LAST UPDATED: 2025

SQL Tuning & Performance Masterclass (2025)

Course Slide for Students

This document is exclusively prepared for existing students to make a quick recap easily about what they have learned throughout the course!

What is SQL(Performance) Tuning and Why We Need?

What is SQL (Performance) Tuning and Why To Do That?

- What is SQL Tuning?
- Oracle is a Very Strong Database
- Better Performance – Less Hardware Cost
- SQL Tuning is the combination of some techniques
- SQL Tuning is not so hard
- Who Will Do the Tuning?

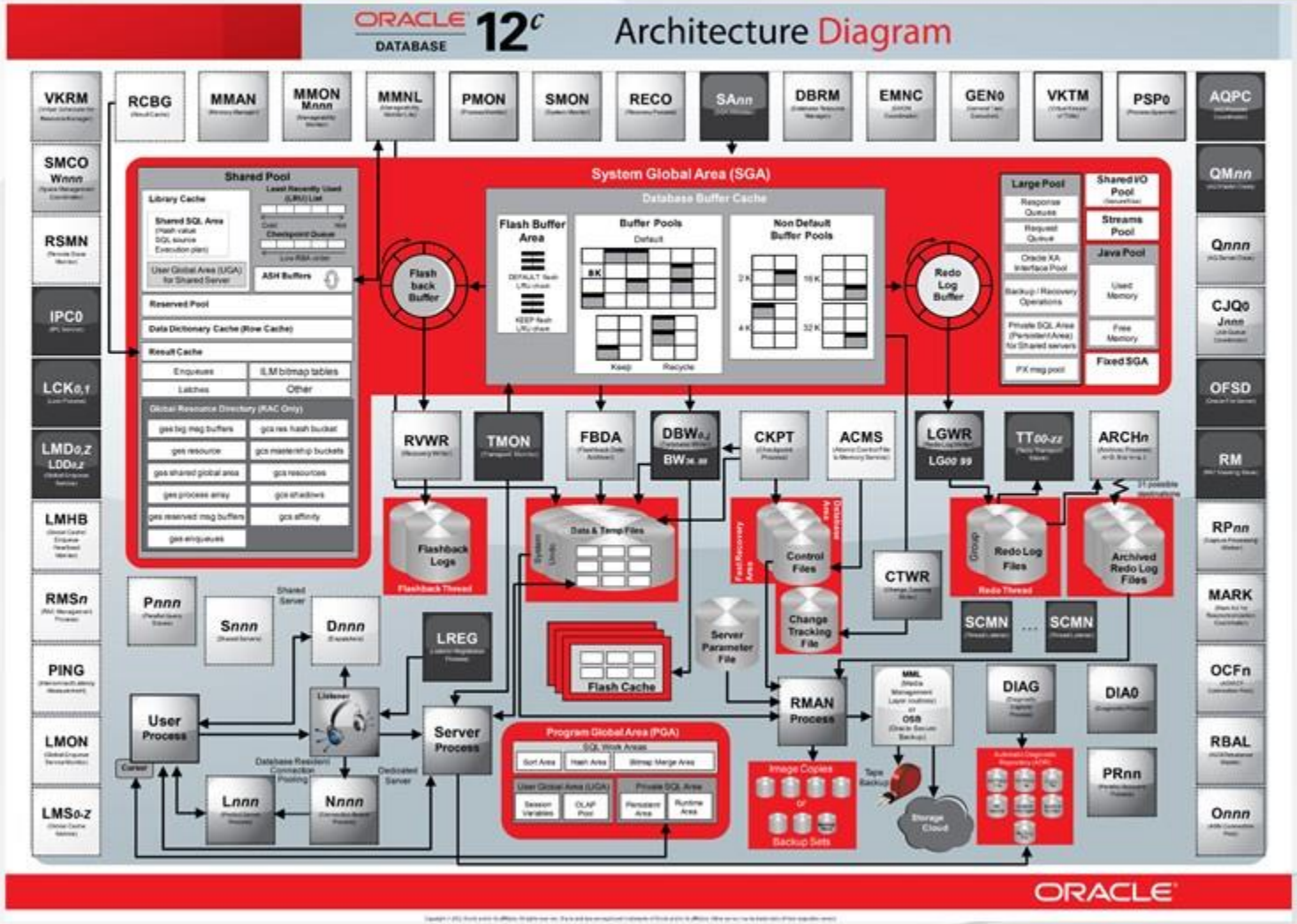
What is SQL(Performance) Tuning and Why We Need?

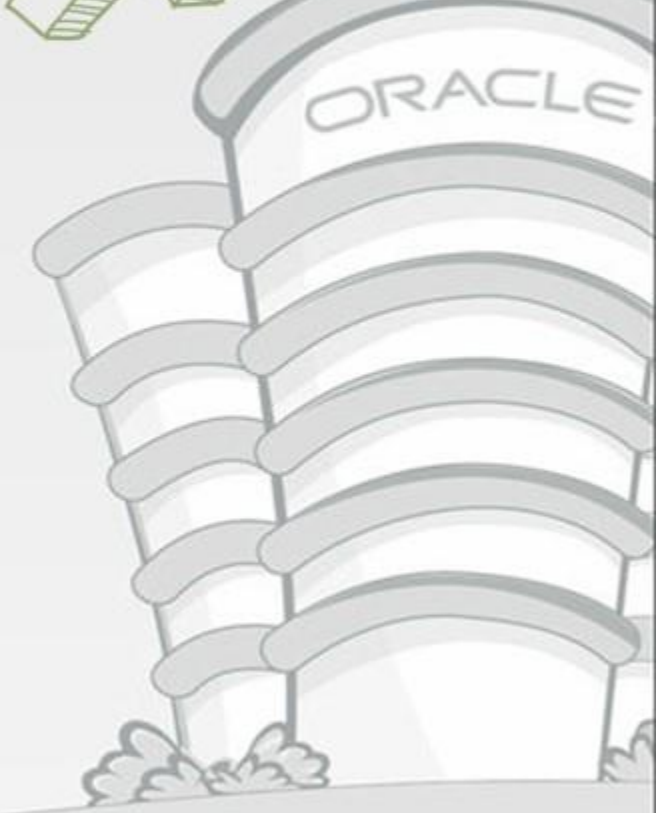
What to know Before Starting?

- No Need to Scare SQL Tuning
- You Need to Know Database Architecture in Basic
- There will be many introductive lectures at first
- SQL Tuning Needs Expertise
- Some Tools will Not Be Explained
- All The Subjects are Organized Strategically
- To sum up!

Oracle Database Architecture

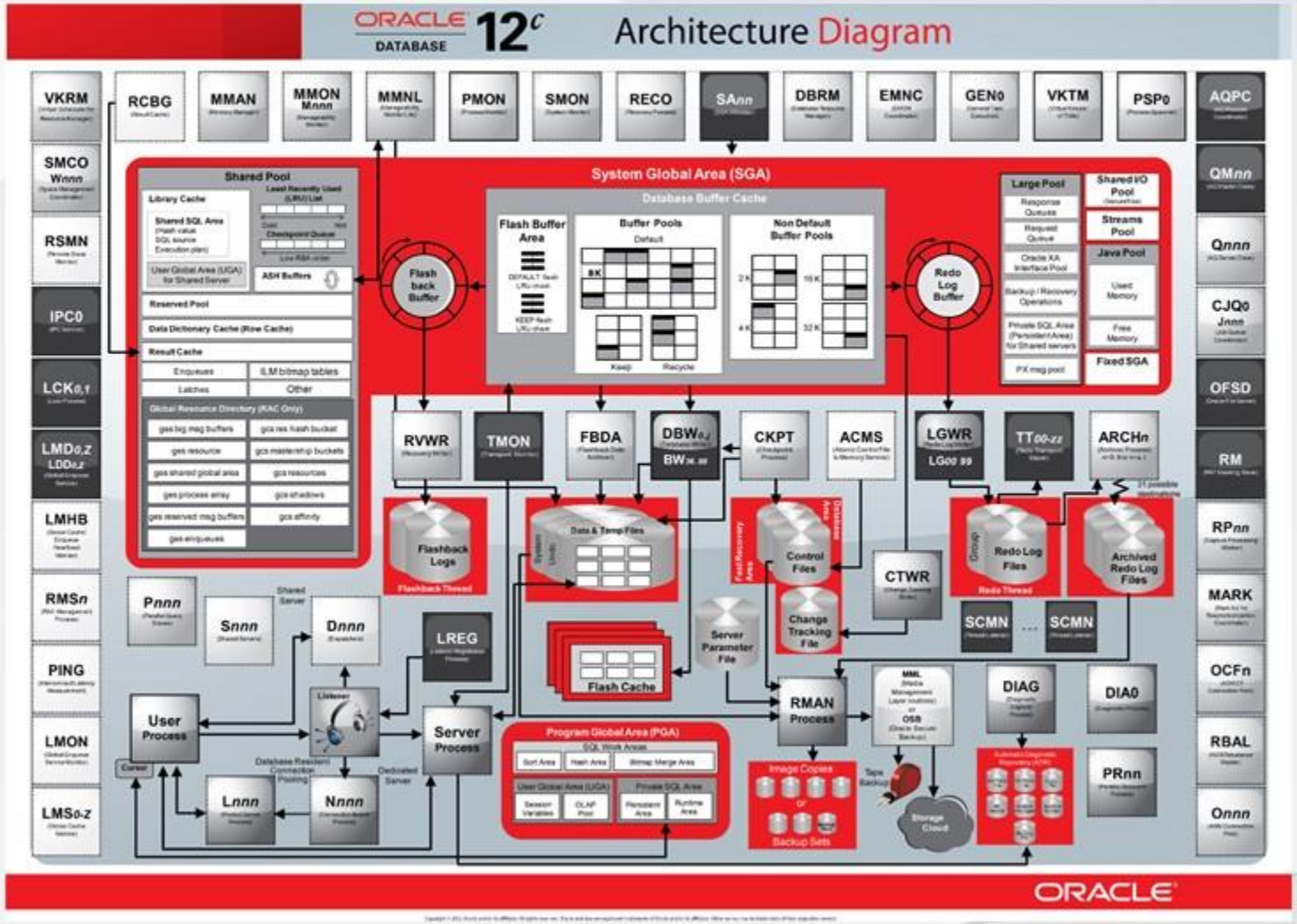
Why to Know the Architecture and How Much to Know?

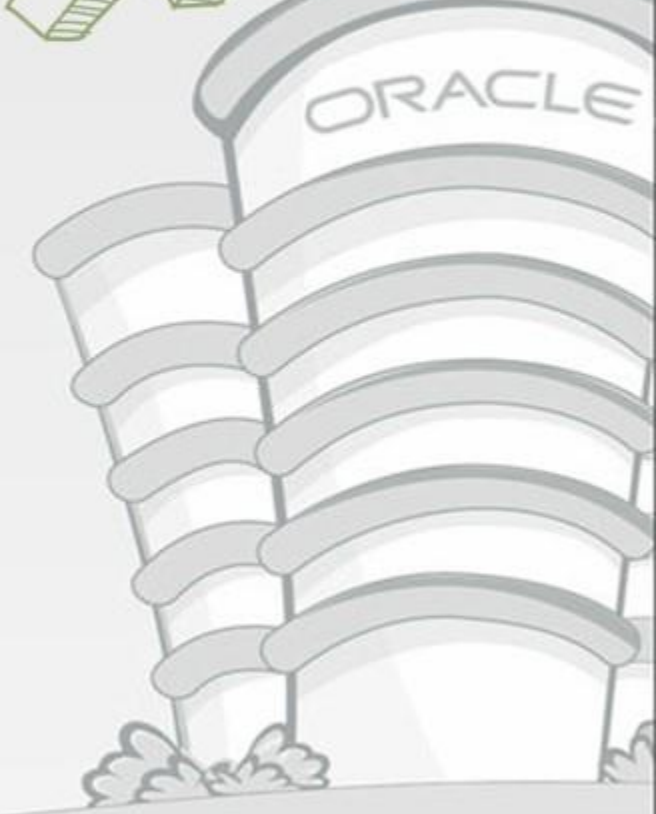




Oracle Database Architecture

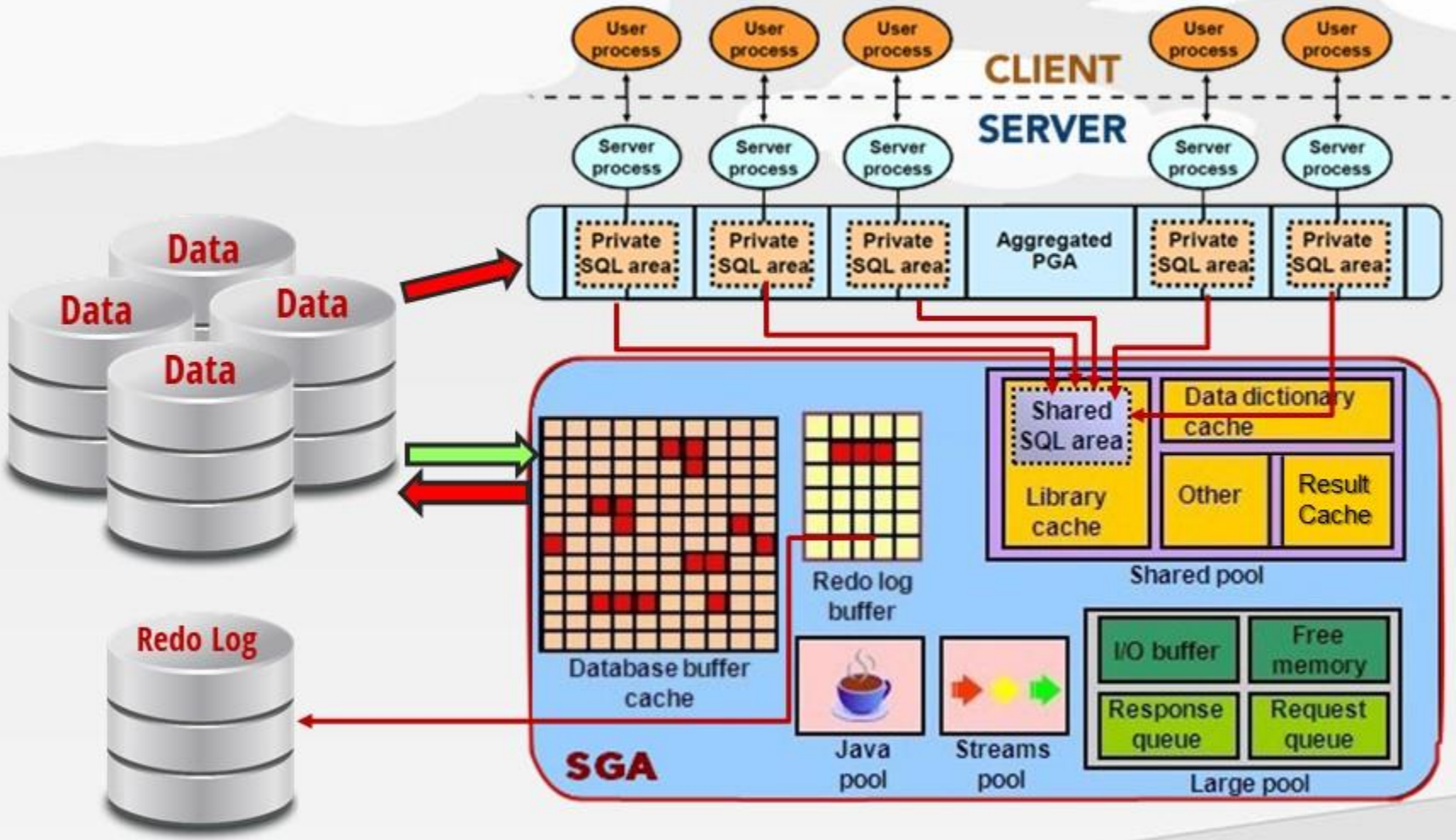
ARCHITECTURE OVERVIEW





Oracle Database Architecture

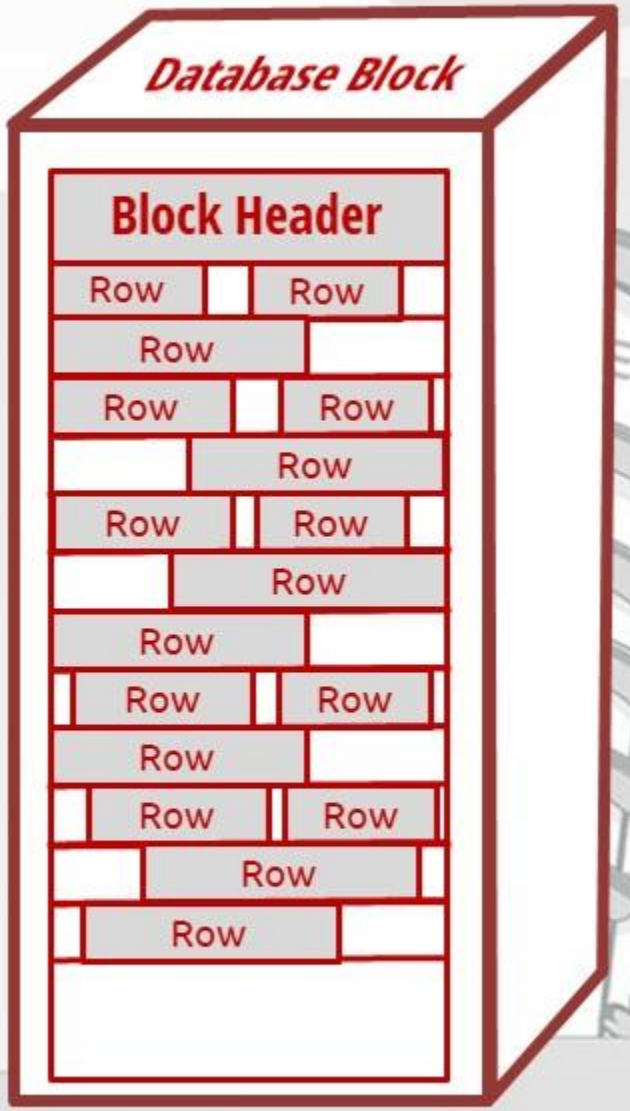
ARCHITECTURE OVERVIEW



Oracle Database Architecture

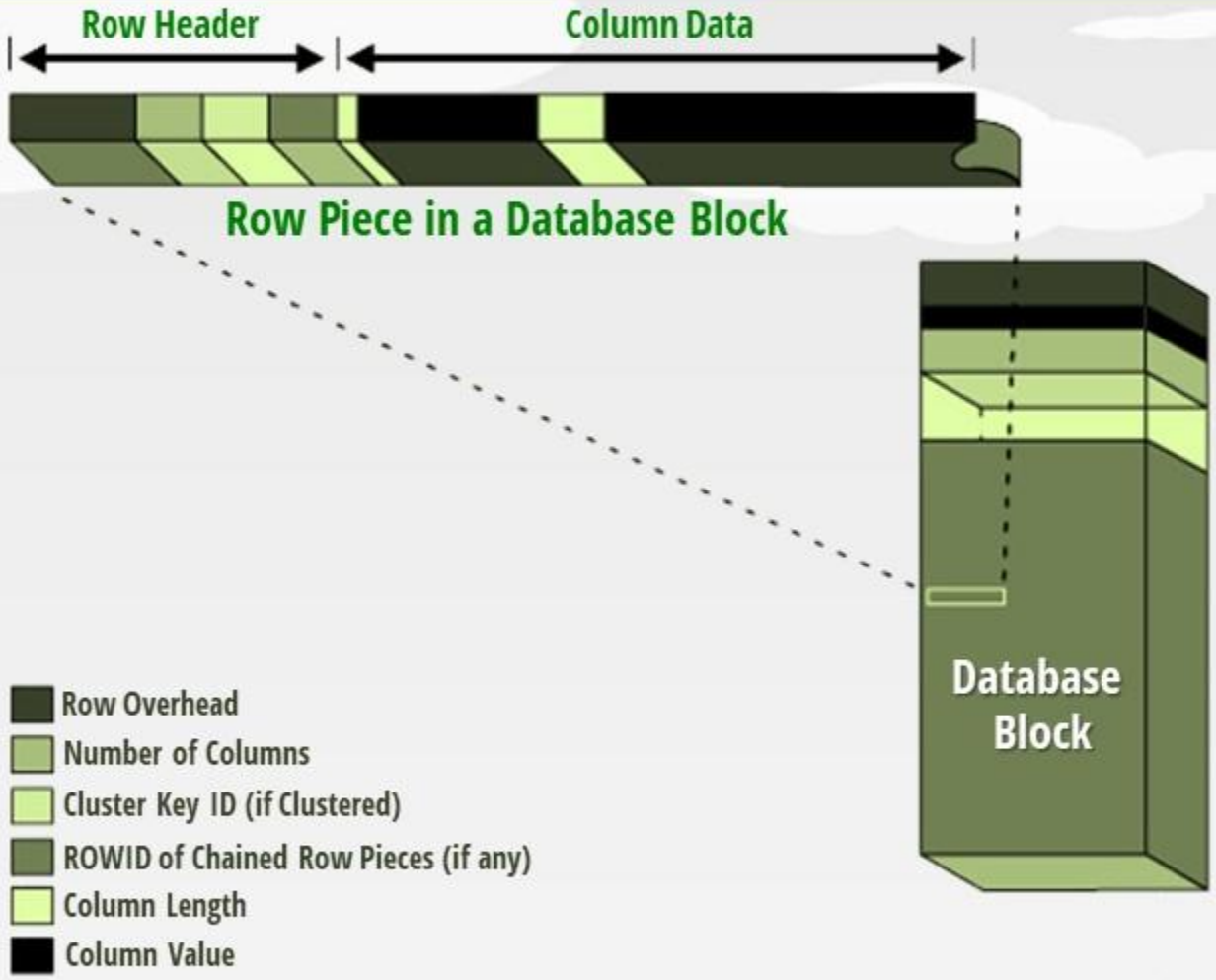
DATABASE BLOCKS in Detail

- All the data is stored in Blocks
- A block is the smallest unit of database storage
- A block can have 2KB to 32KB size (8KB default)
- A block stores row data or index data.
- Block Header Includes :
  - Block Type Information
  - Table Information
  - Row Directory
- We can use PCTFREE or PCTUSE parameters to specify the space size in blocks



Oracle Database Architecture

DATABASE BLOCKS in Detail



Oracle Database Architecture

WHAT IS PGA?

SESSION AREA

- Stores session information of each user.
- Stores session variables, login information, session status, etc.

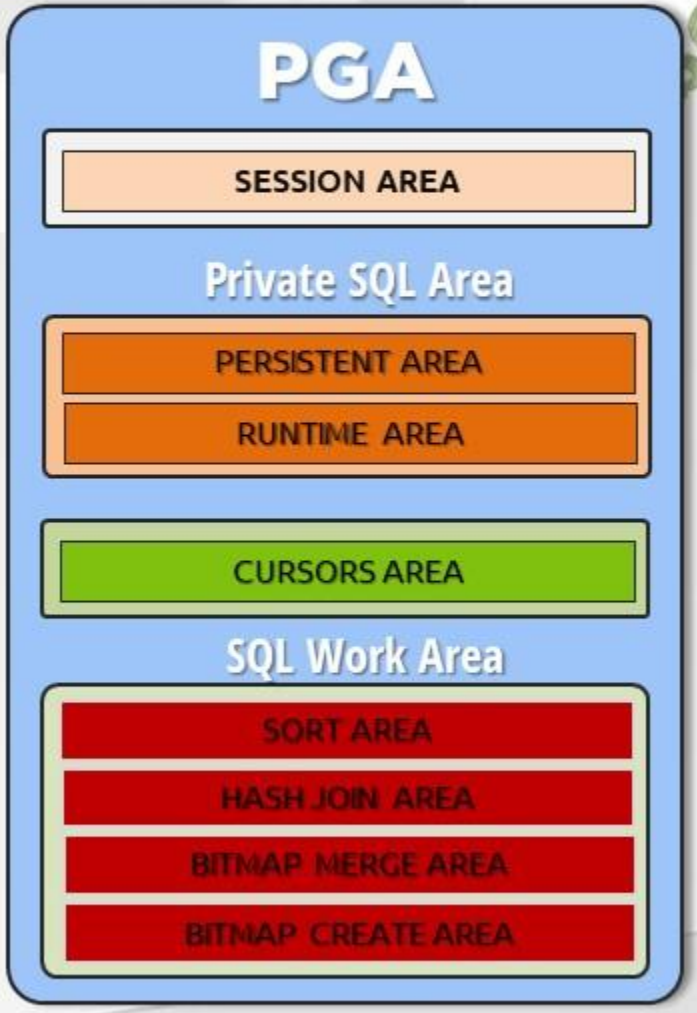
PRIVATE SQL AREA

- Persistent area stores the bind variables
- Runtime area stores the execution state info.

CURSOR AREA

- Stores the information of cursors

SQL WORK AREA



Oracle Database Architecture

WHAT IS SHARED POOL IN DETAIL?

DATA DICTIONARY CACHE

- Stores the definitions of the database objects and their permissions

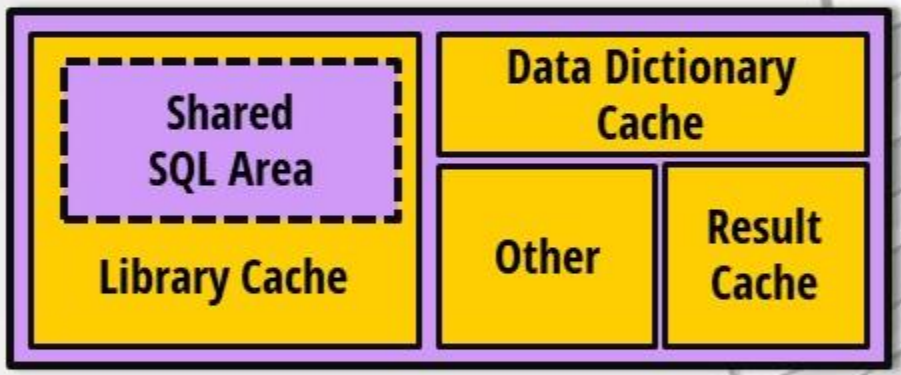
RESULT CACHE

- Stores the result of commonly used queries
- Stores the result of functions

LIBRARY CACHE

- Stores the execution plans
- Stores procedures, packages control structures

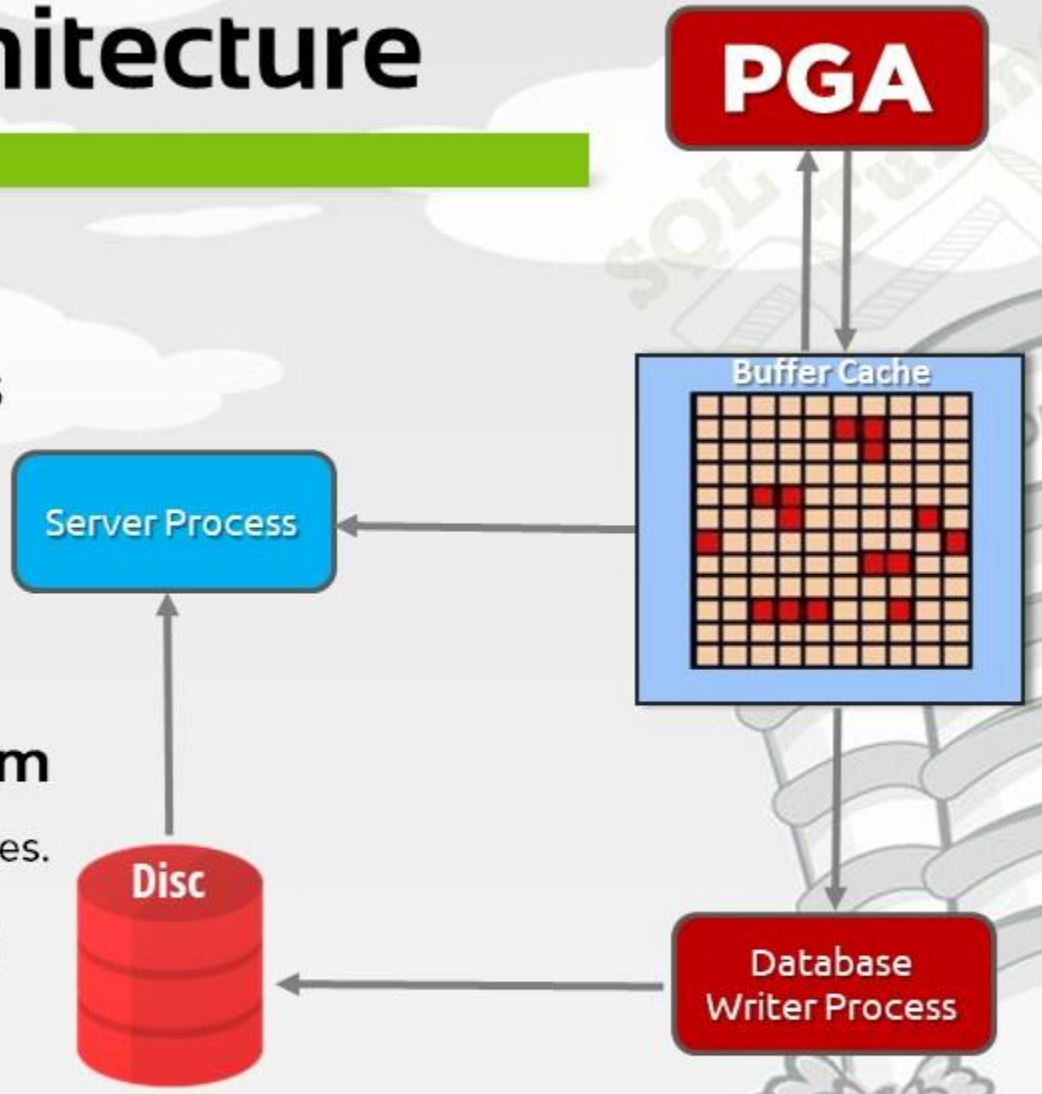
OTHER AREAS



Oracle Database Architecture

WHAT IS BUFFER CACHE?

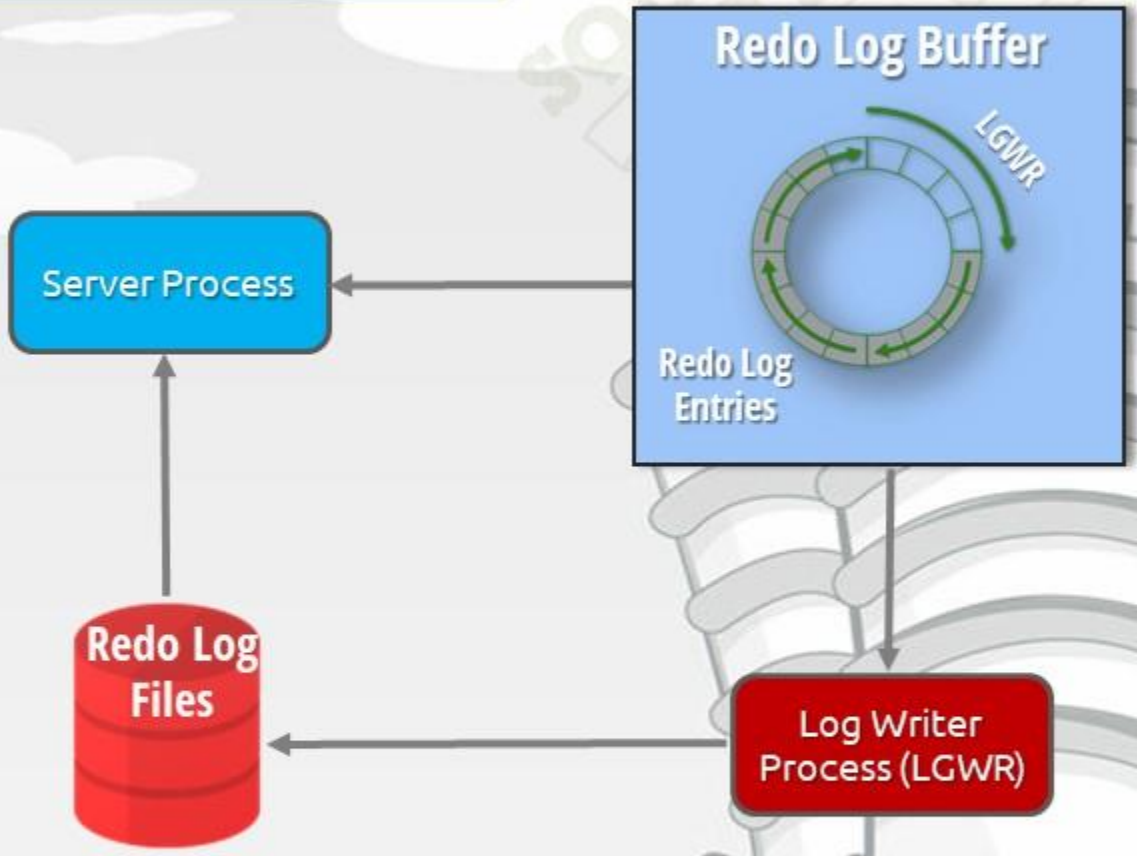
- **Largest memory area of SGA.**
- **Stores the copies of the data blocks read from the disc**
- **Why to read into the buffer cache?**
  - Much faster than discs.
- **Maintained with a complex algorithm**
  - Stores the most recently used & most touched ones.
- **Database write process handles the write operations to the disc.**
- **Stores the index data, too.**



Oracle Database Architecture

WHAT IS REDO LOG BUFFER?

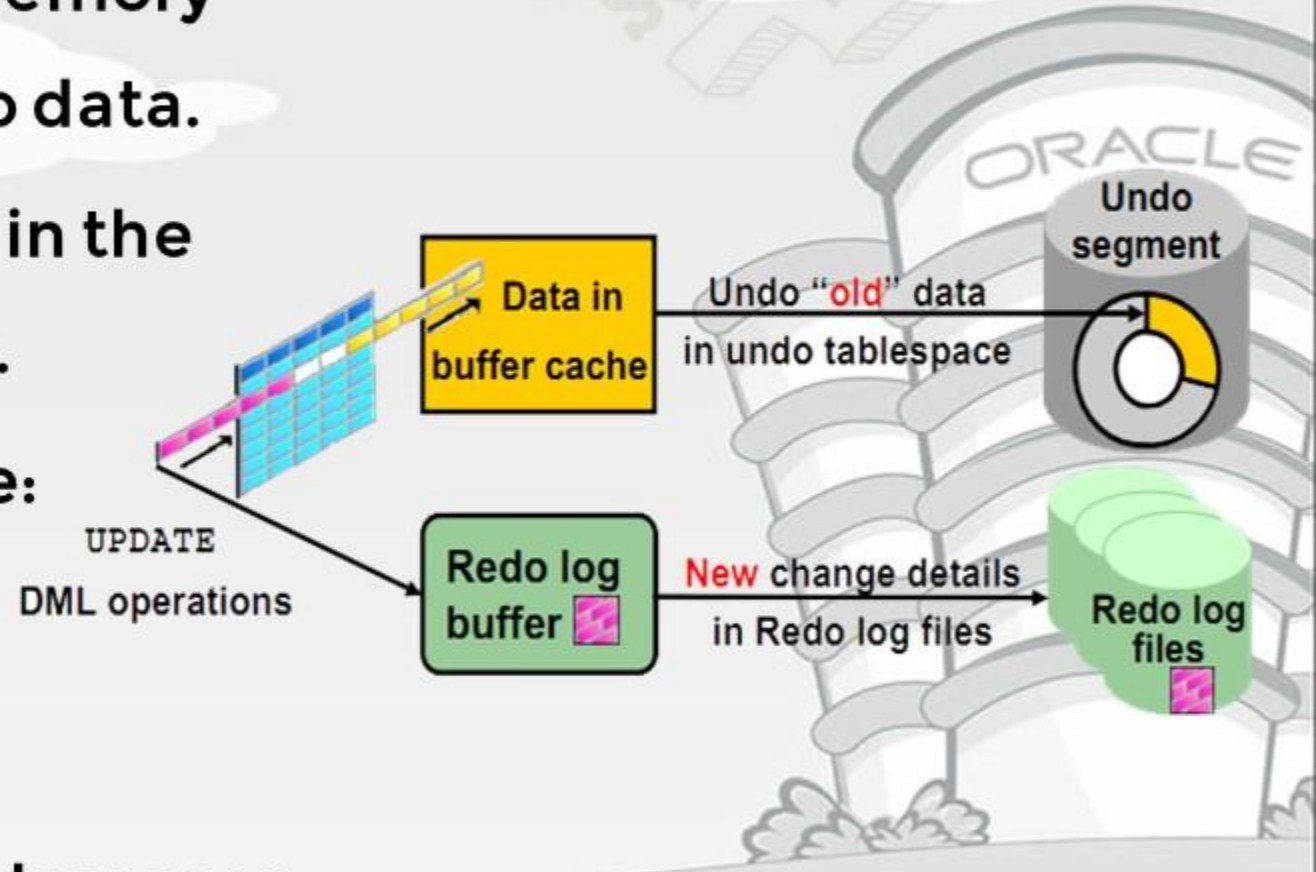
- Oracle Guarantees Not to Lose Data
- A Redo Log Entry is created when insert, update, delete, create, alter, drop occurs
- Redo Log Entries has the changes made to the database
- They are used for recovery operations
- Redo Log Entries are stored in the Redo Log Buffer
- Redo Log Buffer is a circular Buffer
- Rollback is not done with Redo Log Data



Oracle Database Architecture

WHAT IS UNDO?

- The original data stored into the memory (undo tablespace) is called as undo data.
- Another copy of the data is stored in the buffer cache for the modifications.
- Undo data is not modified because:
  - Used for **rollback** operations
  - Used for providing read consistency
  - Used for providing flashback feature
- Blocks > Extents > Segments > Tablespaces

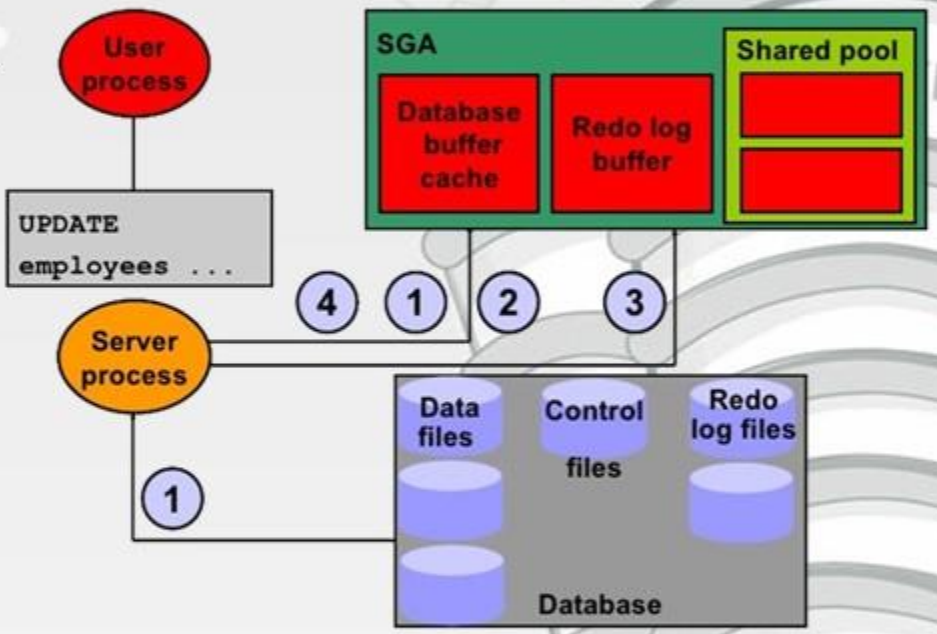


Oracle Database Architecture

HOW A DML IS PROCESSED AND COMMITTED?

When we run a DML code, the server :

- ✓ Checks the Shared SQL Area for similar statements to use.
- ✓ Checks the Data Dictionary Cache and checks if our query is valid.
- ✓ Checks Buffer Cache & Undo Segments for the related data.
- ✓ Locks the related blocks.
- ✓ Makes the change to the blocks in the Buffer Cache.
- ✓ The changes are applied to the Redo Log Buffer before the Buffer Cache.
- ✓ The server returns the feedback for the change.



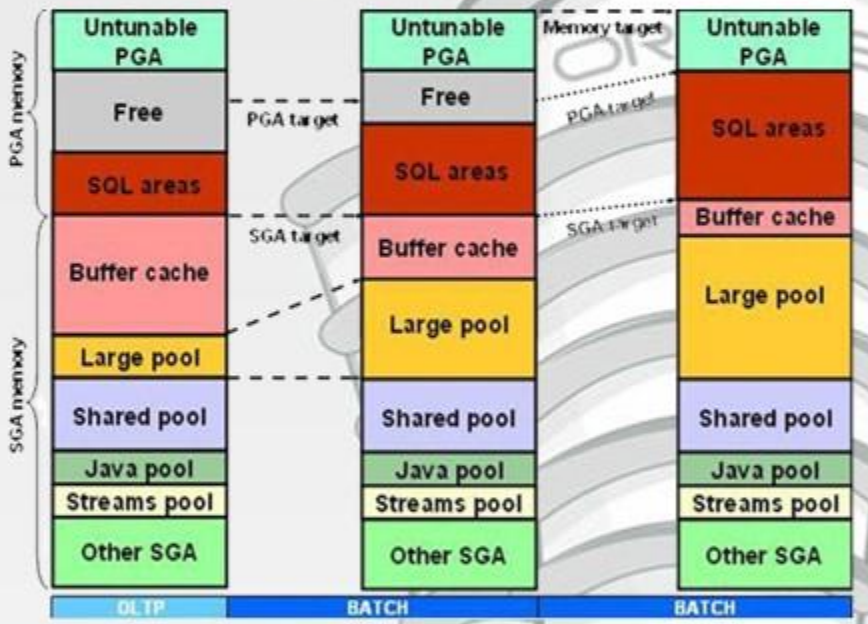
When the user commits :

- ✓ The server creates a commit record with SCN.
- ✓ The LGWr process writes redo log entries in the redo log buffer to the redo log files.
- ✓ The DBWn writes the dirty blocks to the disc & unlocks the blocks.
- ✓ The server returns a feedback about the transaction completion.

Oracle Database Architecture

AUTOMATIC MEMORY MANAGEMENT

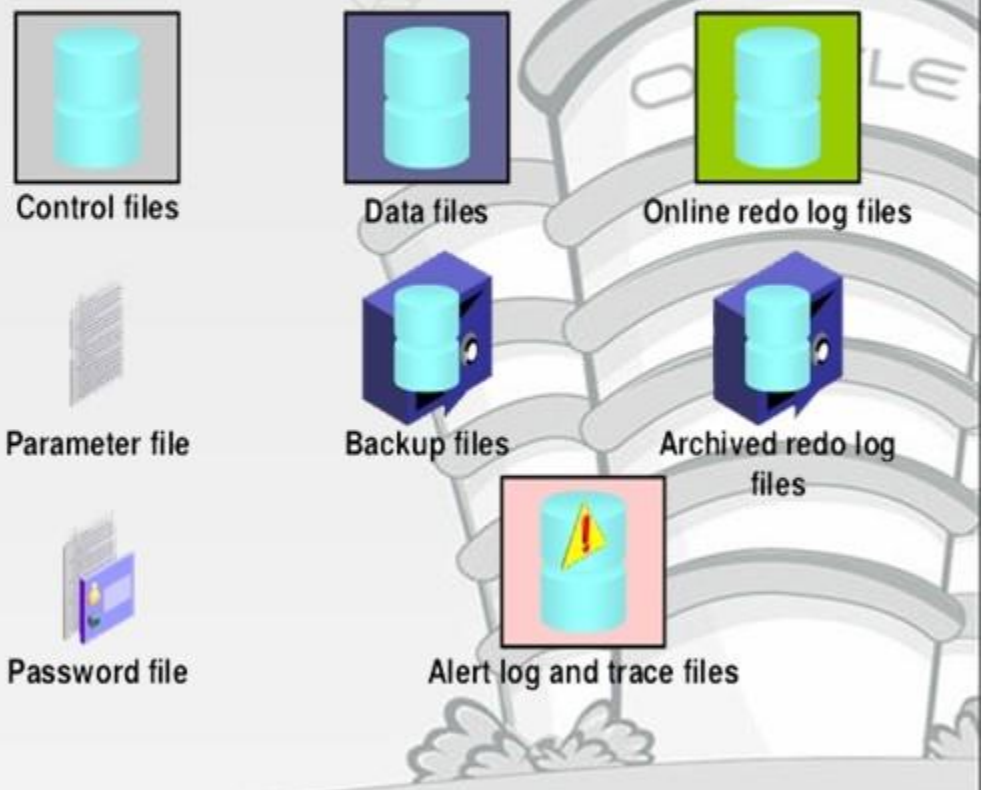
- The size of each memory area is important for the execution performance of your queries
- Oracle can manage the memory automatically
- It can manage both SGA and PGA memories
- It is recommended to leave automatic memory management enabled
- Enabling automatic memory management will prevent out of memory errors



Oracle Database Architecture

ORACLE DATABASE STORAGE ARCHITECTURE

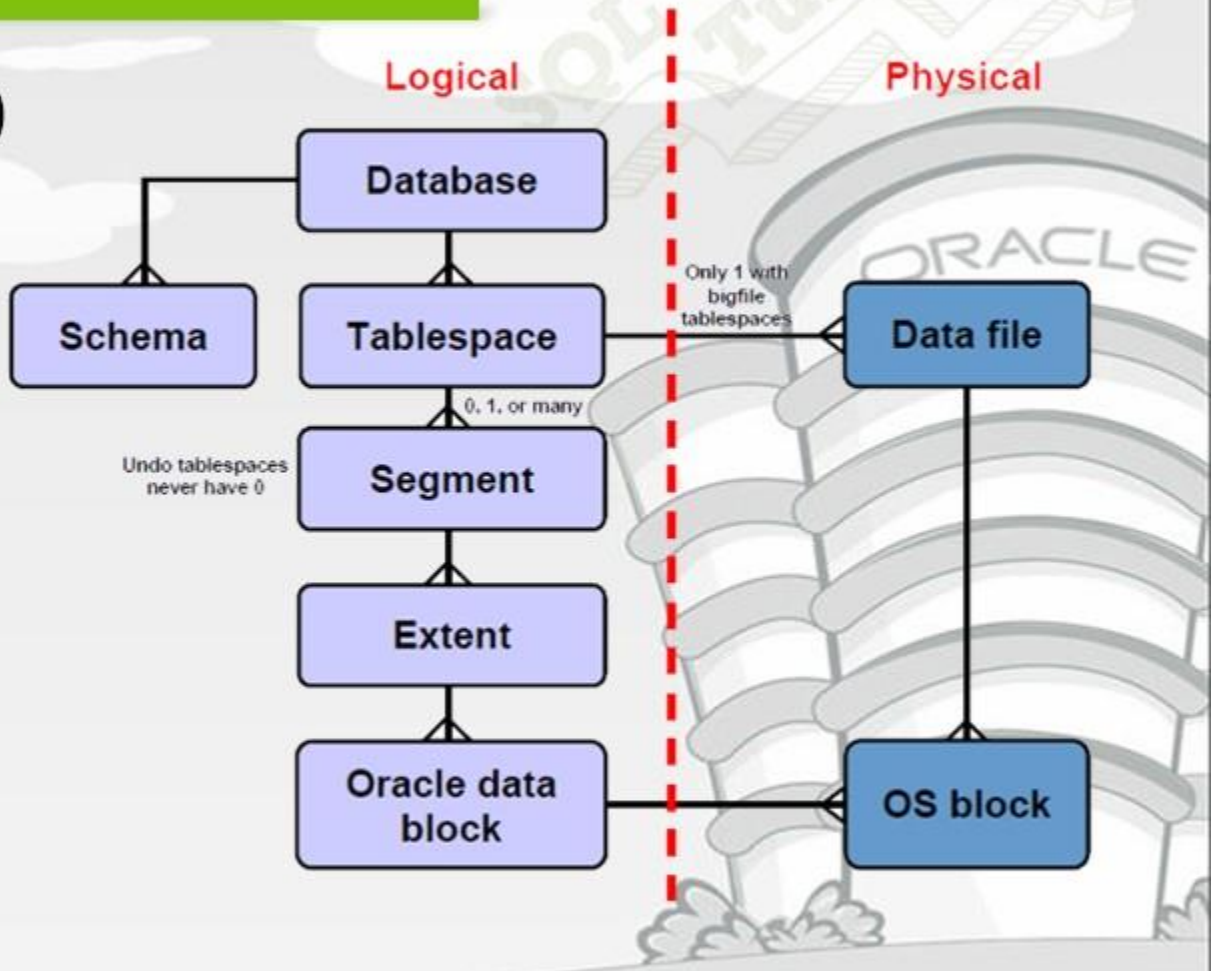
- **Storage = Discs**
- **Control Files:** Stores the pyhsical structure information of the database.
- **Data Files :** Stores Data (Tables, procedures, application data,..)
- **Online Redo Log Files :** Stores redo log entries
- **Archived Redo Log Files :** Online redo log files are constantly moved here
- **Backup Files :** Stores the exact copy of the data files for disaster recovery
- **Parameter File :** Stores the configure data of the database instance
- **Password File :** Stores the passwords of the admin users (sysdba, sysoper, sysasm)
- **Alert Log & Trace Files :** Stores log messages and errors occurred in the database.



Oracle Database Architecture

LOGICAL AND PHYSICAL DATABASE STRUCTURE

- **Blocks :** Smallest units of storage (2KB-32KB)
- **Extents :** Combination of several consecutive data blocks. Used for storing specific type of info.
- **Segments :** Combination of several extents. Used for storing some big data (tables, indexes, etc).
- **Tablespaces :** Combination of many segments. Used for grouping the related data in one container
  - **Temporary Tablespace** : Stores the temporary data of a session
  - **Permanent Tablespace** : Stores the persistent schema objects



Performance Tuning Basics

WHEN TO TUNE?

- SQL Tuning is a continuous process
- You need to tune your queries :
  - While creating
  - After the creation
- When to decide tuning?
  - By checking the top consuming queries frequently
  - After any complaints of bad performance
- The reasons of performance loss
  - Structural changes
  - Changes on the data volume
  - Application Changes
  - Aged Statistics
  - Database Upgrades
  - Database Parameter Changes
  - Operating System & Hardware Changes

Performance Tuning Basics

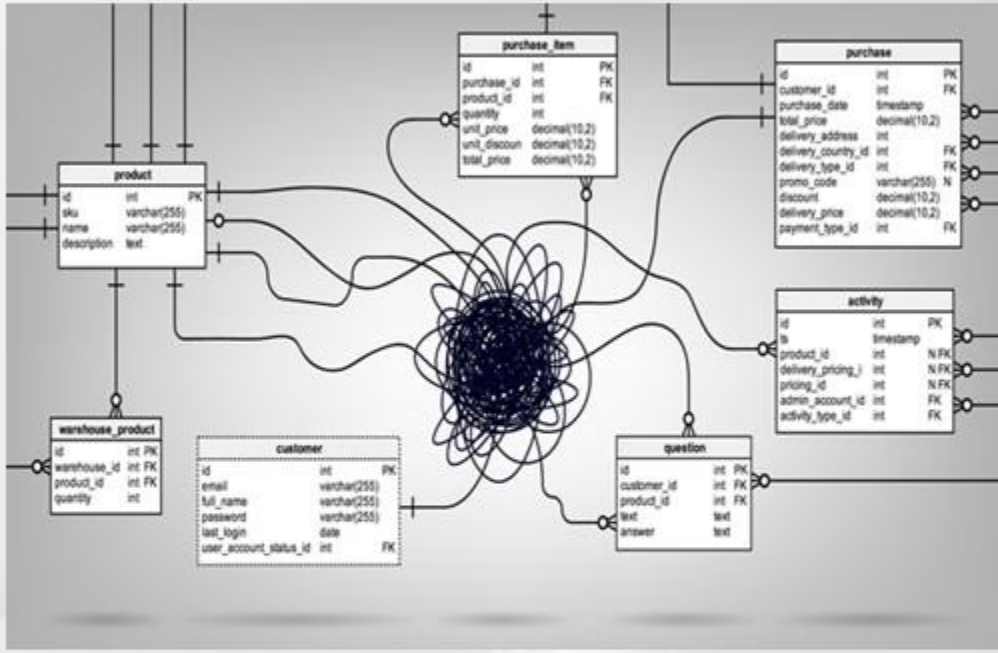
WHAT IS A BAD SQL?

- Bad SQLs use more resources than necessary
- Characteristics of a Bad SQL
  - Unnecessary parse time
  - Unnecessary I/O operations
  - Unnecessary CPU time
  - Unnecessary waits
- Time on Wait (CPU) + Time on Execution = DB TIME
- The reason of a Bad SQL:
  - Bad Design, Poor Coding, Inefficient Execution Plan

Performance Tuning Basics

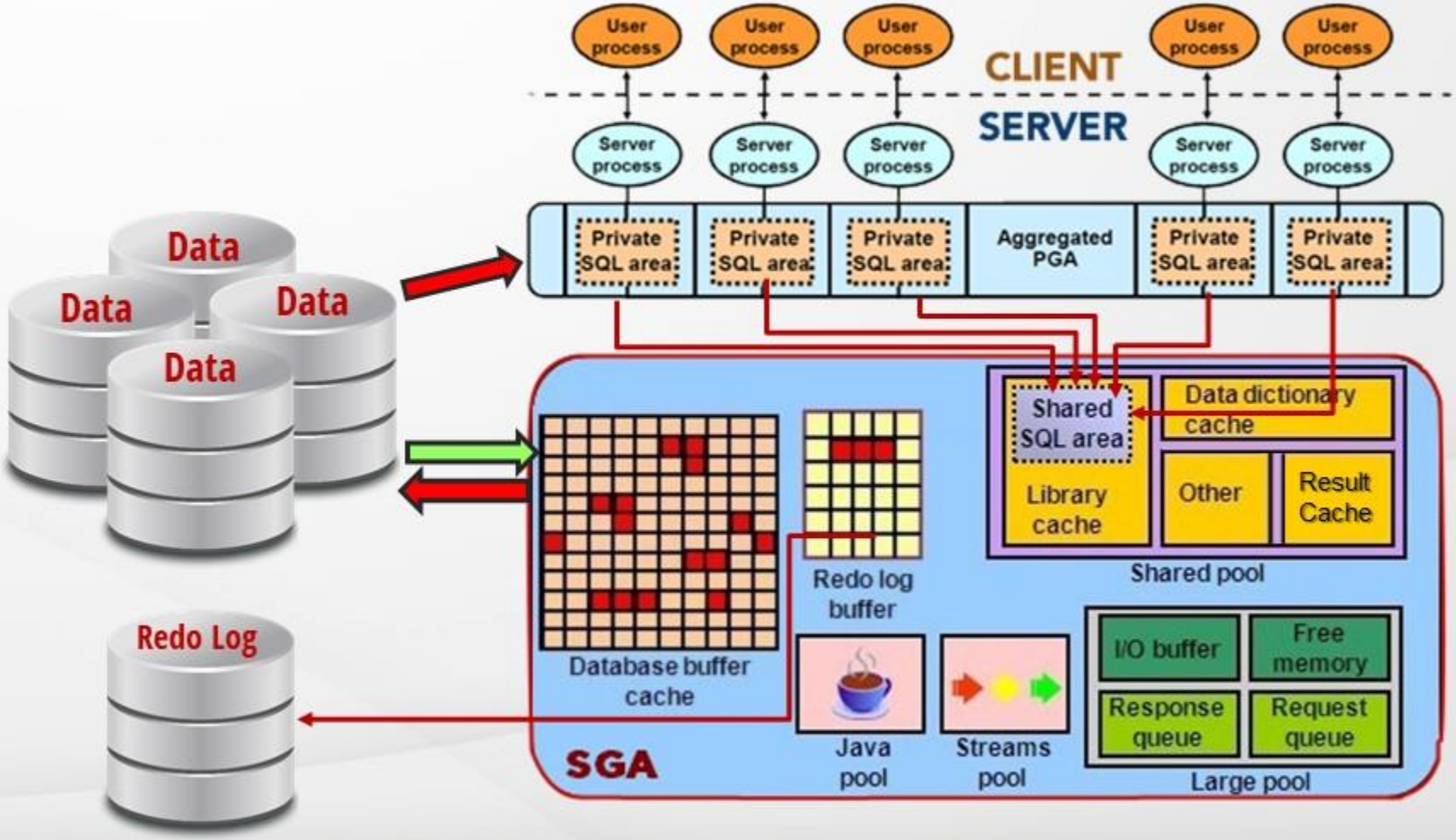
EFFECTIVE SCHEMA DESIGN

- Select the data types carefully
  - Assign data types as much as needed
  - Select exactly the same data type between parent-child keys
- Enforce data integrity
- Use normalization well
- Select right table type
- Create Clusters
- Use indexes often and select index type carefully
- Create index-organized tables (IOT)



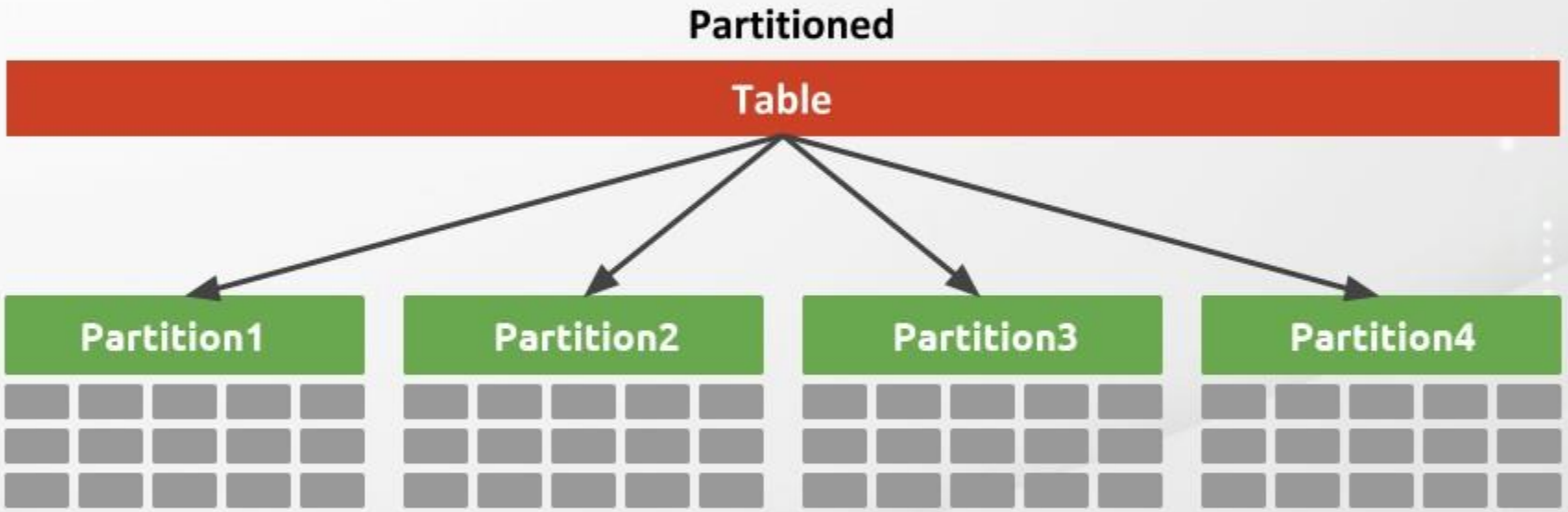
Performance Tuning Basics

TABLE PARTITIONING



Performance Tuning Basics

TABLE PARTITIONING



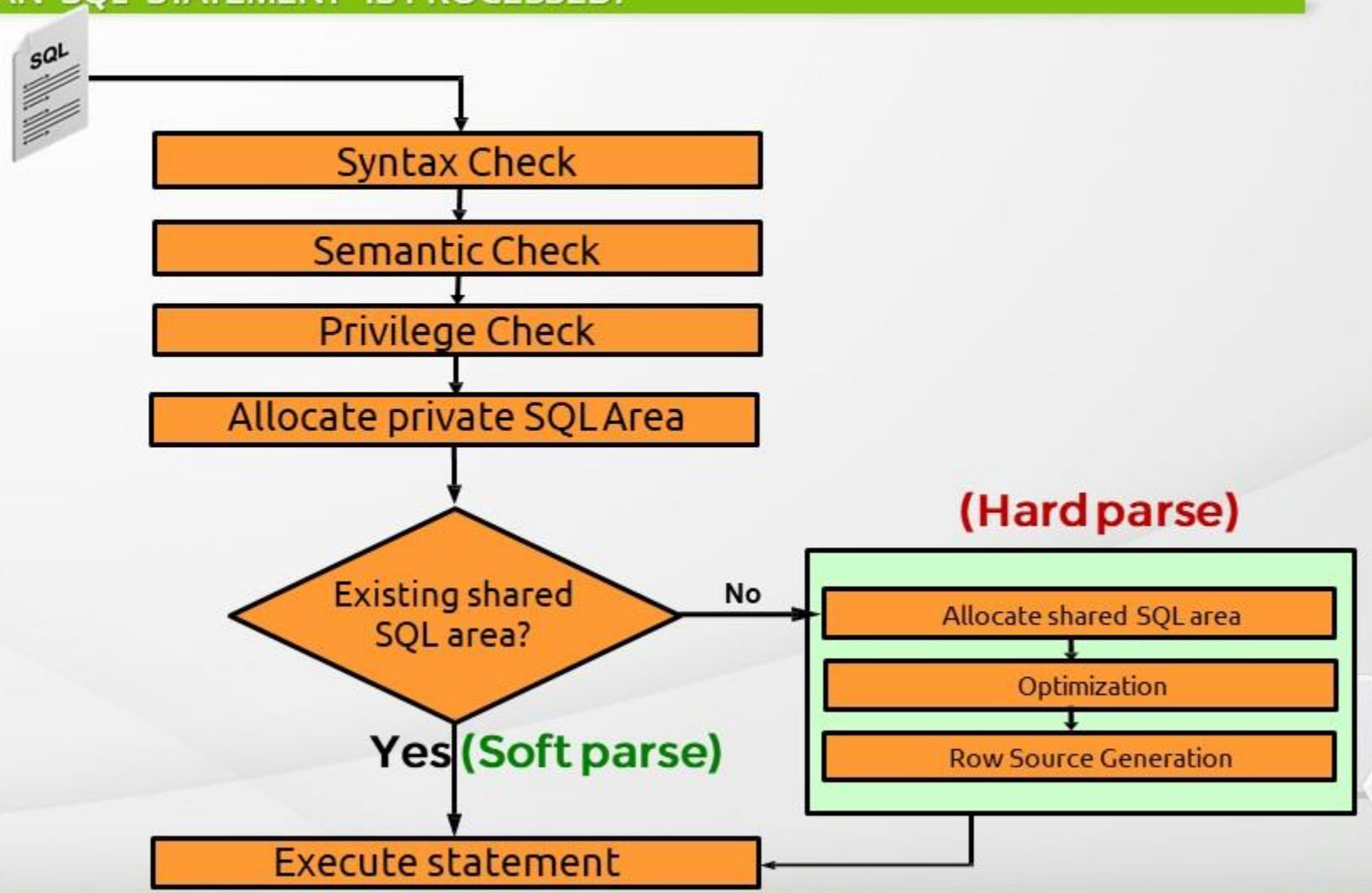
Performance Tuning Basics

TABLE PARTITIONING

```
ALTER TABLE employees MODIFY
  PARTITION BY RANGE (hire_date)
  ( PARTITION P_NAME1 VALUES LESS THAN (TO_DATE('01/01/2018', 'DD/MM/YYYY')) TABLESPACE USERS,
    PARTITION P_NAME2 VALUES LESS THAN (TO_DATE('01/01/2019', 'DD/MM/YYYY')) TABLESPACE USERS,
    PARTITION P_NAME3 VALUES LESS THAN (TO_DATE('01/01/2020', 'DD/MM/YYYY')) TABLESPACE USERS,
    PARTITION P_NAME3 VALUES LESS THAN (MAXVALUE) TABLESPACE USERS,
   ) ONLINE
  UPDATE INDEXES
 ( IDX1_SALARY LOCAL,
   IDX2_EMP_ID GLOBAL PARTITION BY RANGE (employee_id)
  ( PARTITION IP1 VALUES LESS THAN (MAXVALUE))
 );
```

Performance Tuning Basics

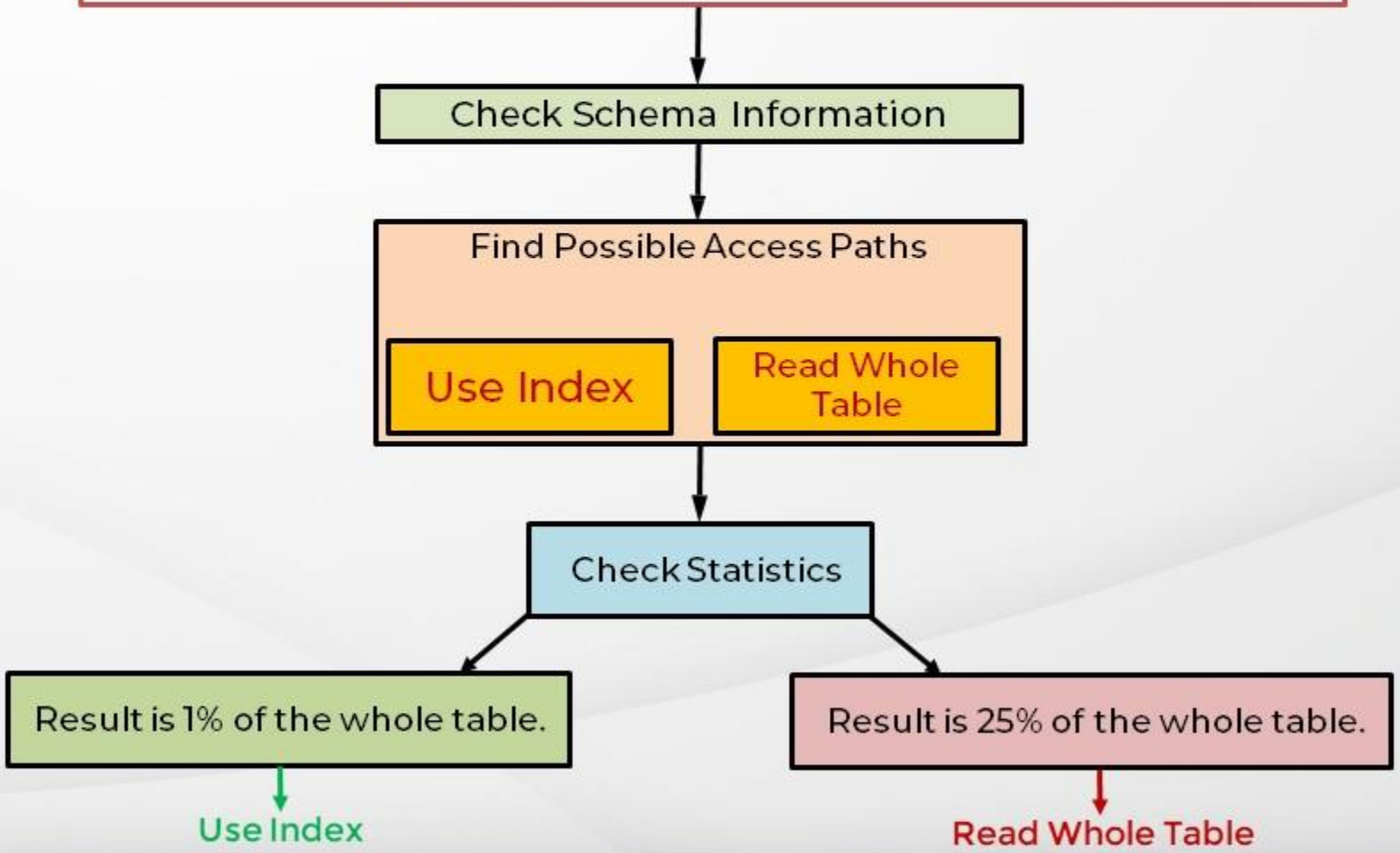
HOW AN SQL STATEMENT IS PROCESSED?



Performance Tuning Basics

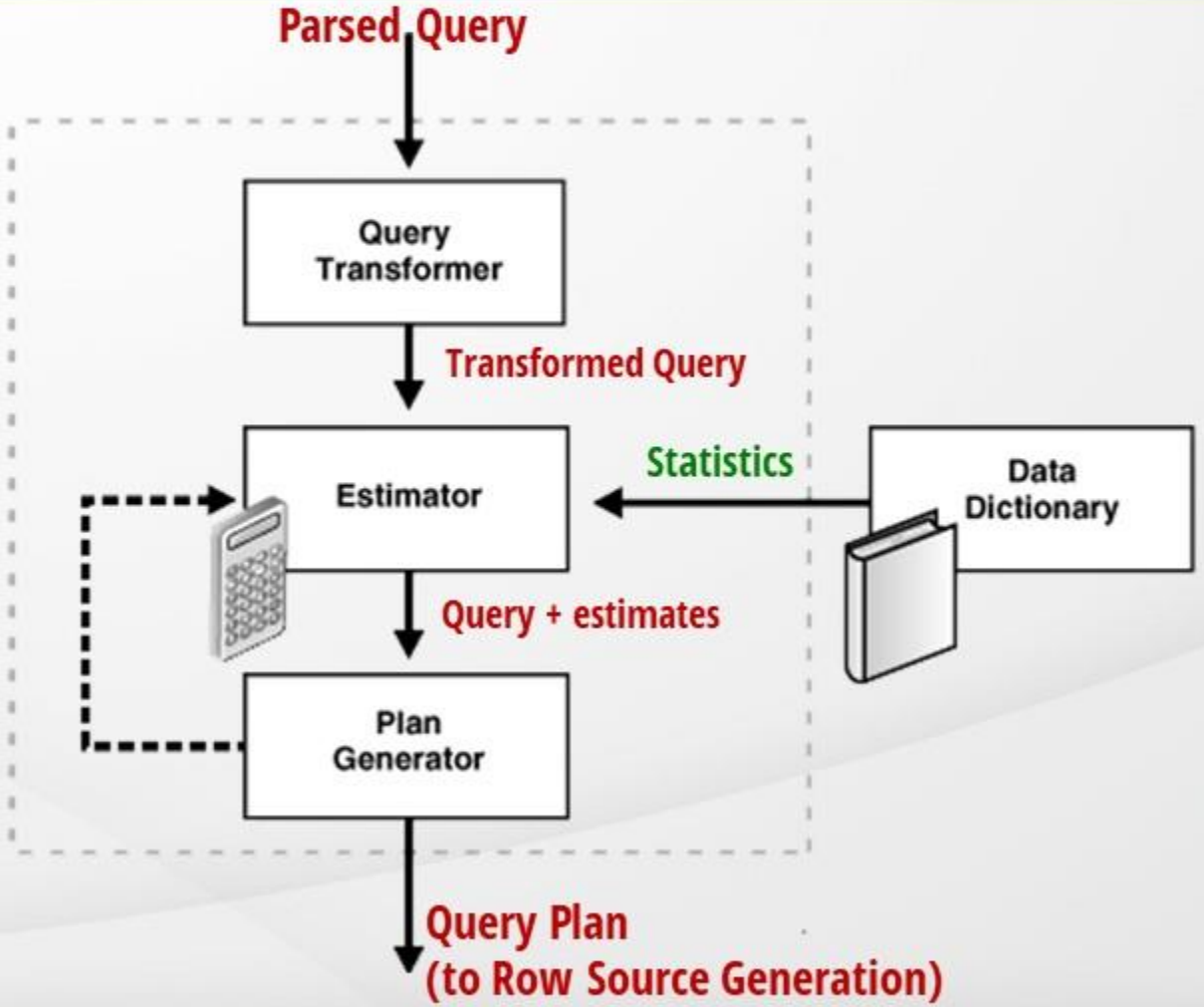
WHY DO WE NEED AN OPTIMIZER?

```
SELECT * FROM products WHERE prod_category = 'Electronics';
```



Performance Tuning Basics

OPTIMIZER OVERVIEW



Performance Tuning Basics

QUERY TRANSFORMER

- Query transformer transforms the query into a better performing one but semantically equivalent of it
- If the transform isn't better, it uses the original one
- Time restriction and old statistics may lead a wrong plan creation
- OR Expansion
  - Using OR in the where clause will prevent index usages

```
SELECT * FROM sales WHERE prod_id = 14 or promo_id = 33;
```

```
SELECT * FROM sales WHERE prod_id = 14
UNION ALL
SELECT * FROM sales WHERE promo_id = 33 and prod_id <> 14;
```

- Subquery Unnesting
  - The optimizer transforms a nested query into a join statement

```
SELECT * FROM sales WHERE cust_id IN
                  (SELECT cust_id FROM customers);
```

```
SELECT sales.*
FROM sales , customers
WHERE sales.cust_id = customers.cust_id;
```

Performance Tuning Basics

SELECTIVITY & CARDINALITY

```
SELECT * FROM sales WHERE promo_id = 999;
```

- Sales Table has **918.843** rows
- The Result has **887.837** rows

```
SELECT * FROM sales WHERE promo_id = 33;
```

- The Result has **2074** rows

$$\text{Selectivity} = \frac{\text{Number of rows returning from the query}}{\text{Total number of rows}}$$

$$\text{Cardinality} = \text{Total number of rows} \times \text{Selectivity}$$

- **Why Selectivity and Cardinality important?**
  - Selectivity affects the estimates in I/O cost
  - Selectivity affects the sort cost
  - Cardinality is used to determine join, sort and filter costs
  - Incorrect selectivity and cardinality = incorrect plan cost estimation

```
SELECT * FROM sales WHERE promo_id = 999;
```

- promo_id column has **4** distinct values.

```
SELECT num_distinct FROM dba_tab_columns
WHERE table_name = 'SALES';
```

```
SELECT * FROM sales WHERE cust_id = 100001;
```

- **7059** distinct values.
- Selectivity : $\frac{1}{7059}$

```
SELECT * FROM sales WHERE cust_id = 100001
AND promo_id = 999 AND channel_id = 9;
```

- Selectivity = $\frac{1}{4} \times \frac{1}{7059} \times \frac{1}{4}$

Performance Tuning Basics

WHAT IS COST IN DETAILS?

- Cost is the optimizer's best estimate of the number of I/Os to execute a statement.
- To estimate the cost, the estimator uses;
  - Disk I/O,
  - CPU usage,
  - Memory usage.

$$\text{Cost} = \frac{\text{Single-block I/O Cost} + \text{Multiblock I/O Cost} + \text{CPU Cost}}{\text{Single-block read time}}$$

Single-block I/O Cost: Number of single-block reads x Single-block read time

Multiblock I/O Cost: Number of multiblock reads x Multiblock read time

CPU Cost: Number of CPU cycles / CPU Speed

Performance Tuning Basics

PLAN GENERATOR

```
SELECT e.first_name, e.last_name, e.department_name from FROM employees e, departments d
    WHERE e.department_id = d.department_id;
```

```
Join order[1]: DEPARTMENTS[D]#0 EMPLOYEES[E]#1
NL Join Cost: 42.25
SM Join cost: 8.48
HA Join cost: 5.20
Best:: JoinMethod: Hash
Cost: 5.2
Join order[2]: EMPLOYEES[E]#1 DEPARTMENTS[D]#0
NL Join Join: 88.28
SM Join cost: 7.57
HA Join cost: 5.50
Join order aborted
Final cost for query block SEL$1 (#0)
All Rows Plan:
Best join order: 1
+-----------------------------------------------------------------+
| Id  | Operation           | Name        |  Rows  | Bytes | Cost |
+-----------------------------------------------------------------+
| 0   | SELECT STATEMENT    |             |        |       |    5 |
| 1   |  HASH JOIN          |             |   106  |  6042 |    5 |
| 2   |   TABLE ACCESS FULL | DEPARTMENTS|    27  |   810 |    3 |
| 3   |   TABLE ACCESS FULL | EMPLOYEES   |   107  |  2889 |    3 |
+-----------------------------------------------------------------+
```

Performance Tuning Basics

ROW SOURCE GENERATOR

- Once the plan generator generates the optimum plan, it handles that to the row source generator
- Row source generator generates an iterative execution plan usable for the database
- Row source is an area that we get the row set (Table, view, result of join or groups)
- Row source generator produces a row source tree (A collection of row sources)
- Row source tree shows the following information :
  - Execution order
  - Access methods
  - Join methods
  - Data operations (filter, sort, ..)

Performance Tuning Basics

ROW SOURCE GENERATOR

```
SELECT prod_name, time_id, max(amount_sold) FROM sales, products
WHERE sales.prod_id = products.prod_id AND promo_id = 33
GROUP BY prod_name,time_id;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST	PARTITION_START	PARTITION_STOP
SELECT STATEMENT		2074	290		
HASH (GROUP BY)		2074	290		
HASH JOIN		2074	289		
Access Predicates					
SALES.PROD_ID=PRODUCTS.PROD_ID					
NESTED LOOPS		2074	289		
NESTED LOOPS					
STATISTICS COLLECTOR					
TABLE ACCESS (FULL)	PRODUCTS	72	3		
PARTITION RANGE (ALL)				1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP AND					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					
BITMAP INDEX (SINGLE VALUE)	SALES_PROD_BIX			1	28
Access Predicates					
SALES.PROD_ID=PRODUCTS.					
TABLE ACCESS (BY LOCAL INDEX ROWID)	SALES	29	286	1	1
PARTITION RANGE (ALL)		2074	286	1	28
TABLE ACCESS (BY LOCAL INDEX ROWID BATCHED)	SALES	2074	286	1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					

Performance Tuning Basics

SQL TUNING PRINCIPLES & STRATEGIES

SQL Tuning Principles

- ✓ Identifying the issue
- ✓ Clarify the details of that issue
- ✓ Collecting data
- ✓ Analyze the data
- ✓ Choose an appropriate tuning strategy

SQL Tuning Strategies

- ✓ Parse time reduction
- ✓ Plan comparison strategy
- ✓ Quick solution strategy
- ✓ Finding & implementing a good plan
- ✓ Query analysis strategy

Performance Tuning Basics

QUERY ANALYSIS STRATEGY

We use Query Analysis Strategy when:

- ✓ Quick tuning strategies did not work, and we have time to work on this problem
- ✓ Query can be modified
- ✓ Determine the underlying cause

What to do on this Strategy?

- ✓ Statistics and Parameters
- ✓ Query Structure
- ✓ Access Paths
- ✓ Join Orders & Join Methods
- ✓ Others

Performance Tuning Basics

WHAT IS THE STRATEGY?

- Collecting Data
  - ✓ Execution Plan
  - ✓ Information of objects in the query
  - ✓ Statistics
  - ✓ Histograms
  - ✓ Parameter Settings
  - ✓ The available tools are (SQLT, DBMS_STATS, TKPROF, AWR Report, etc)
- Pre-Analyze of the Query
  - ✓ Check the volume of resulting data
  - ✓ Check the predicates
  - ✓ Check the problematic constructs
- Analyzing the Execution Plan
  - ✓ Tools to get the execution plan (SQL Trace, TKPROF, V$_SQL_PLAN, DBMS_MONITOR, AWRSQRPT.SQL, etc)
  - ✓ How to read the execution plan :
    - ✓ Check the access paths
    - ✓ Check the join order and the join type
    - ✓ Compare actual & estimated number of rows
    - ✓ Find the steps where cost and logical reads are different significantly
- Analyzing by considering the query tuning techniques
- Find a possible solution
  - Updating statistics
  - Using dynamic statistics
  - Creating or re-creating an index
  - Creating index-organized tables (IOT)
  - Using Hints
  - Others

Execution Plan & Statistics

EXECUTION PLAN & EXPLAIN PLAN IN DETIALS

OPERATION	OBJECT_NAME	CARDINALITY	COST	PARTITION_START	PARTITION_STOP
SELECT STATEMENT		2074	290		
HASH (GROUP BY)		2074	290		
HASH JOIN		2074	289		
Access Predicates					
SALES.PROD_ID=PRODUCTS.PROD_ID					
NESTED LOOPS		2074	289		
NESTED LOOPS					
STATISTICS COLLECTOR					
TABLE ACCESS (FULL)	PRODUCTS	72	3		
PARTITION RANGE (ALL)				1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP AND					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					
BITMAP INDEX (SINGLE VALUE)	SALES_PROD_BIX			1	28
Access Predicates					
SALES.PROD_ID=PRODUCTS.					
TABLE ACCESS (BY LOCAL INDEX ROWID)	SALES	29	286	1	1
PARTITION RANGE (ALL)		2074	286	1	28
TABLE ACCESS (BY LOCAL INDEX ROWID BATCHED)	SALES	2074	286	1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					

Execution Plan & Statistics

GENERATING THE STATISTICS

- Statistic Types
  - System Statistics
  - Optimizer Statistics
- System Statistics
  - Used by the Optimizer to estimate I/O and CPU costs
  - Should be generated regularly
  - Should be gathered during a normal workload

```
EXEC dbms_stats.gather_system_stats('Start');
```

```
SELECT * FROM sys.aux_stats$;
```

- Optimizer Statistics
  - Can be gathered manually or automatically

```
ANALYZE TABLE <table_name> COMPUTE STATISTICS;
```

  - GATHER_DATABASE_STATS Procedure

```
EXEC dbms_stats.gather_database_stats;
```

  - GATHER_DICTIONARY_STATS Procedure

```
EXEC dbms_stats.gather_dictionary_stats;
```

  - GATHER_SCHEMA_STATS Procedure

```
EXEC dbms_stats.gather_schema_stats(ownname=>'SH');
```

  - GATHER_TABLE_STATS Procedure

```
EXEC dbms_stats.gather_table_stats(ownname=>'SH',
                tabname=>'SALES', cascade=>true);
```

  - GATHER_INDEX_STATS Procedure

Execution Plan & Statistics

GENERATING THE STATISTICS

How can we see the Optimizer Statistics?

- DBA_TABLES
- DBA_TAB_STATISTICS
- DBA_TAB_COL_STATISTICS
- DBA_INDEXES
- DBA_CLUSTERS
- DBA_TAB_PARTITIONS
- DBA_IND_PARTITIONS
- DBA_PART_COL_STATISTICS

Execution Plan & Statistics

GENERATING EXECUTION PLANS

- To analyze an execution plan :
  - Explain Plan
  - Autotrace
  - V$SQL_PLAN
- EXPLAIN PLAN

```
EXPLAIN PLAN FOR <QUERY>;
```

  - Generates the explain plan and saves into plan_table

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
EXPLAIN PLAN SET statement_id = 'MyID' FOR SELECT FROM EMPLOYEES where employee_id = 100;
```

```
EXPLAIN PLAN SET statement_id = 'MyID' INTO MyPlanTable FOR SELECT FROM EMPLOYEES where employee_id = 100;
```

Execution Plan & Statistics

AUTOTRACE

- Autotrace traces our query and produces the execution plan and the statistics

```
SET AUTOTRACE ON;
```

```
SET AUTOTRACE ON [EXPLAIN|STATISTICS];
```

```
SET AUTOTRACE TRACE[ONLY] ON [EXPLAIN|STATISTICS];
```

```
SET AUTOTRACE OFF;
```

- Autotrace uses plan_table like the explain plan.

Execution Plan & Statistics

V$SQL_PLAN VIEW

- There are a lot of performance views that can be used for tuning
  - V$SQLAREA
  - V$SQL_WORKAREA
  - V$SQL
  - V$SQL_PLAN
  - V$SQL_PLAN_STATISTICS
  - V$SQL_PLAN_STATISTICS_ALL
- V$SQL_PLAN
  - Actual execution plans are stored here
  - It is very similar to plan_table
  - It is connected to V$SQL view

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR('5d4xt4hva9h94'));
```

Execution Plan & Statistics

Analyzing the Execution Plans

- **Where to look?**
  - Cost
  - Access Methods
  - Cardinality
  - Join Methods & Join Types
  - Partition Pruning
  - Others

```
SELECT p.prod_id,p.prod_name, s.amount_sold, s.quantity_sold
FROM sales s, products p, customers c
 WHERE s.prod_id = p.prod_id
 AND s.cust_id = c.cust_id
 AND s.cust_id IN (2,3,4,5);
```

```
| Id  | Operation                                | Name          | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |               |   521 | 27092 |   211   (0)| 00:00:01 |       |       |
|   1 |  NESTED LOOPS                            |               |   521 | 27092 |   211   (0)| 00:00:01 |       |       |
|*  2 |   HASH JOIN                              |               |   521 | 24487 |   211   (0)| 00:00:01 |       |       |
|   3 |    TABLE ACCESS FULL                     | PRODUCTS      |    72 |  2160 |     3   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE ALL                   |               |   521 |  8857 |   208   (0)| 00:00:01 |     1 |    28 |
|   5 |     INLIST ITERATOR                      |               |       |       |            |          |       |       |
|   6 |      TABLE ACCESS BY LOCAL INDEX ROWID BATCHED| SALES    |   521 |  8857 |   208   (0)| 00:00:01 |     1 |    28 |
|   7 |       BITMAP CONVERSION TO ROWIDS        |               |       |       |            |          |       |       |
|*  8 |        BITMAP INDEX SINGLE VALUE         | SALES CUST BIX|       |       |            |          |     1 |    28 |
|*  9 |   INDEX UNIQUE SCAN                      | CUSTOMERS PK  |     1 |     5 |     0   (0)| 00:00:01 |       |       |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("S"."PROD ID"="P"."PROD ID")
   8 - access("S"."CUST ID"=2 OR "S"."CUST ID"=3 OR "S"."CUST ID"=4 OR "S"."CUST ID"=5)
   9 - access("S"."CUST ID"="C"."CUST ID")
       filter("C"."CUST ID"=2 OR "C"."CUST ID"=3 OR "C"."CUST ID"=4 OR "C"."CUST ID"=5)
```

Table & Index Access Paths

What are Indexes & How They Work in Details

- Types of Indexes
  - ✓ B-TREE Indexes
    - Normal Index
    - Function-Based Index
    - Index-Organized Table (IOT)
  - ✓ BITMAP Indexes
- Attributes of Indexes
  - Key Compression
  - Reverse Key
  - Ascending – Descending Ordered Indexes

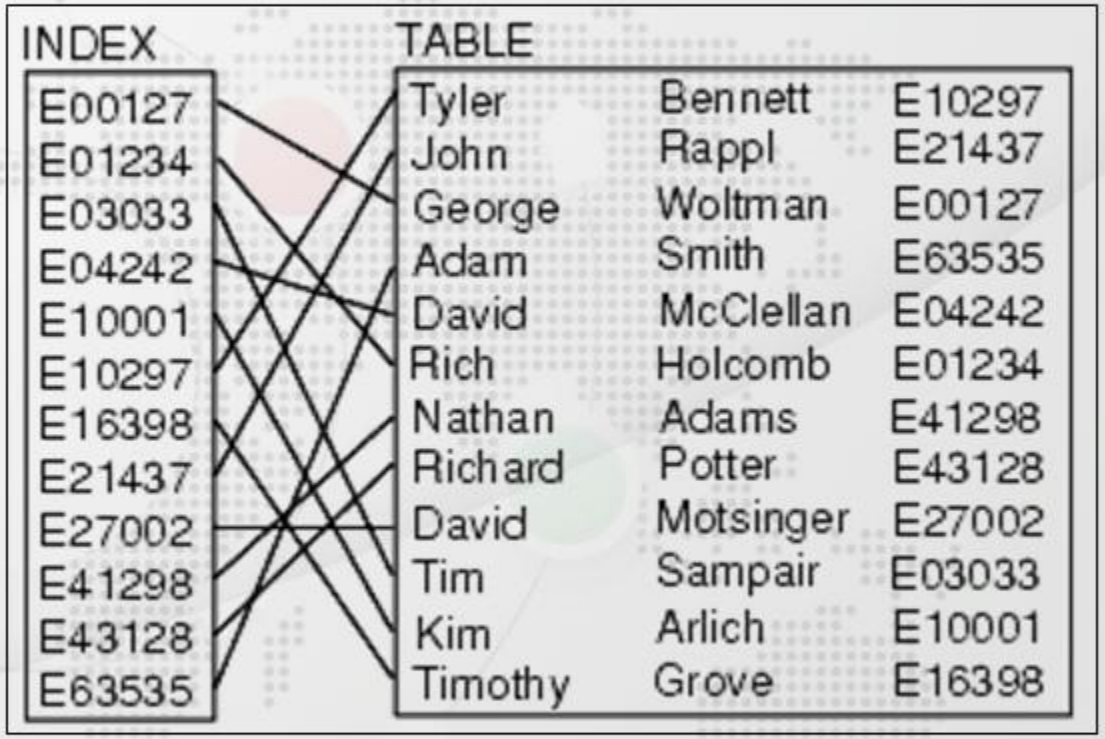


Table & Index Access Paths

What are Indexes & How They Work in Details

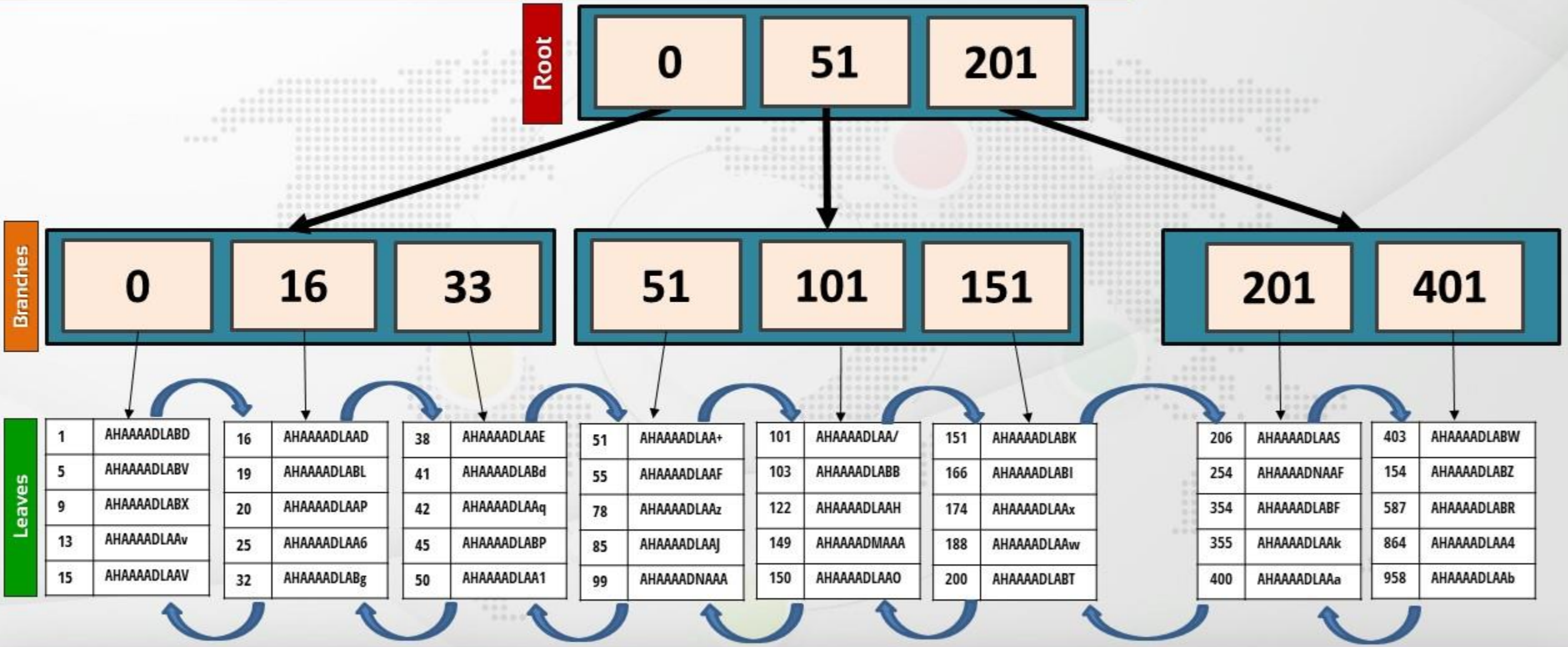
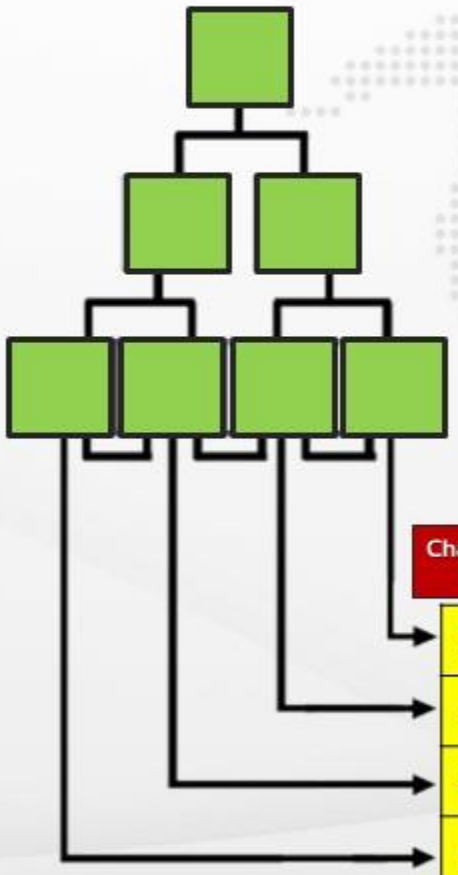


Table & Index Access Paths

What are Indexes & How They Work in Details



Sales Table

ROWID	CUST_ID	PROD_ID	CHANNEL_ID
AAAYtAAAHAAAAKrAED	10007	14	2
AAAYtAAAHAAAAKrAEE	10007	14	3
AAAYtAAAHAAAAKrAEF	10007	14	4
AAAYtAAAHAAAAKrAEG	965	14	4
AAAYtAAAHAAAAKrAEH	10667	14	2
AAAYtAAAHAAAAKrAEI	3047	14	3
AAAYtAAAHAAAAKrAEJ	8318	14	3
AAAYtAAAHAAAAKrAEK	9882	14	3
AAAYtAAAHAAAAKrAEL	10667	14	3
AAAYtAAAHAAAAKrAEM	19683	14	3
AAAYtAAAHAAAAKrAEN	22478	14	3
AAAYtAAAHAAAAKrAEO	3047	14	4
AAAYtAAAHAAAAKrAEP	8318	14	4
AAAYtAAAHAAAAKrAEQ	22478	14	4
AAAYtAAAHAAAAKrAER	5028	14	3
AAAYtAAAHAAAAKrAES	5028	14	4
AAAYtAAAHAAAAKrAET	1385	14	2
AAAYtAAAHAAAAKrAEU	2622	14	3
AAAYtAAAHAAAAKrAEV	27197	14	2
AAAYtAAAHAAAAKrAEW	387	14	3
AAAYtAAAHAAAAKrAEX	2073	14	3
AAAYtAAAHAAAAKrAEY	3947	14	3
AAAYtAAAHAAAAKrAEZ	5847	14	3
AAAYtAAAHAAAAKrAEa	6543	14	3
AAAYtAAAHAAAAKrAEb	10844	14	3
AAAYtAAAHAAAAKrAEc	12605	14	3
AAAYtAAAHAAAAKrAEd	27197	14	3
AAAYtAAAHAAAAKrAEe	3618	14	2
AAAYtAAAHAAAAKrAEf	14	14	3

Channel ID

	START ROWID	END ROWID	BITMAP
2	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0
3	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	0 1 0 0 0 1 1 1 1 1 1 0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 0 1
4	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
9	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Table & Index Access Paths

Types of Table & Index Access Paths

- Table Access Paths
  - ✓ Table Access Full
  - ✓ Table Access by ROWID
  - ✓ Sample Table Scan

CHANNEL_ID	PROD_ID	CUST_ID	ROWID
2	14	10007	AAAYtAAAHAAAAKrAED
3	14	10007	AAAYtAAAHAAAAKrAEE
4	14	10007	AAAYtAAAHAAAAKrAEF
4	14	965	AAAYtAAAHAAAAKrAEG
2	14	10667	AAAYtAAAHAAAAKrAEH
3	14	3047	AAAYtAAAHAAAAKrAEI
3	14	8318	AAAYtAAAHAAAAKrAEJ
3	14	9882	AAAYtAAAHAAAAKrAEK
3	14	10667	AAAYtAAAHAAAAKrAEL
3	14	19683	AAAYtAAAHAAAAKrAEM
3	14	22478	AAAYtAAAHAAAAKrAEN
4	14	3047	AAAYtAAAHAAAAKrAEO
4	14	8318	AAAYtAAAHAAAAKrAEP
4	14	22478	AAAYtAAAHAAAAKrAEQ
3	14	5028	AAAYtAAAHAAAAKrAER
4	14	5028	AAAYtAAAHAAAAKrAES
2	14	1385	AAAYtAAAHAAAAKrAET
3	14	2622	AAAYtAAAHAAAAKrAEU
2	14	27197	AAAYtAAAHAAAAKrAEV
3	14	387	AAAYtAAAHAAAAKrAEW
3	14	2073	AAAYtAAAHAAAAKrAEX
3	14	3947	AAAYtAAAHAAAAKrAEY
3	14	5847	AAAYtAAAHAAAAKrAEZ
3	14	6543	AAAYtAAAHAAAAKrAEa
3	14	10844	AAAYtAAAHAAAAKrAEb
3	14	12605	AAAYtAAAHAAAAKrAEc
3	14	27197	AAAYtAAAHAAAAKrAEd
2	14	3618	AAAYtAAAHAAAAKrAEe
3	14	14	AAAYtAAAHAAAAKrAEf

- Index Access Paths
  - ✓ Index Unique Scan
  - ✓ Index Range Scan
  - ✓ Index Full Scan
  - ✓ Index Fast Full Scan
  - ✓ Index Skip Scan
  - ✓ Index Join Scan
  - ✓ Index Organized Tables
  - ✓ Bitmap Access Paths

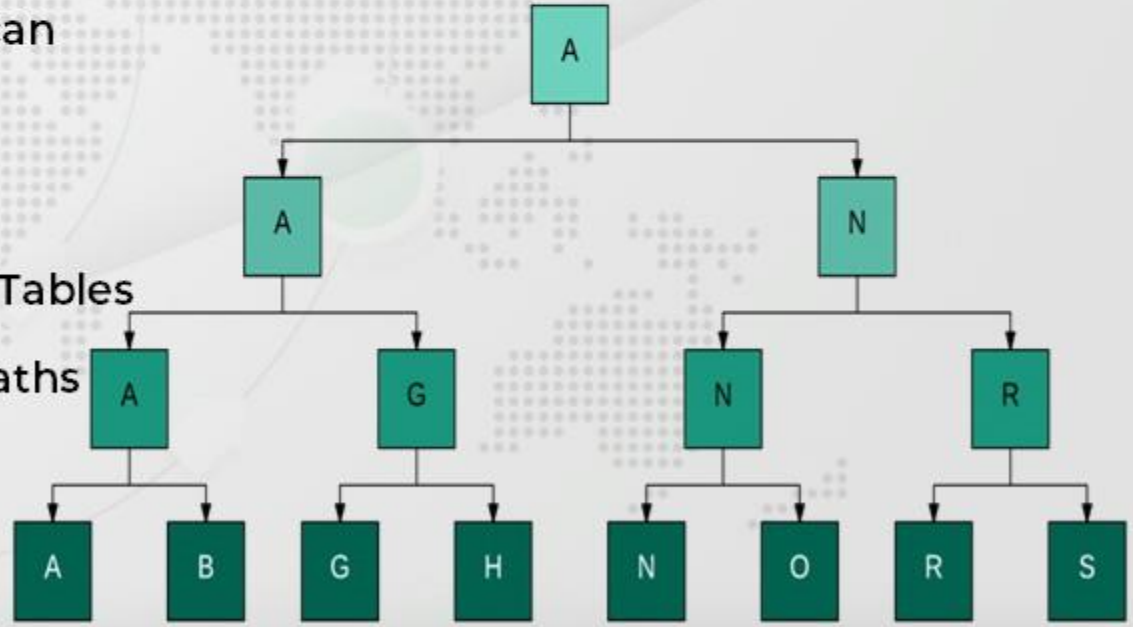
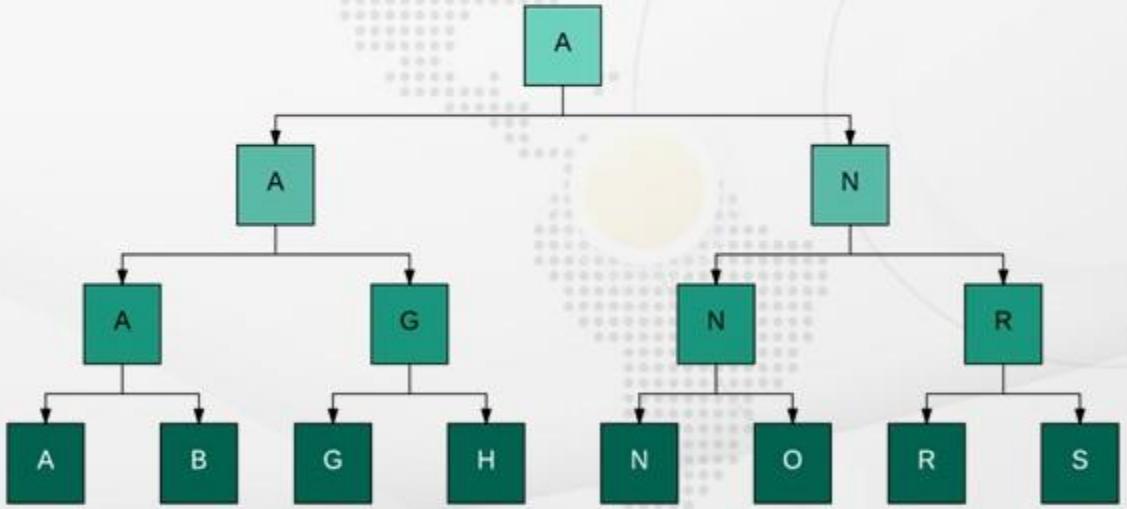


Table & Index Access Paths

Table Access By ROWID

- Table Access by ROWID occurs when :
  - ✓ ROWID is used in the where clause directly
  - ✓ By an Index Scan operation



CHANNEL_ID	PROD_ID	CUST_ID	ROWID
2	14	10007	AAAYtAAAHAAAAKrAED
3	14	10007	AAAYtAAAHAAAAKrAEE
4	14	10007	AAAYtAAAHAAAAKrAEF
4	14	965	AAAYtAAAHAAAAKrAEG
2	14	10667	AAAYtAAAHAAAAKrAEH
3	14	3047	AAAYtAAAHAAAAKrAEI
3	14	8318	AAAYtAAAHAAAAKrAEJ
3	14	9882	AAAYtAAAHAAAAKrAEK
3	14	10667	AAAYtAAAHAAAAKrAEL
3	14	19683	AAAYtAAAHAAAAKrAEM
3	14	22478	AAAYtAAAHAAAAKrAEN
4	14	3047	AAAYtAAAHAAAAKrAEO
4	14	8318	AAAYtAAAHAAAAKrAEP
4	14	22478	AAAYtAAAHAAAAKrAEQ
3	14	5028	AAAYtAAAHAAAAKrAER
4	14	5028	AAAYtAAAHAAAAKrAES
2	14	1385	AAAYtAAAHAAAAKrAET
3	14	2622	AAAYtAAAHAAAAKrAEU
2	14	27197	AAAYtAAAHAAAAKrAEV
3	14	387	AAAYtAAAHAAAAKrAEW
3	14	2073	AAAYtAAAHAAAAKrAEX
3	14	3947	AAAYtAAAHAAAAKrAEY
3	14	5847	AAAYtAAAHAAAAKrAEZ
3	14	6543	AAAYtAAAHAAAAKrAEa
3	14	10844	AAAYtAAAHAAAAKrAEb
3	14	12605	AAAYtAAAHAAAAKrAEc
3	14	27197	AAAYtAAAHAAAAKrAEd
2	14	3618	AAAYtAAAHAAAAKrAEe
3	14	14	AAAYtAAAHAAAAKrAEf

Table & Index Access Paths

Index Unique Scan

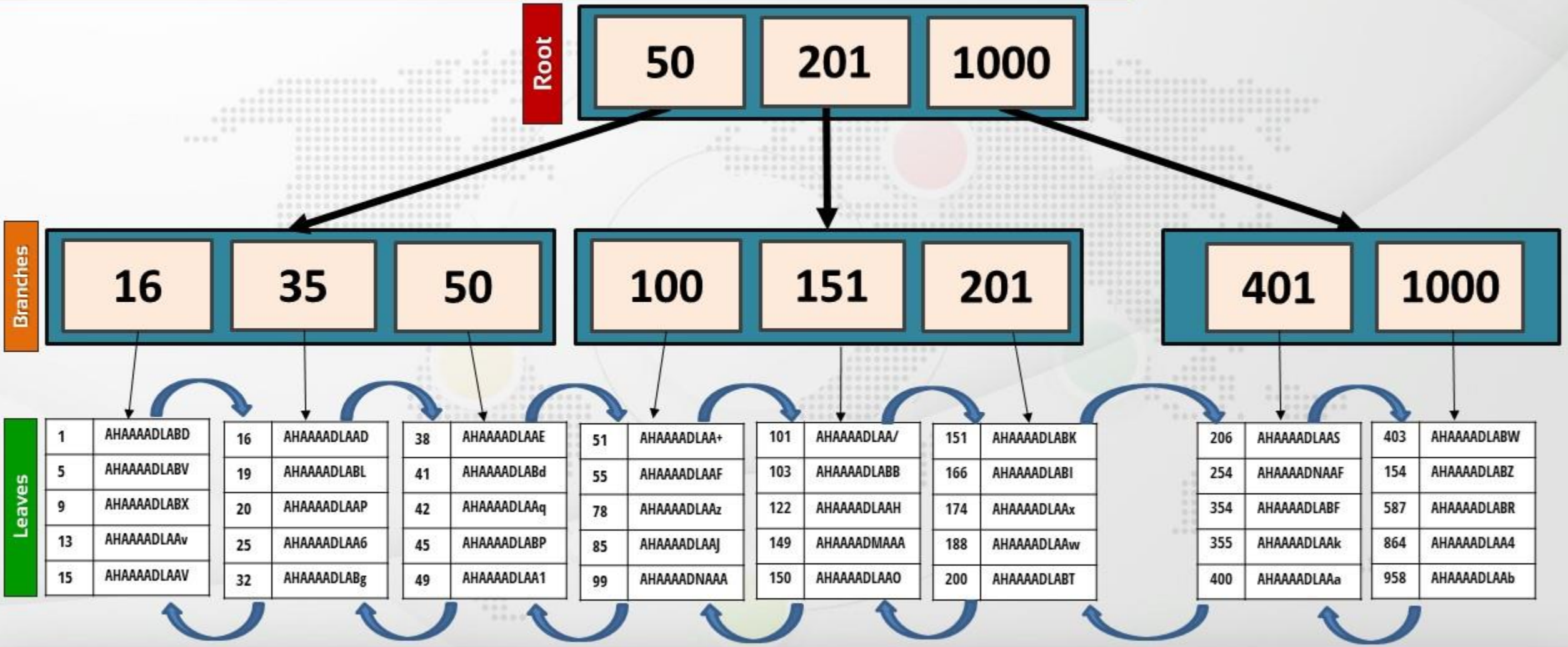


Table & Index Access Paths

INDEX RANGE SCAN

- If the data we queried is bounded from one or both sides, the optimizer can use index range scan.
- Can be applied to b-tree indexes and bitmap indexes.
- Can be applied to unique or non-unique indexes.
- Normally, data is stored in ascending order in the indexes.
- If the optimizer finds one or more leading columns with = > < sings, it will use index range scan.
- If the query includes an order by or group by clauses with the indexing columns, range scan will not do any sort. It is already sorted. It should not have null values.
- If order by clause has desc keyword, it will read the data in descending order.
- You can create your index as descending.

```
CREATE INDEX index_name ON employees (department_id DESC)
```

- Function-based indexes can be accessed as index range scan.
- If wildcard characters are written on the right, it will perform index range scan (`WHERE department_name LIKE 'A%'`)

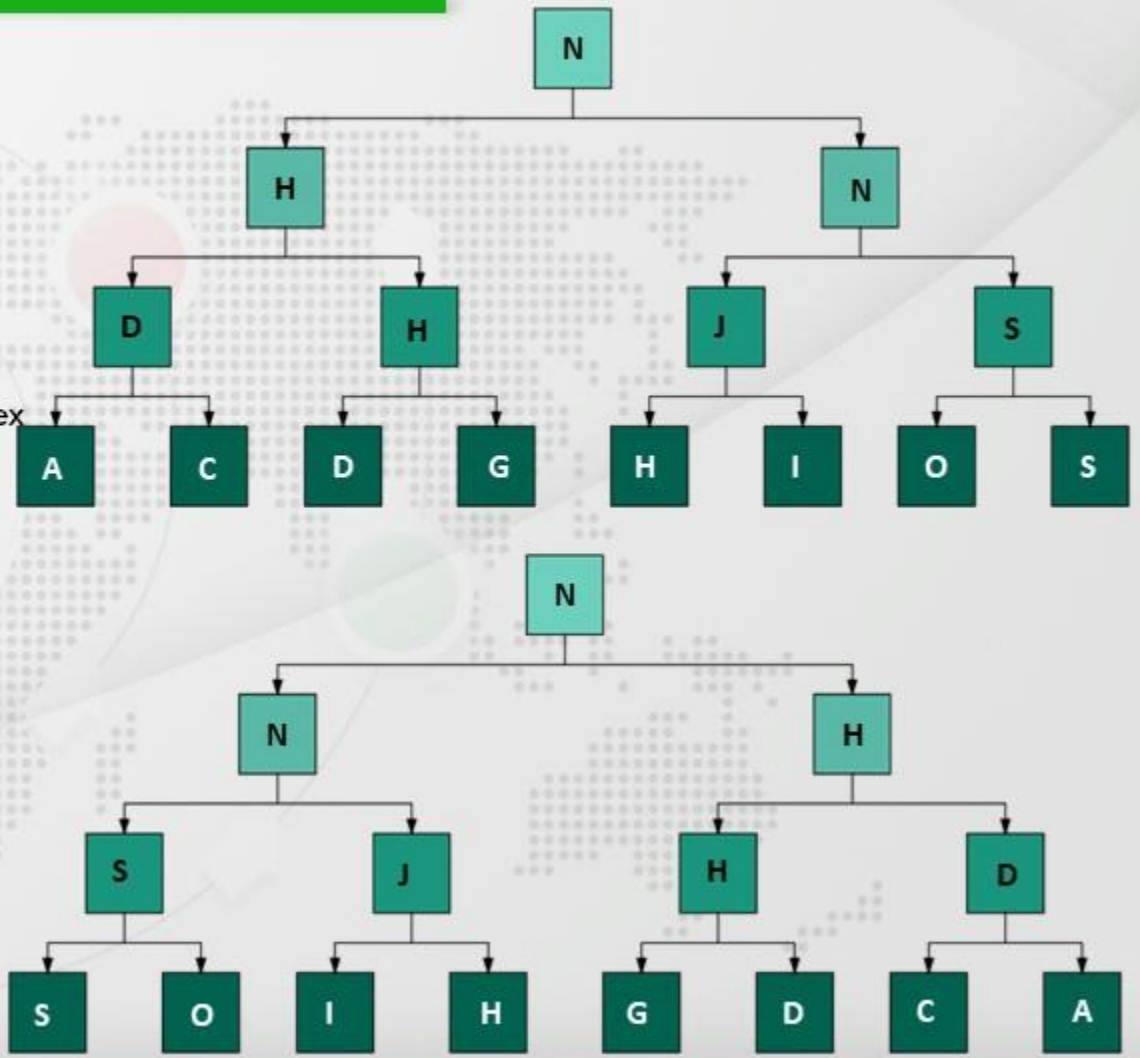


Table & Index Access Paths

INDEX FULL SCAN

- All the rows of the tables are indexed by their indexes
- When the optimizer uses the index full scan?
  - Query has order by clause only with the indexed columns.
  - Query has group by clause only with the indexed columns.
  - Query requires a sort-merge join.

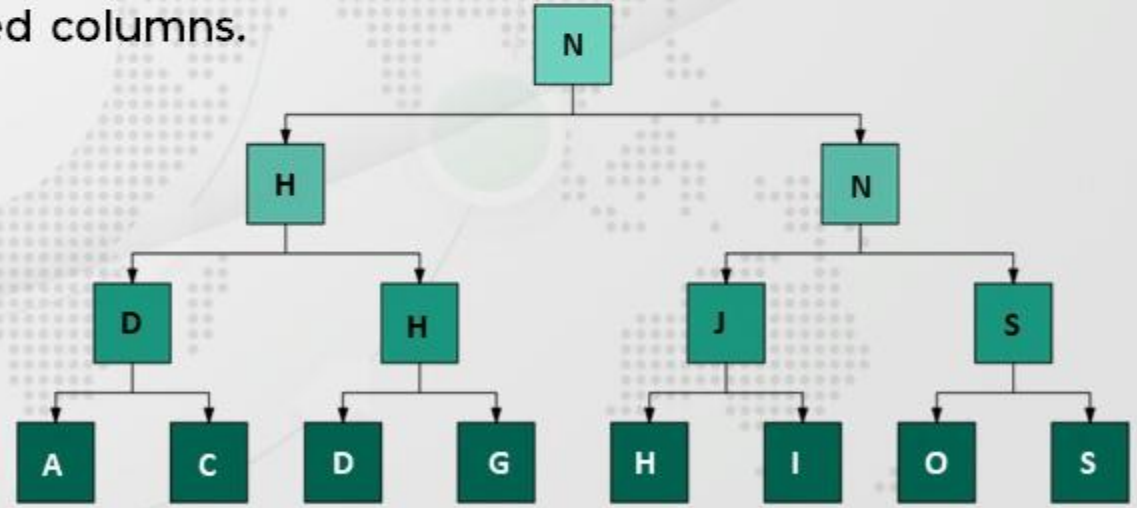


Table & Index Access Paths

INDEX FAST FULL SCAN

- If the query requests only the columns existing in the index, it uses IFF Scan
- Can be applied to both b-tree and bitmap indexes.
- Hints can be used to force the optimizer to use IFF Scan.
- The differences of Index Full Scan vs Index Fast Full Scan
  - Index Fast Full Scan always reads only from the index X Index Full Scan may read from table, too.
  - Index Full Scan reads blocks one by one, sequentially X Index Fast Full scan reads multiple simultaneously, in unordered manner.
  - Index Fast Full Scan is faster than Index Full Scan most of the times.
  - Index Full Scan can be used to eliminate sorting, but Index Fast Full Scan cannot.

Table & Index Access Paths

INDEX SKIP SCAN

- If you don't use the indexed columns on the where clause, the optimizer will not use the indexes.
- We don't create indexes for all the rows because of the costs.
- If the second, third,... column of a composite index is used as an access predicate, the optimizer will consider the index skip scan.
- Index skip scan skips the leaves which do not have any chance to have any matching rows.
- What are the advantages?
  - ✓ Helps to reduce the number of indexes
  - ✓ Decreases the index space
  - ✓ Increases the overall performance by reducing index maintenance

Table & Index Access Paths

INDEX SKIP SCAN

```
SELECT * FROM EMPLOYEES WHERE AGE BETWEEN 20 AND 30;
```

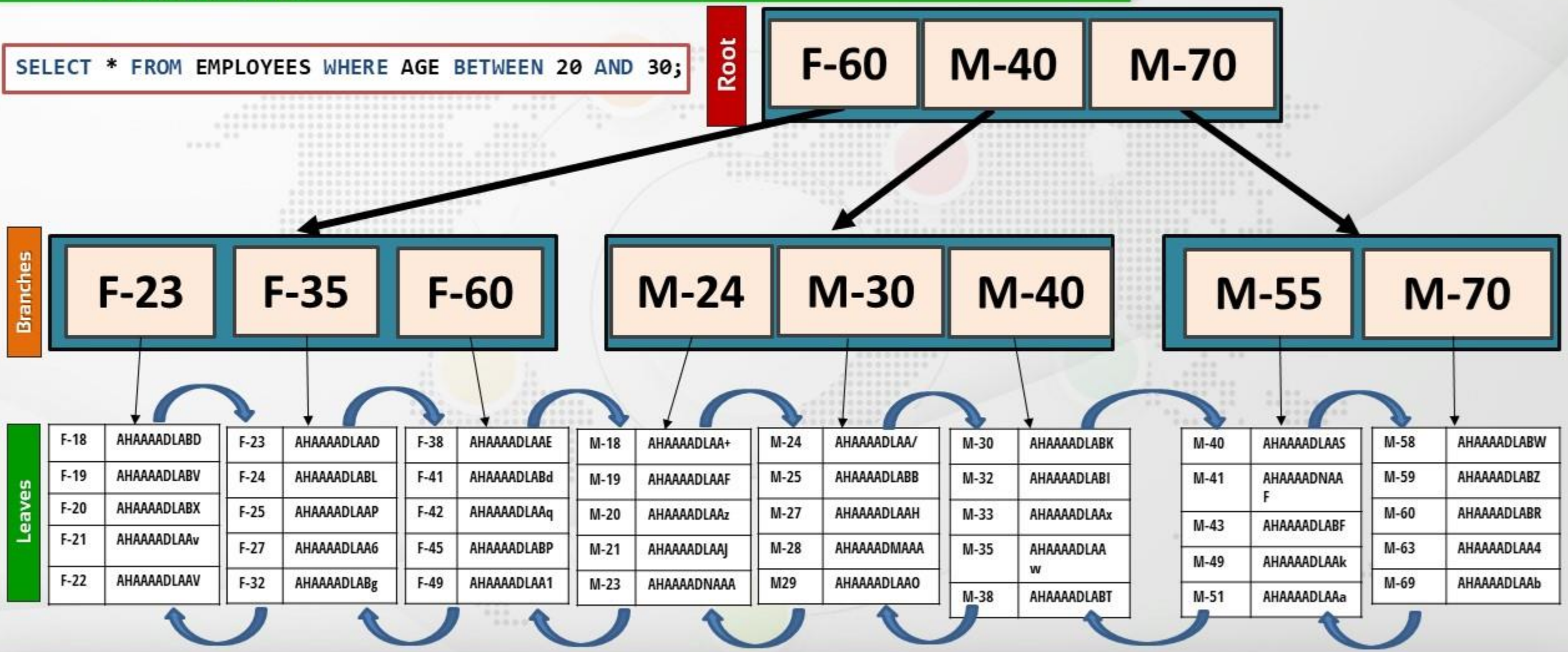


Table & Index Access Paths

INDEX JOIN SCAN

- If an index stores the columns of a query, the optimizer will perform index fast full scan.
- If the combination of multiple indexes store the columns of a query, the optimizer will join them and read the data from that join. (INDEX JOIN SCAN)
- What to know about Index Join Scan?
  - ✓ The combination of indexes must have every column of the select clause.
  - ✓ There is no join limit. More than two indexes can be joined together to get the data.
  - ✓ There might be any index access path before the index join scan
  - ✓ If you write ROWID in the select clause, it will NOT perform index join scan.

Using Optimizer Hints

WHAT ARE THE HINTS AND WHY TO USE THEM?

- To command the optimizer, we use optimizer hints.
- Optimizer hints force the optimizer to pick a specific action.
- The optimizer may not follow your hints.
- If the hint is not reasonable, the optimizer will ignore it.
- Hints can be operating on a single table, multi-tables, a query block, a specific statement.
- Categories of the hints:
  - ✓ Hints for optimization approaches
  - ✓ Access Paths Hints
  - ✓ Query Transformation Hints
  - ✓ Join Order Hints
  - ✓ Join Operations Hints
  - ✓ Parallel Executions Hint
  - ✓ Others

Using Optimizer Hints

HOW TO USE HINTS?

```
SELECT /*+ hint_name(p1 p2 p3..) */ first_name FROM EMPLOYEES;
```

- Hints can be used after a SELECT, UPDATE or DELETE keywords.
- You can use the table name or its alias as the hint parameter. But if there is an alias, you cannot use the table name!
- There can be only one hint area.
- Be careful on the hints you selected, especially if you are using multiple hints. You may lead the optimizer to a bad execution plan.

Join Operations

JOIN METHODS OVERVIEW

- To build an execution plan the optimizer checks :
  - ✓ Access Paths
  - ✓ Join Methods
  - ✓ Join Orders
- Join methods and join orders do not mean left join, outer join, etc.
- Join methods are the ways to join the row sources to build a new row source
- The existing join methods are:
  - ✓ Nested Loop Join
  - ✓ Sort Merge Join
  - ✓ Hash Join
  - ✓ Cartesian Join

- Nested loop join is efficient when joining row sources are small.
- Sort merge join is better than nested loop join if table is big and (or) one side is sorted.
- Hash join is better than sort merge join for most cases if both sides are not sorted already.
- Cartesian product is the most costly one. It joins all the rows of one side with all the rows of the other side.

Join Operations

NESTED LOOP JOIN

PLAN_TABLE_OUTPUT

```
| Id  | Operation                      | Name             |
-------------------------------------------------------------
|   0 | SELECT STATEMENT               |                  |
|   1 |  NESTED LOOPS                  |                  |
|   2 |   NESTED LOOPS                 |                  |
|*  3 |    TABLE ACCESS FULL           | DEPARTMENTS      |
|*  4 |    INDEX RANGE SCAN            | EMP DEPARTMENT IX |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES        |
-------------------------------------------------------------
```

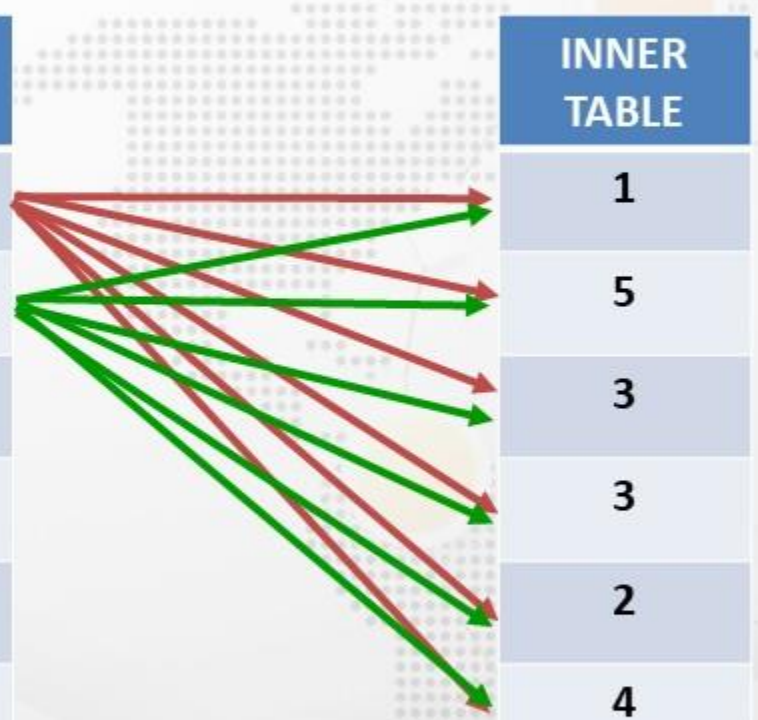
- A join operation is done by the driving table (external table, outer table) and the inner table.
- Table is the general name, but it is actually a row source
- If two row sources are small, or bigger one has an index, the optimizer may perform a nested loop
- Nested loop returns may be efficient if you need some rows immediately
- You can use USE_NL(table1 table2) hint to force the optimizer to use nested loops

Join Operations

NESTED LOOP JOIN

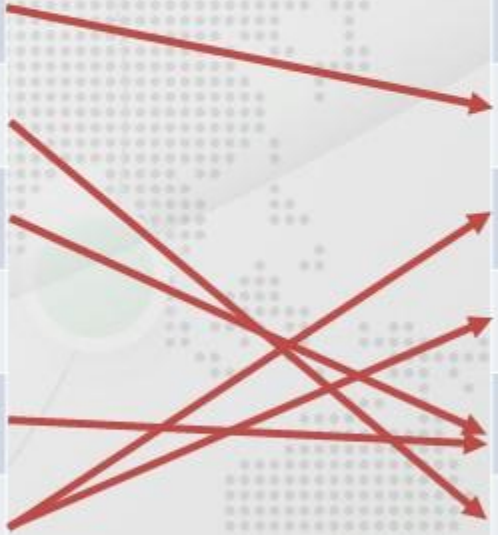
DRIVING TABLE
5
4
2
7
2
3

INNER TABLE
1
5
3
3
2
4



DRIVING TABLE
5
4
2
7
2
3

INNER TABLE
1
5
3
3
2
4



Join Operations

SORT MERGE JOIN

- Sort-merge join sorts both row sources
- If the sort exceeds the sort area in PGA, it will write the sorted data into the disc. But this increases the cost so much.
- Index Full Scan or Index Range Scan will be very useful in sort-merge join, since the data is already sorted in indexes.
- Sort is done on joining keys.
- After the sorts, both row sources are merged.
- Sort Merge is better than Nested Loop Join if table if row sources' sizes are large
- In sort merge, there is no driving table or inner table.
- If the row source is already sorted, there will be no sort key in the execution plan.
- Sort merge join is efficient when the join condition is not an equijoin
- use_merge(table1 table2) hint is used to force the optimizer to perform sort merge join

Join Operations

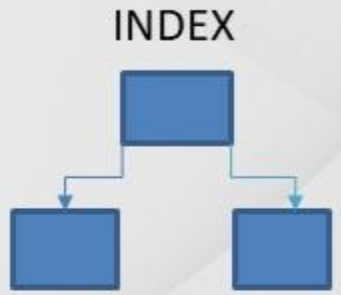
SORT MERGE JOIN

SOURCE A
2
3
4
4
5
7

SOURCE B
1
2
3
3
4
5

SOURCE A
5
4
2
7
2
3

SOURCE B
1
5
3
3
2
4



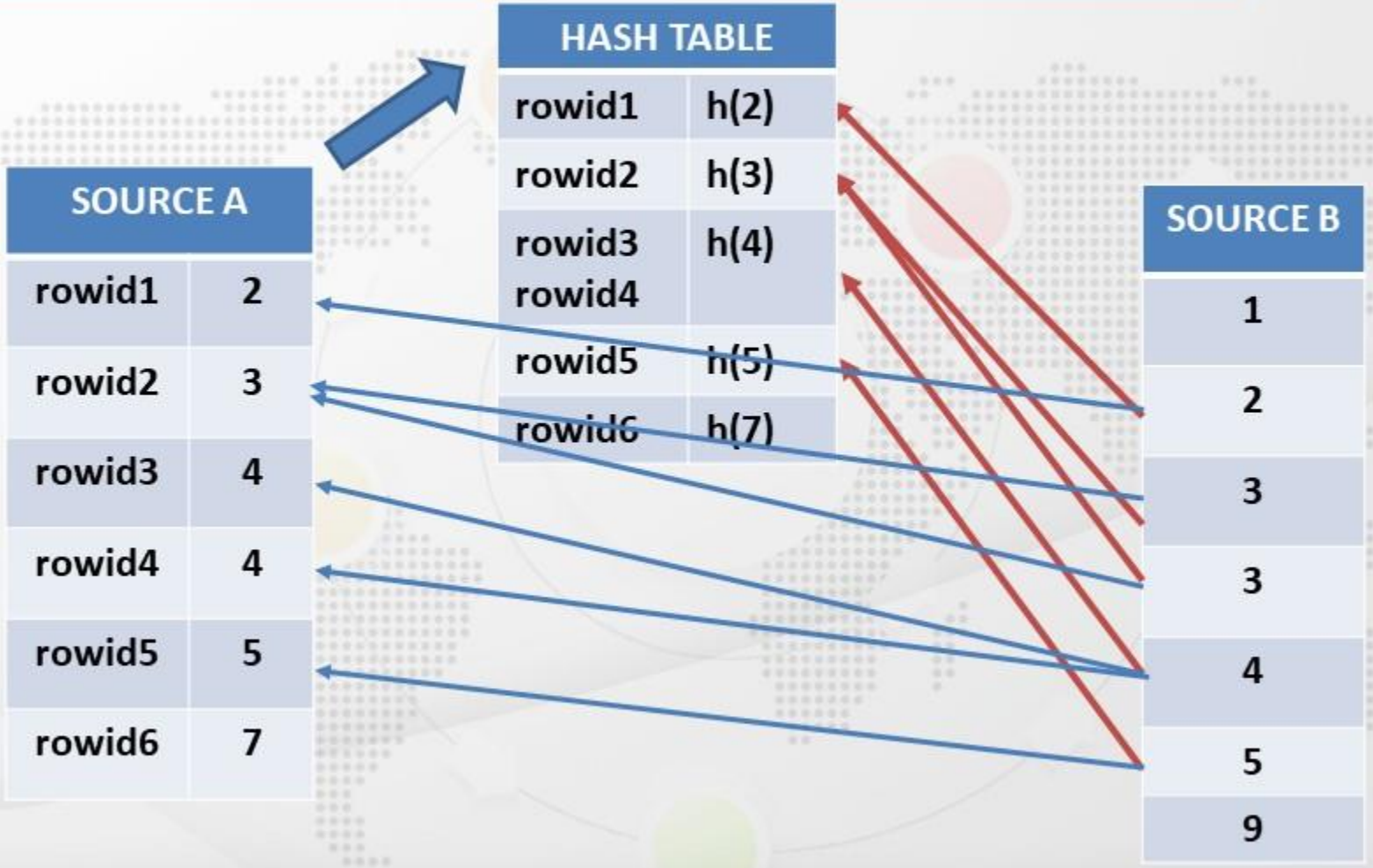
Join Operations

HASH JOIN

- A hash table is built for the smallest row source
- Hash table is created by generating the hash values of the key columns
- The keys of second row source are hashed and checked against the hash table
- Full table scan is performed to the table that will be hashed
- Hash join is performed only when an equijoin is used
- `use_hash(table1 table2)` hint is used to force the optimizer to perform hash join

Join Operations

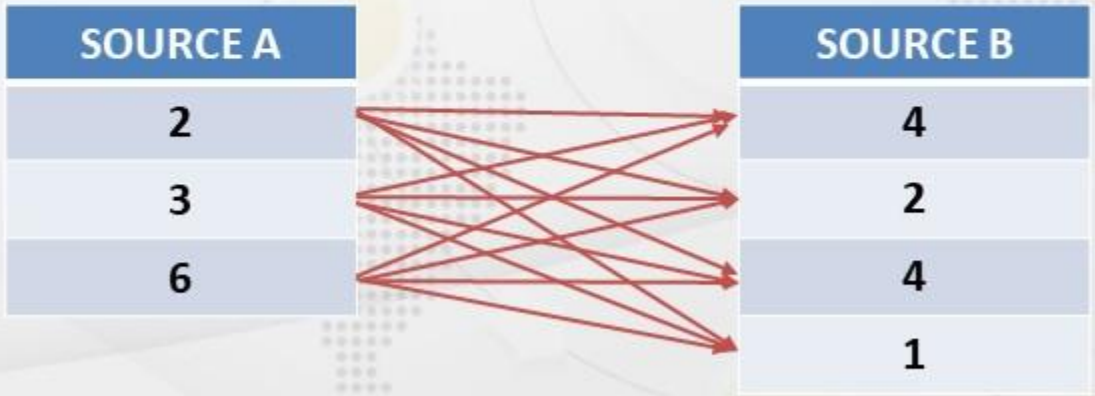
HASH JOIN



Join Operations

CARTESIAN JOIN

- Joins all the rows of a table with all the rows of the other table
- It is not a realistic join for business
- It is mostly done by mistake by forgetting to write the join condition
- It results a very high cost.



Join Operations

JOIN TYPES OVERVIEW

- There are 4 join types :
  - ✓ Equijoins and Nonequijoins
  - ✓ Outer Joins
  - ✓ Semijoins
  - ✓ Antijoins
- Equijoins return the matching rows with the equality operator
- Joining other than equality operator is called as nonequijoins
- Outer joins return matching and nonmatching rows
- Semijoins return the rows matching with the EXISTS subquery
- Antijoins return the rows which does not match with the NOT IN subquery

Join Operations

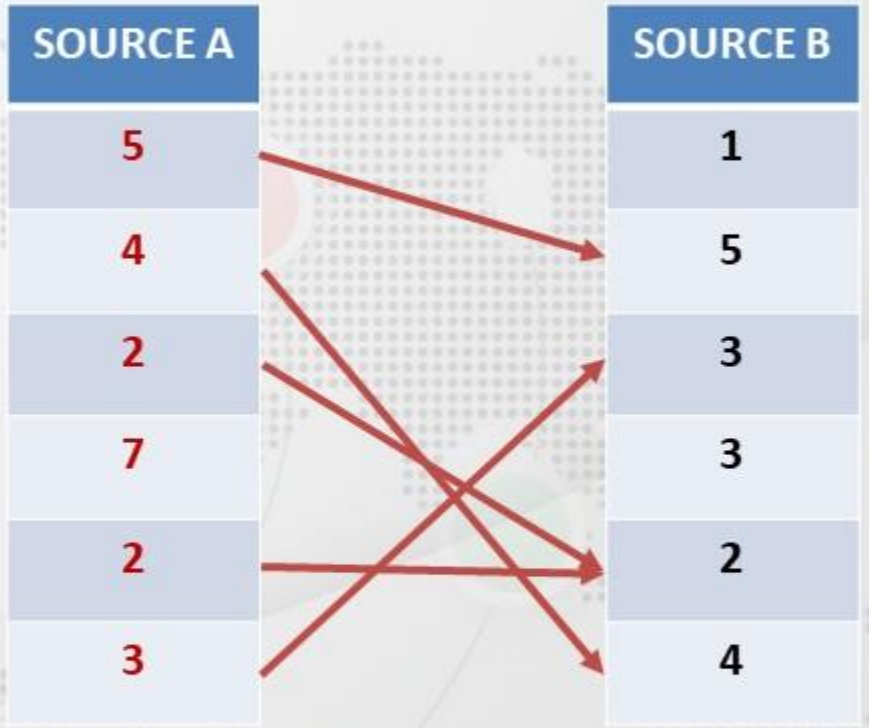
EQUIJOINS & NONEQUIJOINS

- If the join condition contains an equality operator, it is an equijoin.
- Equijoins are the most commonly used join types
- To improve performance, you should use equijoins whenever you can
- If you use another operator than the equality operator, it is nonequijoin.
- Equijoins can use all join methods
- Nonequijoins cannot use the hash join method

Join Operations

OUTER JOIN

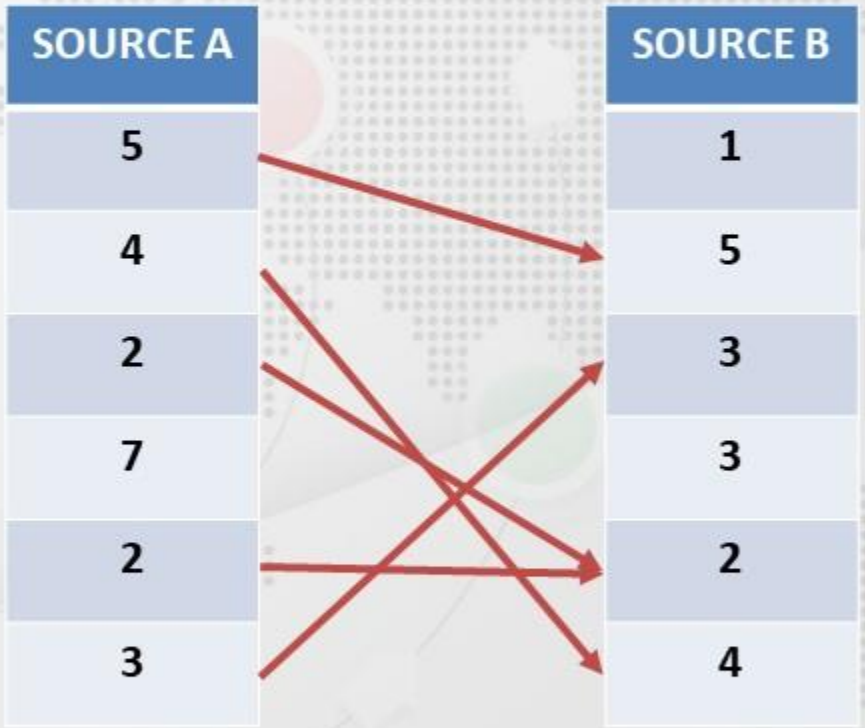
- Outer joins return the matched and unmatched rows of the sources.
- Outer joins can be used with all join methods
- With Nested Loop and Outer Join, inner table is the one whose nonmatching rows will return.
- With Hash Join and Outer Join, hash table is built for the one whose nonmatching rows will NOT return.



Join Operations

SEMIJOIN

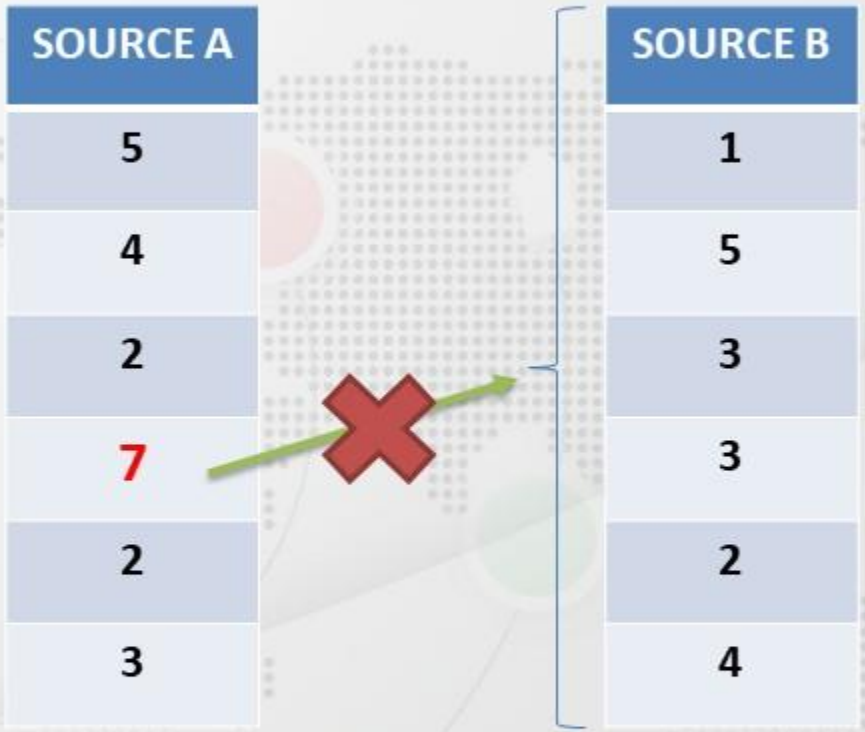
- Semijoin returns the first match
- Semijoin is the way of transforming the EXISTS subquery into a join
- Sometimes the optimizer may select a different method than semijoin even if you used the EXISTS subquery
- Semijoins can be used with all join methods
- Use EXISTS instead of IN if possible



Join Operations

ANTIJOIN

- Antijoins return the rows that do not match with the NOT IN subquery.
- By default antijoins are used with sort-merge joins.
- The optimizer may select a different one or you can use HASH_AJ or NL_AJ hints to change the join method.



Other Optimizer Operators

Result Cache Operator

- Result Cache is a memory area in SGA to store the results of queries for some time to increase the performance.
- There are two ways to store results in result cache:
  - MANUAL (DEFAULT - Needs result_cache hint)
  - FORCE (no_result_cache hint is used not to store in the result cache)
- DBMS_RESULT_CACHE package has statistics, information and some memory managing abilities
- V$RESULT_CACHE_OBJECTS view has the result cache data.
- Table annotations can be used as the default storage option to the result cache.

```
CREATE TABLE table_name (…) RESULT_CACHE (MODE DEFAULT|FORCE);
```

```
ALTER TABLE table_name (…) RESULT_CACHE (MODE DEFAULT|FORCE);
```

Other Optimizer Operators

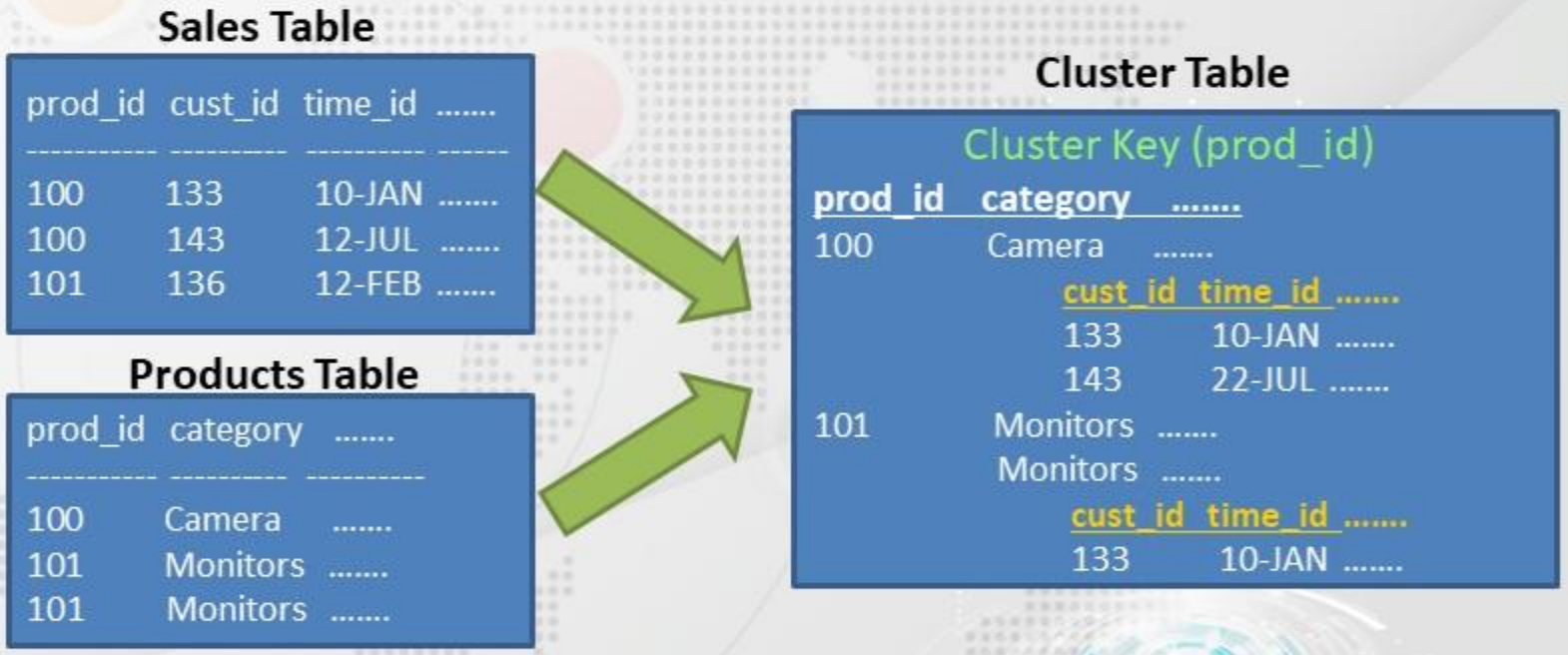
View Operator

- Every VIEW operator in the operations area does not mean that a view is there in the query
- Each seperate query in a query is pretended as an inline view.
- What is View Merging?
- View Merging means joining the inner query and the outer for a better performance
- Eventually, every query inside the outer query is shown with VIEW operator unless they cannot be merged with the outer query.

Other Optimizer Operators

CLUSTERS

- The main goal of clustering is improving the performance with a different way of storage
- Types of clusters:
  - ✓ INDEX CLUSTERS
  - ✓ HASH CLUSTERS
  - ✓ SINGLE TABLE HASH CLUSTERS
  - ✓ SORTED HASH CLUSTERS



Other Optimizer Operators

Sort Operators

- Sort Operator Types :
  - SORT AGGREGATE Operator
  - SORT UNIQUE Operator
  - SORT JOIN Operator
  - SORT GROUP BY Operator
  - SORT ORDER BY Operator
  - HASH GROUP BY Operator
  - HASH UNIQUE Operator
  - BUFFER SORT Operator

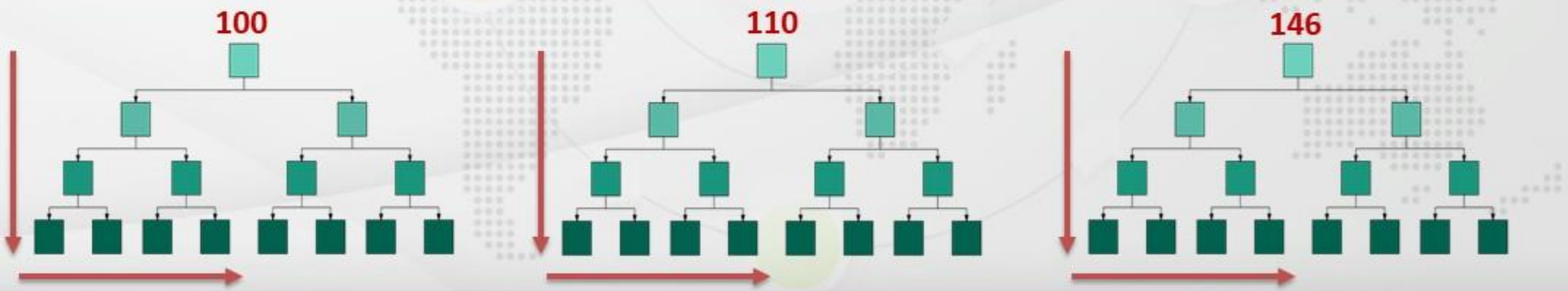
IF YOU WANT THE DATA RETURNED IN ORDER, YOU NEED TO USE THE ORDER BY CLAUSE. THESE ORDER TYPES DOES NOT GUARANTEE THAT THE ROWS WILL RETURN IN ORDER!

Other Optimizer Operators

Inlist Operator

- When we use IN clause, if the values in IN clause are not too many, the optimizer tends to use the INLIST Operator
- For INLIST Operator usage, the search must be on the indexed columns
- How does INLIST Operator work?

```
SELECT * FROM EMPLOYEES WHERE employee_id IN (100, 110, 146)
```



SQL Tuning Techniques

How to find the performance problem and solve it?

- Query analysis strategy mainly focuses on modifying our queries to a better performing one.
- The steps of solving the performance problem:
  - Check the execution plan basically for the problematic areas.
  - Check the statistics
  - Check your query for the common mistakes
  - Check the execution plans detailed
    - Check the Access paths
    - Check join orders and join types
    - Compare the actual & estimated number of rows
    - Check the operations where the cost and logical reads differ significantly

SQL Tuning Techniques

How to find the performance problem and solve it?

Common reasons for a Bad SQL:

- ✗ Poorly written query
- ✗ Index used or not used
- ✗ There is no index
- ✗ Predicates are not used
- ✗ Wrong types in predicates
- ✗ Wrong join order
- ✗ Other

Common possible solutions:

- ✓ Make the statistics up to date
- ✓ Use dynamic statistics
- ✓ Create or modify indexes
- ✓ Rewrite the query to use an index
- ✓ Use hints
- ✓ Remove wrong hints
- ✓ Change the hints
- ✓ Eliminate implicit data type conversion
- ✓ Create function-based indexes
- ✓ Use index-organized tables
- ✓ Change the optimizer mode
- ✓ Use parallel execution
- ✓ Use materialized views
- ✓ Modify or disable triggers and constraints
- ✓ Other

SQL Tuning Techniques

WAYS OF GETTING THE EXECUTION PLAN & STATISTICS

- There are many different tools to get the execution plans and the statistics.
- Explain plan or Execution Plan?
- There are 4 major tools to get the execution plan and the statistics :
  - ✓ Autotrace
  - ✓ SQL Monitor
  - ✓ TKPROF
  - ✓ DBMS_XPLAN
- AUTOTRACE :
  - It is free.
  - SQL Developer and SQL Plus shows different statistics.
  - Collapse or expand and focus only the problematic area in SQL Developer

SQL Tuning Techniques

WAYS OF GETTING THE EXECUTION PLAN & STATISTICS

- AUTOTRACE (Continues) :
  - We can compare two different execution plans and statistics with SQL Developer.
  - SQL Developer has a hotspot button which shows the problematic area of the plan.
  - We can export a plan and use it later.
  - By default, SQL Developer autotrace doesn't read the all of the rows.
    (Check *Tools->Preferences->Database->Autotrace/Explain Plan* in SQL Developer)
- SQL MONITORING :
  - It shows the execution plan and the statistics of the query being used right now
  - It can be viewed by SQL Developer or Enterprise Manager or by an SQL code
  - It captures the queries running longer than 5 seconds or running in parallel mode
  - If our query doesn't suit to these, we can add MONITOR hint to make it show on the list.
  - Needs Diagnostics and Tuning packs to be licensed.

SQL Tuning Techniques

WAYS OF GETTING THE EXECUTION PLAN & STATISTICS

- TKPROF :
  - Converts Oracle trace files into a human readable format.
  - We need to create a trace file first, and use TKPROF tool for this trace file.
  - Using TKPROF needs a bit more work to do comparing to the others.
  - You need to have the access privilege to the database server to use this.
  - It includes all the SQL statements run between the tracing starts and ends.
  - It breaks down the execution time into parse, execute and fetch times.
- DBMS_XPLAN :
  - It shows performance statistics for each step of the plan
  - If the query has already run before, there is no need to run for DBMS_XPLAN.
  - If it is deleted from the cache, the AWR tool can show earlier plans.

SQL Tuning Techniques

USING REAL-TIME SQL MONITORING TOOL

- What does the Real-Time SQL Monitoring Tool do for us?
  - We can check the real-time execution plans and statistics with graphical views.
  - Captures the execution plans and statistics automatically
  - Very useful to analyze the complex queries and PL/SQL code executions.
- This tool can be used via SQL Developer, Enterprise Manager, or by Code
- We can view SQL Monitoring information from v$sql_monitor and v$sql_plan_monitor database views.
- Captures the queries running longer than 5 seconds or running in parallel mode.
- To see the queries running less than 5 seconds, we can use the MONITOR hint.
- Needs to have Diagnostics and Tuning Packs to be licensed.

SQL Tuning Techniques

USING REAL-TIME SQL MONITORING TOOL

- What are the benefits of this tool?
  - Shows the execution in real-time with lots of details
  - Tracks all the running queries, which enables us to find the top consuming queries
  - We can create active reports of the running queries which lets us analyze offline
  - Monitors the parallel execution
  - Helps to analyze the large plans and complex queries easier
  - We can utilize poorly used indexes
  - We can determine the bind variables of the queries

SQL Tuning Techniques

USING SQL TRACE FILES & TKPROF TOOL

- What are the benefits of SQL Trace files?
  - Performance information of our queries.
  - CPU Time and Elapsed Time
  - Wait Events
  - Execution Plans
  - Row Counts
  - Call Counts (Parse, Execute, Fetch)
  - Physical and Logical Reads

SQL Tuning Techniques

USING SQL TRACE FILES & TKPROF TOOL

```
SELECT value FROM V$DIAG_INFO WHERE name = 'Diag Trace';
```

- How to enable/disable tracing?
  - We can enable/disable tracing for a specific user | specific session | entire database.
  - `dbms_monitor.database_trace_enable();`
  - `dbms_session.set_sql_trace();`
  - `alter session set sql_trace = true;`
  - We can trace user, session, database, application, service, module, etc.

SQL Tuning Techniques

USING SQL TRACE FILES & TKPROF TOOL

- How to track our own session?

```
ALTER SESSION SET SQL_TRACE = TRUE|FALSE;
```

```
DBMS_SESSION.SESSION_TRACE_ENABLE(waits => TRUE, binds =>FALSE);
```

```
DBMS_MONITOR.SESSION_TRACE_ENABLE(session_id=>27,serial_num=>60,
waits=>TRUE, binds=>FALSE)
```

```
DBMS_SESSION.SESSION_TRACE_DISABLE();
```

```
DBMS_MONITOR.SESSION_TRACE_DISABLE(session_id=>27,serial_num=>60);
```

  - trcsess utility can merge multiple trace files.

SQL Tuning Techniques

USING SQL TRACE FILES & TKPROF TOOL

- How to generate TKPROF output?
  - TKPROF takes trace files as input, and generates a formatted output file.
  - Doesn't show the commit and rollback operations.

```
TKPROF trace_file_name output_file_name   [waits=yes|no]
                                          [sort=option]
                                          [print=n]
                                          [aggregate=yes|no]
                                          [insert=insert_file_name]
                                          [sys=yes|no]
                                          [table=schema.table]
                                          [explain=user/password]
                                          [record=record_file_name]
                                          [width=n];
```

SQL Tuning Techniques

GET WHAT YOU NEED ONLY

- Do not use select * for all the queries!
  - The optimizer may select a worse plan if you query for unnecessary columns
  - While joining multiple tables or querying from views, selecting less columns might affect the performance
  - If you use select * , the database needs to check the data dictionary to get the table structure
  - select * will make the database perform more I/O operations
  - select * may decrease the performance significantly if the table has LOBs
  - select * will have a higher overload on the network. So there might be more network waits
  - select * is tend to have problems on maintenance

SQL Tuning Techniques

INDEX USAGE

- How do I make the optimizer use my indexes?
  - If the selectivity of the predicate is high, using indexes may increase the performance very much
  - Add adequate predicates to the queries (Use indexed columns in your queries clearly)
  - Use reasonable hints in your queries
  - If possible, select only the indexed columns

SQL Tuning Techniques

USING CONCATENATION OPERATOR

- How do I make the optimizer use my indexes?
  - To make the optimizer use our indexes, the indexed columns must be used clearly.

BAD

```
SELECT first_name, last_name, department_name FROM employees
WHERE first_name||last_name = 'StevenKING';
```

GOOD

```
SELECT first_name, last_name, department_name FROM employees
WHERE first_name = 'Steven' AND last_name = 'KING';
```

SQL Tuning Techniques

USING ARITHMETIC OPERATORS

- How do I make the optimizer use my indexes?
  - One of the common mistakes for tuning aspect is using the arithmetic operations on the indexed columns.

BAD

```
SELECT prod_id, cust_id, time_id FROM sales
WHERE time_id + 10 = '20-JAN-98';
```

GOOD

```
SELECT prod_id, cust_id, time_id FROM sales
WHERE time_id = '10-JAN-98';
```

GOOD

```
SELECT prod_id, cust_id, time_id FROM sales
WHERE time_id = to_date('20-JAN-98', 'DD-MON-RR')-10;
```

SQL Tuning Techniques

USING LIKE OPERATOR

- How do I make the optimizer use my indexes?

BAD

```
SELECT employee_id, first_name, last_name, salary FROM employees
WHERE last_name LIKE '%on';
```

GOOD

```
SELECT employee_id, first_name, last_name, salary FROM employees
WHERE last_name LIKE 'Ba%';
```

GOOD

```
SELECT employee_id, first_name, last_name, reverse(last_name)
FROM employees WHERE reverse(last_name) LIKE 'rahh%';
```

SQL Tuning Techniques

USING FUNCTIONS ON COLUMNS

- How do I make the optimizer use my indexes?
  - Using functions on the indexed columns may suppress index usage

BAD

```
SELECT employee_id, first_name, last_name, salary FROM employees
WHERE TRUNC(hire_date,'YEAR') = '01-JAN-2002';
```

GOOD

```
SELECT employee_id, first_name, last_name, salary FROM employees
WHERE hire_date BETWEEN '01-JAN-2002' AND '01-JAN-2002';
```

SQL Tuning Techniques

USING IS NULL & IS NOT NULL EFFICIENTLY

- How do I make the optimizer use my indexes?
  - B-Tree indexes do not index the null values and this sometimes may suppress the index usages in our queries.
  - Ways to handle NULL value based performance loss :
    - Use IS NOT NULL condition in your where clause if you don't need to have the NULL values in the resultset
    - Add NOT NULL constraint to your columns and insert a specific value for the NULL values.
    - If reasonable, create a BITMAP index instead of a B-TREE index (BITMAP indexes store the NULL values)

SQL Tuning Techniques

THINGS TO KNOW FOR IN – EXISTS AND NOT IN – NOT EXISTS

```
SELECT * FROM T1 WHERE X IN (SELECT X FROM T2);
```

=

```
SELECT * FROM T1, (SELECT X FROM T2) T2 WHERE T1.X = T2.X);
```

```
SELECT * FROM T1 WHERE EXISTS (SELECT X FROM T2 WHERE T1.X = T2.X);
```

=

```
FOR X IN (SELECT * FROM T1) LOOP
 IF (EXISTS (SELECT X FROM T2)) THEN
  OUTPUT THE RECORD
 END IF;
END;
```

SQL Tuning Techniques

USING IS NULL & IS NOT NULL EFFICIENTLY

- Which one is faster?
  - If the outer table is big and the subquery is small, using IN might have a better performance.
  - If the outer table is small and the inner table is big, using EXISTS might have a better performance.
  - Some wrong beliefs:
    - EXISTS doesn't work better than IN all the times. It depends.
    - NOT EXISTS is not the equivalent of NOT IN. So it cannot be used instead of NOT IN all the times. Especially if there are any null values, the NOT IN will return nothing.
    - The new versions of database generally finds the optimum choice between EXISTS and IN

SQL Tuning Techniques

USING TRUNCATE INSTEAD OF DELETE

- TRUNCATE is always faster than the DELETE command (Truncate doesn't generate UNDO data, but delete generates)
- Things to know about TRUNCATE
  - Truncate operation cannot be rollbacked. Flashback is also not so easy after truncate operations.
  - Truncate is a DDL operation. So it performs commits before and after the truncate operation.
  - We can truncate a single partition as well.
  - Truncate doesn't fire the DML triggers. But it can fire the DDL triggers.
  - Truncate makes unusable indexes usable again. But delete does not.

SQL Tuning Techniques

DATA TYPE MISMATCH

- How do I make the optimizer use my indexes?
  - If the data types of the column and compared value don't match, this may suppress the index usage.

BAD

```
SELECT cust_id, cust_first_name, cust_last_name FROM customers
WHERE cust_postal_code = 60332;
```

GOOD

```
SELECT cust_id, cust_first_name, cust_last_name FROM customers
WHERE cust_postal_code = '60332';
```

SQL Tuning Techniques

TUNING ORDERED QUERIES

- Order by mostly requires sort operations
- The sort operations are done in PGA or discs (If PGA doesn't have enough memory)
- How to tune the order by clauses?
  - Create or modify B-Tree indexes including the column used in the order by clause
  - Increase the PGA size
  - Query for only the indexed columns in the select clause
  - Restrict the returning rows

SQL Tuning Techniques

RETRIEVING MIN & MAX VALUES

- B-Tree indexes increase the performance a lot for the min & max value searches
- To find the min or max value, it needs to read the whole table (if there is no B-Tree index)
- In B-Tree indexes, the rightmost and leftmost leaves have the maximum and minimum values of the indexed column
- If the returning rows are restricted, this time instead of leftmost and rightmost leaves, some other leaves have the min and max values. But again, it doesn't need to read all the rows of the index.
- If the query has multiple aggregate functions or another column, it will perform index full scan or table access full.

SQL Tuning Techniques

UNION & UNION ALL OPERATORS

- If they return the same results or if you don't care about the duplicates, you should use UNION ALL instead of UNION for performance (UNION ALL doesn't perform sort)

BAD

```
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales
 WHERE channel_id = 3;
UNION
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales
 WHERE channel_id = 4;
```

GOOD

```
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales
 WHERE channel_id = 3;
UNION ALL
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales
 WHERE channel_id = 4;
```

SQL Tuning Techniques

AVOID USING THE HAVING CLAUSE

- How can I tune the having clauses?
  - Having clause restricts the rows after they are read!
  - Predicates in the having clause will not be used as access predicates!

BAD

```
SELECT prod_id, SUM(amount_sold) FROM sales
GROUP BY prod_id
HAVING prod_id = 136;
```

GOOD

```
SELECT prod_id, SUM(amount_sold) FROM sales
WHERE prod_id = 136
GROUP BY prod_id;
```

SQL Tuning Techniques

BE CAREFUL ON VIEWS!

- Views on performance aspect
  - Do not use the views out of their purposes
- Things to be careful on views!
  - If you don't need to use all the tables in a view, do not use this view on your queries.
  - Don't join the complex views with a table or another view
  - Avoid performing outer joins to the views
  - Be careful on subquery unnesting
  - Avoid using views inside of a view

SQL Tuning Techniques

CREATE MATERIALIZED VIEWS!

- Unlike the basic and complex views, materialized views store both the query and the data
- The materialized view data can be refreshed manually or by a PL/SQL job, or by auto-refresh on DMLs
- Materialized view maintenance also is a burden to the database
- Using materialized views may increase the performance a lot
- We can create indexes, partitions etc, on materialized views (Its table is an ordinary table)
- When query rewrite is enabled in materialized views or in your session, the optimizer may use your materialized view even if you don't query from it

SQL Tuning Techniques

AVOID COMMIT TOO MUCH OR TOO LESS!

- For each DML operations, the database creates UNDO data
  - Performing commit frequently will not help on performance. (UNDO data is already carried to discs frequently)
- For each DML operations, the database creates REDO data
  - Performing commit frequently will not help on performance (REDO data is already carried to discs frequently)
  - Any changes on the same blocks (which are carried to the redo log files) will be created in redo log buffer again
- Updates and deletes will lock the rows and that will make the other users wait to perform any other operations on these rows
- How often do we need to commit?
  - As soon as we finish the DML operations
  - For some business-specific reasons

SQL Tuning Techniques

USING BULK COLLECT

- SQL Codes are executed in SQL Engine & PL/SQL Codes are executed in PL/SQL Engine
- Transfer of control between SQL Engine and PL/SQL Engine is called as CONTEXT SWITCH
- BULK COLLECT decreases the context switches (It reads multiple rows in one fetch)
- Things to know about BULK COLLECT
  - By default, the bulk collect fetches all the rows in one context switch
  - We can change the fetch count by using the LIMIT keyword
  - Implicit cursors use bulk collect by default

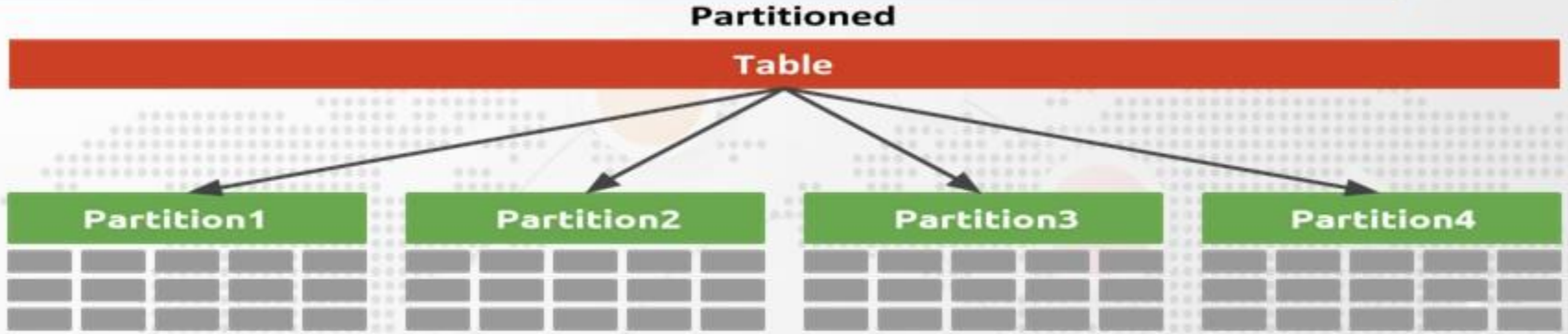
SQL Tuning Techniques

USING BULK COLLECT

- SQL Codes are executed in SQL Engine & PL/SQL Codes are executed in PL/SQL Engine
- Transfer of control between SQL Engine and PL/SQL Engine is called as CONTEXT SWITCH

SQL Tuning Techniques

PARTITION PRUNING



- If the query has low selectivity, the optimizer mostly prefers performing full-table scans
- Creating partitioned tables increases the cost for the queries having low selectivity
- Selecting from specific partitions is called as partition pruning
- How can we prune the partitions?
  - Selecting directly from the partition by using the partition name
  - Adding predicates to the where clause including the partition key (partitioned columns)

SQL Tuning Techniques

TUNING JOIN ORDER

```
SELECT p.prod_name, s.quantity_sold, s.amount_sold FROM sales s, products p
WHERE s.prod_id= p.prod_id;
```

OPERATION	COST	IO_COST	CARDINALITY	BYTES
SELECT STATEMENT	455			
HASH JOIN	455	444	918843	38591406
Access Predicates				
S.PROD_ID=P.PROD_ID				
NESTED LOOPS	455	444	918843	38591406
NESTED LOOPS				
STATISTICS COLLECTOR				
TABLE ACCESS (FULL) PRODUCTS	3	3	72	2160
PARTITION RANGE (ALL)				
BITMAP CONVERSION (TO ROWIDS)				
BITMAP INDEX (SINGLE VALUE) SALES_PROD_BIX				
Access Predicates				
S.PROD_ID=P.PROD_ID				
TABLE ACCESS (BY LOCAL INDEX ROWID) SALES	449	441	12762	153144
PARTITION RANGE (ALL)	449	441	918843	11026116
TABLE ACCESS (FULL) SALES	449	441	918843	11026116

SQL Tuning Techniques

USING WITH CLAUSE

GOOD

```
WITH sum_amount AS
        (SELECT SUM(amount_sold) amt_sold, prod_id FROM sales GROUP BY prod_id),
     num_of_prods AS
        (SELECT COUNT(*) num_prods FROM products)
 SELECT prod_name,
        amt_sold / (SELECT num_prods FROM num_of_prods)
 FROM products P, sum_amount S
 WHERE P.prod_id = S.prod_id
 AND amt_sold > 100000;
```

- If you use similar queries multiple times, using the WITH clause may increase the performance
- With the WITH clause, Oracle stores the result of the queries in user's temp space
- If the result is not very big, it stores the data as a view in memory
- If the result is big, it stores the data in a global temporary table automatically created for that query.

Tuning with Advanced Indexing Techniques

WHY IS INDEXING IMPORTANT?

- Indexing is the most important thing in SQL and Performance Tuning.
- There are lots of very useful index types in Oracle Database.
- Poor indexing strategy will result poor performance.
- Since the data is stored randomly in the discs, indexes will have a much faster access to the data by using their exact location IDs.
- Without indexes, even if you select a very small fraction of the table, it will read the whole table anyway.
- Index keys are selected from the most frequently queried and the most selective columns.
- Indexes store the ROWIDs which are the exact location of the rows in the discs.

Tuning with Advanced Indexing Techniques

WHY IS INDEXING IMPORTANT?

- Why shouldn't we create indexes for all columns?
- Because of indexing cost
  - Storing the similar data more than once
  - Maintenance cost
- Why & How should we use the indexes?
  - If our queries search for a small fraction of the table
  - If the related columns are queried so often
  - We should select the index type carefully
  - Indexed columns must be selective

With Indexes

Without Indexes

Tuning with Advanced Indexing Techniques

INDEX SELECTIVITY & CARDINALITY

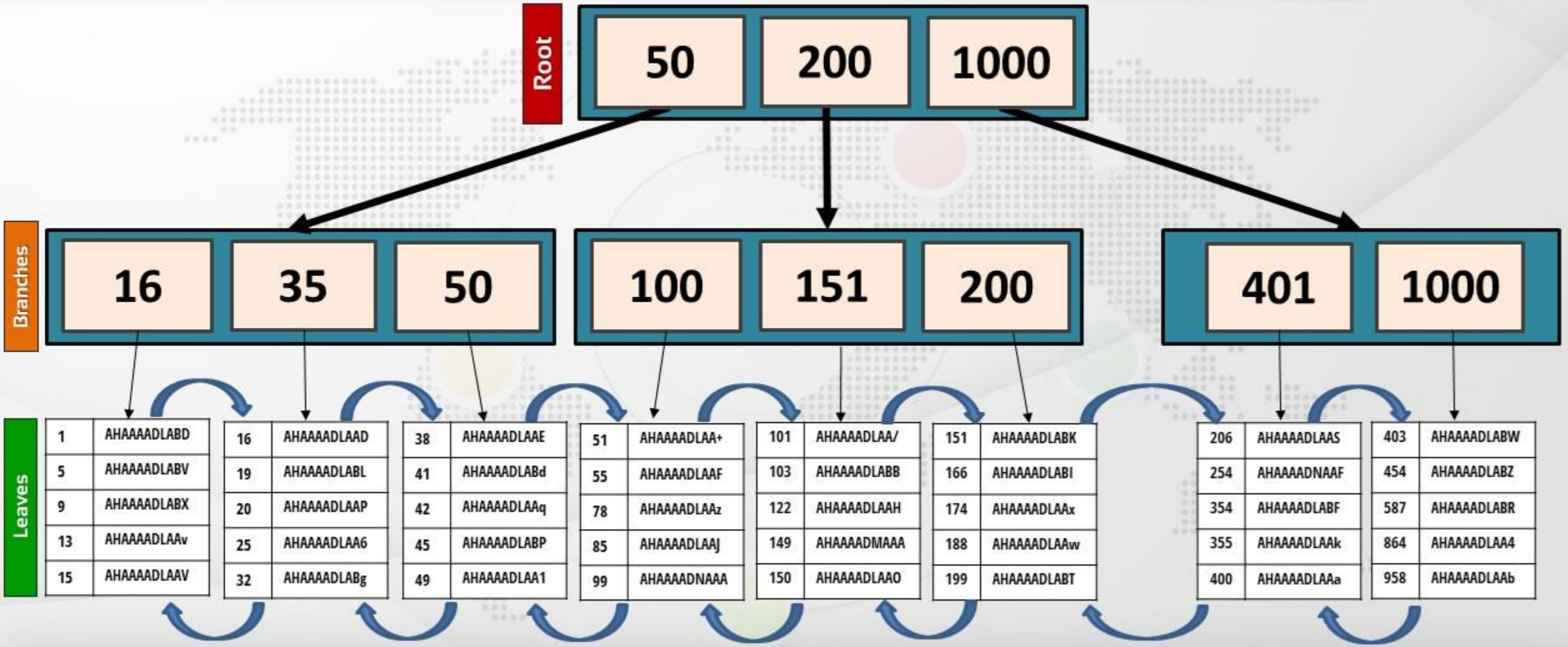
- Index selectivity is very important for index performance
- If the database reads from an index, there are multiple reads here. It first reads from branches, then leaves, and then the table.
- If the index is not selective, its cost might be higher than reading the whole table
- There is not any restriction for the optimizer to or not to use an index.

$$\text{Index Selectivity} = \frac{\text{Number of unique values of the colum or column list}}{\text{Total number of rows}}$$

```
SELECT column_name, num_distict FROM all_tab_colums
WHERE table_name = '<<table_name>>';
```

Tuning with Advanced Indexing Techniques

B-TREE INDEXES IN DETAILS



Tuning with Advanced Indexing Techniques

B-TREE INDEXES IN DETAILS

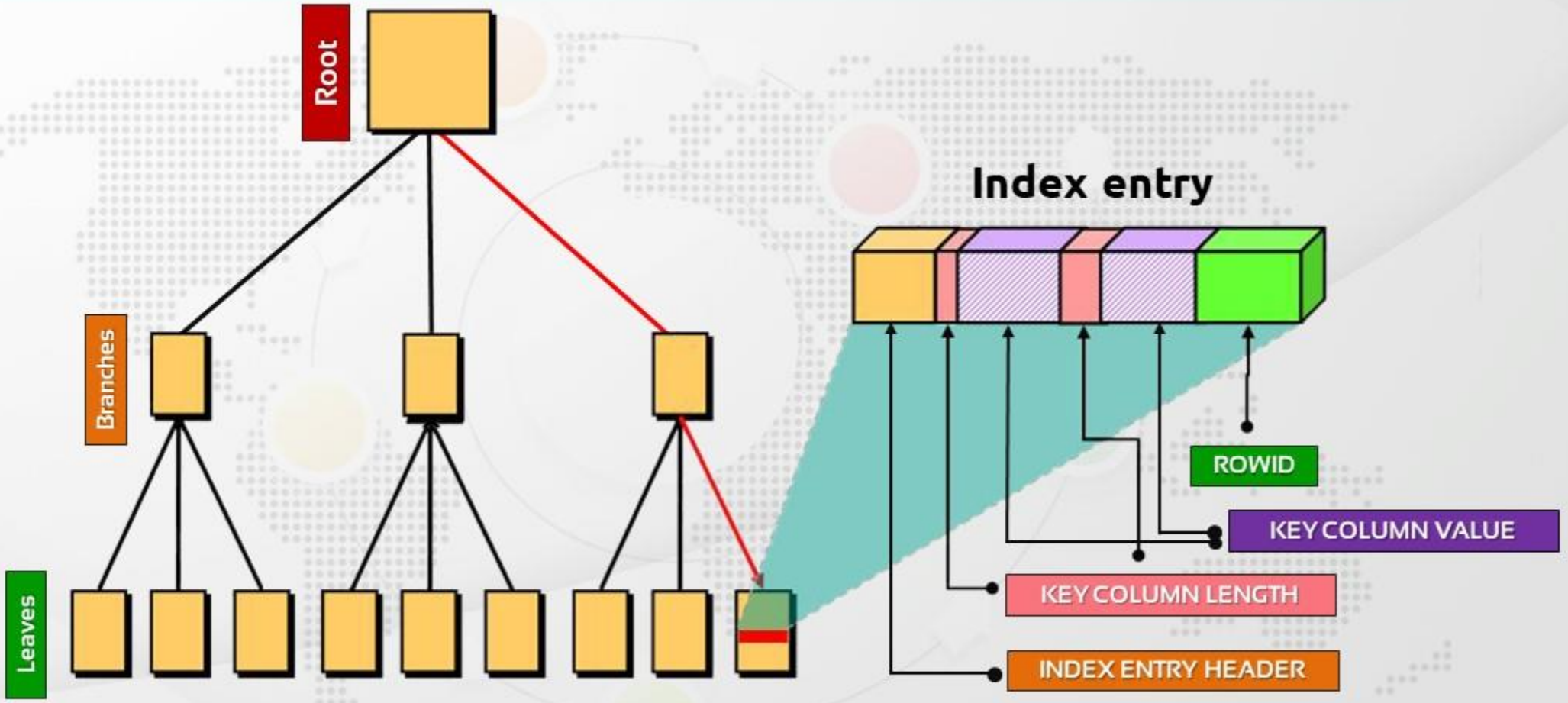
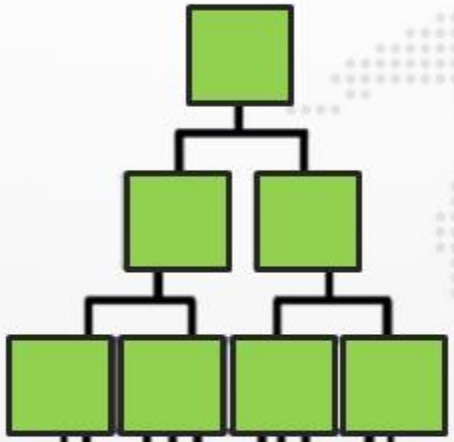


Table & Index Access Paths

BITMAP INDEXES IN DETAILS



Sales Table

ROWID	CUST_ID	PROD_ID	CHANNEL_ID
AAAYtAAAHAAAAKrAED	10007	14	2
AAAYtAAAHAAAAKrAEE	10007	14	3
AAAYtAAAHAAAAKrAEF	10007	14	4
AAAYtAAAHAAAAKrAEG	965	14	4
AAAYtAAAHAAAAKrAEH	10667	14	2
AAAYtAAAHAAAAKrAEI	3047	14	3
AAAYtAAAHAAAAKrAEJ	8318	14	3
AAAYtAAAHAAAAKrAEK	9882	14	3
AAAYtAAAHAAAAKrAEL	10667	14	3
AAAYtAAAHAAAAKrAEM	19683	14	3
AAAYtAAAHAAAAKrAEN	22478	14	3
AAAYtAAAHAAAAKrAEO	3047	14	4
AAAYtAAAHAAAAKrAEP	8318	14	4
AAAYtAAAHAAAAKrAEQ	22478	14	4
AAAYtAAAHAAAAKrAER	5028	14	3
AAAYtAAAHAAAAKrAES	5028	14	4
AAAYtAAAHAAAAKrAET	1385	14	2
AAAYtAAAHAAAAKrAEU	2622	14	3
AAAYtAAAHAAAAKrAEV	27197	14	2
AAAYtAAAHAAAAKrAEW	387	14	3
AAAYtAAAHAAAAKrAEX	2073	14	3
AAAYtAAAHAAAAKrAEY	3947	14	3
AAAYtAAAHAAAAKrAEZ	5847	14	3
AAAYtAAAHAAAAKrAEa	6543	14	3
AAAYtAAAHAAAAKrAEb	10844	14	3
AAAYtAAAHAAAAKrAEc	12605	14	3
AAAYtAAAHAAAAKrAEd	27197	14	3
AAAYtAAAHAAAAKrAEe	3618	14	2
AAAYtAAAHAAAAKrAEf	14	14	3

Channel ID

	START ROWID	END ROWID	BITMAP
2	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0
3	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	0 1 0 0 0 1 1 1 1 1 1 0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 0 1
4	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
9	AAAYtAAAHAAAAKrAED	AAAYtAAAHAAAAKrAEf	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Tuning with Advanced Indexing Techniques

BITMAP INDEXES IN DETAILS

- The benefits of Bitmap Indexes :
  - Works faster than B-Tree indexes for large number of rows in the resultset.
  - Uses less disc space than the B-Tree indexes
    - Bitmap index doesn't store the values for each row. It stores only once for each distinct value for one leaf.
    - Bitmap indexes doesn't store all the rowids. Instead, it stores the intervals and then converts when it needs
    - Bitmap indexes store the bitmaps compressed
  - More efficient when the query contains multiple conditions in the where clause
  - Can be used for parallel DML or parallel queries
  - Indexes the NULL values
  - Bitmap Join indexes are useful for multiple table reads

Tuning with Advanced Indexing Techniques

BITMAP INDEXES IN DETAILS

- What to know about Bitmap indexes for more?
  - Usually easier to remove and re-create than maintain
  - Not suitable for concurrent transactions modifying the indexed column
  - Low selectivity is better for bitmap indexes
  - You need to select global or local index partitioning carefully

Tuning with Advanced Indexing Techniques

BITMAP OPERATIONS

- **Bitmap Conversion To ROWID** – Converts the bitmaps to the corresponding ROWIDs
- **Bitmap Conversion From ROWID** – Generates a bitmap index from b-tree
- **Bitmap Conversion Count** – Calculates the count by using the index
- **Bitmap Index Single Value** – Gets a single value by using the index
- **Bitmap Range Scan** – Performs a range scan over the bitmap index
- **Bitmap Full Scan** – Reads the whole bitmap to return the result
- **Bitmap Merge Scan** – Merges multiple bitmaps (result of a range scan) into one bitmap
- **Bitmap AND** – Performs an AND operation over the bits of two bitmaps
- **Bitmap OR** – Performs an OR operation over the bits of two bitmaps
- **Bitmap Minus** – Performs an AND operation between a bitmap and the negated version of another bitmap
- **Bitmap Key Iteration** – Takes each row from a table row source, finds the corresponding bitmaps and merges them

Tuning with Advanced Indexing Techniques

COMPOSITE INDEXES & ORDER OF INDEXED COLUMNS

- Composite indexes are the ones created for multiple columns
- Advantages of composite indexes :
  - Higher Selectivity
  - Less I/O
  - Can be used for one or multiple columns

- Selecting column order in composite indexes is important!
- Wrong column order will lead a worse plan or not the use the index!
- Select the columns in order of mostly queried & most selective

Tuning with Advanced Indexing Techniques

COVERING INDEXES

- An index including all the columns of the query is called covering index for that query
- Benefits of covering indexes:
  - There is no need to look up the data in the table
  - Needs less I/O operations
- Drawbacks of covering indexes:
  - Increase the index size
  - Will be used for fewer queries
  - Maintenance cost increases

Tuning with Advanced Indexing Techniques

REVERSE KEY INDEXES

- Simultaneous inserts/updates on the indexed tables may have performance problems because of the index maintenance (especially for the sequential values)
- Sequential value inserts may cause contention in the index blocks with some waits or locks
- Reverse key index is not an index used by reverse function on the indexed column!
- Reverse key indexes store the bytes of the indexed columns in reverse order (ROWID's are not reversed)
- Reversing the bytes will lead the database to store them in different index blocks
- Drawbacks of reverse key indexes :
  - It works with only the equality searches
  - It uses more CPU to reverse the key values

```
CREATE INDEX ix ON temp(a,b) REVERSE;
```

Tuning with Advanced Indexing Techniques

BITMAP JOIN INDEXES

- We can create bitmap join indexes over two or more tables
- Bitmap Join Indexes need less space than materialized views.

```
CREATE BITMAP INDEX sales_temp_bjx ON sales(P.prod_subcategory, C.cust_city)
FROM sales S, products P, customers C
WHERE S.prod_id = P.prod_id
AND S.cust_id = C.cust_id
LOCAL;
```

- Maintenance Cost is higher
- Only one table among the indexed tables can be updated concurrently by different transactions
- Parallel DML is only supported on the fact table
- The joined columns of dimension table needs to have a unique or primary key constraint
- If Dimension table has a multi-column primary key, each column of that PK must be in the join
- No table can be joined twice in the index
- Bitmap join indexes cannot be created on temporary tables

Tuning with Advanced Indexing Techniques

COMBINING BITMAP INDEXES

```
SELECT * FROM customers_temp
WHERE cust_city IN ('Aachen','Abingdon','Bolton','Santos')
AND cust_first_name = 'Abigail';
```

1	0	1	1	1	0	0	1	0	1	0	0
1	1	0	0	1	1	0	1	1	0	1	1
1	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**

AND

1	0	1	1	1	0	0	1	0	1	0	0
1	1	0	0	1	1	0	1	1	0	1	1
1	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**

OR

Tuning with Advanced Indexing Techniques

FUNCTION-BASED INDEXES

```
CREATE INDEX emp_last_name_fix ON employees(UPPER(Last_name));
```

- Function-based index stores the result of the function for each row
- Using functions over the columns will suppress the index usages except for the function-based indexes
- We can use any type of functions (built-in or user-defined)
- We can use multiple functions in a column or in multiple columns
- The restrictions:
  - The function needs to be deterministic
  - Aggregate functions cannot be used
  - Function needs to have a fixed-length data type

Tuning with Advanced Indexing Techniques

INDEX - ORGANIZED TABLES

- Store the non-key columns as well, in the index leaves
- There is not a table in addition to an index. Just the index.
- Store the rows in the order of primary key values
- It reads faster than the ordinary indexes over the primary key values
- The changes are only over the index (since there is no table)
- IT needs less storage (no duplicate columns | rows)
- Have full-functionality of ordinary tables (all the objects including indexes can be created over index-organized tables)
- Primary keys can be composite key

Tuning with Advanced Indexing Techniques

Restrictions & Disadvantages of IOTs (Index-Organized Tables) :

- Cannot create with a bitmap index
- Needs to have a unique primary key
- Can have max 100 columns (255 in index portion – rest in the overflow segment)
- Primary keys can have max 32 columns
- Cannot contain virtual columns
- PCTTRESHOLD size cannot be larger than 50% of the index block
- Faster in updates but slower in inserts
- There is no physical rowid in IOTs. There are logical rowids
- Secondary indexes use logical rowids which makes it work slower

Tuning with Advanced Indexing Techniques

INDEX - ORGANIZED TABLES

When to use IOTs?

- If the where clauses mostly have the primary key column, but select clause queries for other columns as well
- Queries returning for small number of rows
- If the table data is not so often changing
- If you don't need additional indexes over the IOTs
- If the table is small in both row count and column count
- If an index already needs the majority of the columns

Tuning with Advanced Indexing Techniques

INDEX - ORGANIZED TABLES

```
CREATE TABLE customers_iot (cust_id  NUMBER,
                            cust_first_name VARCHAR2(20),
                            cust_last_name  VARCHAR2(40),
                            cust_gender     CHAR(1),
                            cust_year_of_birth      NUMBER(4,0),
                            cust_marital_status     VARCHAR2(20),
                           CONSTRAINT cid_pk PRIMARY KEY (cust_id))
    ORGANIZATION INDEX
    TABLESPACE iot_tbs
    PCTTHRESHOLD 20
    INCLUDING cust_year_of_birth
    OVERFLOW TABLESPACE iot_tbs2;
```

Tuning with Advanced Indexing Techniques

CLUSTER INDEXES

- We can create indexes over the clusters
- We cannot create indexes for every clusters
- We can create indexes for the index clusters
- We cannot create indexes for hash-type clusters
- Default cluster type is index cluster
- We cannot make DML operations over the index-clustered tables before the index is created
- Cluster indexes are stored in the index segment
- Cluster indexes store the null values
- Cluster indexes have entries for each cluster key value
- Index Clusters cannot be used without the indexes

```
CREATE INDEX emp_dept_index
ON CLUSTER emp_dep_cluster
TABLESPACE USERS
STORAGE (INITIAL 250K NEXT 50);
```

Tuning with Advanced Indexing Techniques

INVISIBLE INDEXES

- Reasons to make an index invisible :
  - Compare the performance with the new one before changing it
  - Check if dropping that index results in some problems
- Invisible indexes are also alive like the visible indexes. But the optimizer ignores them
- OPTIMIZER_USE_INVISIBLE_INDEXES parameter can be set to TRUE to make the optimizer use the invisible indexes
- Invisible indexes are maintained by the database

```
SELECT * FROM user_indexes WHERE visibility = 'INVISIBLE';
```

```
ALTER INDEX ix INVISIBLE;
```

```
ALTER INDEX ix VISIBLE;
```

```
CREATE INDEX ix ON TEMP(a,b) INVISIBLE;
```

Tuning with Advanced Indexing Techniques

INDEX KEY COMPRESSION

- If there are lots of redundant data in the index, it might be useful to compress it (Will decrease the size and may increase the performance)
- Can be applied to unique and non-unique indexes
- It will work better for non-unique indexes.
- It eliminates the duplicate keys
- Composite indexes can be compressed by the first N keys

```
CREATE INDEX ix ON temp(a,b,c) COMPRESS [N];
```

```
CREATE INDEX ix ON temp(a,b,c) COMPRESS ADVANCED HIGH|LOW;
```

Tuning with Advanced Indexing Techniques

INDEX KEY COMPRESSION

- Things to know about index compression:
  - Column order is important
  - Bitmap indexes cannot be compressed
  - Partitioned indexes cannot be compressed before 11g version
  - Can be alternative to bitmap indexes in some cases
  - Indexes are created as non-compressed by default

Tuning with Advanced Indexing Techniques

FULL-TEXT SEARCH INDEXES

- CONTEXT Type Index :
  - Performs search operations over large documents (PDF, MS Word, XML, HTML, Plain Text)
  - Converts the words in the documents into tokens
  - Documents are stored in BLOB or CLOB type columns

```
CREATE TABLE my_doc (
                    id       NUMBER(10)     NOT NULL,
                    name     VARCHAR2(200)  NOT NULL,
                    document BLOB           NOT NULL);
```

```
CREATE INDEX my_doc_idx ON my_docs(document) INDEXTYPE IS CTXYS.CONTEXT;
```

```
SELECT name FROM my_doc WHERE CONTAINS(document, 'Search Text') > 0;
```

```
SELECT SCORE(1) score, name FROM my_doc
       WHERE CONTAINS(document, 'Search Text', 1) > 0
       ORDER BY SCORE(1) DESC;
```

  - Score Operator returns the relevance score of the row for the specified search text

Tuning with Advanced Indexing Techniques

FULL-TEXT SEARCH INDEXES

- CTXCAT Type Index :
  - Ideal for smaller documents or text fragments
  - Larger than the context index and takes longer to build
  - It creates indexes over the index sets

```
EXEC CTX_DDL.CREATE_INDEX_SET('products_iset');
```

```
EXEC CTX_DDL.ADD_INDEX('products_iset','prod_list_price');
```

```
CREATE INDEX my_products_name_idx ON products (prod_desc) INDEXTYPE IS CTXSYS.CTXCAT
PARAMETERS ('index set products_iset');
```

```
SELECT prod_id, prod_list_price, prod_name, prod_subcategory
FROM   products
WHERE CATSEARCH(prod_desc, 'CD', 'prod_list_price>10') > 0;
```

Tuning with Advanced Indexing Techniques

FULL-TEXT SEARCH INDEXES

- CTXRULE Type Index :
  - Used to build a document classification application
  - Documents inside of the table are classified based on their contets
  - Used for the category searches

```
CREATE INDEX temp_rule ON tem_table(text) INDEXTYPE IS CTXSYS.CTXTULE;
```

```
SELECT CLASSIFICATION FROM temp_table
WHERE matches(text, 'Lionel Messi is a famous footballer from Argentina') > 0;
```

- Synchronizing the Full-Text Indexes

```
EXEC CTX_DDL.SYNC_INDEX('my_docs_idx');
```

```
EXEC CTX_DDL.OPTIMIZE_INDEX('my_docs_idx','FULL');
```

Tuning with Advanced Indexing Techniques

WHY MY INDEX ISN'T USED?

- Indexed columns are not used in the where clause
- First column of the index is not used in the where clause
- Index may not be selective enough
- Leading wildcard(%) characters
- Selectivity of the LIKE clause is low
- Using functions on the indexed columns
- Implicit data type conversion
- The column can contain NULL values
- Invalid hint usage
- Outdated database statistics

Tuning with Advanced Indexing Techniques

OVERALL TIPS FOR INDEXING

- Create the indexes after inserting the table data
- Create indexes for correct tables and columns
- Order the indexed columns carefully
- Do not keep the unncessary indexes
- Consider creating and using indexes in parallel
- Consider creating indexes with NOLOGGING
- Do not drop or disable the constraints having an index without thinking the indexing cost
- Consider coalescing or rebuilding the index

Advanced Tuning Techniques

USING BIND VARIABLES

- Using the BIND Variables may increase the performance by decreasing the parse counts

```
SELECT AVG(salary) FROM employees WHERE department_id = 30;
SELECT AVG(salary) FROM employees WHERE department_id = 40;
SELECT AVG(salary) FROM employees WHERE department_id = 50;

SELECT sql_id,executions,parse_calls,first_load_time,last_load_time,sql_text
FROM v$sql
WHERE sql_text LIKE '%avg(salary) from employees%'
ORDER BY first_load_time DESC;

SELECT AVG(salary) FROM employees WHERE department_id = :b;
```

Tuning with Advanced Indexing Techniques

BEWARE OF BIND VARIABLE PEEKING

- The optimizer peeks the bind variable values for the first execution
- After the first plan is generated, it uses that plan for the next executions
- This may cause the optimizer to select suboptimal plans for the next executions
- Why not it peeks for all the values?
  - To eliminate the hard parses
- When to use the bind variables then?
  - Don't use the bind variables if the cardinality of the values in the column is pretty different
  - If the cardinalities are pretty similar and they all will need the same plan, use bind variables

Tuning with Advanced Indexing Techniques

CURSOR SHARING

- The lifecycle of a query:
  - Open : Allocates memory for that cursor
  - Parse : Syntax analysis, semantic analysis, privilege checks etc
  - Bind : Bind variable values are assigned
  - Define : Defines how you want to see the data
  - Execute
  - Fetch
- The data structure allocated in the database for that query is called as cursor in the server side
- Using these cursors by multiple executions is called as cursor sharing
- Parent cursor stores the SQL statement and child stores the information related to the differences

Tuning with Advanced Indexing Techniques

CURSOR SHARING

- When the database can share the cursors?
  - When Bind variables are used
  - Only if the literals are different
- CURSOR_SHARING parameter should be set to :
  - EXACT : The default cursor_sharing parameter. It allows cursor sharing only if the queries are exactly the same.
  - FORCE : Allows cursor sharing if everything but literals are the same. But it is not guaranteed.
    - Needs extra work to find a similar statements in the shared pool during the soft parse
    - It needs to use more memory
    - Star transformation is not supported
- The cursor sharing can be set by alter session or alter system commands

Tuning with Advanced Indexing Techniques

ADAPTIVE CURSOR SHARING

- The main goal of adaptive cursor sharing is, not to have a new cursor for each bind value, but not to use the same cursor for every query also.
- Enabled by default (It is applied automatically if the query does not have over 14 bind variables)
- It is independent of CURSOR_SHARING parameter
- Benefits of adaptive cursor sharing :
  - It automatically detects if the query needs another execution plan or can use the existing one
  - Decreases the number of generated child cursors to minimum
  - It works automatically. You don't need to start it

Tuning with Advanced Indexing Techniques

ADAPTIVE CURSOR SHARING

Bind-Sensitive Cursor

```
SELECT * FROM customers_temp WHERE country_id = :country_id;
```

Bind-Aware Cursor

1 — 52787 – 0.151217

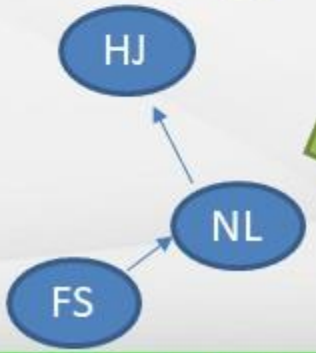
- Lowest Selectivity : 0.126162
- Highest Selectivity : 0.367063

2 — 52790 – 0.001602

- Lowest Selectivity : 0.001476
- Highest Selectivity : 0.001804

3 — 52790 – 0.001602

- Lowest Selectivity : 0.001476
- Highest Selectivity : 0.001804



Tuning with Advanced Indexing Techniques

ADAPTIVE CURSOR SHARING

- Useful views for adaptive cursor sharing :
  - V$SQL – Stores if the query is bind sensitive or bind aware
  - V$SQL_CS_SELECTIVITY – Stores the lowest and highest acceptable selectivity values
  - V$SQL_CS_STATISTICS – Stores some extra info like buffer gets, CPU time, etc.
  - V$SQL_CS_HISTOGRAM – Stores the histogram statistics of the queries using bind variables
- Hints about adaptive cursor sharing :
  - BIND_AWARE – Makes the database skip monitoring that query to check bind-sensitivity
  - NO_BIND_AWARE – Makes the database ignore that query for bind-sensitiveness

Tuning with Advanced Indexing Techniques

ADAPTIVE PLANS

- The statistics used by the optimizer :
  - Table Statistics :
    - Number of rows
    - Number of blocks
  - Column Statistics
    - Number of distinct values in that column
    - Number of NULL values in that column
    - Data distribution statistics (Histograms)
    - Extended statistics
  - Index Statistics
    - Number of leaf blocks
    - Number of branch levels
    - Number of distinct keys
    - Index clustering factor
  - System Statistics
    - I/O performance
    - CPU performance

The statistics are gathered by the optimizer by using the DBMS_STATS package

Tuning with Advanced Indexing Techniques

ADAPTIVE PLANS

- Before database version 12c the execution plan was determined before the execution and this plan was applied
- Starting with 12c the optimizer can change the plan on runtime.
- While executing the query, the statistics collector gathers some new statistics about cardinality and histograms
- If the new statistics do not match with the first statistics, the optimizer picks one of its sub-plans it stored
- It writes 'This is an adaptive plan' on the execution plan to express that
- It is enabled by default

Tuning with Advanced Indexing Techniques

DYNAMIC STATISTICS (DYNAMIC SAMPLING)

- Starting with 10g, dynamic sampling is introduced. It allows the optimizer to gather additional information at the parse time.
- Dynamic sampling gathers the statistics by some recursive calls before generating the plan
- Scans a fraction of random samples from the table blocks and calculates the statistics based on these random blocks
- You can control the dynamic sampling with :
  - ALTER SYSTEM SET OPTIMIZER_DYNAMIC_SAMPLING = 4;
  - ALTER SESSION SET OPTIMIZER_DYNAMIC_SAMPLING = 9;
  - /*+ DYNAMIC_SAMPLING(11) */
- Before 12c, the dynamic sampling level can be set to between 0 to 10 (Default is 2)
- Dynamic sampling is renamed to dynamic statistics in 12 and beyond.
- New level 11 has automatic sampling

Tuning with Advanced Indexing Techniques

DYNAMIC STATISTICS (DYNAMIC SAMPLING)

- Why to use dynamic statistics?
  - If the current statistics are not enough to create an optimal plan
  - If the query is executed multiple times
  - If the time for gathering the dynamic statistics is ignorable compared to the overall execution time
- When to use dynamic statistics?
  - Statistics are missing
  - Statistics are stale
  - Statistics are insufficient
  - There is a parallel execution
  - There is a SQL Plan directive

Tuning with Advanced Indexing Techniques

DYNAMIC STATISTICS (DYNAMIC SAMPLING)

Dynamic Sampling Level	Estimated Rows	Actual Rows
No Hint	90	932
Level **0**	90	932
Level **1**	90	932
Level **2**	90	932
Level **3**	90	932
Level **4**	981	932
Level **5**	874	932
Level **6**	985	932
Level **7**	966	932
Level **8**	943	932
Level **9**	932	932
Level **10**	932	932
Level **11**	826	932