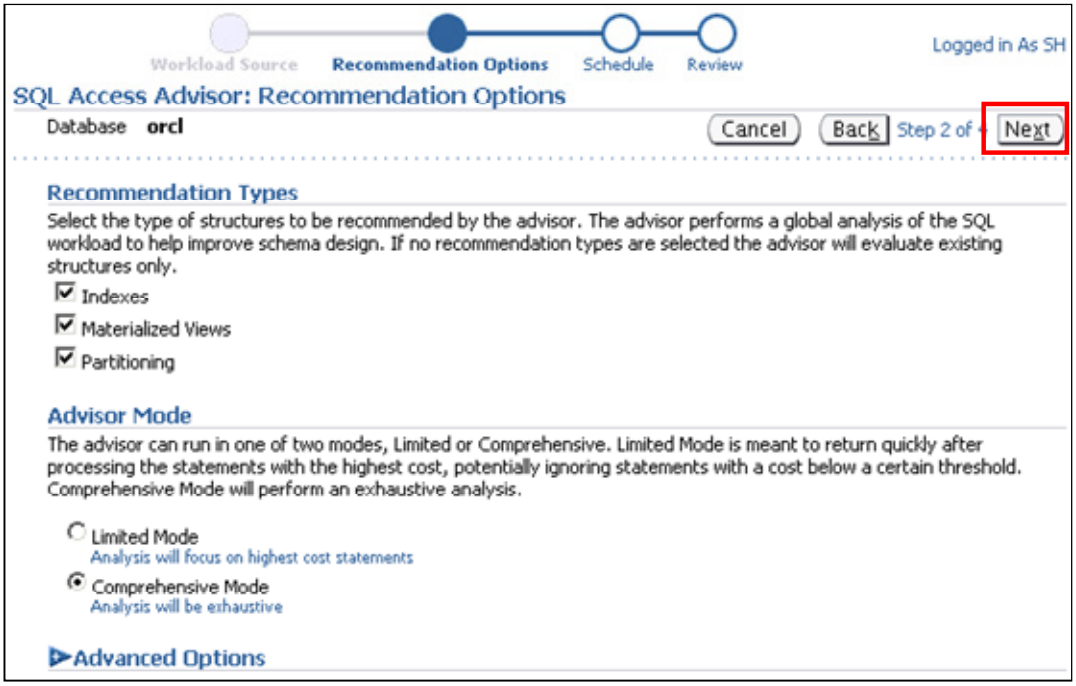
You can select your workload source from three different sources:

- **Current and Recent SQL Activity:** This source corresponds to SQL statements that are still cached in your System Global Area (SGA).
- **Use an existing SQL Tuning Set:** You also have the possibility of creating and using a SQL Tuning Set that holds your statements.
- **Hypothetical Workload:** This option provides a schema that allows the advisor to search for dimension tables and produce a workload. This option is very useful to initially design your schema.

Using the Filter Options section, you can further filter your workload source. Filter options are:

- Resource Consumption (number of statements ordered by Optimizer Cost, Buffer Gets, CPU Time, Disk Reads, Elapsed Time, Executions)
- Users
- Tables
- SQL Text
- Module IDs
- Actions

SQL Access Advisor: Recommendation Options

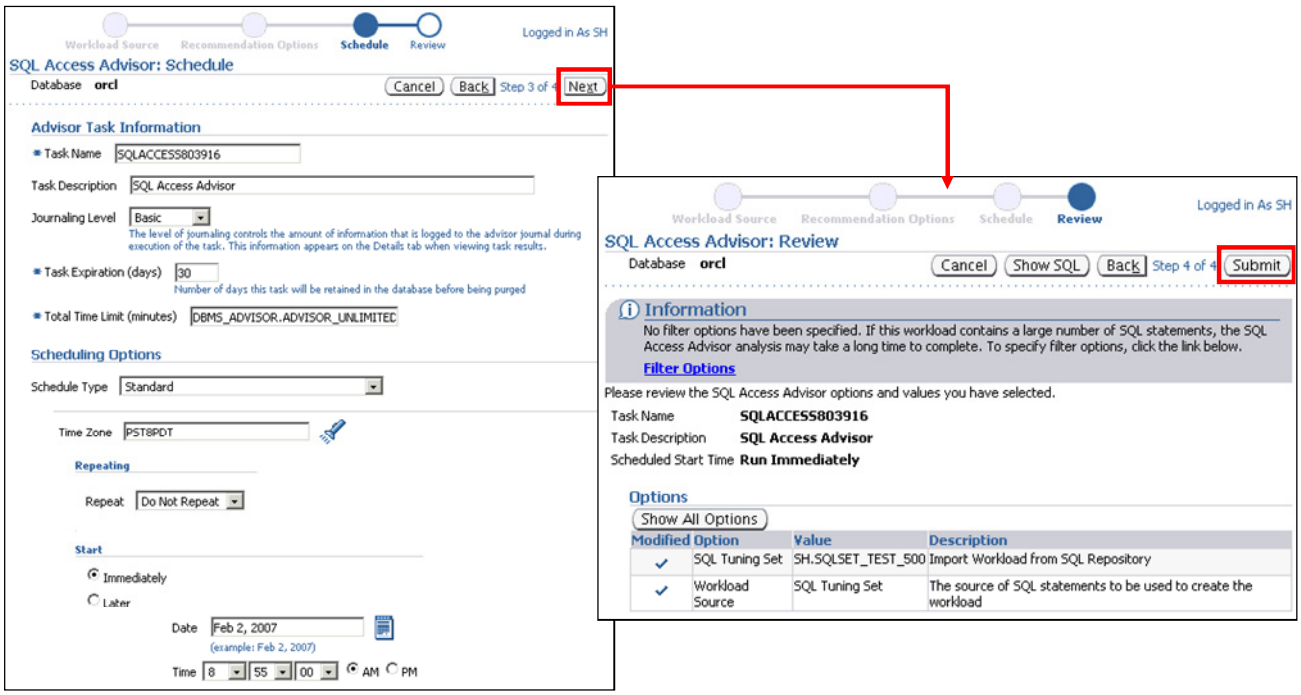


On the Recommendation Options page, you can select whether to limit SQL Access Advisor to recommendations based on a single access method. You can select the type of structures to be recommended by the advisor. If none of the three possible structures are chosen, the advisor evaluates existing structures instead of trying to recommend new ones.

You can use the Advisor Mode section to run the advisor in one of the two modes. These modes affect the quality of recommendations as well as the length of time required for processing. In Comprehensive Mode, the advisor searches a large pool of candidates, resulting in recommendations of the highest quality. In Limited Mode, the advisor performs quickly, limiting the candidate recommendations by working only on the highest-cost statements.

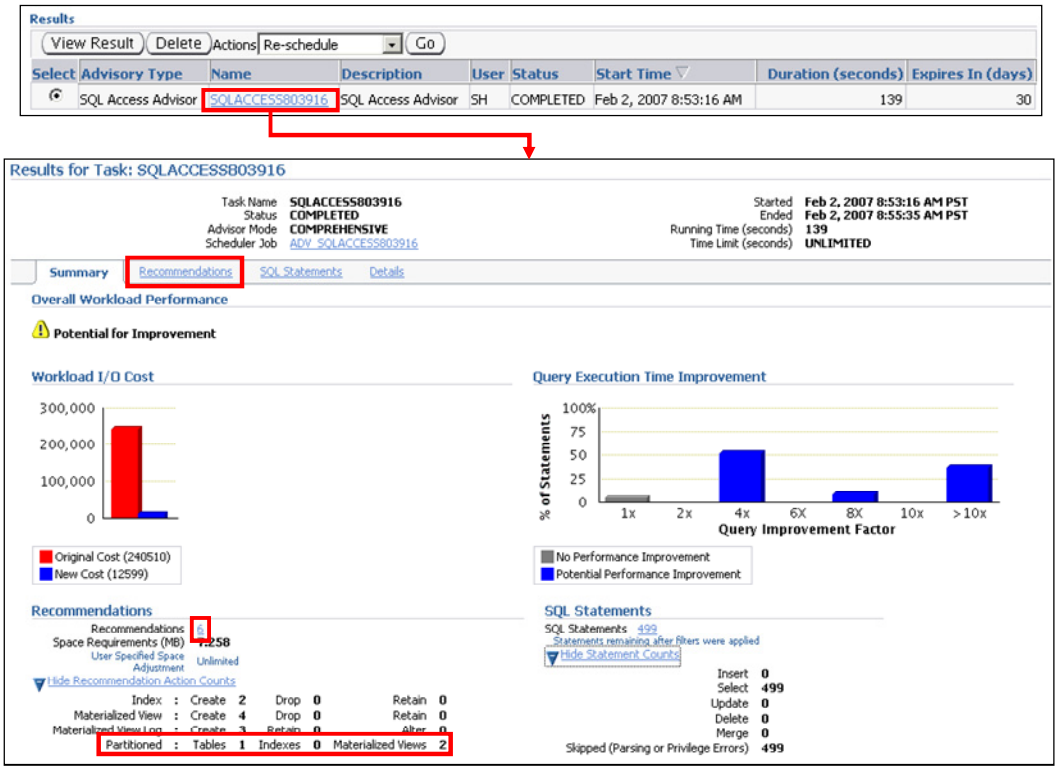
Note: You can click Advanced Options to show or hide options that allow you to set space restrictions, tuning options, and default storage locations.

SQL Access Advisor: Schedule and Review



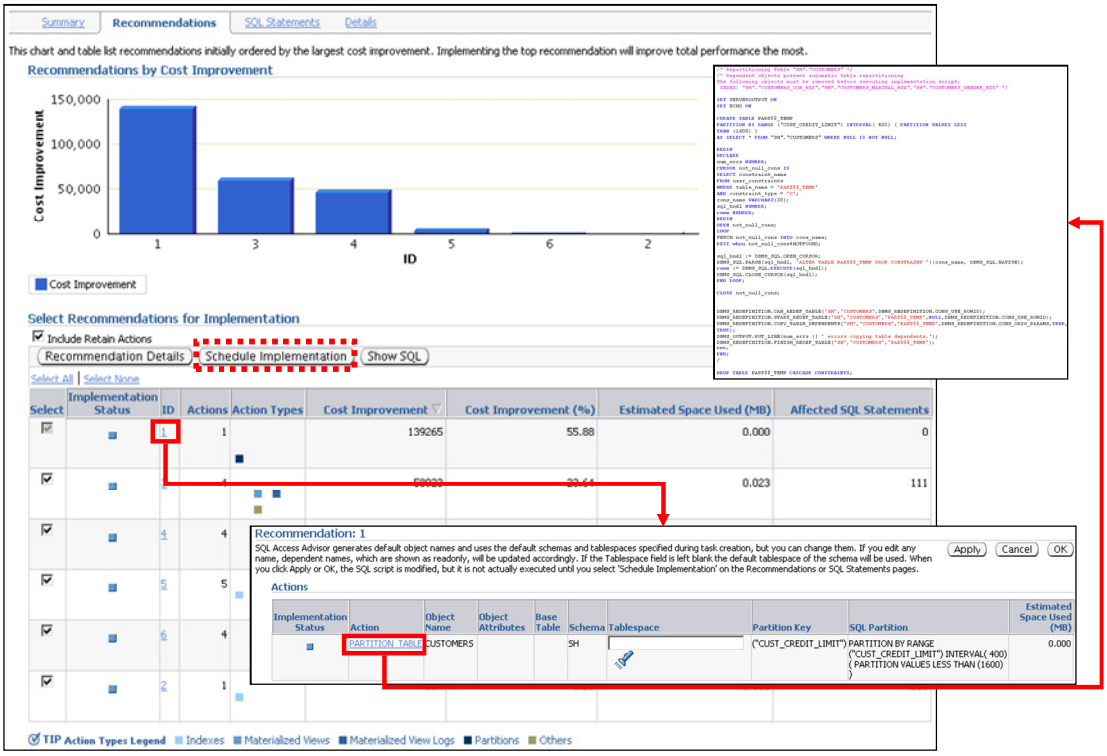
You can then schedule and submit your new analysis by specifying various parameters to the scheduler. The possible options are shown in the slide.

SQL Access Advisor: Results



On the Advisor Central page, you can retrieve the task details for your analysis. By selecting the task name in the Results section, you can access the Results for Task Summary page, where you can see an overview of the SQL Access Advisor findings. The page shows you charts and statistics that provide overall workload performance and potential for improving query execution time for the recommendations. You can use the page to show statement counts and recommendation action counts.

SQL Access Advisor: Results and Implementation



To see other aspects of the results for the SQL Access Advisor task, click one of the three other tabs on the page: Recommendations, SQL Statements, or Details.

On the Recommendations page, you can drill down to each of the recommendations. For each of them, you see important information in the Select Recommendations for Implementation table. You can then select one or more recommendations and schedule their implementation.

If you click the ID for a particular recommendation, you are taken to the Recommendations page that displays all actions for the specified recommendation and, optionally, to modify the tablespace name of the statement. When you complete any changes, click OK to apply the changes. On the Recommendations page, you can view the full text of an action by clicking the link in the Action field for the specified action. You can view the SQL statements for all actions in the recommendation by clicking Show SQL.

The SQL Statements page (not shown here) gives you a chart and a corresponding table that lists SQL statements initially ordered by the largest cost improvement. The top SQL statement is improved the most by implementing its associated recommendation.

The Details page shows you the workload and task options that were used when the task was created. This page also gives you all journal entries that were logged during the task execution.

You can also schedule implementation of the recommendations by clicking the Schedule Implementation button.

Quiz

Identify two available workload analysis methods.

a. Comprehensive
b. Complete
c. Partial
d. Limited

Answer: a, d

Quiz

SQL Access Advisor identifies but cannot help resolve performance problems relating to the execution of SQL statements.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to use SQL Access Advisor.

Exploring the Oracle Database Architecture

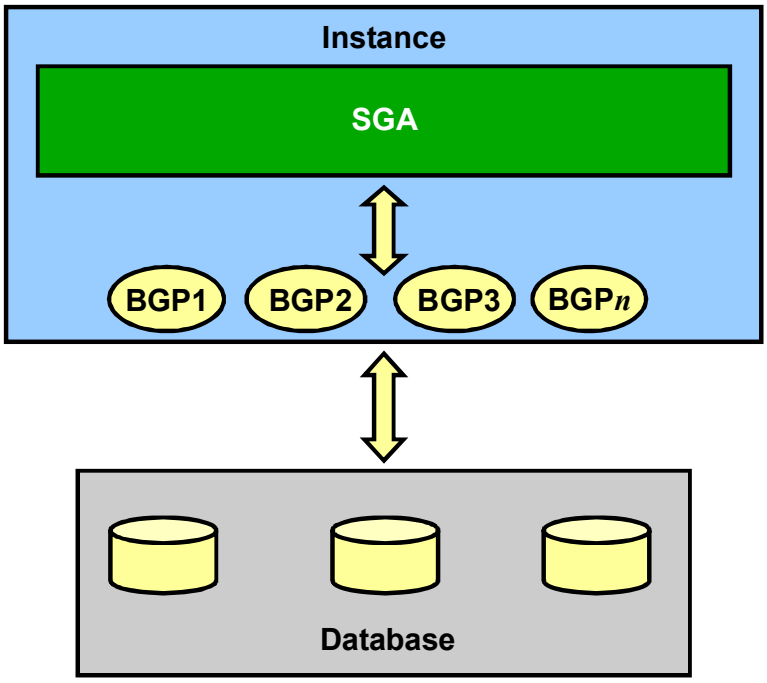
Objectives

After completing this lesson, you should be able to:

- List the major architectural components of the Oracle Database server
- Explain memory structures
- Describe background processes
- Correlate logical and physical storage structures

This lesson provides an overview of the Oracle Database server architecture. You learn about its architectural components, memory structures, background processes, and logical and physical storage structures.

Oracle Database Server Architecture: Overview



An Oracle Database server consists of an Oracle Database and one or more Oracle Database instances. An instance consists of memory structures and background processes. Every time an instance is started, a shared memory area called the System Global Area (SGA) is allocated and the background processes are started.

The SGA contains data and control information for one Oracle Database instance.

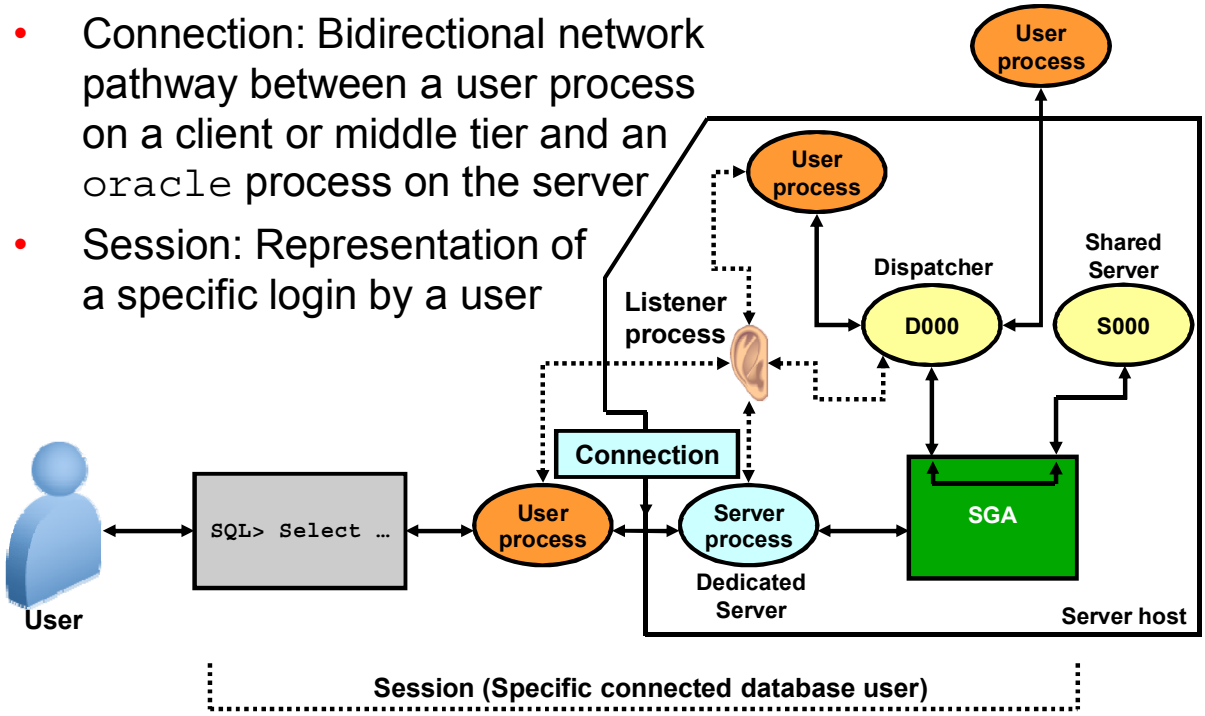
The background processes consolidate functions that would otherwise be handled by multiple Oracle Database server programs running for each user process. They may asynchronously perform input/output (I/O) and monitor other Oracle Database processes to provide increased parallelism for better performance and reliability.

The database consists of physical files and logical structures discussed later in this lesson. Because the physical and logical structures are separate, the physical storage of data can be managed without affecting access to the logical storage structures.

Note: Oracle Real Application Clusters (Oracle RAC) comprises two or more Oracle Database instances running on multiple clustered computers that communicate with each other by means of an interconnect and access the same Oracle Database.

Connecting to the Database Instance

- Connection: Bidirectional network pathway between a user process on a client or middle tier and an `oracle` process on the server
- Session: Representation of a specific login by a user



ORACLE

When users connect to an Oracle Database server, they are connected to an Oracle Database instance. The database instance services those users by allocating other memory areas in addition to the SGA and by starting other processes in addition to the Oracle Database background processes:

- User processes, which are sometimes called client or foreground processes, are created to run the software code of an application program. Most environments have separate machines for the client processes. A user process also manages communication with a corresponding server process through a program interface.
- The Oracle Database server creates server processes to handle requests from connected user processes. A server process communicates with the user process and interacts with the instance and the database to carry out requests from the associated user process.

An Oracle Database instance can be configured to vary the number of user processes for each server process. In a dedicated server configuration, a server process handles requests for a single user process.

A shared server configuration enables many user processes to share a small number of shared server processes, minimizing the number of server processes and maximizing the use of available system resources. One or more dispatcher processes are then used to queue user process requests in the SGA and dequeue shared server responses.

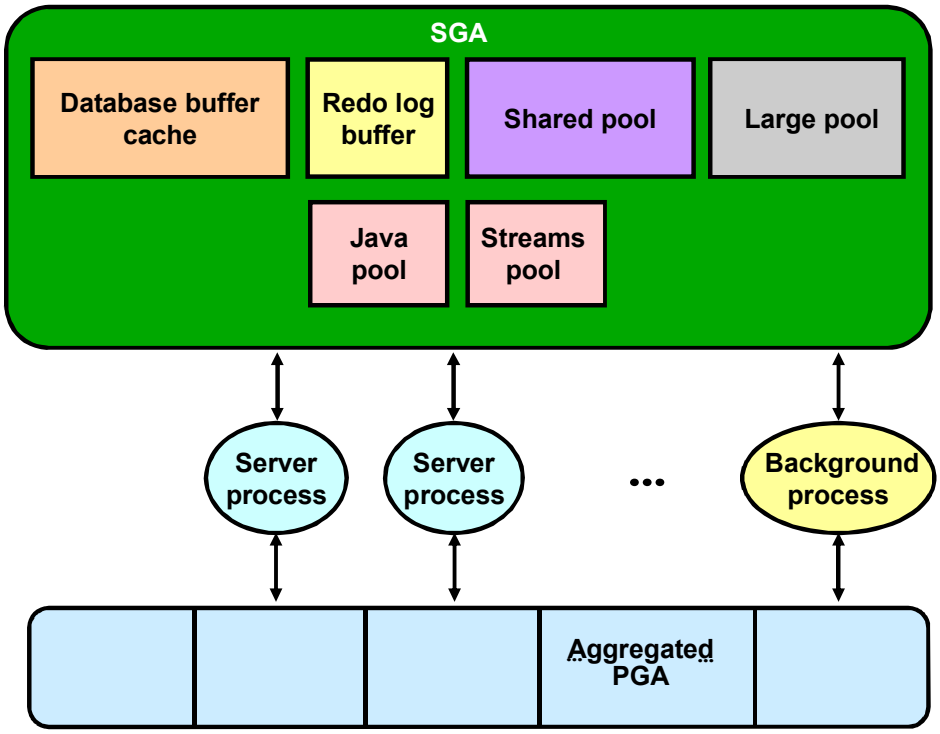
The Oracle Database server runs a listener that is responsible for handling network connections. The application connects to the listener that creates a dedicated server process or handles the connection to a dispatcher.

Connections and sessions are closely related to user processes, but are very different in meaning:

- A connection is a communication pathway between a user process and an Oracle Database instance. A communication pathway is established by using available interprocess communication mechanisms (on a computer that runs both the user process and Oracle Database) or network software (when different computers run the database application and the Oracle Database, and communicate through a network).
- A session represents the state of a current database user login to the database instance. For example, when a user starts Oracle SQL*Plus, the user must provide a valid database username and password, and then a session is established for that user. A session lasts from the time when a user connects until the user disconnects or exits the database application.

Note: Multiple sessions can be created and exist concurrently for a single Oracle Database user by using the same username. For example, a user with the username/password of `HR`/`HR` can connect to the same Oracle Database instance several times.

Oracle Database Memory Structures: Overview

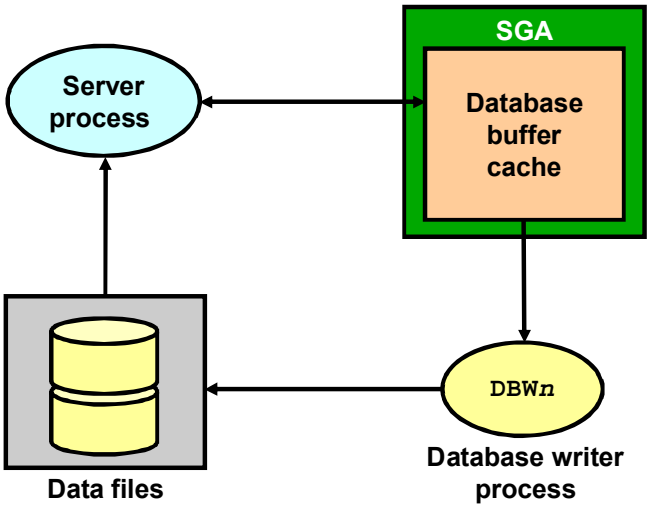


Oracle Database allocates memory structures for various purposes. For example, memory stores the program code that is run, data that is shared among users, and private data areas for each connected user. Two basic memory structures are associated with an instance:

- **System Global Area (SGA):** The SGA is shared by all server and background processes. The SGA includes the following data structures:
  - **Database buffer cache:** Caches blocks of data retrieved from the database files
  - **Redo log buffer:** Caches recovery information before writing it to the physical files
  - **Shared pool:** Caches various constructs that can be shared among sessions
  - **Large pool:** Optional area used for certain operations, such as Oracle backup and recovery operations, and I/O server processes
  - **Java pool:** Used for session-specific Java code and data in the Java Virtual Machine (JVM)
  - **Streams pool:** Used by Oracle Streams to store information about the capture and apply processes
- **Program Global Areas (PGA):** Memory regions that contain data and control information about a server or background process. A PGA is suballocated from the aggregated PGA area.

Database Buffer Cache

- Is a part of the SGA
- Holds copies of data blocks that are read from data files
- Is shared by all concurrent processes



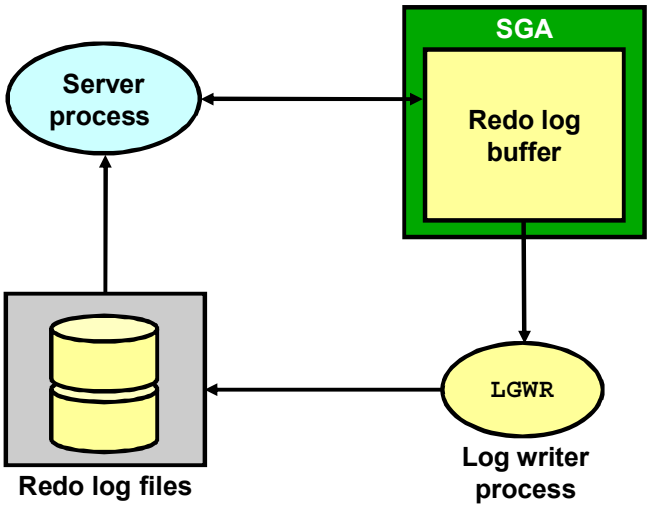
The database buffer cache is the portion of the SGA that holds copies of data blocks which are read from data files. All users concurrently connected to the instance share access to the database buffer cache.

The first time an Oracle Database server process requires a particular piece of data, it searches for the data in the database buffer cache. If the process finds the data already in the cache (a cache hit), it can read the data directly from memory. If the process cannot find the data in the cache (a cache miss), it must copy the data block from a data file on disk into a buffer in the cache before accessing the data. Accessing data through a cache hit is faster than accessing data through a cache miss.

The buffers in the cache are managed by a complex algorithm that uses a combination of least recently used (LRU) lists and touch count. The `DBWn` (Database Writers) processes are responsible for writing modified (dirty) buffers in the database buffer cache to disk when necessary.

Redo Log Buffer

- Is a circular buffer in the SGA (based on the number of CPUs)
- Contains redo entries that have the information to redo changes made by operations, such as DML and DDL



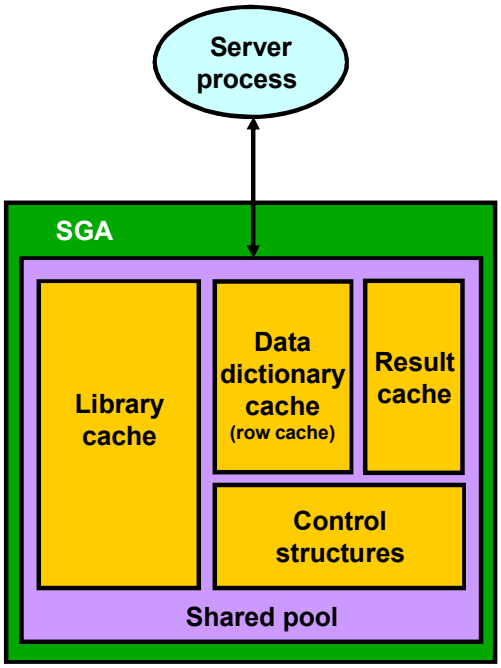
The redo log buffer is a circular buffer in the SGA that holds information about changes made to the database. This information is stored in redo entries. Redo entries contain the information necessary to reconstruct (or redo) changes that are made to the database by `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `ALTER`, or `DROP` operations. Redo entries are used for database recovery, if necessary.

Redo entries are copied by Oracle Database server processes from the user's memory space to the redo log buffer in the SGA. The redo entries take up continuous, sequential space in the buffer. The `LGWR` (log writer) background process writes the redo log buffer to the active redo log file (or group of files) on disk. `LGWR` is a background process that is capable of asynchronous I/O.

Note: Depending on the number of CPUs on your system, there may be more than one redo log buffer. They are automatically allocated.

Shared Pool

- Is part of the SGA
- Contains:
  - Library cache
    - Shared parts of SQL and PL/SQL statements
  - Data dictionary cache
  - Result cache:
    - SQL queries
    - PL/SQL functions
  - Control structures
    - Locks

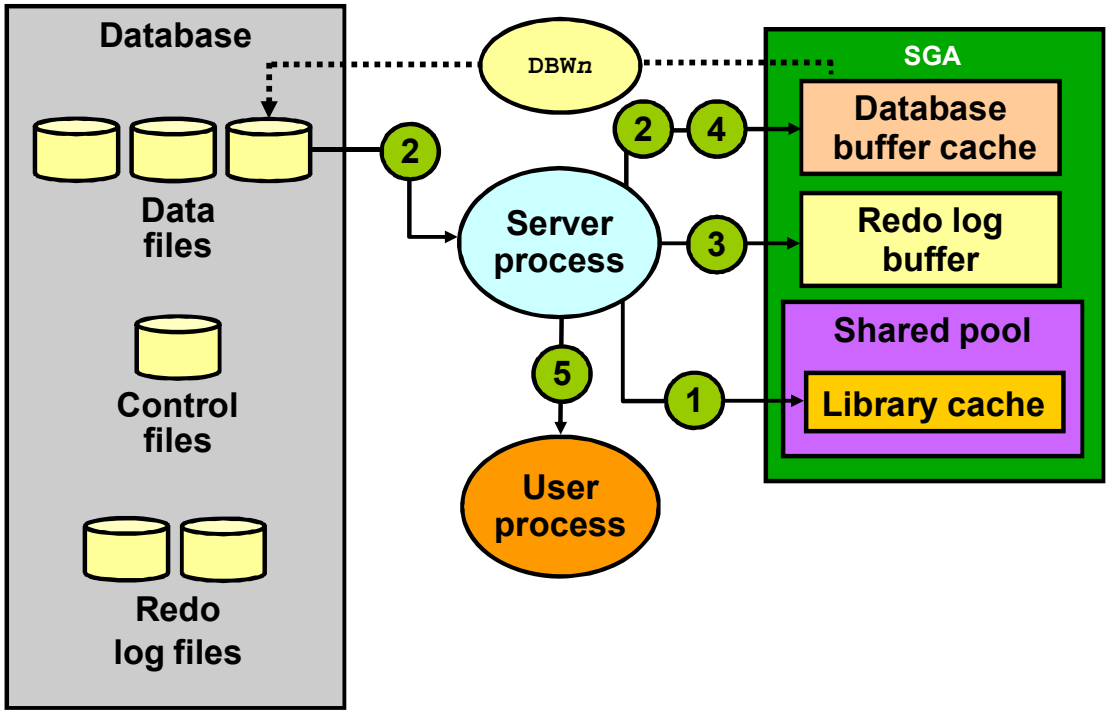


The shared pool portion of the SGA contains the following main parts:

- The library cache includes the shareable parts of SQL statements, PL/SQL procedures, and packages. It also contains control structures, such as locks.
- The data dictionary is a collection of database tables containing reference information about the database. The data dictionary is accessed so often by the Oracle Database that two special locations in memory are designated to hold dictionary data. One area is called the data dictionary cache, also known as the row cache, and the other area is called the library cache. All Oracle Database server processes share these two caches for access to data dictionary information.
- The result cache is composed of the SQL query result cache and the PL/SQL function result cache. This cache is used to store results of SQL queries or PL/SQL functions to speed up their future executions.
- Control structures are essentially lock structures.

Note: In general, any item in the shared pool remains until it is flushed according to a modified LRU algorithm.

Processing a DML Statement: Example

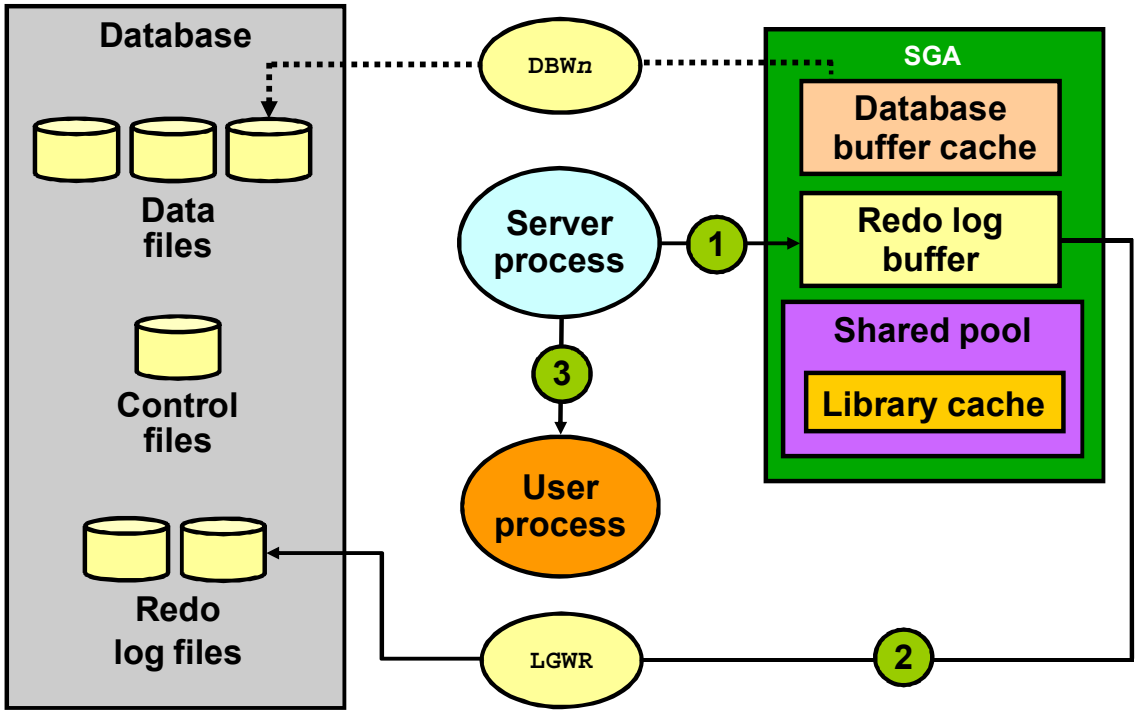


The steps involved in executing a data manipulation language (DML) statement are:

1. The server process receives the statement and checks the library cache for any shared SQL area that contains a similar SQL statement. If a shared SQL area is found, the server process checks the user's access privileges for the requested data, and the existing shared SQL area is used to process the statement. If not, a new shared SQL area is allocated for the statement, so that it can be parsed and processed.
2. If the data and undo segment blocks are not already in the buffer cache, the server process reads them from the data files into the buffer cache. The server process locks the rows that are to be modified.
3. The server process records the changes to be made to the data buffers as well as the undo changes. These changes are written to the redo log buffer before the in-memory data and undo buffers are modified. This is called write-ahead logging.
4. The undo segment buffers contain values of the data before it is modified. The undo buffers are used to store the before image of the data so that the DML statements can be rolled back, if necessary. The data buffers record the new values of the data.
5. The user gets the feedback from the DML operation (such as how many rows were affected by the operation).

Note: Any changed blocks in the buffer cache are marked as dirty buffers; that is, the buffers are not the same as the corresponding blocks on the disk. These buffers are not immediately written to disk by the `DBWn` processes.

COMMIT Processing: Example



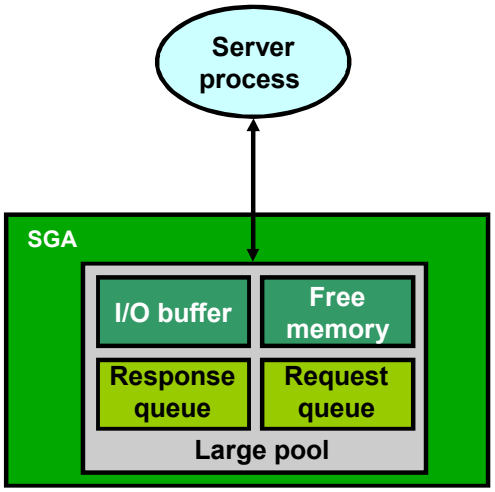
When COMMIT is issued, the following steps are performed:

1. The server process places a commit record, along with the system change number (SCN), in the redo log buffer. The SCN is monotonically incremented and is unique within the database. It is used by Oracle Database as an internal time stamp to synchronize data and to provide read consistency when data is retrieved from the data files. Using the SCN enables Oracle Database to perform consistency checks without depending on the date and time of the operating system.
2. The LGWR background process performs a contiguous write of all the redo log buffer entries up to and including the commit record to the redo log files. After this point, Oracle Database can guarantee that the changes are not lost even if there is an instance failure.
3. If modified blocks are still in the SGA, and if no other session is modifying them, then the database removes lock-related transaction information from the blocks. This process is known as commit cleanout.
4. The server process provides feedback to the user process about the completion of the transaction.

Note: If not done already, DBW*n* eventually writes the actual changes back to disk based on its own internal timing mechanism.

Large Pool

- Provides large memory allocations for:
  - Session memory for the shared server and Oracle XA interface
  - Parallel execution buffers
  - I/O server processes
  - Oracle Database backup and restore operations
- Optional pool better suited when using the following:
  - Parallel execution
  - Recovery Manager
  - Shared server



ORACLE

You can configure an optional memory area called the large pool to provide large memory allocations for the following components:

- Session memory for the shared server, the Oracle XA interface (used where transactions interact with more than one database), or parallel execution buffers
- I/O server processes
- Oracle Database backup and restore operations

By allocating the memory components from the large pool, Oracle Database can use the shared pool primarily for caching the shared part of SQL and PL/SQL constructs. The shared pool was originally designed to store SQL and PL/SQL constructs. Using the large pool helps avoid fragmentation issues associated with large and small allocations that share the same memory area. Unlike the shared pool, the large pool does not have an LRU list.

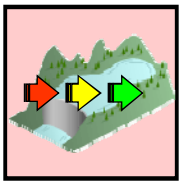
You should consider configuring a large pool if your instance uses any of the following:

- **Parallel execution:** Parallel query uses shared pool memory to cache parallel execution message buffers.
- **Recovery Manager:** Recovery Manager uses the shared pool to cache I/O buffers during backup and restore operations.
- **Shared server:** In a shared server architecture, the session memory for each client process is included in the shared pool.

Java Pool and Streams Pool

- Java pool memory is used in server memory for all session-specific Java code and data in the JVM.
- Streams pool memory is used exclusively by Oracle Streams to:
  - Store buffered queue messages
  - Provide memory for Oracle Streams processes

Java pool



Streams pool

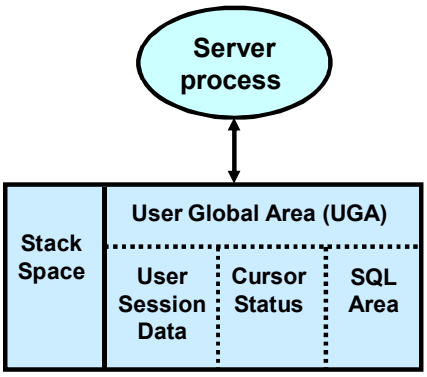
Java pool memory is used for all session-specific Java code and data in the JVM. Java pool memory is used in different ways, depending on the mode in which Oracle Database runs.

Oracle Streams enables the propagation and management of data, transactions, and events in a data stream either within a database or from one database to another. The Streams pool is used exclusively by Oracle Streams. The Streams pool stores buffered queue messages, and it provides memory for Oracle Streams capture and apply processes.

Note: A detailed discussion of Java programming and Oracle Streams is beyond the scope of this course.

Program Global Area

- PGA is a memory area that contains:
  - Session information
  - Cursor information
  - SQL execution work areas:
    - Sort area
    - Hash join area
    - Bitmap merge area
    - Bitmap create area



- The size of the work area influences SQL performance.
- Work areas can be managed automatically or manually.

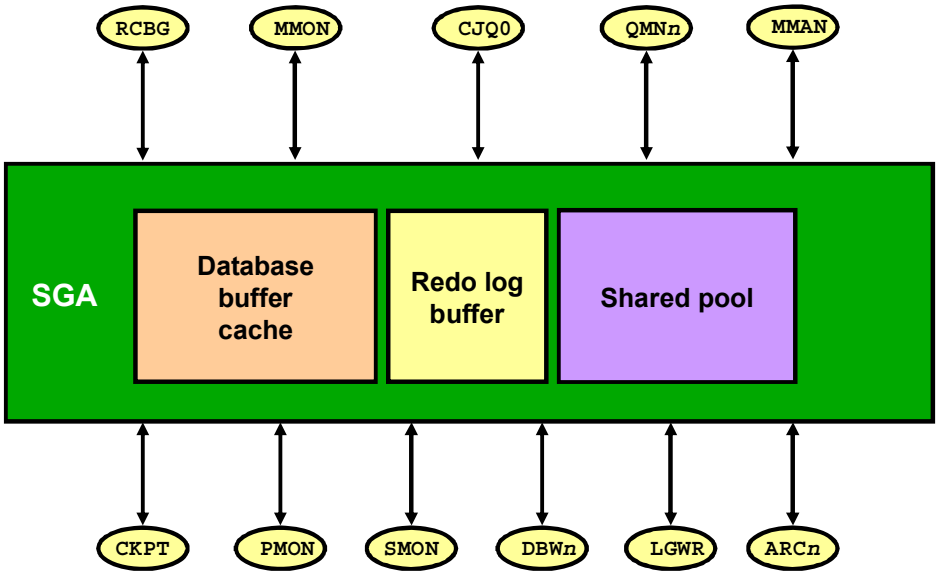
ORACLE

The PGA can be compared to a temporary countertop workspace used by a file clerk (the server process) to perform a function on behalf of a customer (client process). The clerk clears a section of the countertop, uses the workspace to store details about the customer's request, and then gives up the space when the work is done.

Generally, the PGA memory is divided into the following areas:

- Session memory is the memory allocated to hold a session's variables (login information) and other information related to the session. For a shared server, the session memory is shared and not private.
- Cursors are handles to private memory structures of specific SQL statements.
- SQL work areas are allocated to support memory-intensive operators, such as those listed in the slide. Generally, bigger work areas can significantly improve the performance of a particular operator at the cost of higher memory consumption.

Background Process

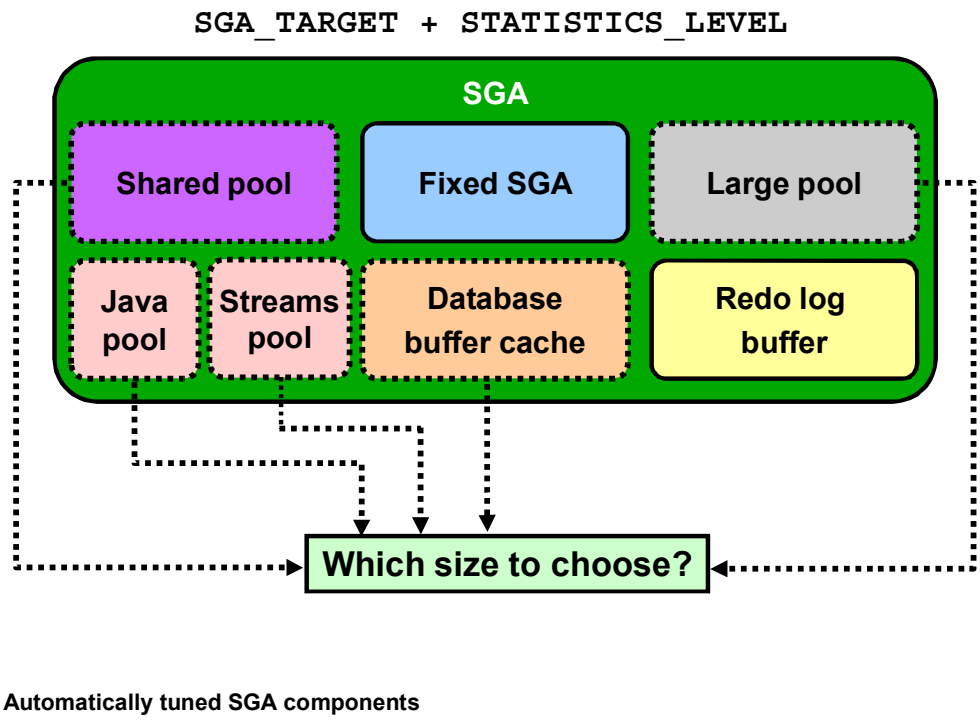


The background processes commonly seen in non-RAC, non-ASM environments can include the following:

- **Database writer process (`DBWn`):** Asynchronously writes modified (dirty) buffers in the database buffer cache to disk
- **Log writer process (`LGWR`):** Writes the recovery information called redo information in the log buffer to a redo log file on disk
- **Checkpoint process (`CKPT`):** Records checkpoint information in control files and each data file header
- **System Monitor process (`SMON`):** Performs recovery at instance startup and cleans up unused temporary segments
- **Process monitor process (`PMON`):** Performs process recovery when a user process fails
- **Result cache background process (`RCBG`):** Used to maintain the result cache in the shared pool

- **Job queue process (`CJQ0`):** Runs user jobs used in batch processing through the Scheduler
- **Archiver processes (`ARC`*n*):** Copies redo log files to a designated storage device after a log switch has occurred
- **Queue monitor processes (`QMN`*n*):** Monitors the Oracle Streams message queues
- **Manageability monitoring process (`MMON`):** Performs manageability-related background tasks
- **Memory Manager background process (`MMAN`):** Used to manage SGA and PGA memory components automatically

Automatic Shared Memory Management

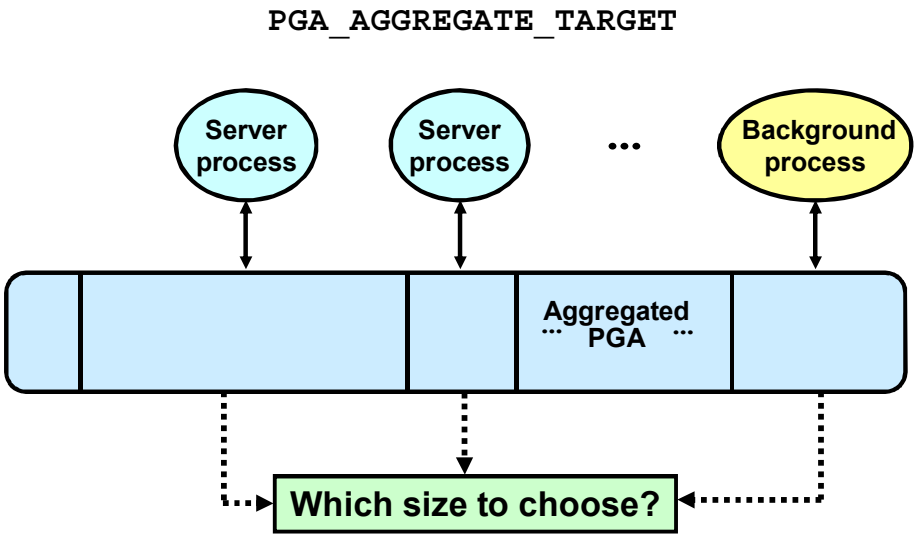


You can use the Automatic Shared Memory Management (ASMM) feature to enable the database to automatically determine the size of each of the memory components listed in the slide within the limits of the total SGA size.

The system uses an SGA size parameter (`SGA_TARGET`) that includes all memory in the SGA, including all automatically-sized components, manually-sized components, and any internal allocations during startup. ASMM simplifies the configuration of the SGA by enabling you to specify a total memory amount to be used for all SGA components. Oracle Database then periodically redistributes memory between the automatically-tuned components, according to workload requirements.

Note: You must set `STATISTICS_LEVEL` to `TYPICAL` or `ALL` to use ASMM.

Automated SQL Execution Memory Management



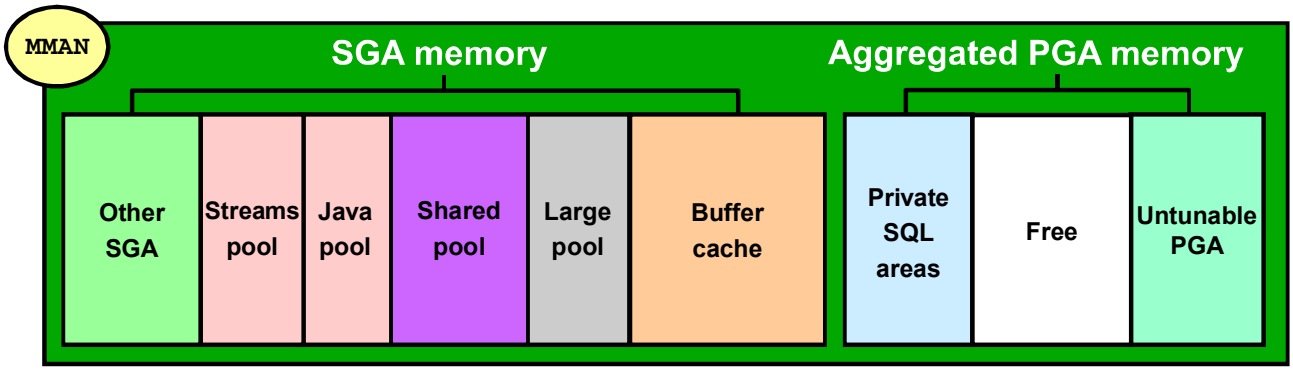
This feature provides an automatic mode for allocating memory to working areas in the PGA. You can use the `PGA_AGGREGATE_TARGET` parameter to specify the total amount of memory that should be allocated to the PGA areas of the sessions of the instance. In the automatic mode, working areas that are used by memory-intensive operators (sorts and hash joins) can be adjusted automatically and dynamically.

This feature offers several performance and scalability benefits for decision support system (DSS) workloads and mixed workloads with complex queries. The overall system performance is maximized, and the available memory is allocated more efficiently among queries to optimize both throughput and response time. In particular, the savings that are gained from improved use of memory translate to better throughput at high loads.

Note: In earlier releases of the Oracle Database server, you had to manually specify the maximum work area size for each type of SQL operator, such as sort or hash join. This method proved to be very difficult because the workload changes constantly. Although the current release of Oracle Database supports this manual PGA memory management method that might be useful for specific sessions, it is recommended that you leave automatic PGA memory management enabled.

Automatic Memory Management

- Sizing of each memory component is vital for SQL execution performance.
- It is difficult to manually size each component.
- Automatic memory management automates memory allocation of each SGA component and aggregated PGA.



MEMORY_TARGET + STATISTICS_LEVEL

As seen already, the size of the various memory areas of the instance directly impacts the speed of SQL processing. Depending on the database workload, it is difficult to size those components manually.

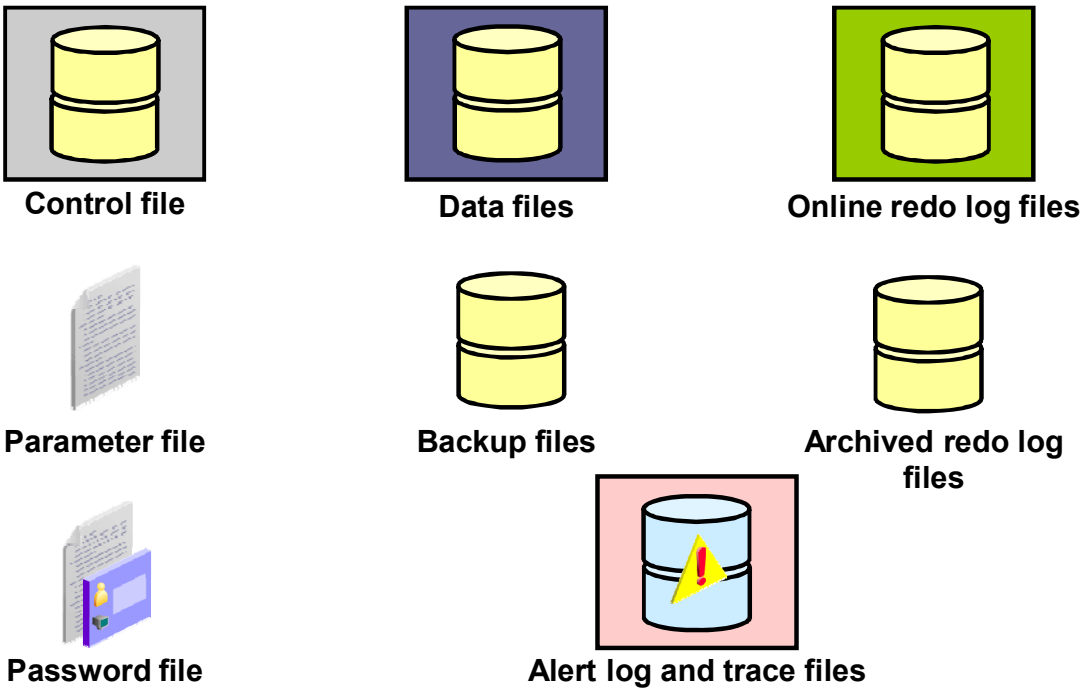
With Automatic Memory Management, the system automatically adapts the size of each memory's components to your workload memory needs.

When you set your `MEMORY_TARGET` initialization parameter for the database instance, the `MMAN` background process automatically tunes to the target memory size, redistributing memory as needed between the internal components of the SGA and between the SGA and the aggregated PGAs.

The ASSM feature uses the SGA memory broker that is implemented by two background processes: Manageability Monitor (MMON) and Memory Manager (MMAN). Statistics and memory advisory data are periodically captured in memory by MMON. MMAN coordinates the sizing of the memory components according to MMON decisions.

Note: Currently, this mechanism is implemented only on Linux, Solaris, HP-UX, AIX, and Windows.

Database Storage Architecture



The files that constitute an Oracle database are organized into the following:

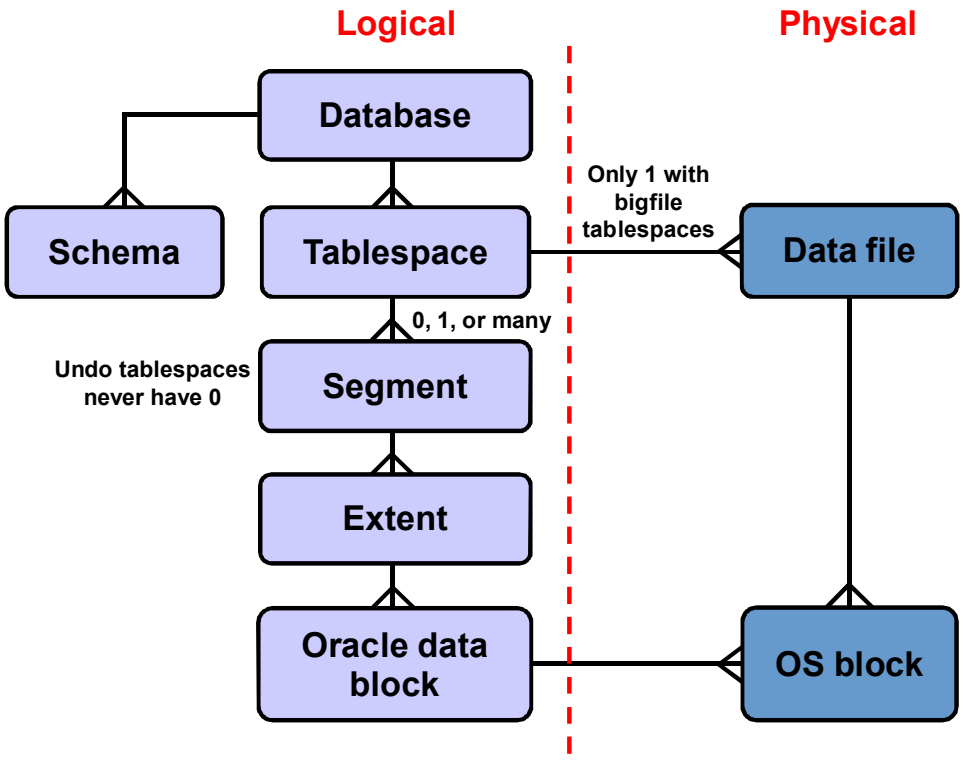
- **Control files:** Contain data about the database itself (that is, physical database structure information). These files are critical to the database. Without them, you cannot open data files to access the data in the database.
- **Data files:** Contain the user or application data of the database, as well as metadata and the data dictionary.
- **Online redo log files:** Allow for instance recovery of the database. If the database server crashes and does not lose any data files, the instance can recover the database with the information in these files.

The following additional files are important for the successful running of the database:

- **Parameter file:** Is used to define how the instance is configured when it starts up
- **Password file:** Allows `sysdba`, `sysoper`, and `sysasm` to connect remotely to the database and perform administrative tasks
- **Backup files:** Are used for database recovery. You typically restore a backup file when a media failure or user error has damaged or deleted the original file.
- **Archived redo log files:** Contain an ongoing history of the data changes (redo) that are generated by the instance. By using these files and a backup of the database, you can recover a lost data file; that is, archive logs enable the recovery of restored data files.

- **Trace files:** Each server and background process can write to an associated trace file. When an internal error is detected by a process, the process dumps information about the error to its trace file. Some of the information written to a trace file is intended for the developer, whereas other information is for Oracle Support Services.
- **Alert log file:** These files are special trace entries. The alert log of a database is a chronological log of messages and errors. Each instance has one alert log file. It is recommended that you periodically review this file.

Logical and Physical Database Structures



The database has logical structures and physical structures.

Tablespaces

A database is divided into logical storage units called tablespaces, which group related logical structures together. For example, tablespaces commonly group all the objects of an application to simplify some administrative operations. You may have a tablespace for different applications.

Databases, Tablespaces, and Data Files

The relationship among databases, tablespaces, and data files is illustrated in the slide. Each database is logically divided into one or more tablespaces. One or more data files are explicitly created for each tablespace to physically store the data of all logical structures in a tablespace. If it is a TEMPORARY tablespace instead of a tablespace containing data, the tablespace has a temporary file.

Schemas

A schema is a collection of database objects that are owned by a database user. Schema objects are the logical structures that directly refer to the data contained in a database. Schema objects include structures, such as tables, views, sequences, stored procedures, synonyms, indexes, clusters, and database links. In general, schema objects include everything that your application creates in the database.

Data Blocks

At the finest level of granularity, the data contained in an Oracle Database is stored in data blocks. One data block corresponds to a specific number of bytes of physical database space on the disk. A data block size is specified for each tablespace when it is created. A database uses and allocates free database space in Oracle data blocks.

Extents

The next level of logical database space is an extent. An extent is a specific number of contiguous data blocks (obtained in a single allocation) that are used to store a specific type of information.

Segments

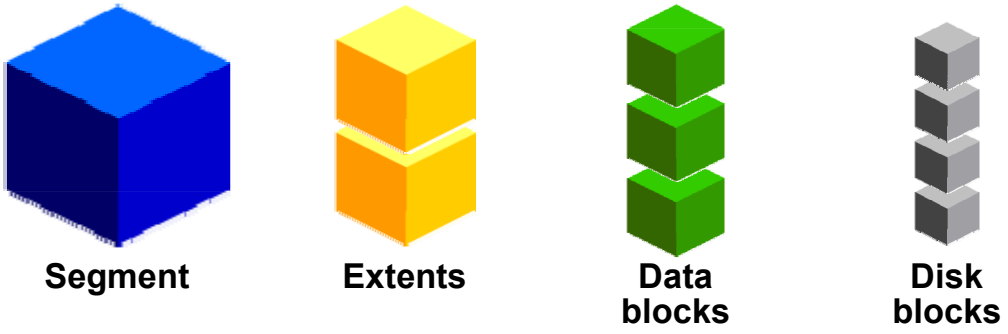
The level of logical database storage above an extent is called a segment. A segment is a set of extents that are allocated for a certain logical structure. Different types of segments include:

- **Data segments:** Each nonclustered, non-index-organized table has a data segment, with the exception of external tables and global temporary tables that have no segments, and partitioned tables in which each table has one or more segments. All the data contained in the table is stored in the extents of its data segment. For a partitioned table, each partition has a data segment. Each cluster has a data segment. The data of every table in the cluster is stored in the data segment of the cluster.
- **Index segments:** Each index has an index segment that stores all of its data. For a partitioned index, each partition has an index segment.
- **Undo segments:** One `UNDO` tablespace is created for each database instance. This tablespace contains numerous undo segments to temporarily store undo information. The information in an undo segment is used to generate read-consistent database information and, during database recovery, to roll back uncommitted transactions for users.
- **Temporary segments:** Temporary segments are created by the Oracle Database when a SQL statement needs a temporary work area to complete execution. When the statement finishes execution, the temporary segment's extents are returned to the instance for future use. Specify either a default temporary tablespace for every user, or a default temporary tablespace that is used across the database.

Oracle Database dynamically allocates space. When the existing extents of a segment are full, additional extents are added. Because extents are allocated as needed, the extents of a segment may or may not be contiguous on the disk.

Segments, Extents, and Blocks

- Segments exist in a tablespace.
- Segments are collections of extents.
- Extents are collections of data blocks.
- Data blocks are mapped to disk blocks.



Database objects, such as tables and indexes, are stored as segments in tablespaces. Each segment contains one or more extents. An extent consists of contiguous data blocks, which means that each extent can exist only in one data file. Data blocks are the smallest units of I/O in the database.

When the database requests a set of data blocks from the operating system (OS), the OS maps this set to an actual file system or disk block on the storage device. Because of this, you do not need to know the physical address of any of the data in your database. This also means that a data file can be striped or mirrored on several disks.

The size of the data block can be set at the time of database creation. The default size of 8 KB is adequate for most databases. If your database supports a data warehouse application that has large tables and indexes, a larger block size may be beneficial.

If your database supports a transactional application in which reads and writes are random, specifying a smaller block size may be beneficial. The maximum block size depends on your OS. The minimum Oracle block size is 2 KB; it should rarely (if ever) be used.

You can have tablespaces with a nonstandard block size. For details, see the *Oracle Database Administrator's Guide*.

SYSTEM and SYSAUX Tablespaces

- The SYSTEM and SYSAUX tablespaces are mandatory tablespaces that are created at the time of database creation. They must be online.
- The SYSTEM tablespace is used for core functionality (for example, data dictionary tables).
- The auxiliary SYSAUX tablespace is used for additional database components (such as the Enterprise Manager Repository).

Each Oracle Database must contain a SYSTEM tablespace and a SYSAUX tablespace, which are automatically created when the database is created. The system default is to create a smallfile tablespace. You can also create bigfile tablespaces, which enable the Oracle database to manage ultralarge files (up to 8 exabytes in size).

A tablespace can be online (accessible) or offline (not accessible). The SYSTEM tablespace is always online when the database is open. It stores tables that support the core functionality of the database, such as the data dictionary tables.

The SYSAUX tablespace is an auxiliary tablespace to the SYSTEM tablespace. The SYSAUX tablespace stores many database components, and it must be online for the correct functioning of all database components.

Note: The SYSAUX tablespace may be taken offline for performing tablespace recovery, whereas this is not possible in the case of the SYSTEM tablespace. Neither of them may be made read-only.

Quiz

The first time an Oracle Database server process requires a particular piece of data, it searches for the data in the:

a. Database buffer cache
b. PGA
c. Redo log buffer
d. Shared pool

Answer: a

Quiz

Which of the following is not a database logical structure?

a. Tablespace
b. Data file
c. Schema
d. Segment

Answer: b

Quiz

The SYSAUX tablespace is used for core functionality, and the SYSTEM tablespace is used for additional database components, such as the Enterprise Manager Repository.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- List the major architectural components of the Oracle Database server
- Explain memory structures
- Describe background processes
- Correlate logical and physical storage structures