How The "Cost Based Optimizer" Works?

20. - 22. November 2012, Nürnberg, Gremany

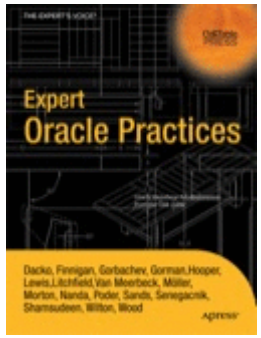
Jože Senegačnik

joze.senegacnik@dbprof.com

About the Speaker

Jože Senegačnik

- Registered private researcher
- First experience with Oracle Version 4 in 1988
- 24 years of experience with Oracle RDBMS.
- Proud member of the OakTable Network www.oaktable.net
- Oracle ACE Director
- Co-author of the OakTable book "Expert Oracle Practices" by Apress (Jan 2010)
- VP of Slovenian OUG (SIOUG) board
- CISA – Certified IS auditor
- Blog about Oracle: http://joze-senegacnik.blogspot.com

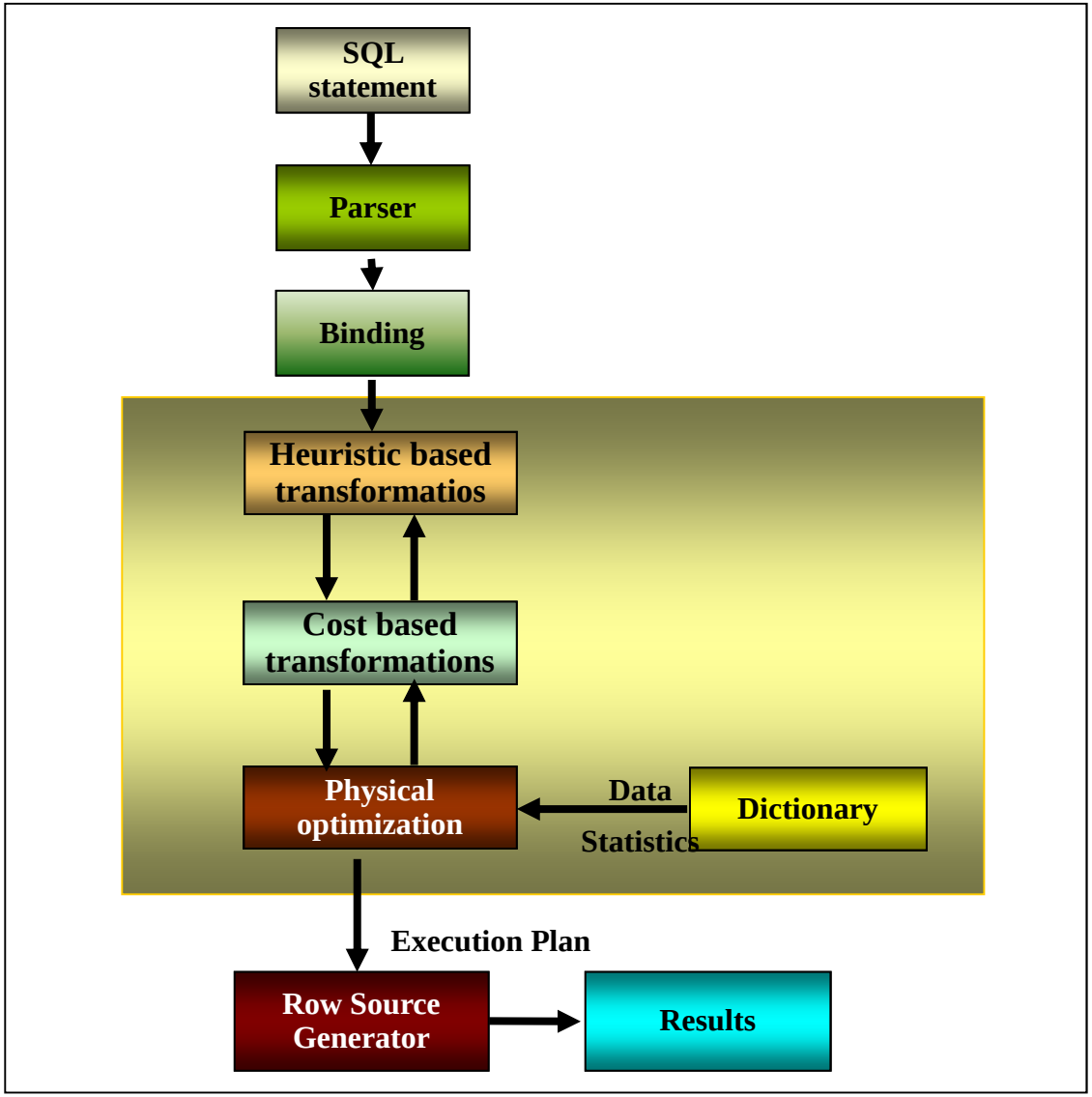


- PPL(A) – private pilot license PPL(A) / instrument rated IR/SE
- Blog about flying: http://jsenegacnik.blogspot.com
- Blog about Building Ovens, Baking and Cooking: http://senegacnik.blogspot.com

Optimizer Operations on SQLs

- For any SQL statement processed the optimizer performs the following operations:
  - Evaluation of expressions and conditions (parsing)
  - Optional bind variable peeking
  - Statement transformation (logical optimization)
  - Choice of access paths (physical optimization)
  - Choice of optimizer approaches (physical optimization)
  - Choice of join orders (physical optimization)
  - Choice of join methods (physical optimization)

SQL Statement Processing



Parsing SQL

- During the parse phase the kernel checks for the following:
  - Verifies that the statement is a valid SQL statement
  - Checks table and column definitions against the data dictionary
  - Acquires parse locks on required objects to prevent changes during the parsing phase
  - Checks for privileges on referenced schema objects
  - Finds out the optimal execution plan (if there are no bind variables present)
  - Loads the SQL statement into the library cache (V$SQL, …)
  - Routes all or part of distributed statement to remote nodes if necessary

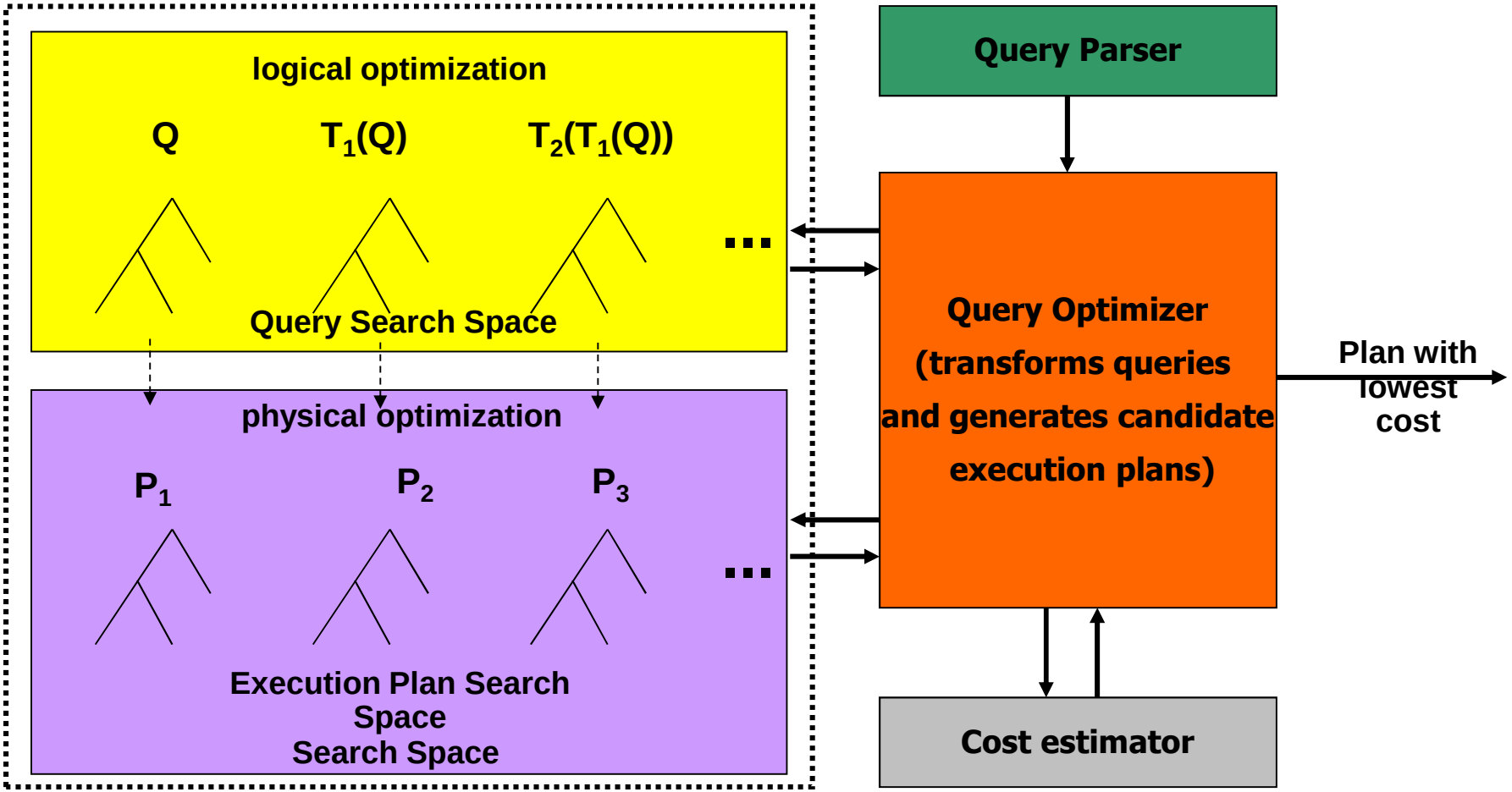
Query Optimization

- The query optimization is performed in two phases
  1. **Logical optimization** (query transformation)
     - Among others also constraints are used by the CBO to generate additional clauses during the query transformation phase.
  2. **Physical optimization** – finds information
     - Possible access method to every table (full scan, index lookup,...)
     - Possible join method for every join (HJ, SM, NL)
     - Join order for the query tables (join(join(A,B), C)
       or (join(join(C,B), A) ...

Why Query Transformations?

- The goal of transformation is to enhance the query performance.
- Transformation generates semantically equivalent form of statement, which produces the same results, but significantly differs in performance.
- Transformation rely on algebraic properties that are not always expressible in SQL, e.g, anti-join and semi-join.

Query Optimization



Query Optimization

- Query optimization is performed in two phases
  - **Logical optimization** (query transformation)
  - **Physical optimization** – finds information
    - Possible access method to every table (full scan, index lookup,...)
    - Possible join method for every join (HJ, SM, NL)
    - Join order for the query tables (join(join(A,B), C)

Query Transformations (Logical Optimization)

- The following abbreviations for transformations are used in the optimizer trace:
  - CBQT - cost-based query transformation
  - JPPD - join predicate push-down
  - OJPPD - old-style (non-cost-based) JPPD
  - FPD - filter push-down
  - PM - predicate move-around
  - CVM - complex view merging
  - SPJ - select-project-join
  - SJC - set join conversion
  - SU - subquery unnesting
  - OBYE - order by elimination
  - ST - star transformation
  - CNT - count(col) to count(*) transformation
  - JE - Join Elimination

COST BASED OPTIMIZATION

Optimizer Measures Used During Optimization

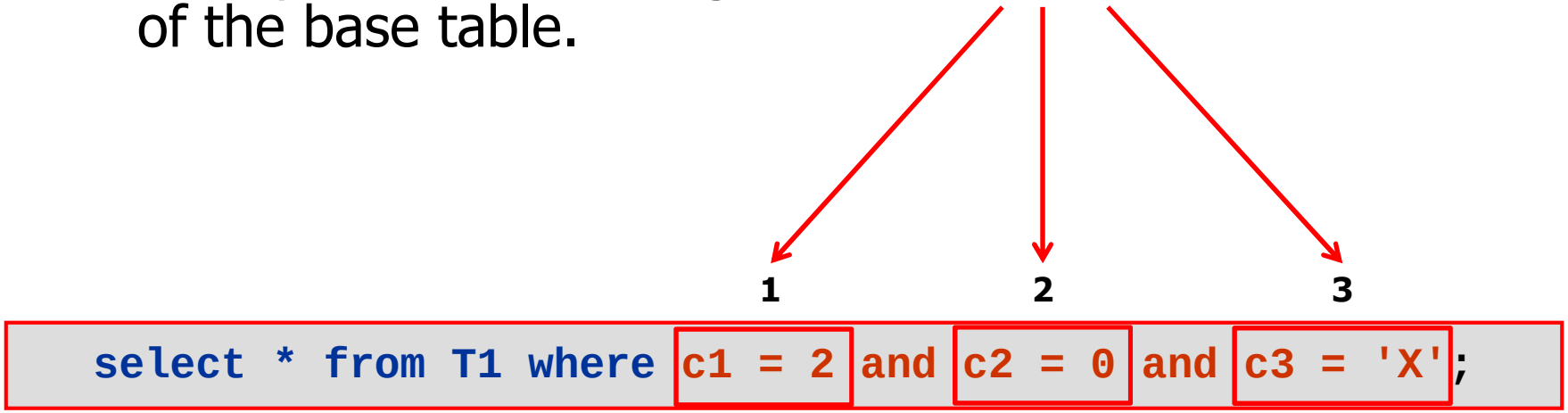
- Estimator
  - The goal is to estimate the overall cost of any of possible plans.
  - The Estimator computes the following measures
    - selectivity,
    - cardinality
    - cost

Selectivity Measure

- Selectivity
  - a fraction of rows from a row set where row set can be a base table or a result of a 'join' or a 'group by' operator.
  - Selectivity lies in a value range from 0.0 to 1.0 (or in other words from 0% to 100%)
  - Selectivity is the reciprocal of the number of distinct values.
  - Histograms help the estimator to generate good selectivity estimates for columns with non-uniform data distribution.

Cardinality

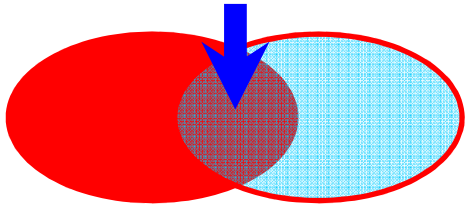
- **Cardinality** represents the number of rows in a row source (table, result of previous operations).
- The **computed cardinality** is computed as the product of the table cardinality (base cardinality) and combined selectivity of all predicates specified in where clause.
- Each predicate is acting as a successive filter on the rows of the base table.



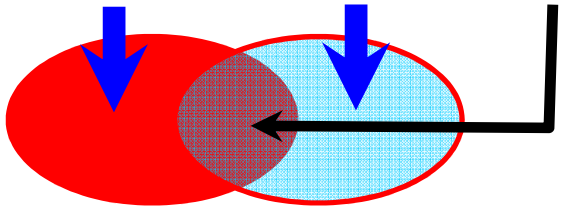
Combining Predicates

P1 AND P2 ➔ FF1 * FF2

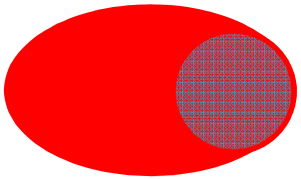
FF=Filtering factor (selectivity)



P1 OR P2 ➔ FF1 + FF2 - FF1 * FF2



NOT P1 ➔ 1 - FF1

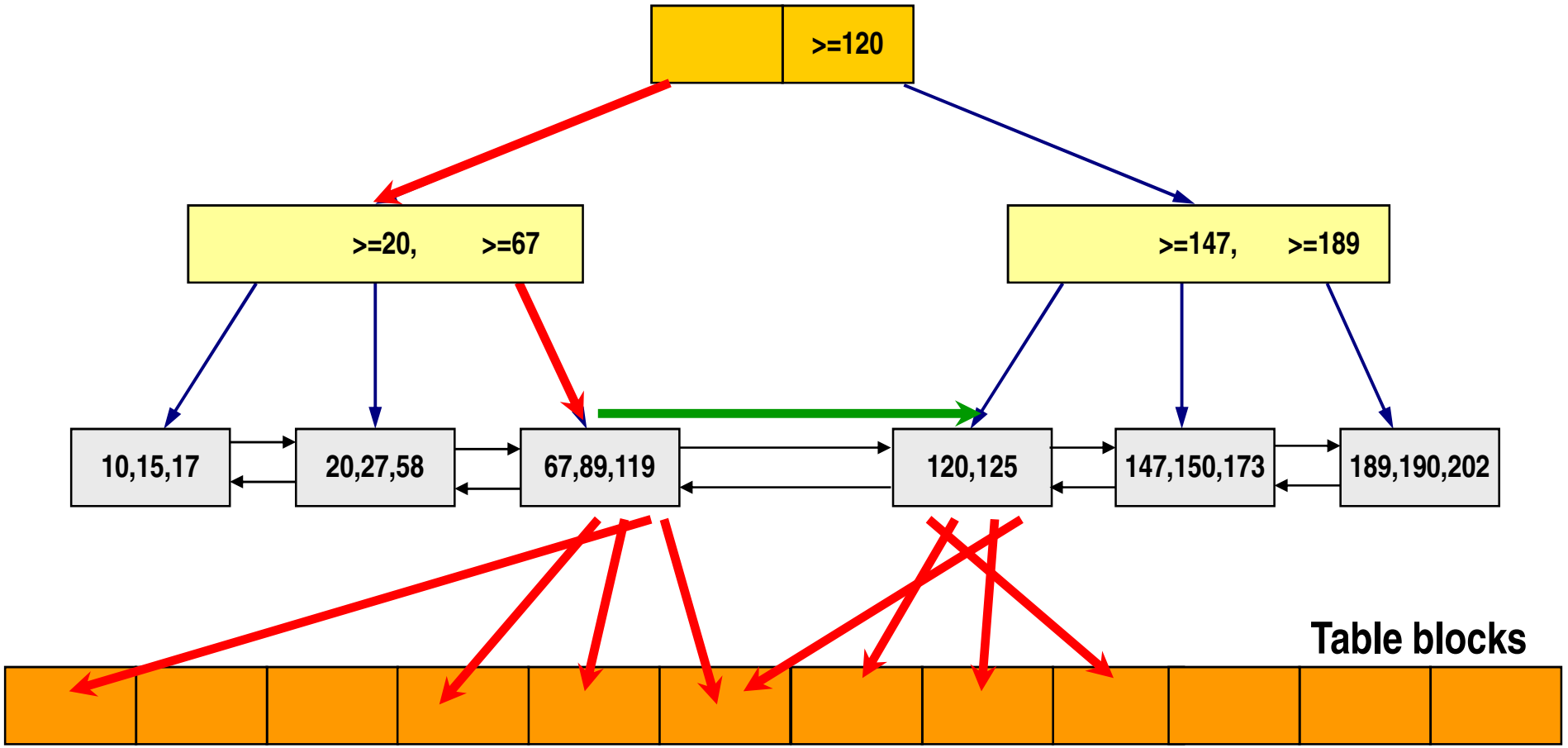


Cost

- The cost used by the CBO represents an estimate of the number of disk I/Os and the amount of CPU and memory used in performing an operation.
- "Cost" is the result of the "price" of the access method and the estimated cardinality of the row source.
- One can think about cost also as "time used" to execute the SQL.
- It can only be treated as the **CBO's internal measure** that is used in the process of selecting the optimal plan.
- Incorrectly estimated selectivity and an inaccurate base cardinality of the table have same bad effect on cost calculation.
- The theoretical execution plan, produced by the *explain plan* command, also contains the estimated cardinality of the final result and of all execution steps.

INDEX RANGE SCAN (2)

```
select * from t1 where c1 between 90 and 123
```



Case: Calculating the Cost Of Index Scan

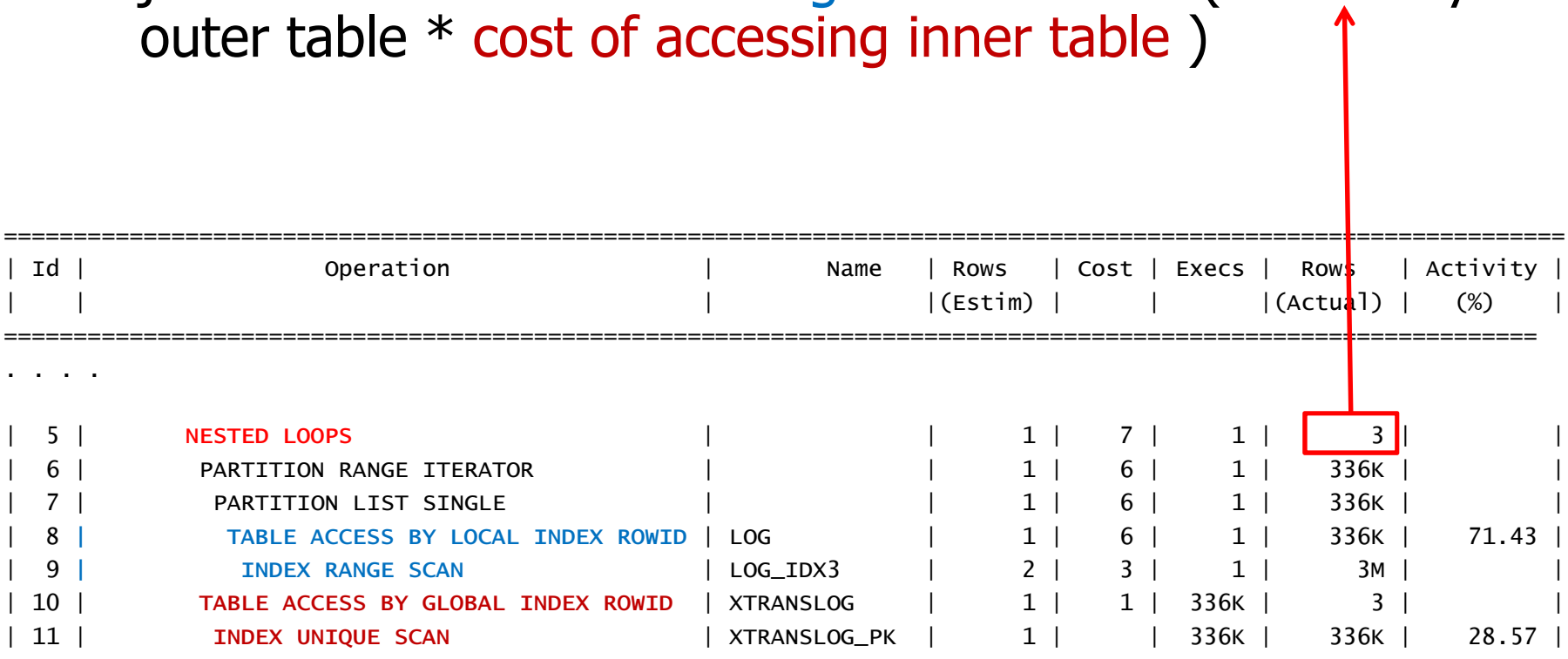
- The formula essentially equates to the count of blocks that Oracle has to traverse in order to get all qualifying rows:
  - From the root page follow the tree down to the leaf page blevel blocks
  - Access all qualifying leaf blocks *leaf blocks * filter factor*
  - Access all qualifying data blocks *clustering factor * filter factor*

General Cost Calculation Formulas for Joins

- NL - NESTED LOOP JOIN
  - join cost = cost of accessing outer table + (cardinality of outer table * cost of accessing inner table)
- SM – SORT MERGE JOIN
  - join cost = (cost of accessing outer table + outer sort cost) + (cost of accessing inner table + inner sort cost)
- HA – HASH JOIN
  - join cost = (cost of accessing outer table) + (cost of building hash table) + (cost of accessing inner table)

Nested Loop Join

- NL - NESTED LOOP JOIN
  - join cost = cost of accessing outer table + (cardinality of outer table * cost of accessing inner table)



Id	Operation	Name	Rows (Estim)	Cost	Execs	Rows (Actual)	Activity (%)
. . . .							
5	NESTED LOOPS		1	7	1	3	
6	PARTITION RANGE ITERATOR		1	6	1	336K	
7	PARTITION LIST SINGLE		1	6	1	336K	
8	TABLE ACCESS BY LOCAL INDEX ROWID	LOG	1	6	1	336K	71.43
9	INDEX RANGE SCAN	LOG_IDX3	2	3	1	3M	
10	TABLE ACCESS BY GLOBAL INDEX ROWID	XTRANSLOG	1	1	336K	3	
11	INDEX UNIQUE SCAN	XTRANSLOG_PK	1		336K	336K	28.57

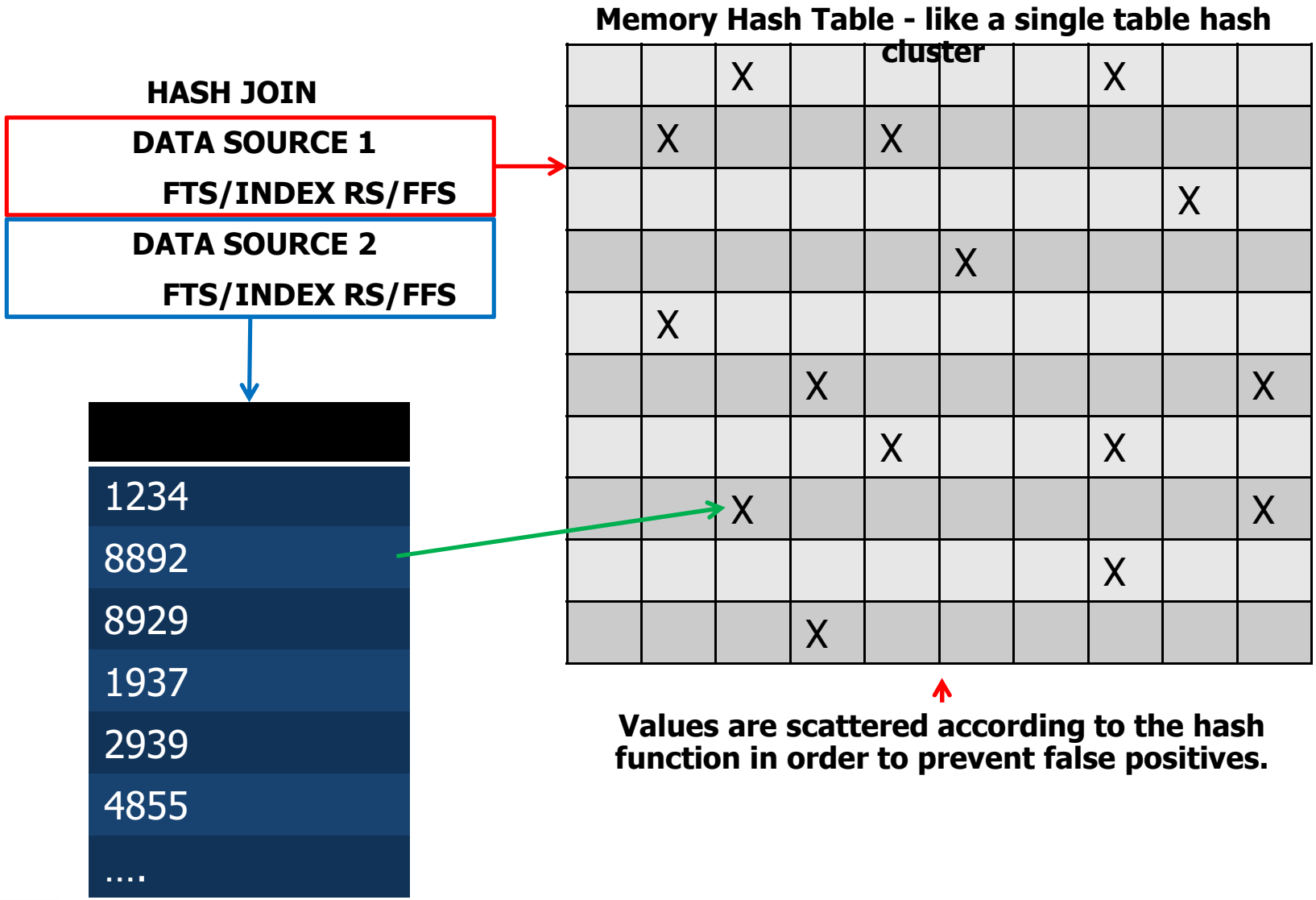
Hash Join

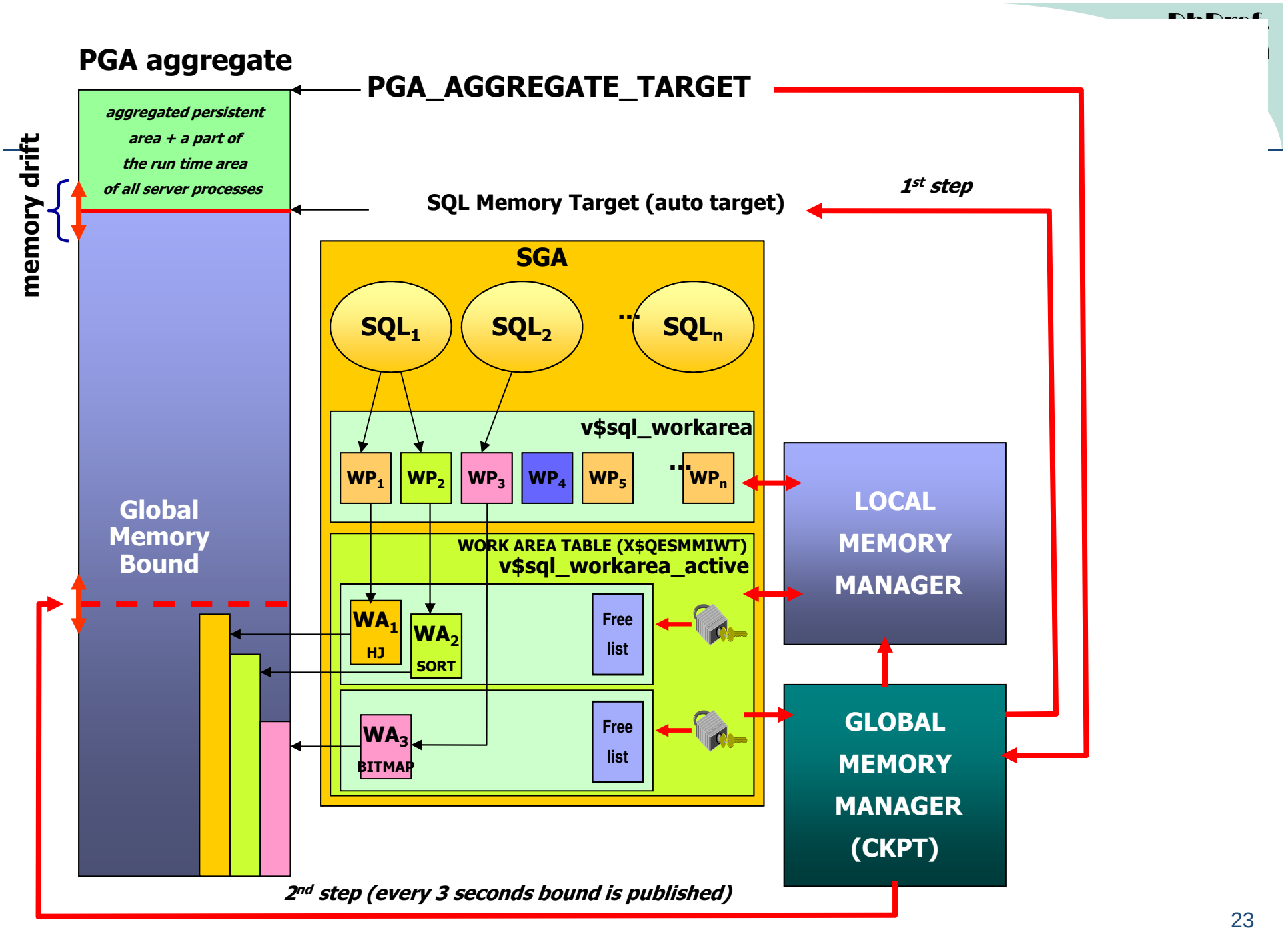
- HASH JOIN
  - join cost = (cost of accessing outer table) + (cost of building hash table) + (cost of accessing inner table)

```
=============================================================================================================
| Id |                 Operation                |        Name         |  Rows    | Cost  | Execs |   Rows    | Activity |
|    |                                          |                     | (Estim)  |       |       | (Actual)  |   (%)    |
=============================================================================================================
|  0 | SELECT STATEMENT                         |                     |          |       |     1 |      1594 |          |
|  1 |   SORT ORDER BY                          |                     |      203 | 34258 |     1 |      1594 |          |
|  2 |    HASH JOIN                             |                     |      203 | 34257 |     1 |      1594 |          |
|  3 |     HASH JOIN                            |                     |       38 |    12 |     1 |        11 |          |
|  4 |      TABLE ACCESS FULL                   | C_TYPE              |       13 |     4 |     1 |        14 |          |
|  5 |      TABLE ACCESS FULL                   | C_PROD_DEF          |       78 |     7 |     1 |        78 |          |
. . .
```

- First data source is (considered) smaller and thus candidate for creating a hash table in memory.
- The second data source is accessed and probved against the memory hash table.

Hash Join


HASH JOIN
DATA SOURCE 1
FTS/INDEX RS/FFS
DATA SOURCE 2
FTS/INDEX RS/FFS
1234
8892
8929
1937
2939
4855
....
Memory Hash Table - like a single table hash cluster
Values are scattered according to the hash function in order to prevent false positives.


PGA aggregate
PGA_AGGREGATE_TARGET
aggregated persistent area + a part of the run time area of all server processes
memory drift
SQL Memory Target (auto target)
1st step
SGA
SQL1
SQL2
...
SQLn
v$sql_workarea
WP1
WP2
WP3
WP4
WP5
...
WPn
Global Memory Bound
WORK AREA TABLE (X$QESMMIWT)
v$sql_workarea_active
WA1
HJ
WA2
SORT
Free list
WA3
BITMAP
Free list
LOCAL MEMORY MANAGER
GLOBAL MEMORY MANAGER (CKPT)
2nd step (every 3 seconds bound is published)

Demo SQL Statement

```
select t.*,b.*
from t, b
where t.id = b.id
and t.id <= 100;
```

CPU Costing Model – Intoduced in 9i

- The CPU costing model:

```
Cost =  (#SRds * sreadtim + #MRds * mreadtim +
          #CPUCycles / cpuspeed ) / sreadtim
```

- where
  - #SRDs - number of single block reads
  - #MRDs - number of multi block reads
  - #CPUCycles - number of CPU Cycles *)
  - sreadtim - single block read time
  - mreadtim - multi block read time
  - cpuspeed - CPU cycles per second

- *CPUCycles includes
  - CPU cost of query processing ("pure" CPU cost) and
  - CPU cost of data retrieval (CPU cost of the buffer cache get).

Table Statistics

- Stored in
  - DBA_TABLES
  - DBA_TAB_[SUB]PARTITIONS
- Statistics
  - NUM_ROWS
  - BLOCKS (always exact)
  - AVG_ROW_LEN
  - EMPTY_BLOCKS
  - AVG_SPACE
  - CHAIN_CNT

Column Statistics

- Stored in
  - DBA_TAB_COL_STATISTICS
  - DBA_TAB_HISTOGRAMS
  - DBA_[SUB]PART_COL_STATISTICS
  - DBA_[SUB]PART_HISTOGRAMS
- The following statistics are collected:
  - NUM_DISTINCT
  - LOW_VALUE
  - HIGH_VALUE
  - NUM_NULLS
  - DENSITY
  - NUM_BUCKETS

Index Statistics

- Stored in:
  - DBA_INDEXES
  - DBA_IND_[SUB]PARTITIONS

- The following statistics are collected:
  - BLEVEL (always exact)
  - LEAF_BLOCKS
  - CLUSTERING_FACTOR
  - DISTINCT_KEYS
  - AVG_LEAF_BLOCKS_PER_KEY
  - AVG_DATA_BLOCKS_PER_KEY
  - NUM_ROWS

Histograms

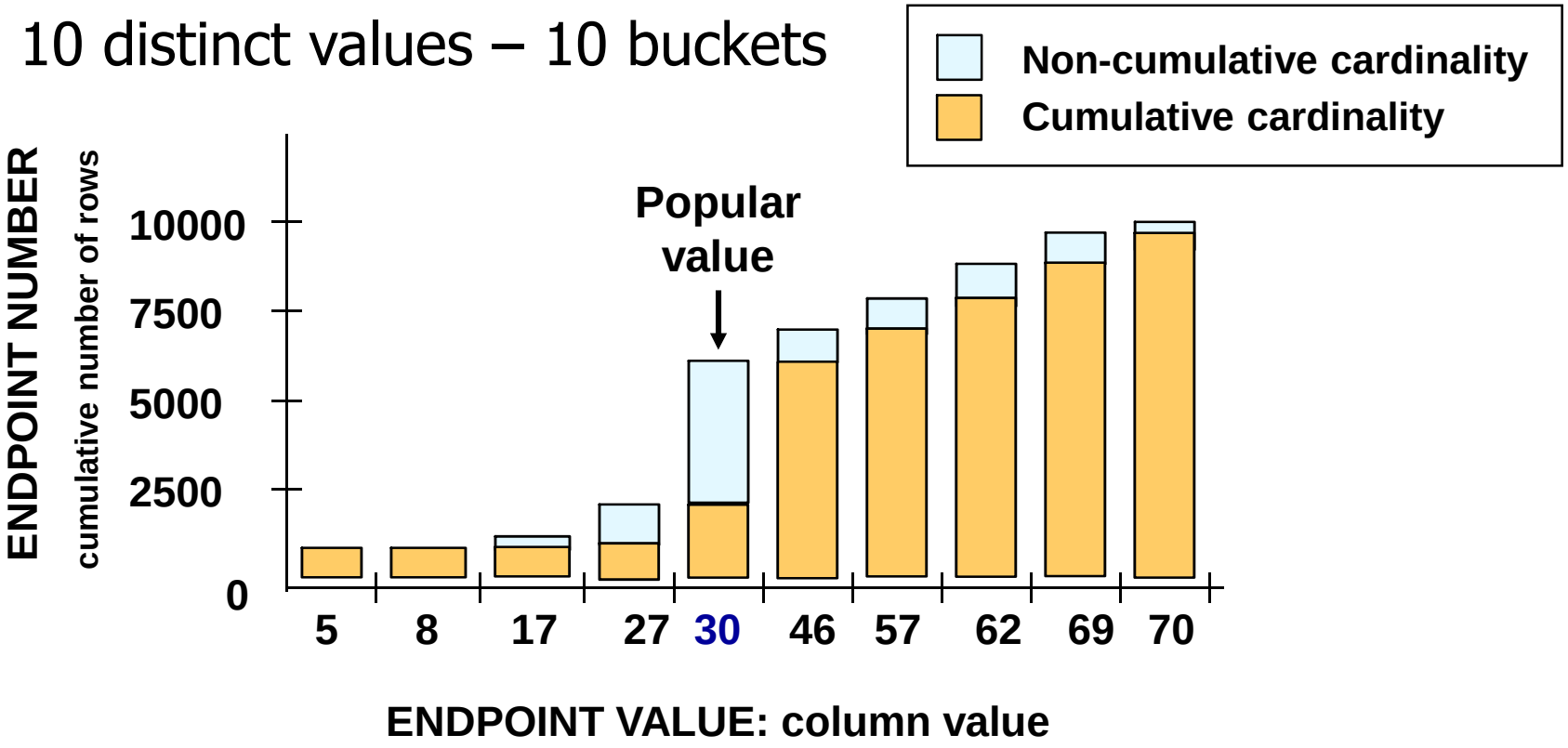
Histograms

- By default Oracle assumes uniform distribution. If this is not a case we need a histogram.
- When histograms are available they are used in calculations of selectivity (filtering factor (FF)):
  - the optimizer uses the density in its filter factor calculation, not number of distinct values.
  - the density is calculated differently for columns with histograms, not simply as 1/number of distinct values.
- *The cause for a poor access plan is more often the incorrect estimate of the cardinality of a row source than the incorrect estimate of an index access cost.*

Types of Histograms

- Histogram Types
  - Height balancing: Buckets have the same cardinality
  - Frequency histograms: One bucket for each distinct value, storing exact cardinalities

Frequency Histogram

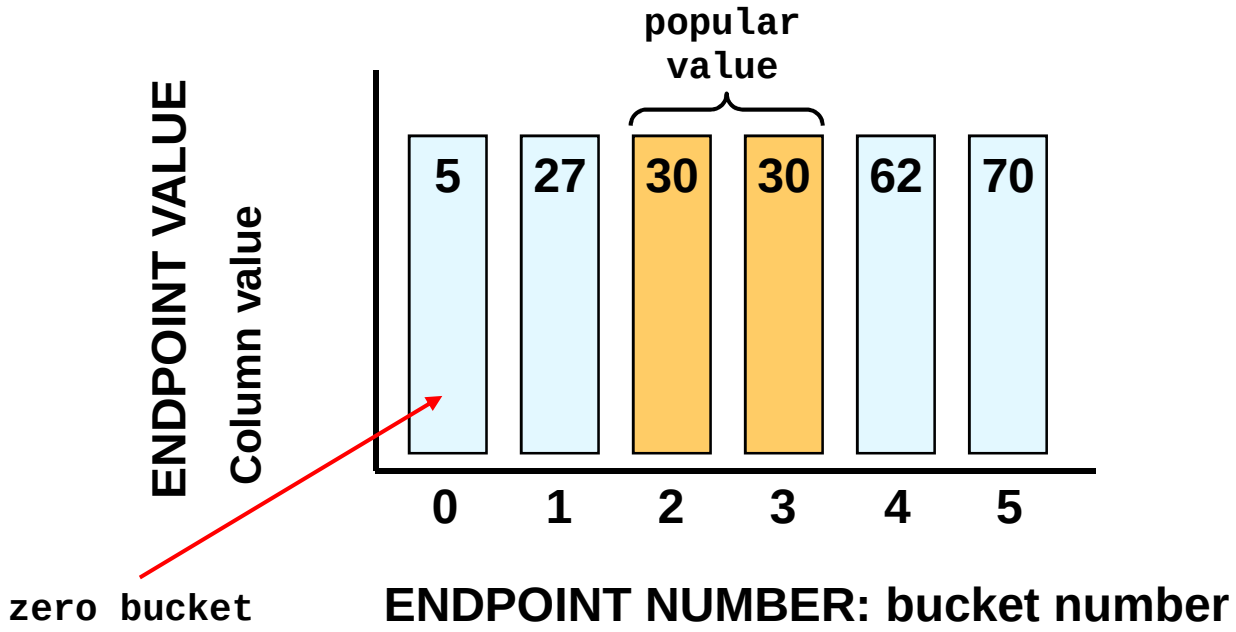


Height Balanced Histograms

- In height balanced histograms column values are divided up so that each bucket contains the same number of values
  - Maximum values are recorded as endpoints.
  - The special zero bucket records the minimum value.
  - Column values occurring more than once as endpoint are popular values.
  - For non-popular values the density statistic is used to estimate selectivity for equality predicates.

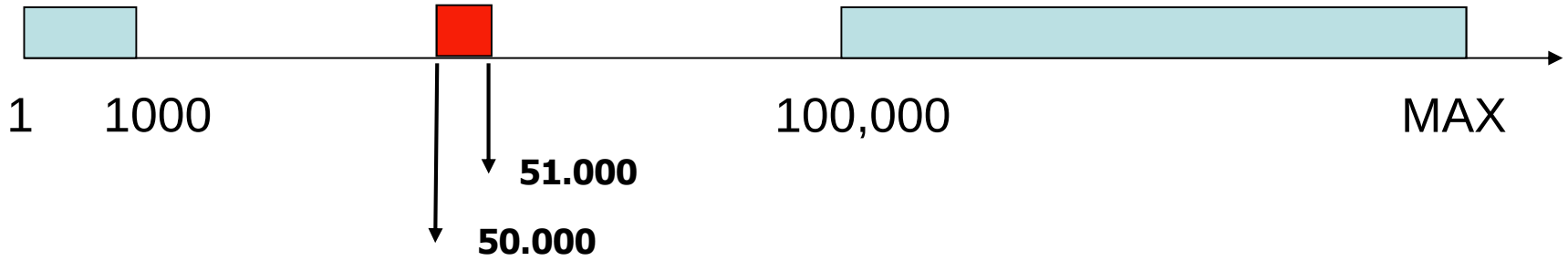
Height Balanced Histograms

5 buckets, 10 distinct values
(2000 rows per bucket)



Skew Data

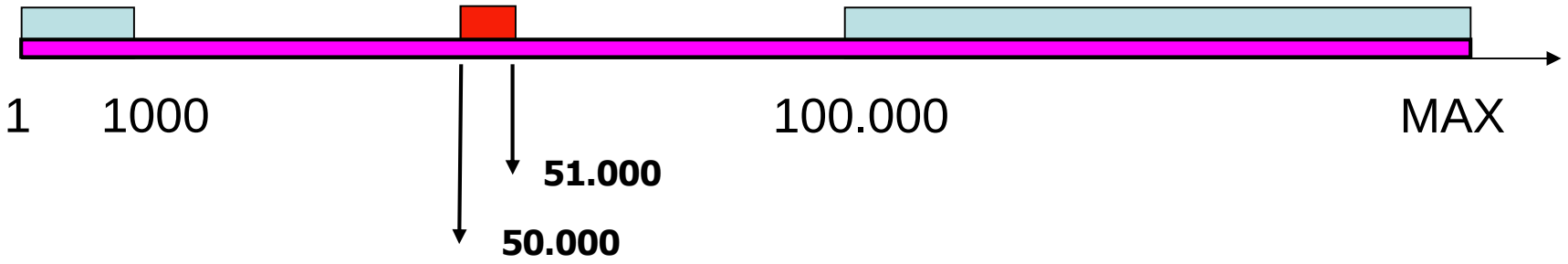
- Not only un-uniform distribution but gaps in values as well
  - values that are used for null or not known values
    - regular range of values 1-15, uknown 99
    - null date – 31.12.4000
    - gaps in primary keys 1 - 1 000, 10 000 →
- Histogram can help a lot!



Optimizer's Estimation

```
SQL> select * from bt
     where c1 between 50000 and 51000;

-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |   697 |
|*  1 |  TABLE ACCESS FULL| BT   |   697 |
-------------------------------------------
```



Estimation Explanation

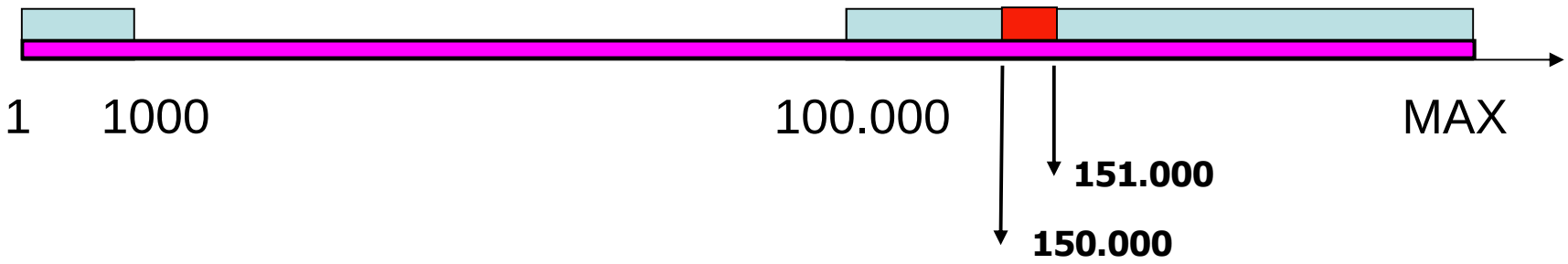
Selectivity = (ending value – starting value) /
(max column value –minimum column value)

Selectivity = (51000 – 50000) / (325000 – 1) = 0,003077

Cardinality = selectivity * table cardinality =
226001 * 0,003077 = 695,389832 = 695 rows.

Conclusions

- Table contains NO rows with values between 1.001 and 99.999 in column c1
- Estimation error is huge!
- The estimation for a range where data exists would be more reliable but still not close to the reality.



Test With A Histogram

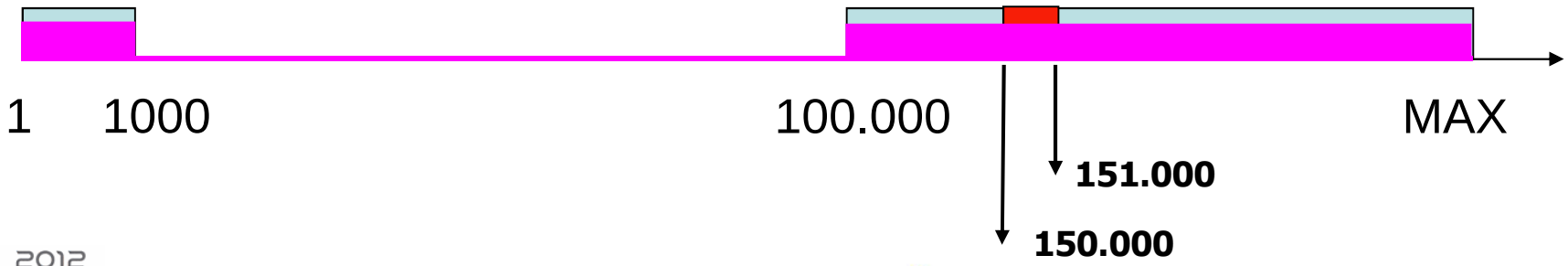
- Histogram
  - First test – 100 buckets
  - Second test – 200 buckets

```
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    22 |
|*  1 |  TABLE ACCESS FULL | BT   |    22 |
-----------------------------------------
```

Histogram (2)

```
SQL> select * from bt
     where c1 between 150000 and 151000;
```

```
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |   999 |
|*  1 |  TABLE ACCESS FULL | BT   |   999 |
-----------------------------------------
```



Using User-defined Functions in SQLs

- A lot of developers like to use their own PL/SQL functions in SQL Statements.
- Nothing wrong with this if the function is really needed. However, many of them were developed instead of :
  - using simple decode statement
  - joining to lookup table(s)
- Executing functions means that database has to switch from SQL Engine to PL/SQL engine and then again to SQL Engine and .... what is very costly.

CBO's Default Costing For Functions

Selectivity	1% (0.01)
CPU cost	3000
IO cost	0
Network cost	0

Default Cost

- The **default cost** is defined as the **CPU, I/O** and the **NETWORK** cost. The latter is currently ignored by the CBO.

```
SQL> ASSOCIATE STATISTICS WITH FUNCTIONS
     DEMO_FUNC1 DEFAULT SELECTIVITY 10,
     DEFAULT COST (3994087, 5, 0);
```

- We can define the default cost for functions, packages, types, indexes and indextypes.
- We can use DBMS_MONITOR package and TKPROF to obtain actual numbers for I/O cost and execution time.
- Run the function 100 or 1000 times to get the number of LIOs performed and CPU time used per execution.
- We can do this also for the package as whole.

Estimating CPU Cost

- Use DBMS_ODCI.ESTIMATE_CPU_UNITS function to convert the CPU time per execution to the approximate number of CPU instructions.
- The returned number is defined in 1000 of instructions.
- To calculate CPU cost for a function that always executes in 0.002 second:

```
SQL> variable a number
SQL> begin :a := round(1000*DBMS_ODCI.ESTIMATE_CPU_UNITS(0.002),0);
  2  end;
  3  /

PL/SQL procedure successfully completed.

SQL> print a

         A
----------
   3994087
```

CBO and PL/SQL Functions

- One should properly define the cost and selectivity of a function.
- If no statistics is associated with functions then one should at least be careful about the order how they are defined in the where clause which defines the default order of execution.
- For simple cases we can use **default selectivity** and **default cost** associations.
- For complex cases and user-defined indexes we must implement statistics types.

Conclusions

- CBO is a very complex piece of software and as any other SW requires good input in order to produce good output (execution plans).
- Never compare cost but rather look at cardinalities – estimated and actual.
- You know your data much better then CBO will ever know.
- By rewriting a SQL statement one can achieve more then CBO is capable to do (manual query transformation).

Thank you for your interest!

Q&A