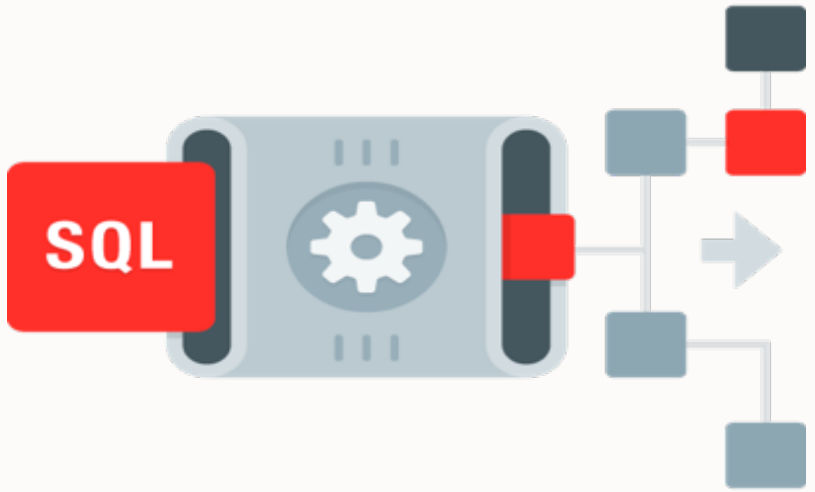
ORACLE

Understanding the Oracle Optimizer

Part 1

Maria Colgan
Master Product Manager
Oracle Database
February 2020
@SQLMaria



SQL Maria

Safe harbor statement

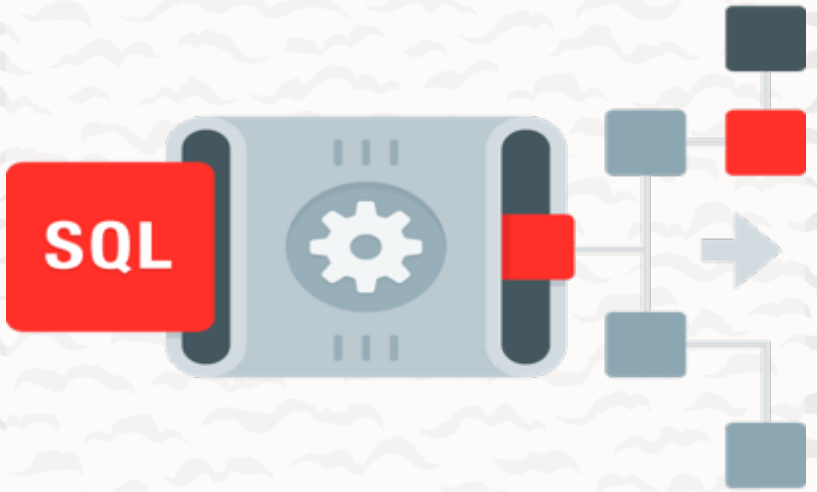
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.

The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Agenda

1. Understanding the Oracle Optimizer
2. Best Practices for Managing Optimizer Statistics
3. Explain the Explain Plan
4. Harnessing the Power of Optimizer Hints
5. SQL Tuning

How the Oracle Optimizer Operates



“ In the beginning there were rules …”

1979 – 1991 RIP

Rule Based Optimizer

Oracle Version 6 and earlier

- Rule Based Optimizer (RBO) is a heuristic based Optimizer
  - Uses a ranked list of access paths (rules)
  - 17 possible access paths
  - Lower ranked access paths assumed to operate more efficiently
- Plans chosen based on access paths available and their rank
  - If multiple access paths exist, path with the lowest rank is chosen
- Only very simple physical optimizations done automatically
  - OR Expansion: multiple OR predicates rewritten as UNION ALL

Famous tricks to work around RBO

Got an index access but want a full table scan

- Only way to influence RBO was to change the SQL text
- Concatenating an empty string to the column prevents the index from being used

```
SELECT count(*)
FROM   emp
WHERE  ename || '' = 'SMITH';
```

```
-----------------------------------------
| Id  | Operation                 | Name |
-----------------------------------------
|   0 | SELECT STATEMENT          |      |
|   1 |  SORT AGGREGATE           |      |
|*  2 |   TABLE ACCESS STORAGE FULL| EMP  |
-----------------------------------------
```

Dawn of a new era:

“ .. there is cost .. “

1992

Cost Based Optimizer

Oracle 7 - dawn of a new era

- Database features become more and more complex
  - Partitioning
  - Parallel execution
- No easy way to extend Rule Based Optimizer to accommodate so many additional access paths
- Having only one plan per statement regardless of the objects size or structure was no longer the best approach

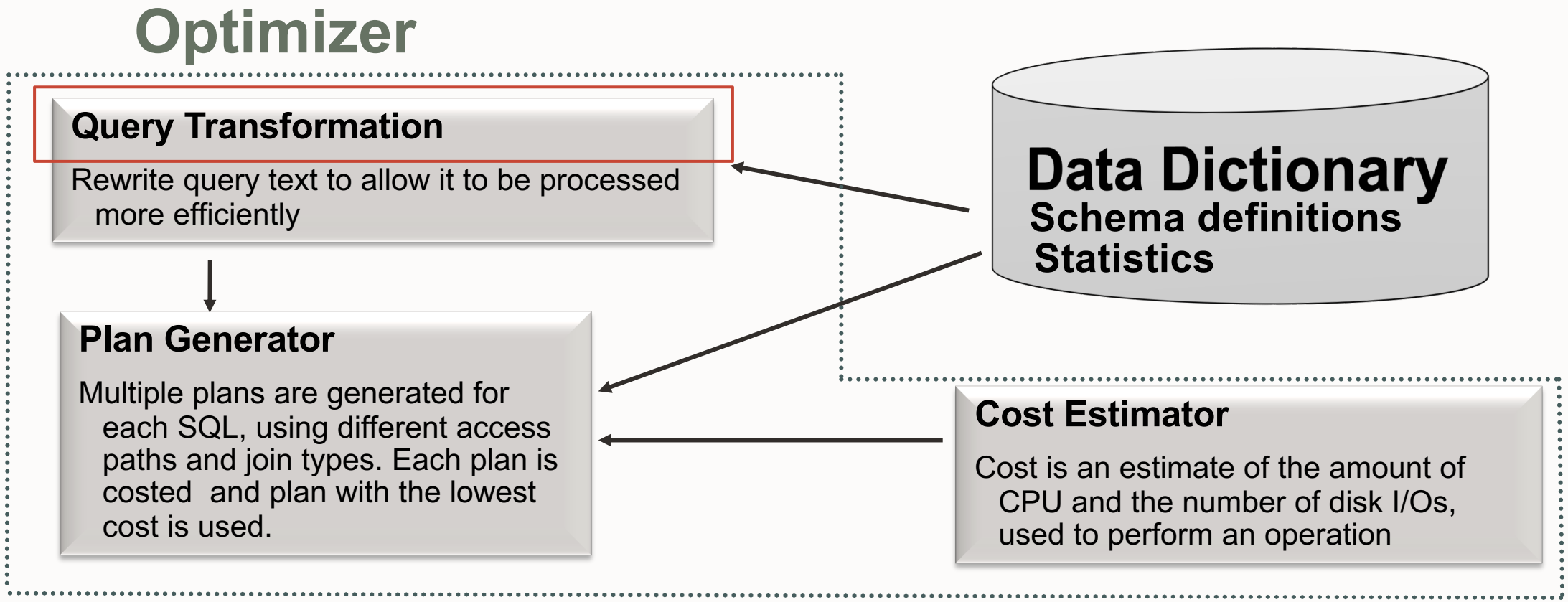
Optimizer must evolve to become cost based

Cost Based Optimizer

Oracle 7 - dawn of a new era

- Initial design based on IBM research paper
  - Access Path Selection in a Relational Database Management System (1979)
- Approach outlined in the paper was
  - Multiple execution plans are generated for a statement
  - Estimated cost is computed for each plan
  - Optimizer selects the plan with the lowest estimated cost

Understanding how the Optimizer works



Optimizer Transformations

- Translates statements into semantically equivalent SQL that can be processed more efficiently
- Initial transformations were heuristic based
  - Applied to SQL statements based on their structural properties only
- Predominately cost based now
- Transformations include
  - Subquery Unnesting
  - View Merging
  - OR Expansion
  - Star transformation

Subquery Unnesting

- A correlated subquery is one that refers to a column from a table outside the subquery
- In this case `C.cust_id` is referenced in the subquery
- Without subquery unnesting the correlated subquery must be evaluated for each row in the Customers tables

```
SELECT C.cust_last_name, C.country_id
FROM    customers C
WHERE EXISTS(SELECT 1
             FROM   sales S
             WHERE  C.cust_id=S.cust_id
             AND    S.quantity_sold > 1000);
```

*Compares the cost of the best plan with and without the transformation

Subquery Unnesting

After the Transformation

- Transformation rewrites the `EXISTS` subquery to an `ANY` subquery
- `ANY` subquery is no longer correlated
- `ANY` subquery returns a set of `CUST_IDs` if any match the predicate will return true

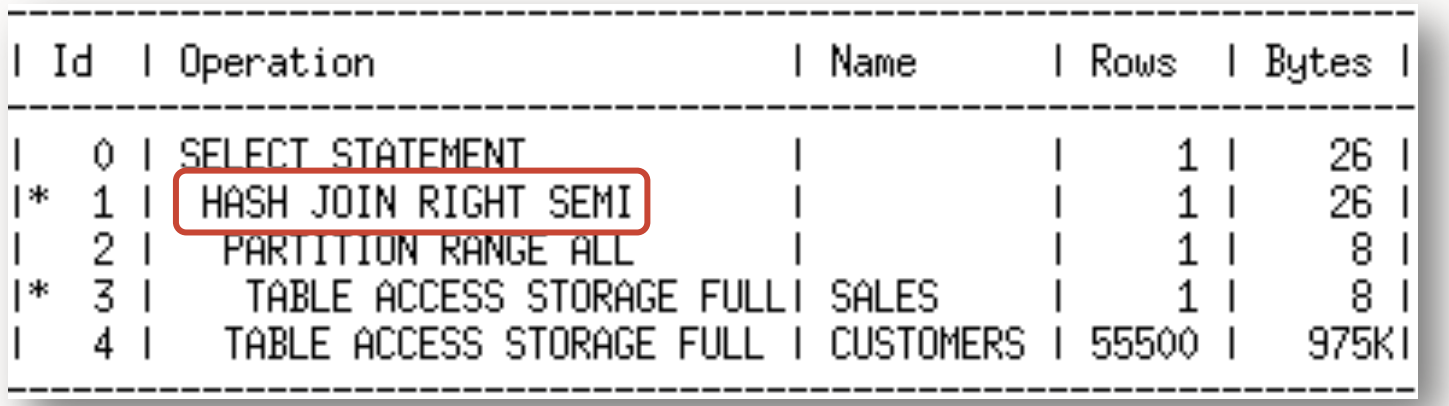
```
SELECT C.cust_last_name, C.country_id
FROM    customers C
WHERE C.cust_id = ANY(SELECT S.cust_id
                      FROM   sales S
                      WHERE S.quantity_sold > 1000);
```

*Compares the cost of the best plan with and without the transformation

Subquery Unnesting

After the Transformation

- Transformation allows subquery to be evaluated as a `SEMI` join
- Subquery returns a set of `CUST_IDs` those `CUST_IDs` are joined to the customers table via a SEMI Hash Join



Id	Operation	Name	Rows	Bytes
0	SELECT STATEMENT		1	26
* 1	HASH JOIN RIGHT SEMI		1	26
2	PARTITION RANGE ALL		1	8
* 3	TABLE ACCESS STORAGE FULL	SALES	1	8
4	TABLE ACCESS STORAGE FULL	CUSTOMERS	55500	975K

*Compares the cost of the best plan with and without the transformation

Complex View Merging

- Complex view merging refers to the merging of group by and distinct views
- Allows the optimizer to consider additional join orders and access paths
- Group-by/distinct operations can be delayed until after the joins have been evaluated

```
CREATE View cust_prod_totals_v as
SELECT SUM(s.quantity_sold) total, s.cust_id, s.prod_id
FROM   sales s
GROUP BY s.cust_id, s.prod_id;

SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM   customers c,
       cust_prod_totals_v v,
       products p
WHERE  c.country_id = 'US'
AND    c.cust_id =v.cust_id
AND    v.total > 100
AND    v.prod_id = p.prod_id
AND    p.prod_name = 'T3 Faux Fur-Trimmed Sweater';
```

Complex View Merging

After the Transformation

- After transformation GROUP BY operation occurs after SALES is joined to CUSTOMERS and PRODUCTS
- Number of rows in GROUP BY greatly reduced after join
- May not always be best to delay the GROUP BY or DISTINCT operation

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM   customers c,
       products p,
       sales s
WHERE  c.country_id = 'US'
AND    c.cust_id =s.cust_id
AND    s.prod_id = p.prod_id
AND    p.prod_name = 'T3 Faux Fur-Trimmed Sweater'
GROUP BY s.cust_id, s.prod_id, s.cust_id, s.prod_id,
         p.rowid, c.rowid, c.cust_last_name,
         c.cust_first_name, c.cust_id
HAVING SUM(s.quantity_sold) > 100
;
```

OR Expansion

- Without the transformation Optimizer treats OR predicate as a single unit
- Can’t use index on either column
- Or Expansion transforms queries that contain `OR` predicates into the form of a `UNION ALL` query of two or more branches

```
SELECT *
  FROM products p
 WHERE prod_category ='Photo'
    OR prod_subcategory ='Camera Media';
```

OR Expansion

After the Transformation

- The transformation adds an `LNNVL()` function to the second branch in order to avoid duplicates being generated across branches

- The `LNNVL` function returns `TRUE`, if the predicate evaluates to `FALSE` or if the predicate involved is `NULL`; otherwise it will return `FALSE`
  - **`lnnvl(true) is FALSE, lnnvl(false||null) is TRUE`**

```
SELECT *
  FROM products p
 WHERE prod_subcategory ='Camera Media'
UNION ALL
SELECT *
  FROM products p
 WHERE prod_category ='Photo'
    AND lnnvl(prod_subcategory =
                        'Camera Media')
;
```

OR Expansion

Transformation allows an index access to be considered for each branch of the UNION ALL

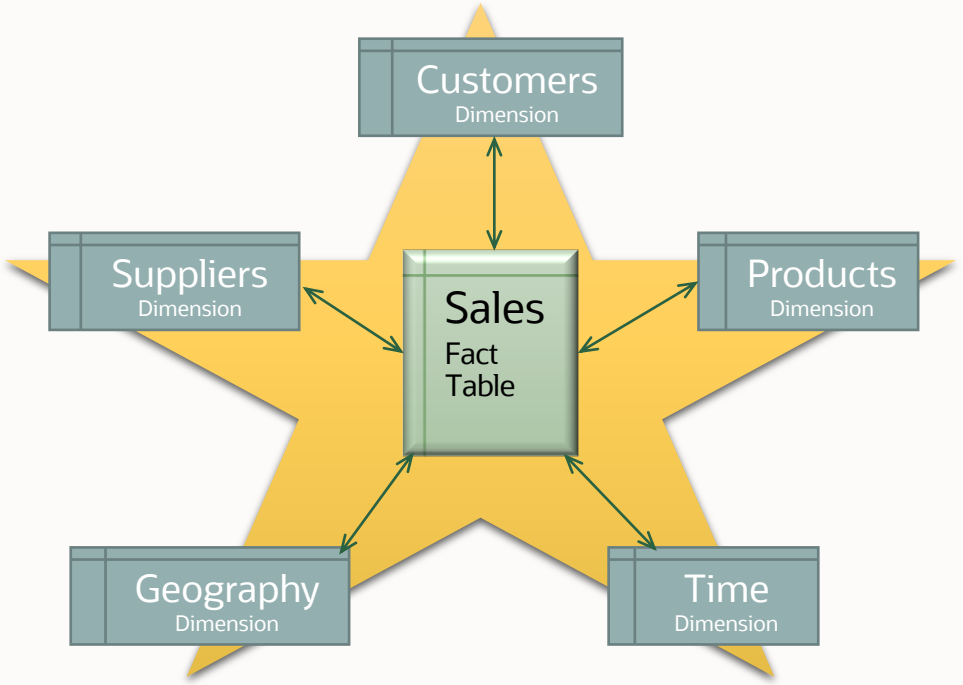
```
-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                               |       |       |     4 (100)|          |
|   1 |  UNION-ALL                          |                               |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID       | PRODUCTS                      |     3 |   519 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | PRODUCTS_PROD_SUBCAT_IX       |     3 |       |     1   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS BY INDEX ROWID       | PRODUCTS                      |    14 |  2422 |     2   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN                 | PRODUCTS_PROD_CAT_IX          |    14 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   3 - access("PROD_SUBCATEGORY"='Camera Media')
   4 - filter(LNNVL("PROD_SUBCATEGORY"='Camera Media'))
   5 - access("PROD_CATEGORY"='Photo')
```

Star Transformation

- Cost-based* transformation designed to execute star queries more efficiently
- Relies on bitmap indexes on foreign key columns to access rows in the fact table
- Controlled by parameter `STAR_TRANSFORMATION_ENABLED`



Star Schema - one or more large fact table and many smaller dimension tables

*Compares the cost of the best plan with and without the transformation

Star Transformation

- Traditionally a star query only defines predicates on the dimension tables
- No efficient way to access rows in the fact table
- By rewriting the query new access paths become available on the fact table

```
SELECT c.cust_city, t.cal_quarter_desc,
       SUM(s.amount_sold) sales_amt
FROM   sales s, times t, customers c,
       channels ch
WHERE  s.time_id = t.time_id
AND    s.cust_id = c.cust_id
AND    s.channel_id = ch.channel_id
AND    c.cust_state_province = 'CA'
AND    ch.channel_desc = 'Internet'
AND    t.calendar_quarter_desc IN ('2019-
04', '2020-01')
GROUP BY c.cust_city, t.cal_quarter_desc;
```

*Compares the cost of the best plan with and without the transformation

Star Transformation

After the Transformation

- Converts original query to include 3 sub-queries on the fact

```
SELECT c.cust_city, t.cal_quarter_desc,
       SUM(s.amount_sold) sales_amt
FROM   sales s, times t, customers c,
       channels ch
WHERE  s.time_id = t.time_id
AND    s.cust_id = c.cust_id
AND    s.channel_id = ch.channel_id
AND    c.cust_state_province = 'CA'
AND    ch.channel_desc = 'Internet'
AND    t.calendar_quarter_desc IN ('2019-04', '2020-01')
AND    s.time_id IN (SELECT time_id
                     FROM   times
                     WHERE cal_quarter_desc
                                 IN('2019-01','2020-01'))
AND    s.cust_id IN (SELECT cust_id
                     FROM   customers
                     WHERE cust_state_province='CA')
AND s.channel_id IN (SELECT channel_id
                     FROM   channels
                     WHERE channel_desc = 'Internet')
GROUP BY c.cust_city, t.cal_quarter_desc;
```

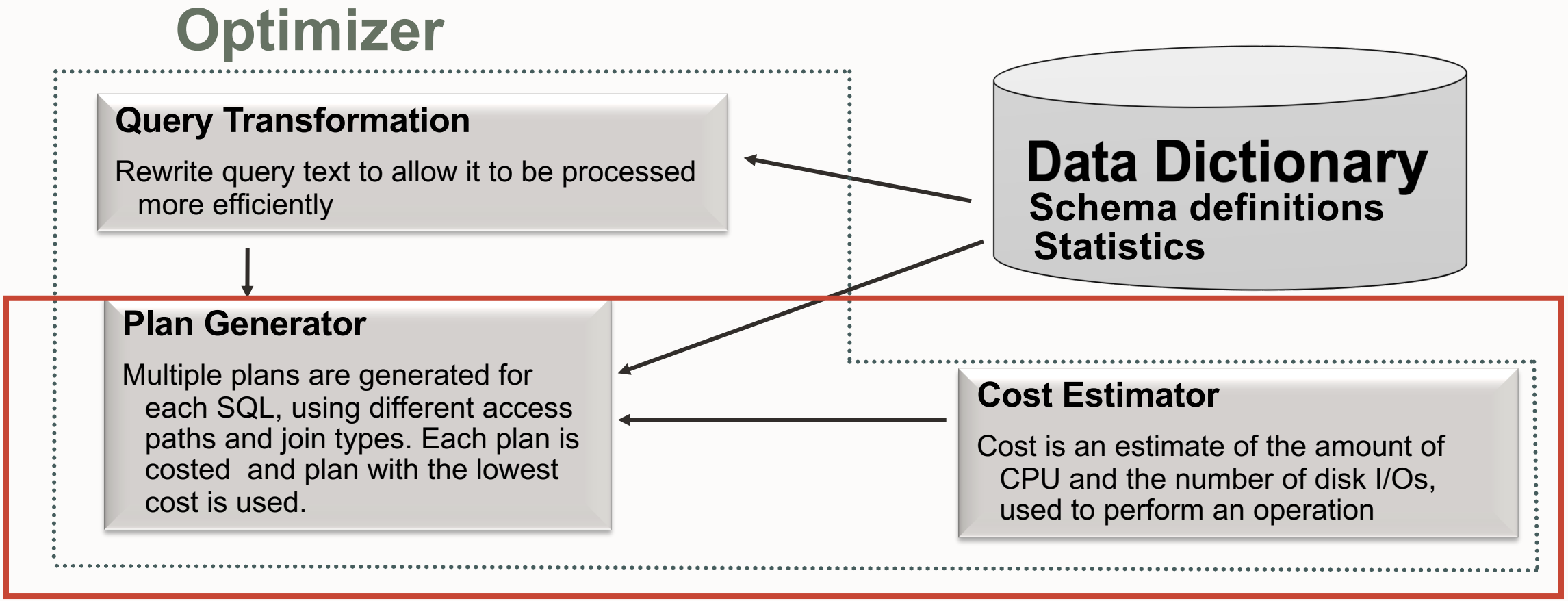
Star Transformation

After the Transformation

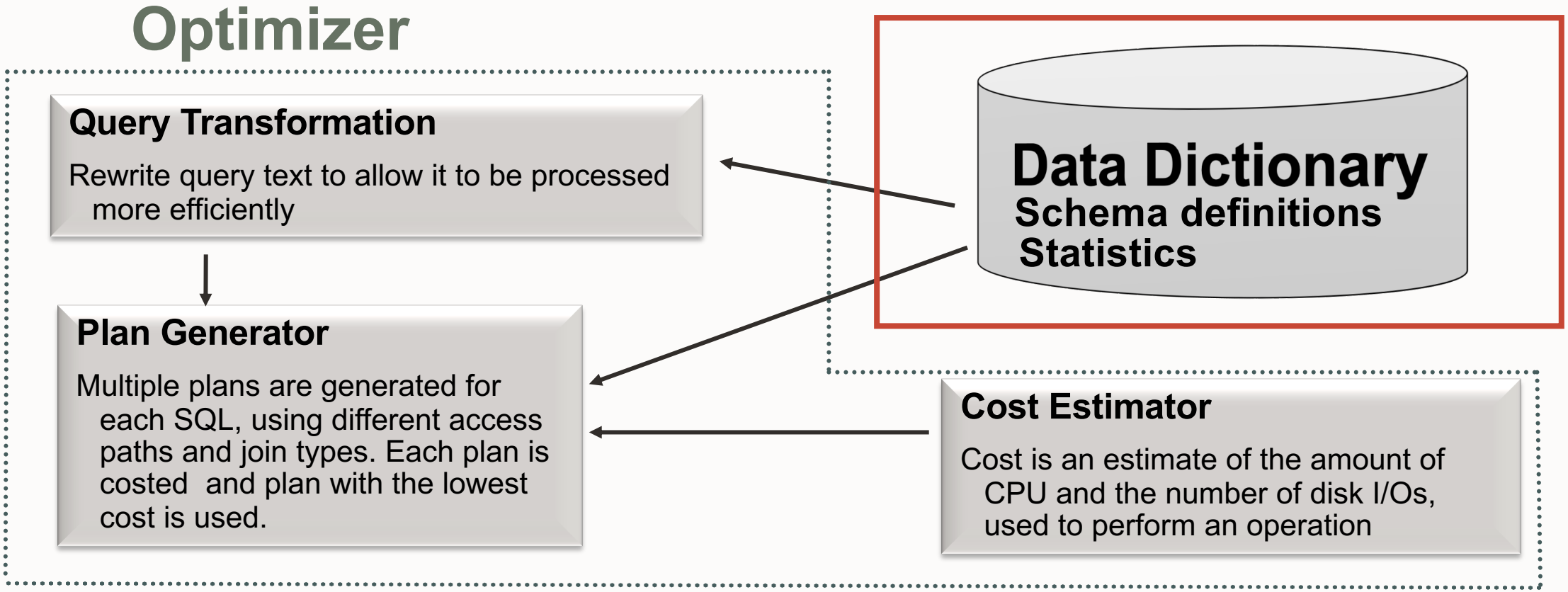
- Converts original query to include 3 sub-queries on the fact
- Fact table accessed first via bitmap index and then joins out to dimension tables
- Result of sub-queries may be saved in temp tables

```
-------------------------------------------------------------------
| Id | Operation                          | Name           |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |                |
|   1 |  HASH GROUP BY                    |                |
|*  2 |   HASH JOIN                       |                |
|*  3 |    TABLE ACCESS FULL              | CUSTOMERS      |
|*  4 |    HASH JOIN                      |                |
|*  5 |     TABLE ACCESS FULL             | TIMES          |
|   6 |     VIEW                          | VW_ST_B1772830 |
|   7 |      NESTED LOOPS                 |                |
|   8 |       PARTITION RANGE SUBQUERY    |                |
|   9 |        BITMAP CONVERSION TO ROWIDS|                |
|  10 |         BITMAP AND                |                |
|  11 |          BITMAP MERGE             |                |
|  12 |           BITMAP KEY ITERATION    |                |
|  13 |            BUFFER SORT            |                |
|* 14 |            TABLE ACCESS FULL      | CHANNELS       |
|* 15 |           BITMAP INDEX RANGE SCAN | SALES_CHANNEL_BIX|
|  16 |         BITMAP MERGE              |                |
|  17 |          BITMAP KEY ITERATION     |                |
|  18 |           BUFFER SORT             |                |
|* 19 |           TABLE ACCESS FULL       | TIMES          |
|* 20 |          BITMAP INDEX RANGE SCAN  | SALES_TIME_BIX |
|  21 |        BITMAP MERGE               |                |
|  22 |         BITMAP KEY ITERATION      |                |
|  23 |          BUFFER SORT              |                |
|* 24 |          TABLE ACCESS FULL        | CUSTOMERS      |
|* 25 |        BITMAP INDEX RANGE SCAN    | SALES_CUST_BIX |
|  26 | TABLE ACCESS BY USER ROWID        | SALES          |
-------------------------------------------------------------------
```

Understanding how the Optimizer works



Understanding how the Optimizer works



Go to PART 2 – Best Practices for Managing Optimizer Statistics

Join the Conversation

https://twitter.com/SQLMaria

https://blogs.oracle.com/optimizer/

https://sqlmaria.com

https://www.facebook.com/SQLMaria

Related Information

- White paper on Cost-Based Query Transformation in Oracle

  http://dl.acm.org/citation.cfm?id=1164215