Oracle SQL Tunning

Filipe Matos

oracle@filipematos.com

linkedin.com/in/filipe-matos-b1a29a26/

AGENDA

1. Efficient SQL
2. Execution Plan
3. Indexes
4. Materialized Views
5. Partitioning
6. Parallel Processing

1. EFFICIENT SQL

1. EFFICIENT SQL

STRUCTURED QUERY LANGUAGE (SQL)

The industry standard for interacting with a relational database is SQL—officially pronounced as "S-Q-L," but many still use the pronunciation "sequel." Structured Query Language (SQL) is not considered a programming language, such as VB.NET, COBOL, or Java.

SQL Command Types:

- **Query**: Retrieve data values.
  - SELECT
- **Data manipulation language (DML)**: Create or modify data values.
  - INSERT, UPDATE, DELETE
- **Data definition language (DDL)**: Define data structures.
  - CREATE, ALTER, DROP
- **Transaction control (TC)**: Save or undo data value modifications.
  - COMMIT, ROLLBACK
- **Data control language (DCL)**: Set permissions to access database structures.
  - GRANT, REVOKE

1. EFFICIENT SQL

Oracle Server Architecture

An Oracle database is a set **of files on disk**. It exists until these files are deliberately deleted.

Access to the database is through the Oracle instance. The instance is a set of processes and memory structures: it exists **on the CPU(s)** and in the memory of the server node, and this existence is temporary.

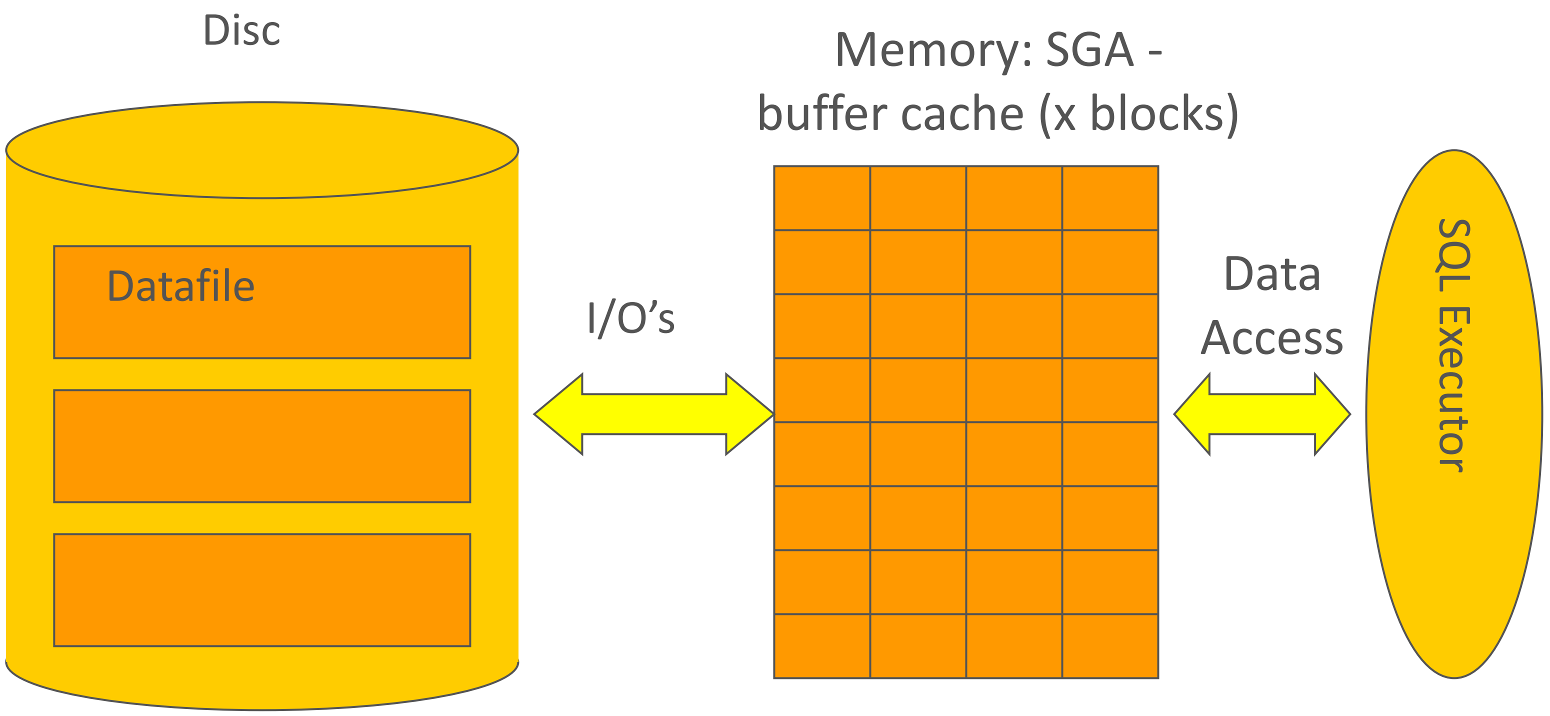
The processing model implemented by the Oracle server is that **of client-server processing**, often referred to as two-tier. In the client-server model, the generation of the user interface and much of the application logic is separated from the management of the data

1. EFFICIENT SQL

Storage I/O

I/O's are done at 'block level'

- LRU list controls who 'makes place' in the cache

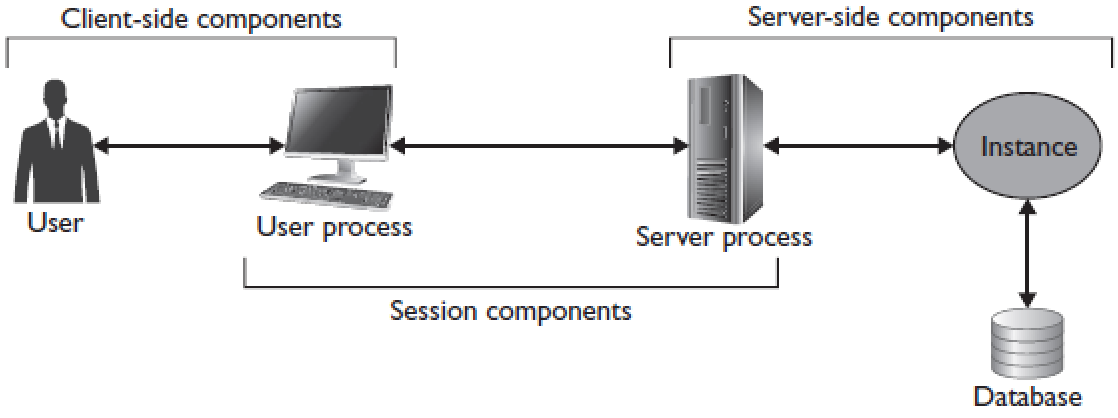


1. EFFICIENT SQL

Oracle Server Architecture

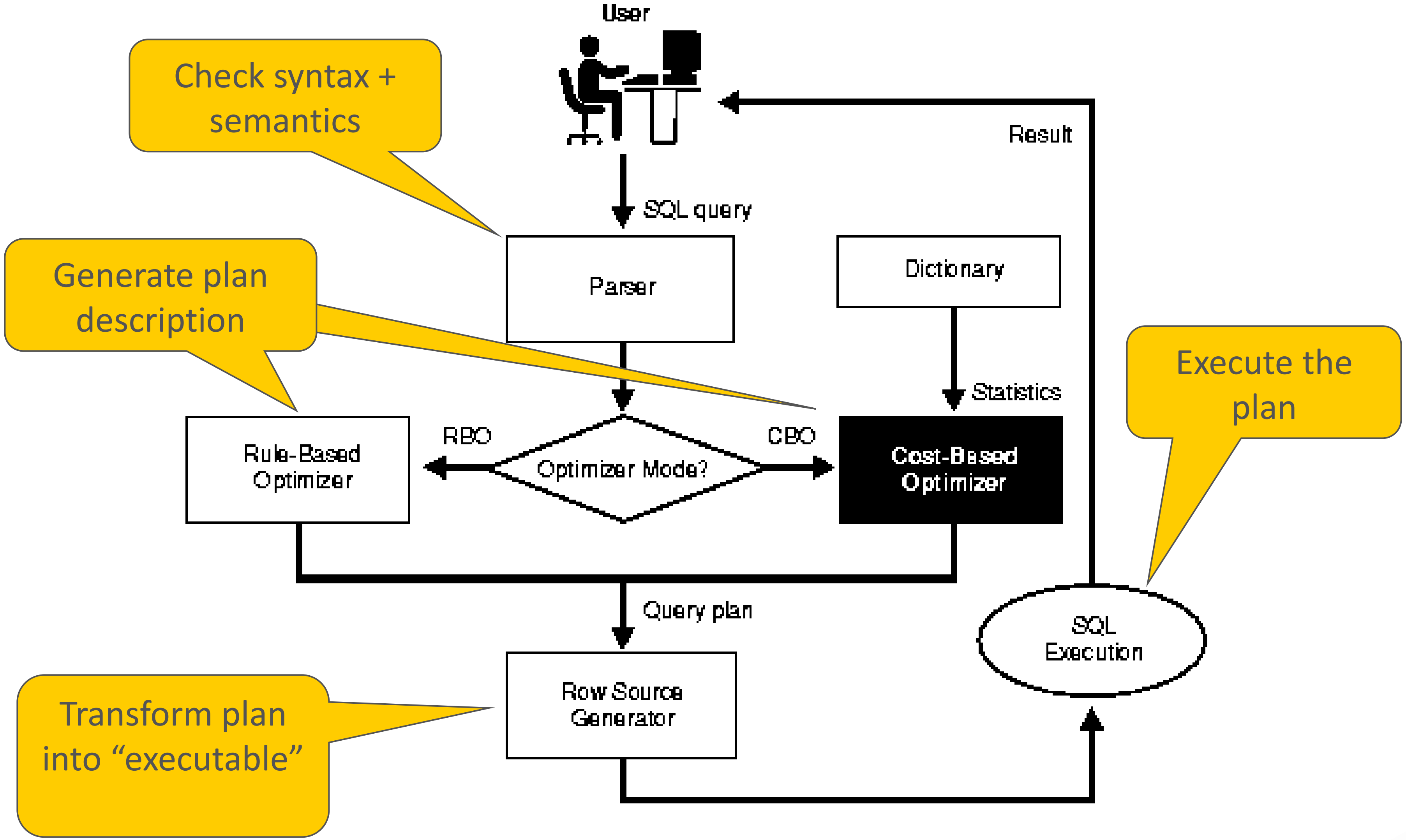
The client tier consists of two components: the **users and the user processes**. The server tier has three components: the server processes that execute the SQL, the instance, and the database itself. Each user interacts with a user process. Each user **process interacts with a server process**, usually across a local area network.

A session is a user process in communication with a server process. There will usually be one user process per user and one server process per user process. The user and server processes that make up sessions are launched on demand by users and terminated when no longer required.



1. EFFICIENT SQL

Optimizer Overview



1. EFFICIENT SQL

Cost vs. Rule

Rule

- Hardcoded heuristic rules determine plan
  - “Access via index is better than full table scan”
  - “Fully matched index is better than partially matched index”
  - …

Cost (2 modes)

- Statistics of data play role in plan determination
  - Best throughput mode: retrieve **all rows** asap
    - First compute, then return fast
  - Best response mode: retrieve **first row** asap
    - Start returning while computing (if possible)

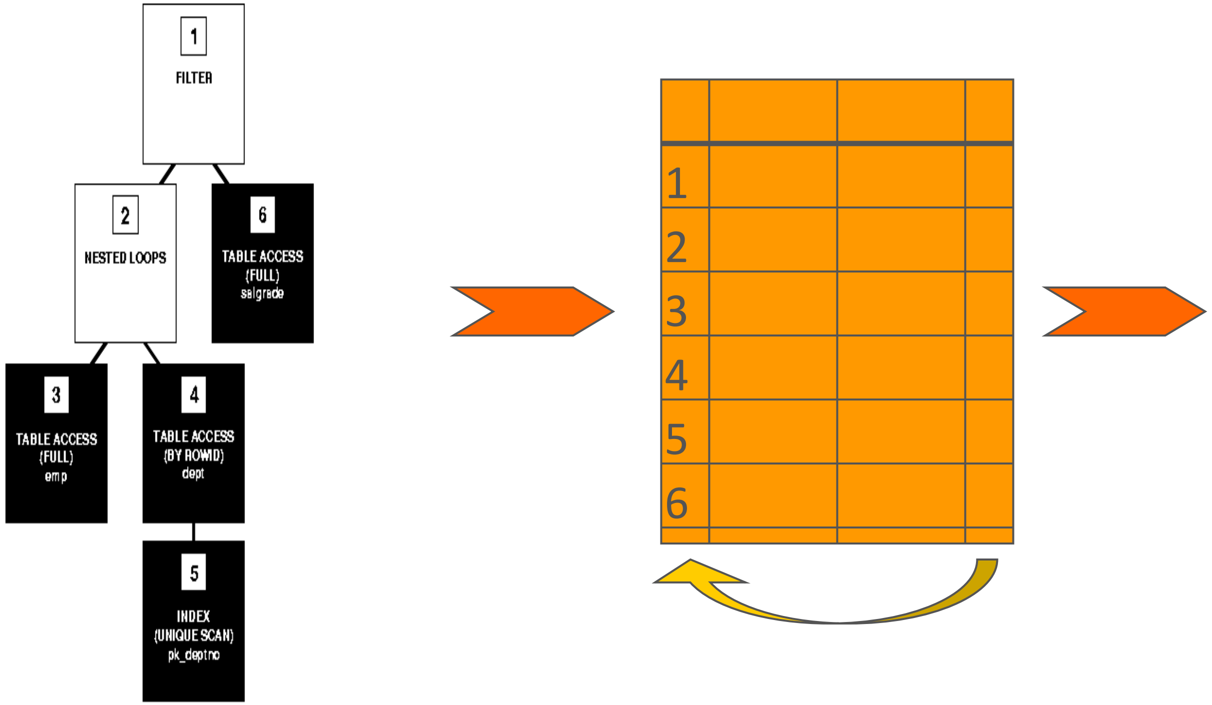
2. Execution Plan

2. Execution Plan

Explain Plan Utility

Explain plan for <SQL-statement>

- Stores plan (row-sources + operations) in Plan_Table
- View on Plan_Table (or 3rd party tool) formats into readable plan



>Filter
>....NL
>........TA-full
>........TA-rowid
>...........Index Uscan
>....TA-full

2. Execution Plan

Explain Plan Utility

```
create table PLAN_TABLE (
  statement_id    varchar2(30),   operation       varchar2(30),
  options         varchar2(30),   object_owner    varchar2(30),
  object_name     varchar2(30),   id              numeric,
  parent_id       numeric,        position        numeric,
  cost            numeric,        bytes           numeric);
```

```
create or replace view PLANS(STATEMENT_ID,PLAN,POSITION) as
select statement_id,
  rpad('>',2*level,'.')||operation||
  decode(options,NULL,'',' (')||nvl(options,' ')||
  decode(options,NULL,'',') ')||
  decode(object_owner,NULL,'',object_owner||'.')||object_name plan,
  position
from plan_table
start with id=0
connect by prior id=parent_id
       and prior nvl(statement_id,'NULL')=nvl(statement_id,'NULL')
```

2. Execution Plan

Exemplo: Single Table

```
SELECT *
FROM emp;
```

```
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

Full table scan (FTS)

- All blocks read sequentially into buffer cache
  - Also called “buffer-gets”
  - Done via multi-block I/O’s (db_file_multiblock_read_count)
  - Till high-water-mark reached (truncate resets, delete not)
- Per block: extract + return all rows
  - Then put block at LRU-end of LRU list (!)
  - All other operations put block at MRU-end

2. Execution Plan

Exemplo: Single Table

```
SELECT *
FROM emp
WHERE sal > 100000;
```

```
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

Full table scan with filtering

- Read all blocks
- Per block extract, filter, then return row
  - **Simple where-clause filters never shown in plan**
  - FTS with: rows-in < rows-out

2. Execution Plan

Exemplo: Single Table

```
SELECT *
FROM emp
ORDER BY ename;
```

```
>.SELECT STATEMENT
>...SORT order by
>.....TABLE ACCESS full emp
```

FTS followed by sort on ordered-by column(s)

- "Followed by" Ie. SORT won't return rows to its parent row-source till its child row-source fully completed
- SORT order by: rows-in = rows-out
- Small sorts done in memory (SORT_AREA_SIZE)
- Large sorts done via TEMPORARY tablespace
  - Potentially many I/O's

3 INDEXES

3. INDEXES

B-Tree Indexes

Balanced trees

- Indexed column(s) sorted and stored seperately
  - NULL values are excluded (not added to the index)
- Pointer structure enables logarithmic search
  - Access index first, find pointer to table, then access table

B-trees consist of

- Node blocks
  - Contain pointers to other node, or leaf blocks
- Leaf blocks
  - Contain actual indexed values
  - Contain rowids (pointer to rows)

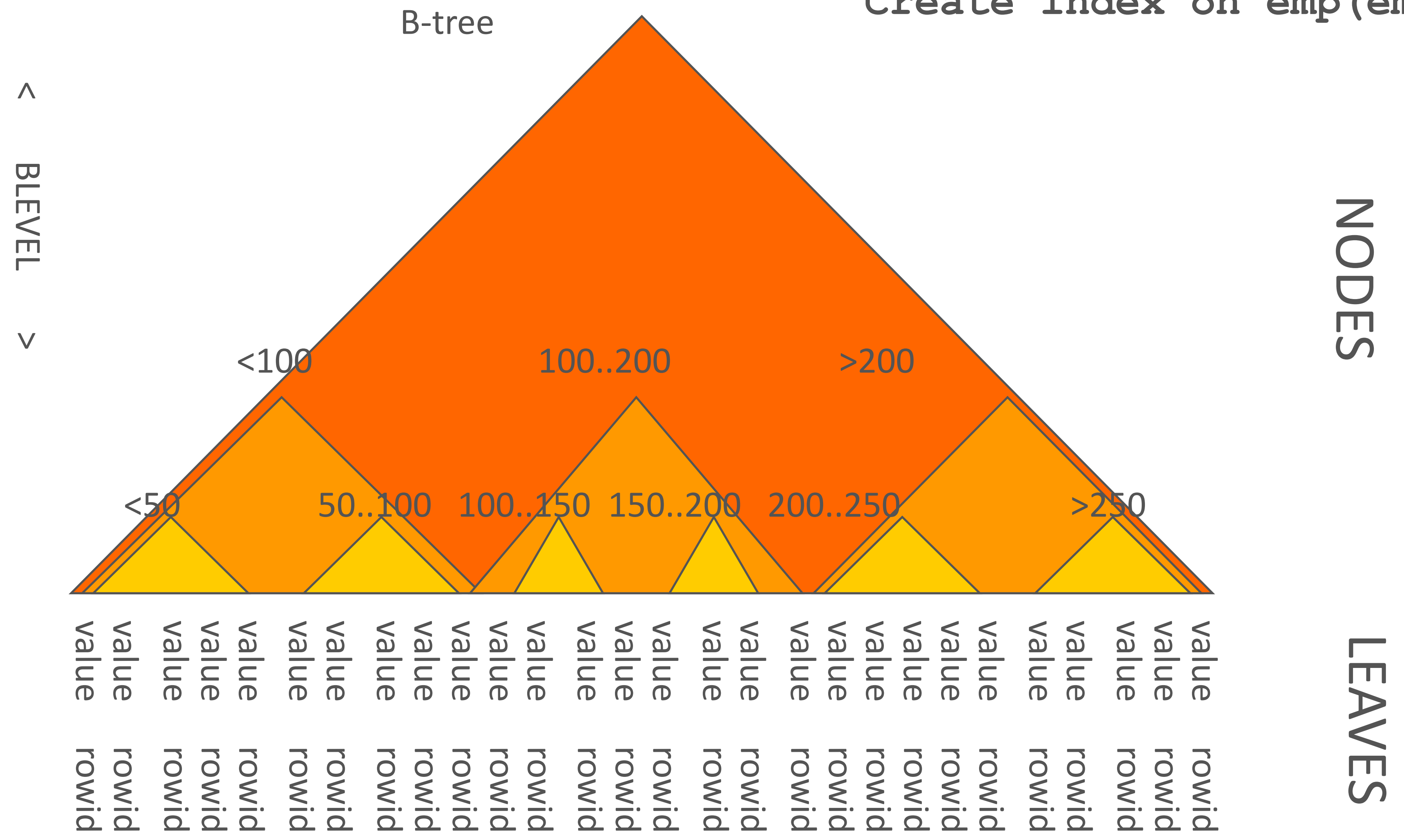
Also stored in blocks in datafiles

- Proprietary format

3. INDEXES

Data Storage

`Create index on emp(empno)`



3. INDEXES

Explain Plan

```
SELECT *
FROM emp
WHERE empno=174;

Unique emp(empno)
```

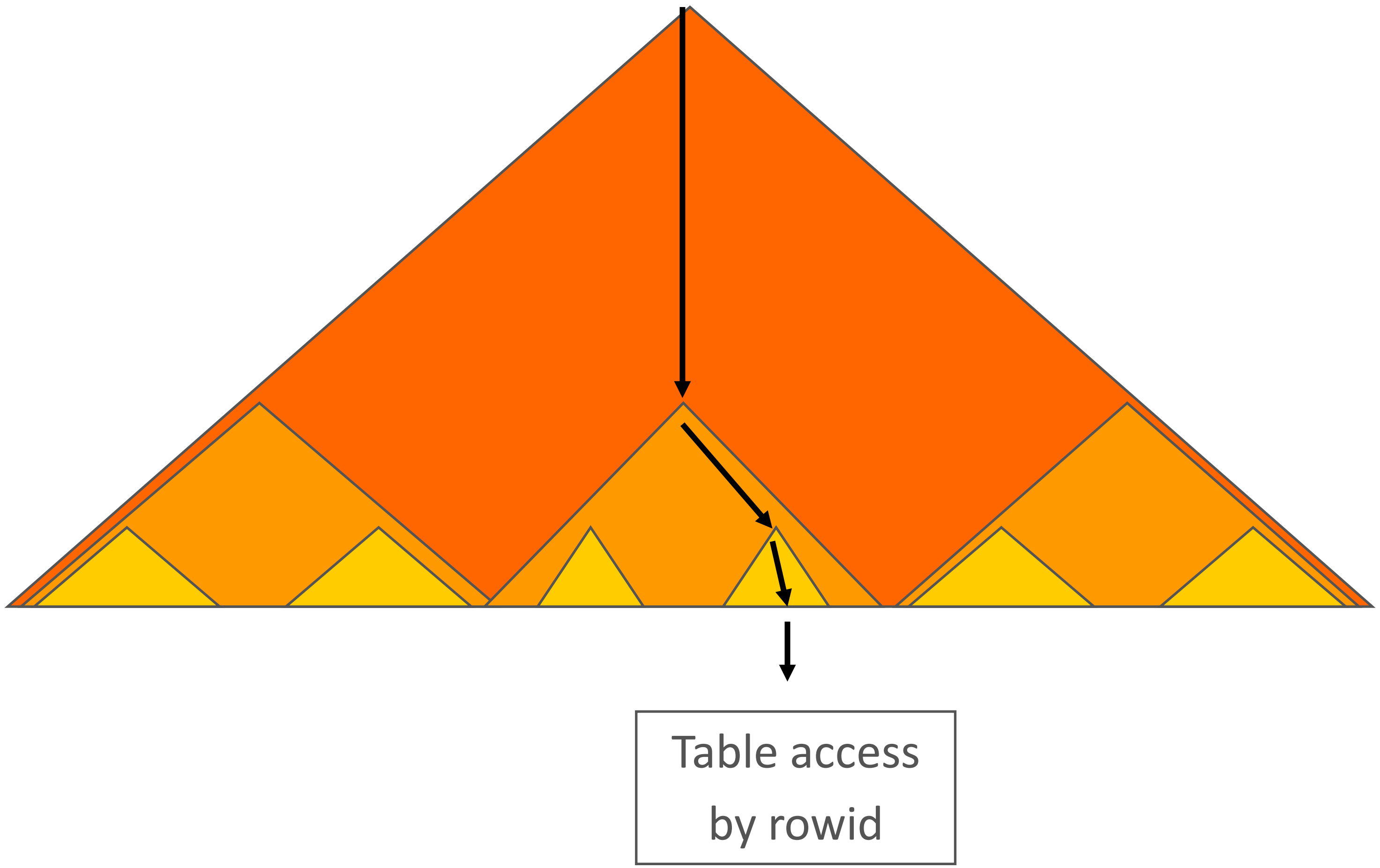
```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX unique scan i_emp_pk
```

Index Unique Scan

- Traverses the node blocks to locate correct leaf block
- Searches value in leaf block (if not found => done)
- Returns rowid to parent row-source
  - Parent: accesses the file+block and returns the row

3. INDEXES

Explain Plan



3. INDEXES

Explain Plan

SELECT *

FROM emp

WHERE job='manager';

emp(job)

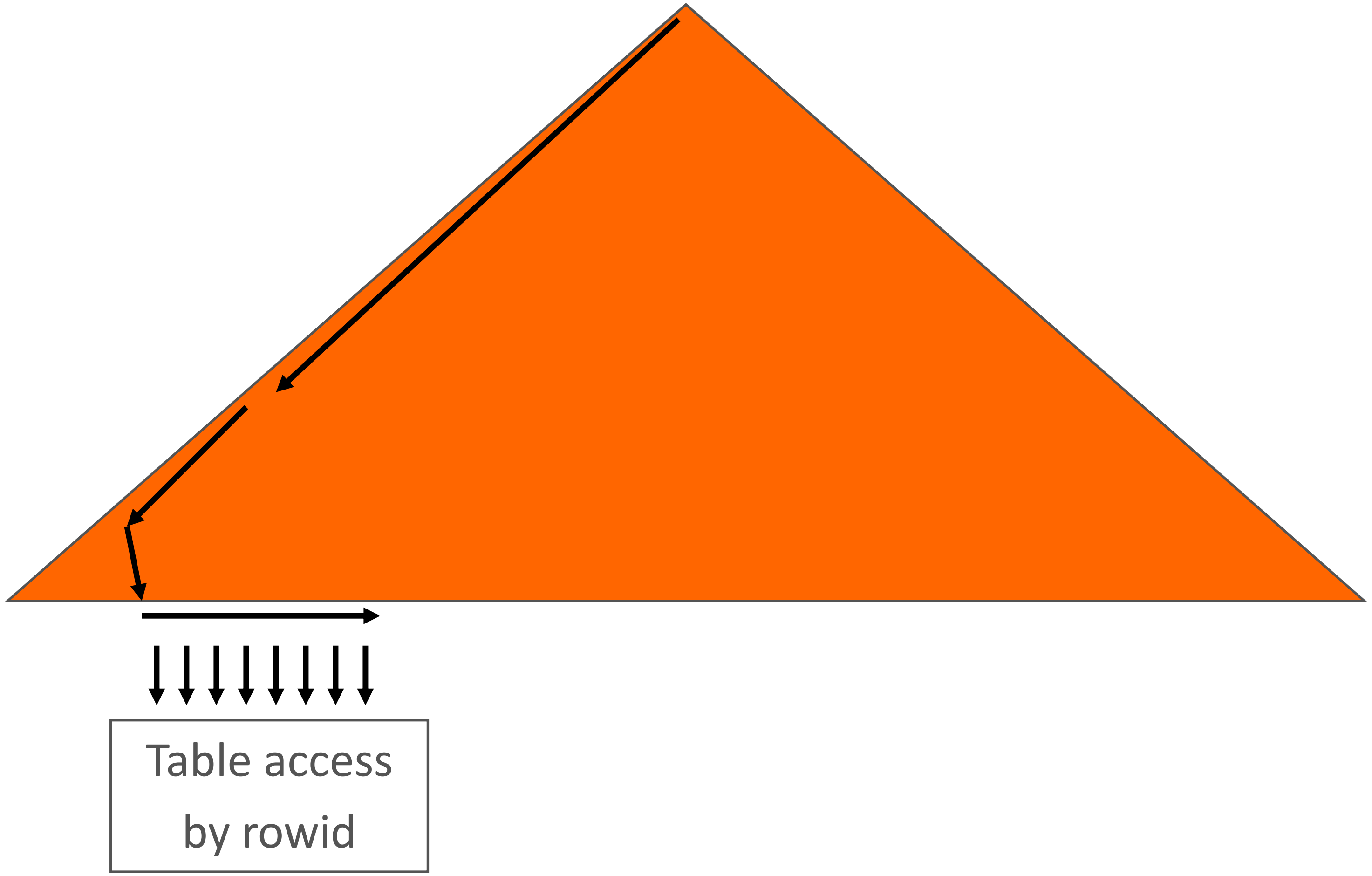
```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_job
```

(Non-unique) Index Range Scan

- Traverses the node blocks to locate most left leaf block
- Searches 1st occurrence of value in leaf block
- Returns rowid to parent row-source
  - Parent: accesses the file+block and returns the row
- Continues on to next occurrence of value in leaf block
  - Until no more occurences

3. INDEXES

Explain Plan



3. INDEXES

BitMap Indexes

Empno	Status	Region	Gender	Info
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

3. INDEXES

Data Storage

```
SELECT COUNT(*)
FROM CUSTOMER
WHERE MARITAL_STATUS = 'married‘
AND REGION IN ('central','west');
```

status = 'married'		region = 'central'		region = 'west'						
0		0		0		0		0		0
1		1		0		1		1		1
1	AND	0	OR	1	=	1	AND	1	=	1
0		0		1		0		1		0
0		1		0		0		1		0
1		1		0		1		1		1

3. INDEXES

Explain Plan

```
SELECT count(*)
FROM customer
WHERE status='M'
AND region in ('C','W');
```

```
>.......TABLE ACCESS (BY INDEX ROWID) cust
>.........BITMAP CONVERSION to rowids
>...........BITMAP AND
>.............BITMAP INDEX single value cs
>.............BITMAP MERGE
>...............BITMAP KEY ITERATION
>.................BITMAP INDEX range scan cr
```

Bitmap OR's, AND's and CONVERSION

- Find Central and West bitstreams (bitmap key-iteration)
- Perform logical OR on them (bitmap merge)
- Find Married bitstream
- Perform logical AND on region bitstream (bitmap and)
- Convert to actual rowid's
- Access table

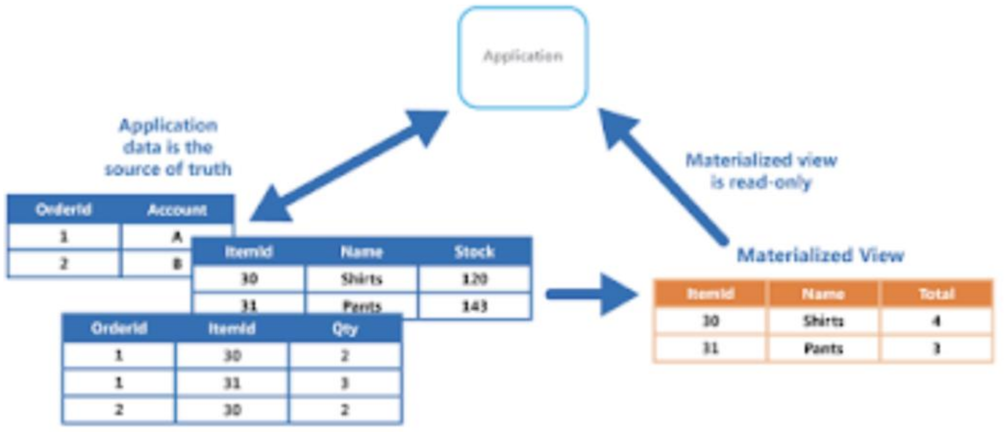
4. Materialized Views

4. MATERIALIZED VIEWS

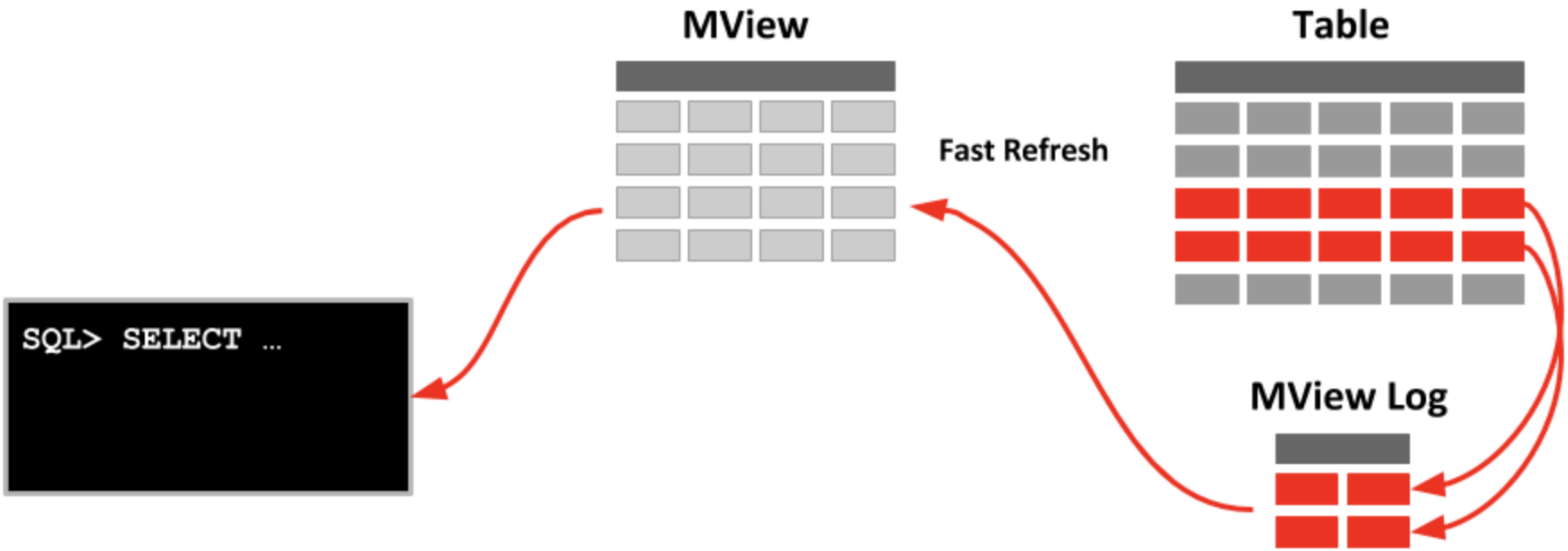
What is?

In real-world systems, materialized views are beneficial in four key areas:

- **Easing network loads**: Read-only materialized views have been a game-changer in helping distribute network loads.
- Creating a mass deployment environment.
- **Enabling data subsetting**: Data subsetting allows me to create copies that contain only portions of the entire database. I can then focus only on what information I need.
- Enabling disconnected computing: Manually refreshing my materialized view on-demand, as opposed to using a continuous data steam,



4. MATERIALIZED VIEWS
Data Flow



4. MATERIALIZED VIEWS

Example

Items

ItemId	categoryID	itemName
111	201	Cooker
222	301	Dryer
333	201	Fridge

Sales

saleId	staffId	ItemId	price
100	1	111	3000.00
200	2	333	2500.00
300	3	111	3000.00

Total sold per item?

sales_by_item

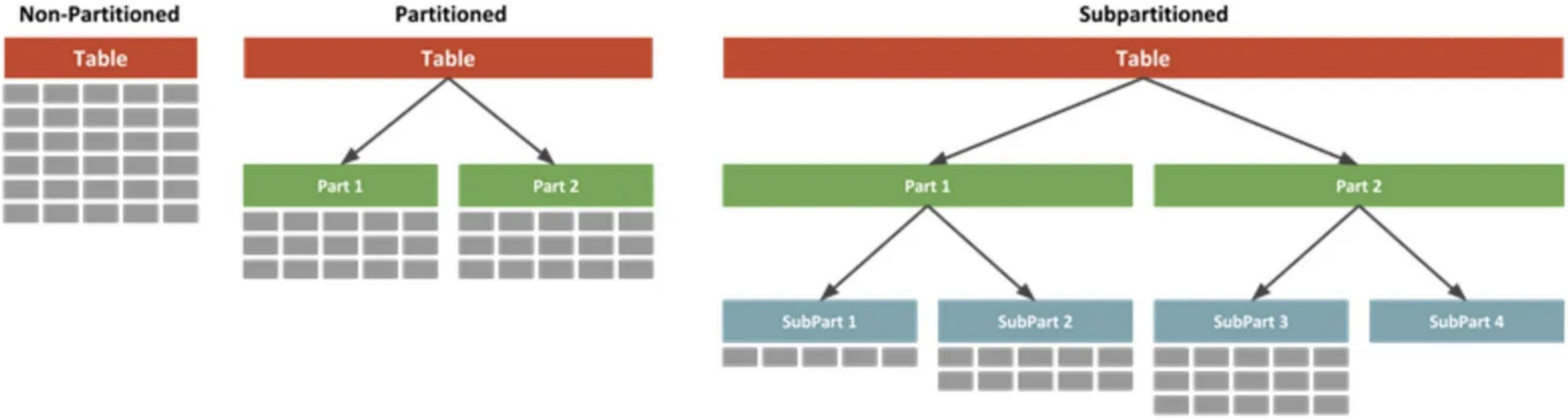
ItemId	totalSale
111	6000.00
333	2500.00

Materialized View

5. Partitioning

5. PARTITIONING

How to Partition



5. PARTITIONING

Partition Pruning

Q: What was the total sales for the weekend of May 20 - 22 2012?

```
Select sum(sales_amount)
From SALES
Where sales_date between
to_date('05/20/2012','MM/DD/YYYY')
And
to_date('05/22/2012','MM/DD/YYYY');
```

Only the 3 relevant partitions are accessed

Sales Table

- May 18th 2012
- May 19th 2012
- May 20th 2012
- May 21st 2012
- May 22nd 2012
- May 23rd 2012
- May 24th 2012

5. PARTITIONING

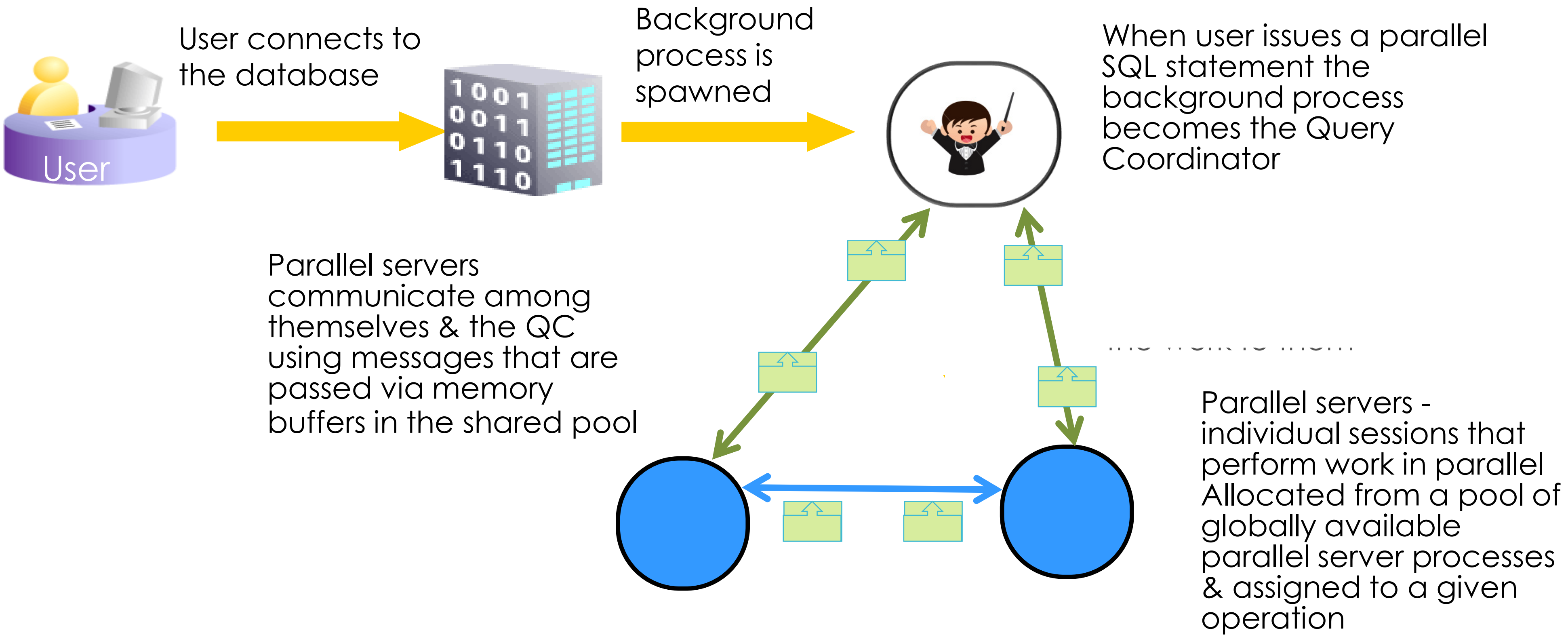
Explain Plan

```
select sum(amount_sold) from sales s, times t where t.time_id =
s.time_id and t.calendar_month_desc in
('MAR-2004','APR-2004','MAY-2004')

Plan hash value: 1350851517

-------------------------------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |       |       |       |    13 (100)|          |       |       |
|   1 |  SORT AGGREGATE           |       |     1 |    28 |            |          |       |       |
|   2 |   NESTED LOOPS            |       |     2 |    56 |    13   (0)| 00:00:01 |       |       |
|*  3 |    TABLE ACCESS FULL      | TIMES |     2 |    32 |    13   (8)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE ITERATOR|      |     2 |    24 |     0   (0)|          |   KEY |   KEY |
|*  5 |     TABLE ACCESS FULL     | SALES |     2 |    24 |     0   (0)|          |   KEY |   KEY |
-------------------------------------------------------------------------------------------------------

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

   3 - filter(("T"."CALENDAR_MONTH_DESC"='APR-2004' OR "T"."CALENDAR_MONTH_DESC"='MAR-2004'
              OR "T"."CALENDAR_MONTH_DESC"='MAY-2004'))
   5 - filter("T"."TIME_ID"="S"."TIME_ID")


26 rows selected.
```

6 PARALLEL EXECUTION

6. PARALLEL EXECUTION

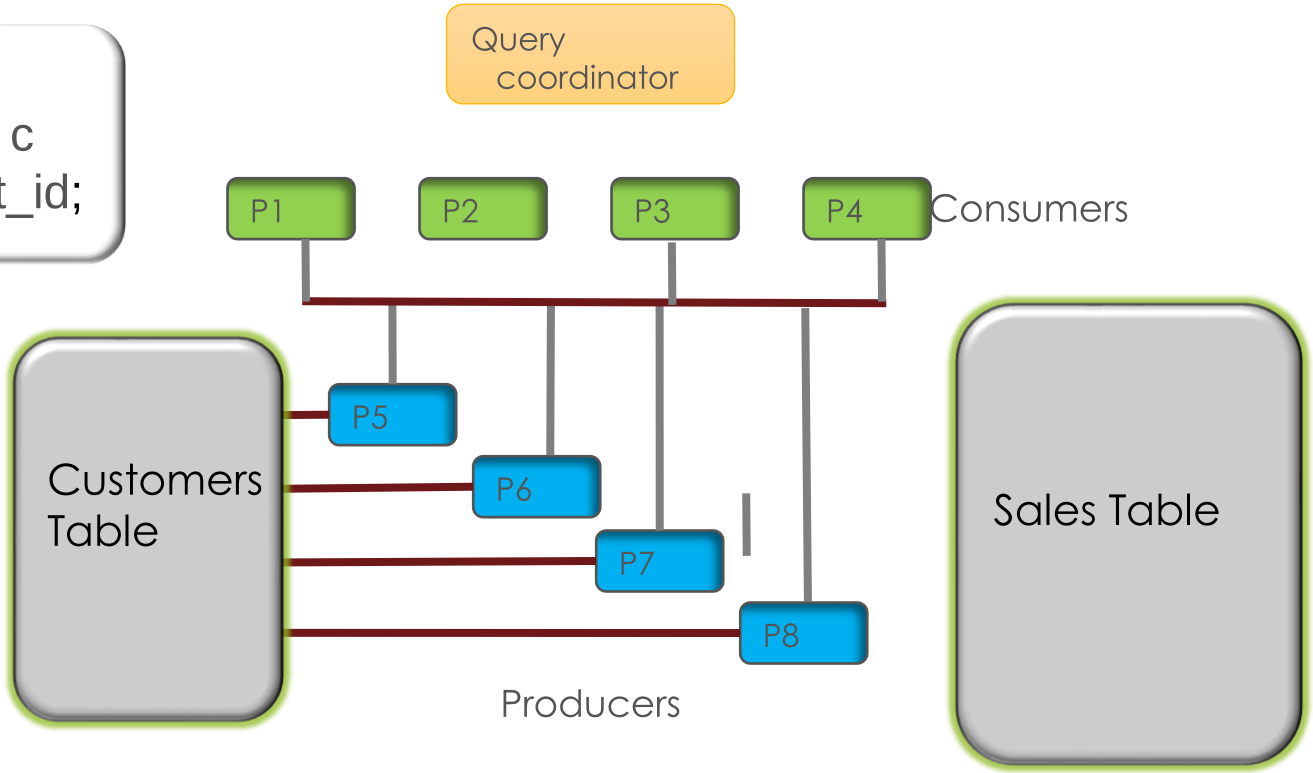
How parallel execution works



6. PARALLEL EXECUTION

Execution Flow

SELECT ………..
FROM sales s, customers c
WHERE s.cust_id = c.cust_id;



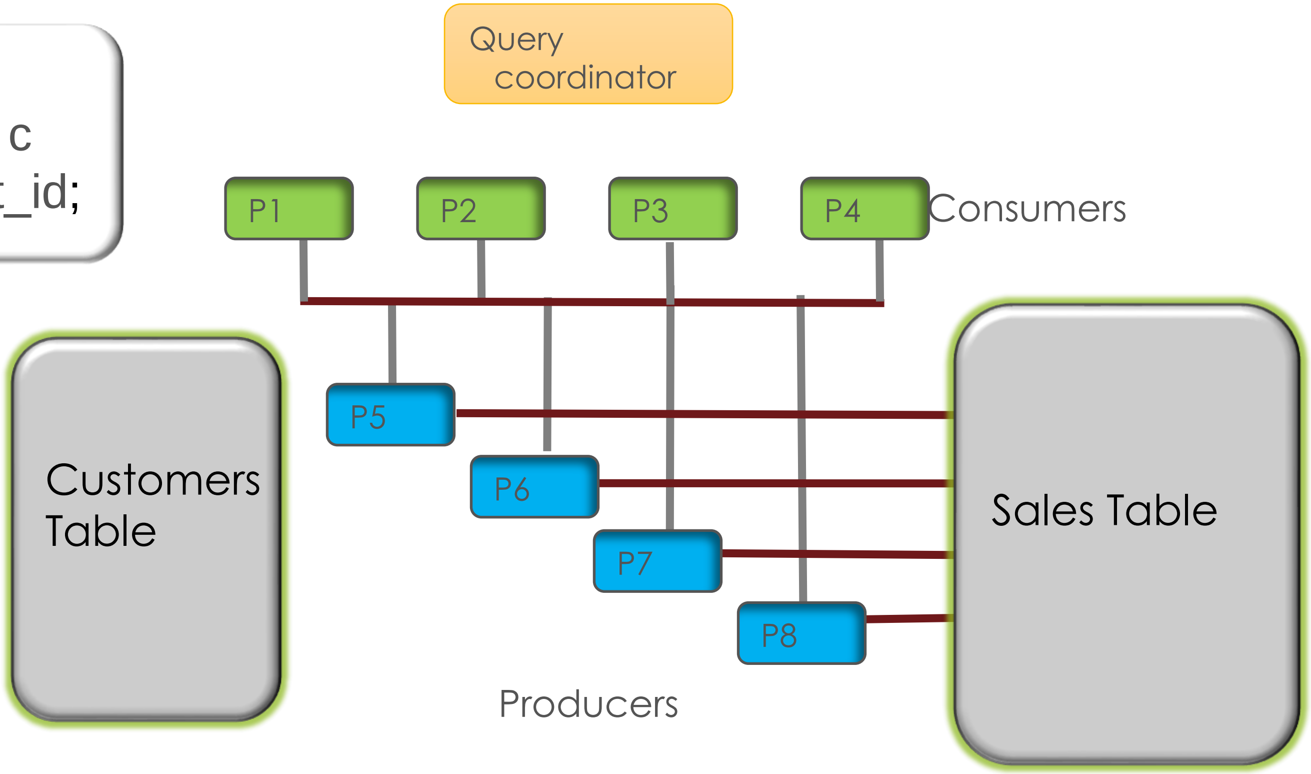
Hash join always begins with a scan of the smaller table. In this case that's is the customer table. The 4 producers scan the customer table and send the resulting rows to the consumers

6. PARALLEL EXECUTION

Execution Flow

```
SELECT ………..
FROM sales s, customers c
WHERE s.cust_id = c.cust_id;
```

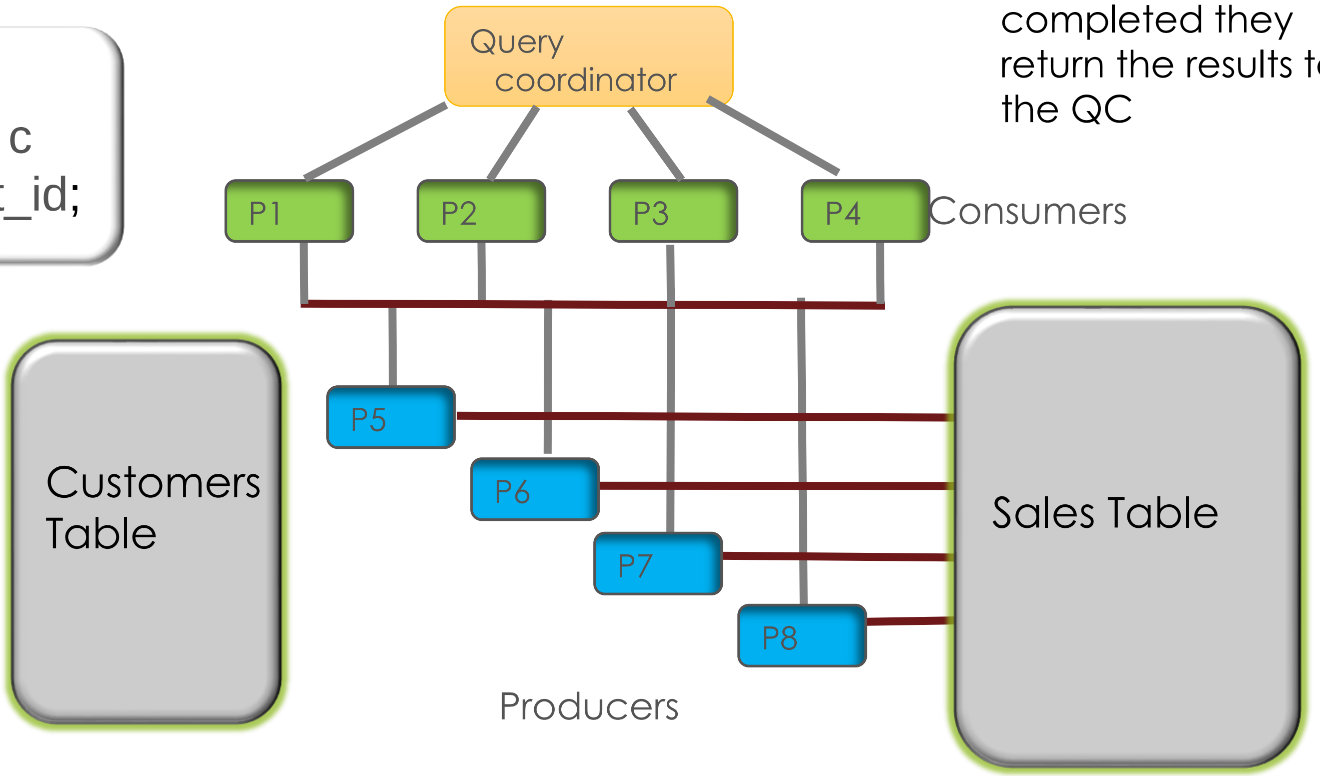
Once the 4 producers finish scanning the customer table, they start to scan the Sales table and send the resulting rows to the consumers



6. PARALLEL EXECUTION
Execution Flow

Once the consumers receive the rows from the sales table they begin to do the join. Once completed they return the results to the QC

```
SELECT ………..
FROM sales s, customers c
WHERE s.cust_id = c.cust_id;
```



6. PARALLEL EXECUTION

Explain Plan

```sql
SELECT c.cust_last_name, s.time_id, s.amount_sold
FROM sales s, customers c
WHERE s.cust_id = c.cust_id;
```

Query Coordinator

```
--------------------------------------------------------------------------------------
| Id  | Operation               | Name       | Rows   | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |            |        |       |   311 (100)|          |
|   1 |  PX COORDINATOR         |            |        |       |            |          |
|   2 |   PX SEND QC (RANDOM)   | :TQ10002   |  1049K|    31M|   311    (2)| 00:00:04 |
|*  3 |    HASH JOIN BUFFERED   |            |  1049K|    31M|   311    (2)| 00:00:04 |
|   4 |     PX RECEIVE          |            | 55500 |   704K|   112    (0)| 00:00:02 |
|   5 |      PX SEND HASH       | :TQ10000   | 55500 |   704K|   112    (0)| 00:00:02 |
|   6 |       PX BLOCK ITERATOR |            | 55500 |   704K|   112    (0)| 00:00:02 |
|*  7 |        TABLE ACCESS FULL| CUSTOMERS  | 55500 |   704K|   112    (0)| 00:00:02 |
|   8 |     PX RECEIVE          |            |  1049K|    18M|   196    (2)| 00:00:03 |
|   9 |      PX SEND HASH       | :TQ10001   |  1049K|    18M|   196    (2)| 00:00:03 |
|  10 |       PX BLOCK ITERATOR |
|* 11 |        TABLE ACCESS FULL| SALES
--------------------------------------------------------------------------------------
```

Parallel Servers do majority of the work

Obrigado