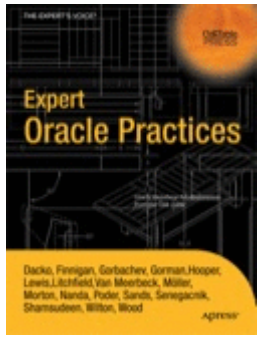

Joining Tables –
Isn't that simple?
Jože Senegačnik
joze.senegacnik@dbprof.com
1

About the Speaker

Jože Senegačnik

- Owner of Dbprof d.o.o.
- First experience with Oracle Version 4.1 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) / IR(SE) – private pilot license, instrument rating
- Blog about flying: http://jsenegacnik.blogspot.com
- Blog about Building Ovens, Baking and Cooking: http://senegacnik.blogspot.com

Join Methods

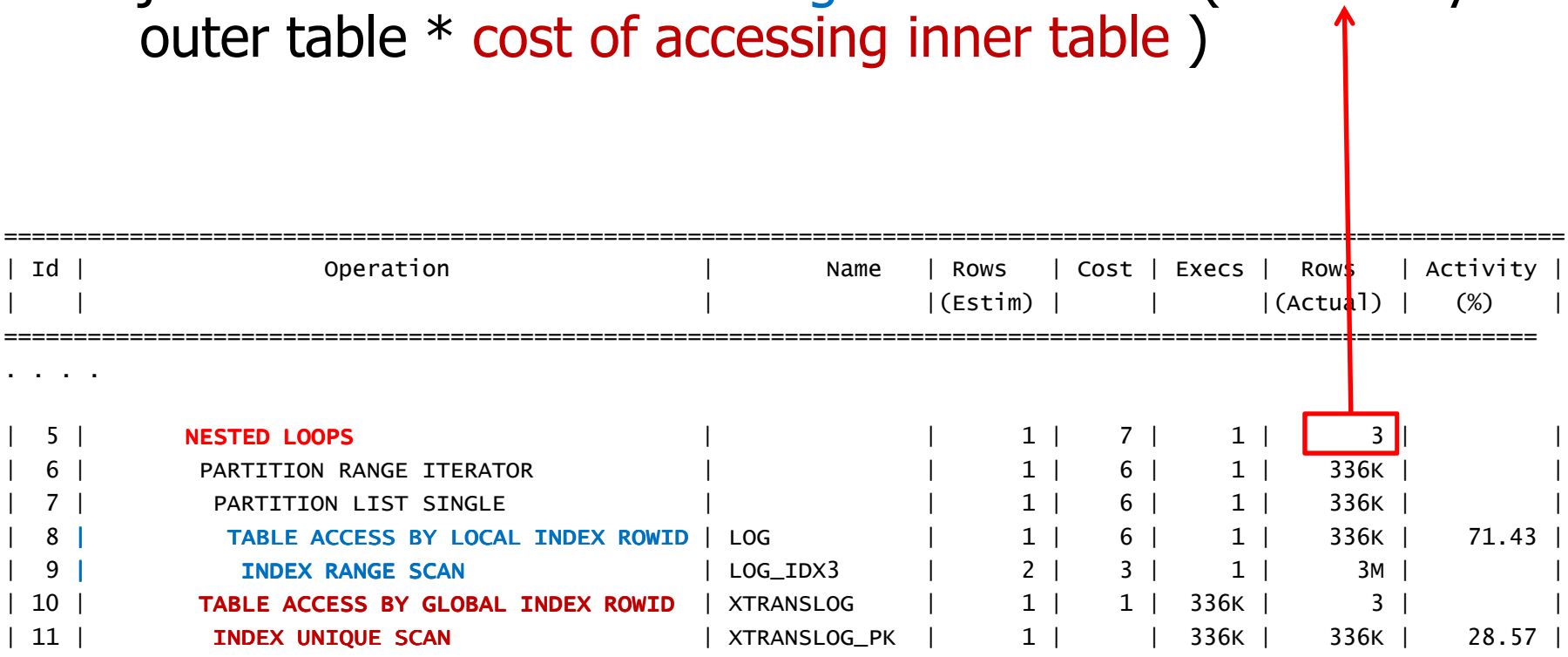
- CBO uses three join methods:
  - Nested-loop
    - inner-joins, outer-joins, semi-joins, and anti-joins
  - Sort-merge
    - inner-joins, outer-joins, semi-joins, and anti-joins
  - Hash join
    - inner-joins, outer-joins, semi-joins, and anti-joins
- Every method is good for a different join cardinality and usually the CBO selects the most appropriate one, however there are cases, when unfortunately this is not true.
- How we can determine that the join method is not the optimal one?
  - Most likely the response time is beyond the expected one ☺
  - Checking the execution plan to determine which type of join was used
  - Looking at the estimated and actual cardinalities (after the statement is executed at least once)

Join Methods Availability

- Not always are all join methods available.
- Some limitations:
  - HASH JOIN works only for equality join predicates, but not for any other like <>, >=, <=, …
  - CARTESIAN JOIN is only possible in SORT MERGE join

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)



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

Inefficient Nested-loop Join

Id	Operation	Name	Rows (Estim)	Cost	Execs	Rows (Actual)	Activity (%)
0	SELECT STATEMENT				1	3	
1	SORT ORDER BY		1	9	1	3	
2	FILTER				1	3	
3	NESTED LOOPS				1	3	
4	NESTED LOOPS		1	8	1	3	
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
12	INDEX RANGE SCAN	MT_STATUS_UIDX	1		3	3	
13	TABLE ACCESS BY INDEX ROWID	MT_STATUS	1	1	3	3	

SORT MERGE JOIN

SM – SORT MERGE JOIN
join cost = (cost of accessing outer table + outer sort cost) +
(cost of accessing inner table + inner sort cost)

```
-------------------------------------------------------------------------
| Id  | Operation             | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |  2250K|    25M|  1171   (98)|
|   1 |  MERGE JOIN           |      |  2250K|    25M|  1171   (98)|
|   2 |   SORT JOIN           |      | 30000 |   175K|    23   (31)|
|   3 |    TABLE ACCESS FULL  | T1   | 30000 |   175K|    17    (6)|
|*  4 |   FILTER              |      |       |       |             |
|*  5 |    SORT JOIN          |      | 30000 |   175K|    23   (31)|
|   6 |     TABLE ACCESS FULL| T2   | 30000 |   175K|    17    (6)|
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  4 - filter("T1"."C2">="T2"."C2")
  5 - access(INTERNAL_FUNCTION("T1"."C1")>=INTERNAL_FUNCTION("T2"."C1"))
      filter(INTERNAL_FUNCTION("T1"."C1")>=INTERNAL_FUNCTION("T2"."C1"))
```

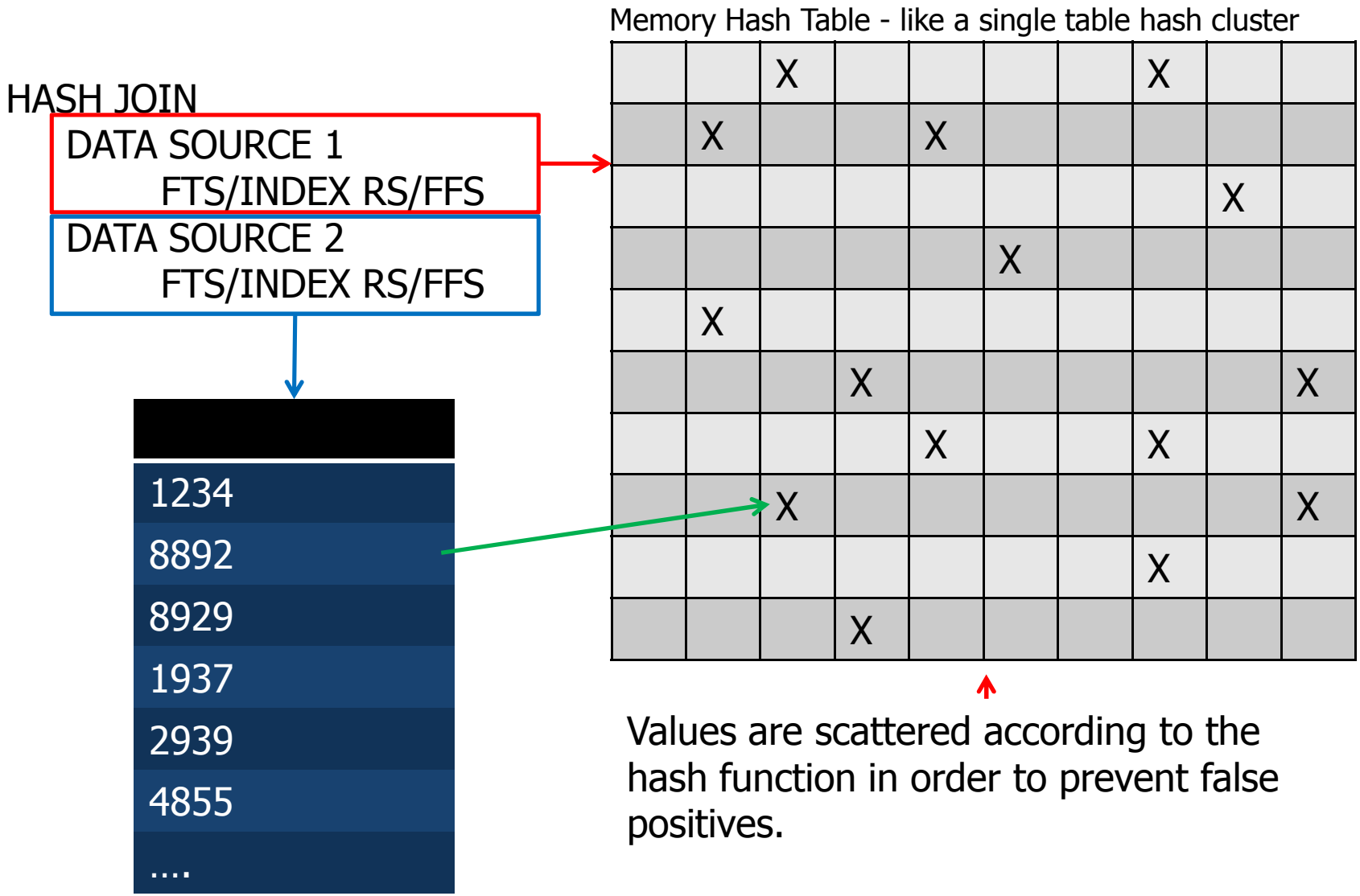
HASH JOIN

- 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 |          |
. . .
```

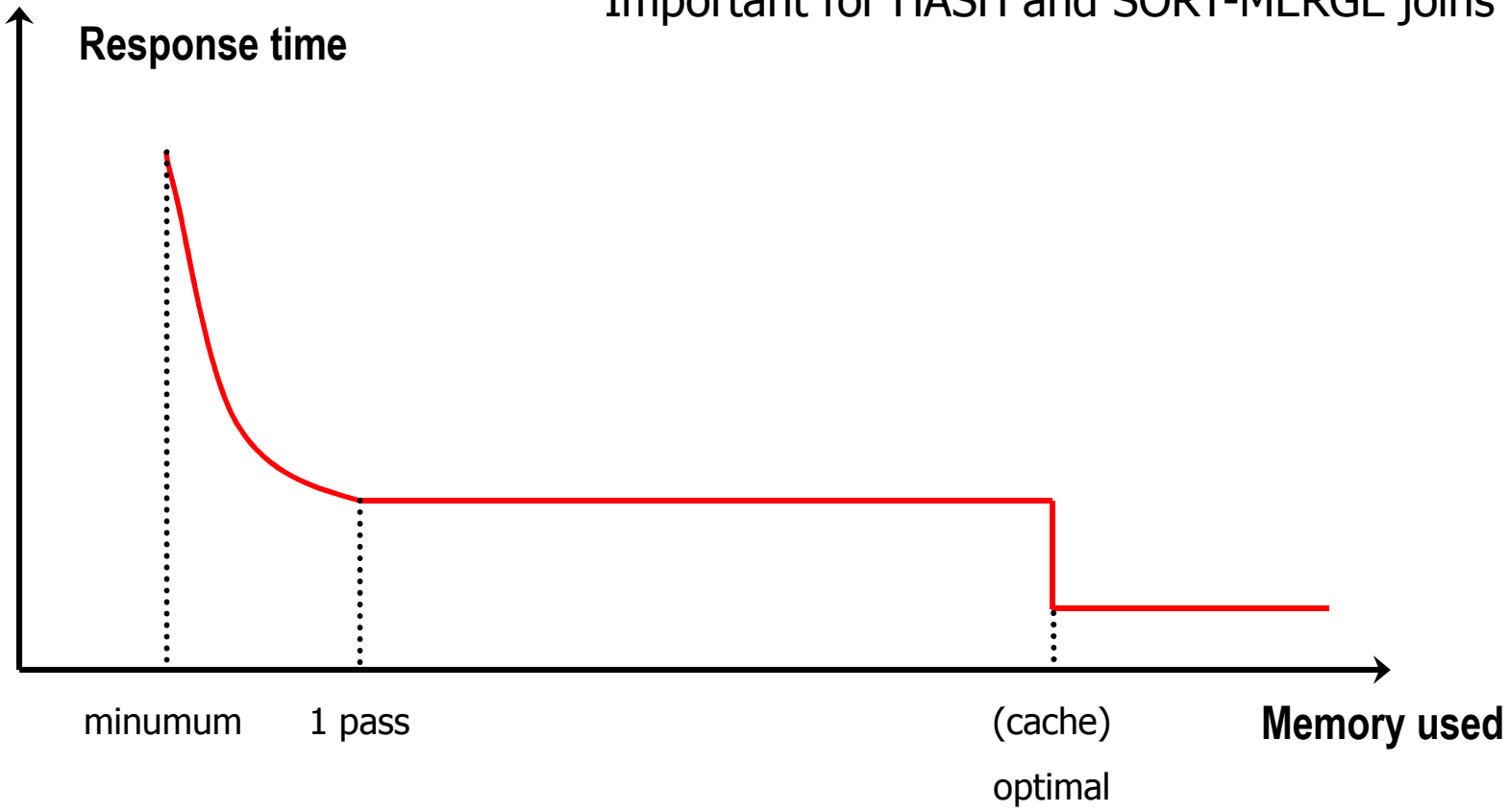
- First data source is (considered) smaller and thus candidate for creating a hash table in memory.
- The second data source is accessed and probved against the memory hash table.

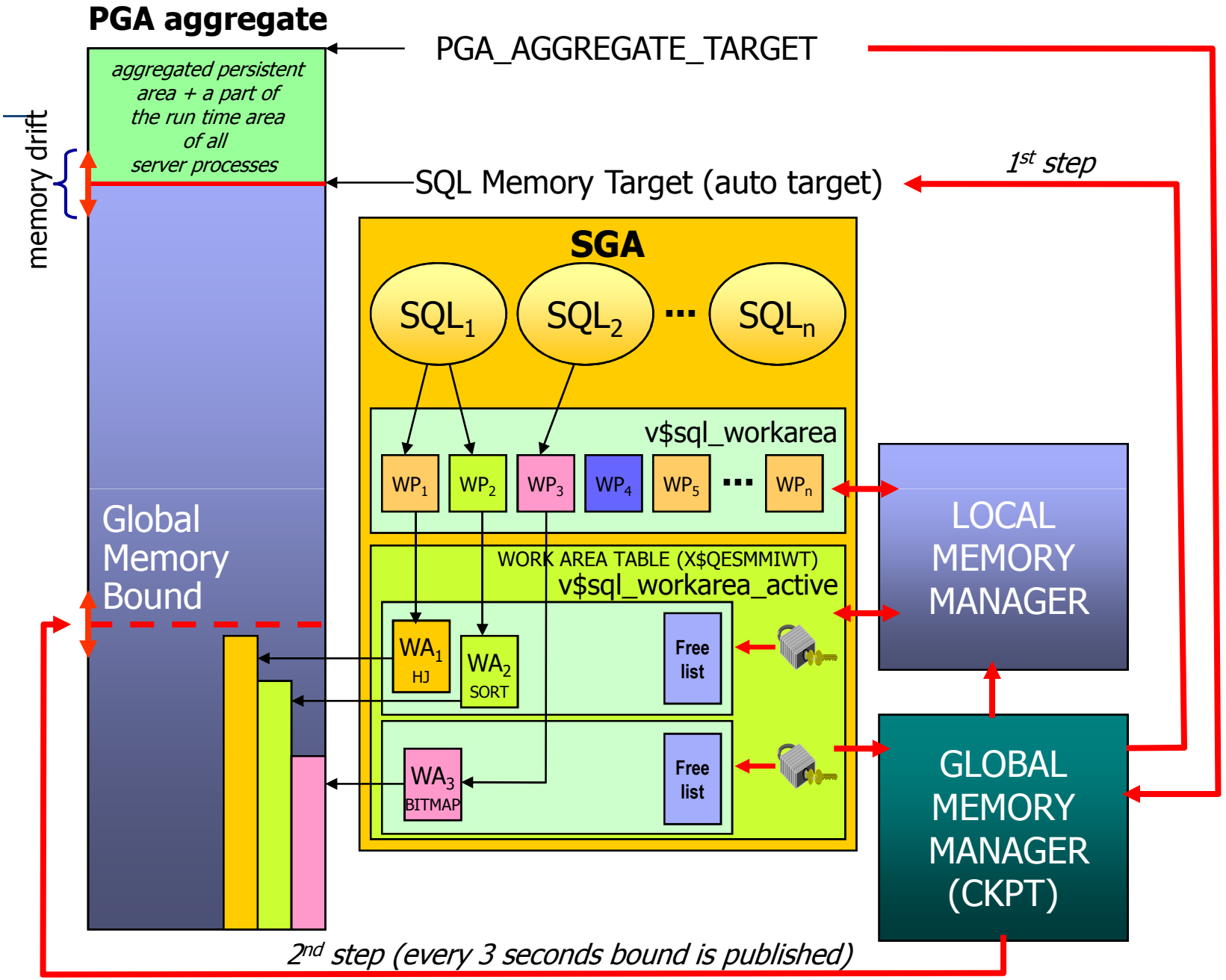
HASH JOIN


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

Work Area Memory Requirements

Important for HASH and SORT-MERGE joins




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)

Clustering of Data

- Many times one can spot that the most expensive operation was
  - `TABLE ACCESS BY LOCAL INDEX ROWID`
- Isn't the ROWID access the best one? – ROWID is actually the pointer to the row.
- Unfortunately the distribution of rows in previous case was such that we had to read many blocks because most likely each block contained only one matching row.
- What can be the remedy?
  - For frequent queries we can organize data in such order that frequent queries will benefit of the actual data clustering.
  - Possible solutions:
    - Using Index organized table (IOT) where the rows are clustered by the value of the primary key (composite key)
    - Sorting data in the partition after there are no more changes in that partition
    - The goal is simple – reduce the number of blocks visited.

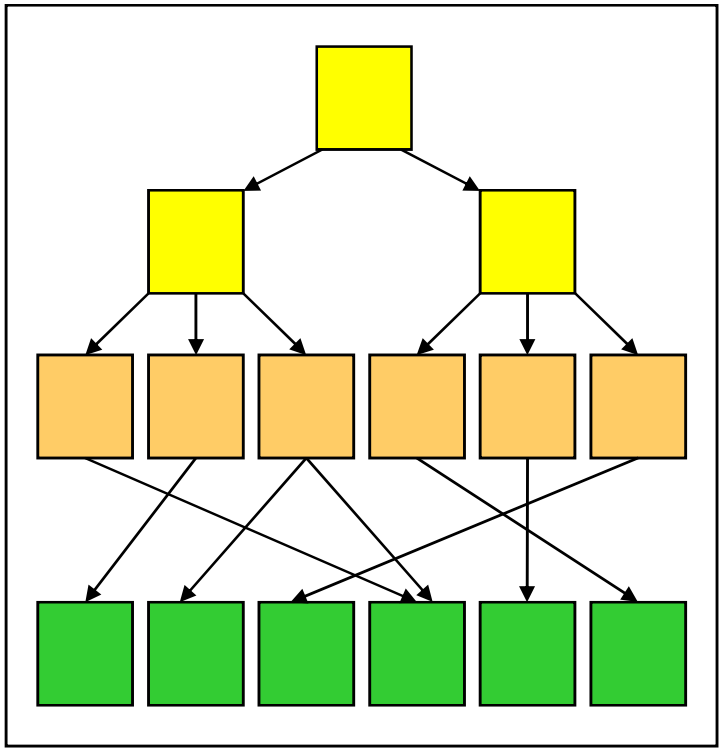
Clustering of Data

- How many times you insert data?
- How many times you update data?
- How many times you access data?

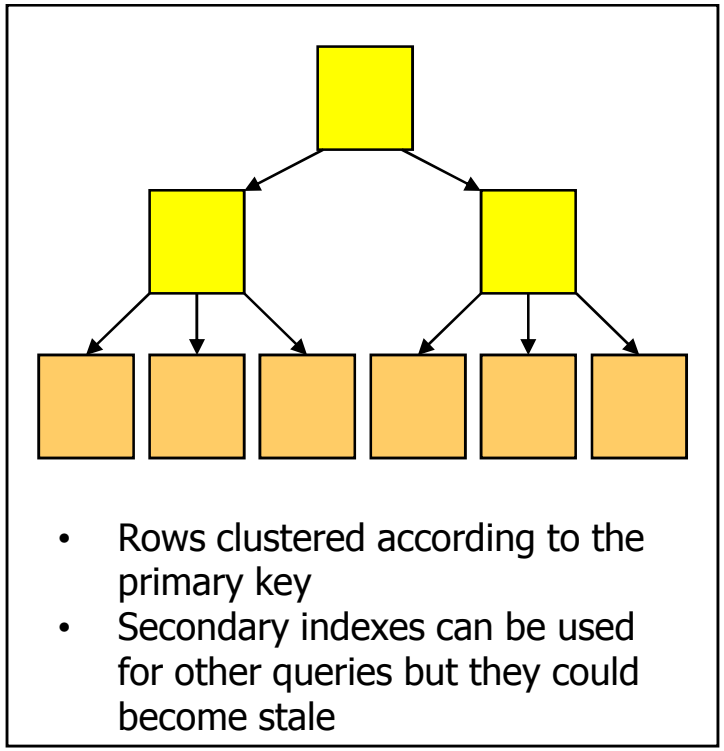
- It is obvious for which operation we should optimize.
- However, optimizing for one scenario may substantially degrade another – so don't forget to test first!!!

Heap and Index Organized Tables

Heap Organized Table



Index Organized Table

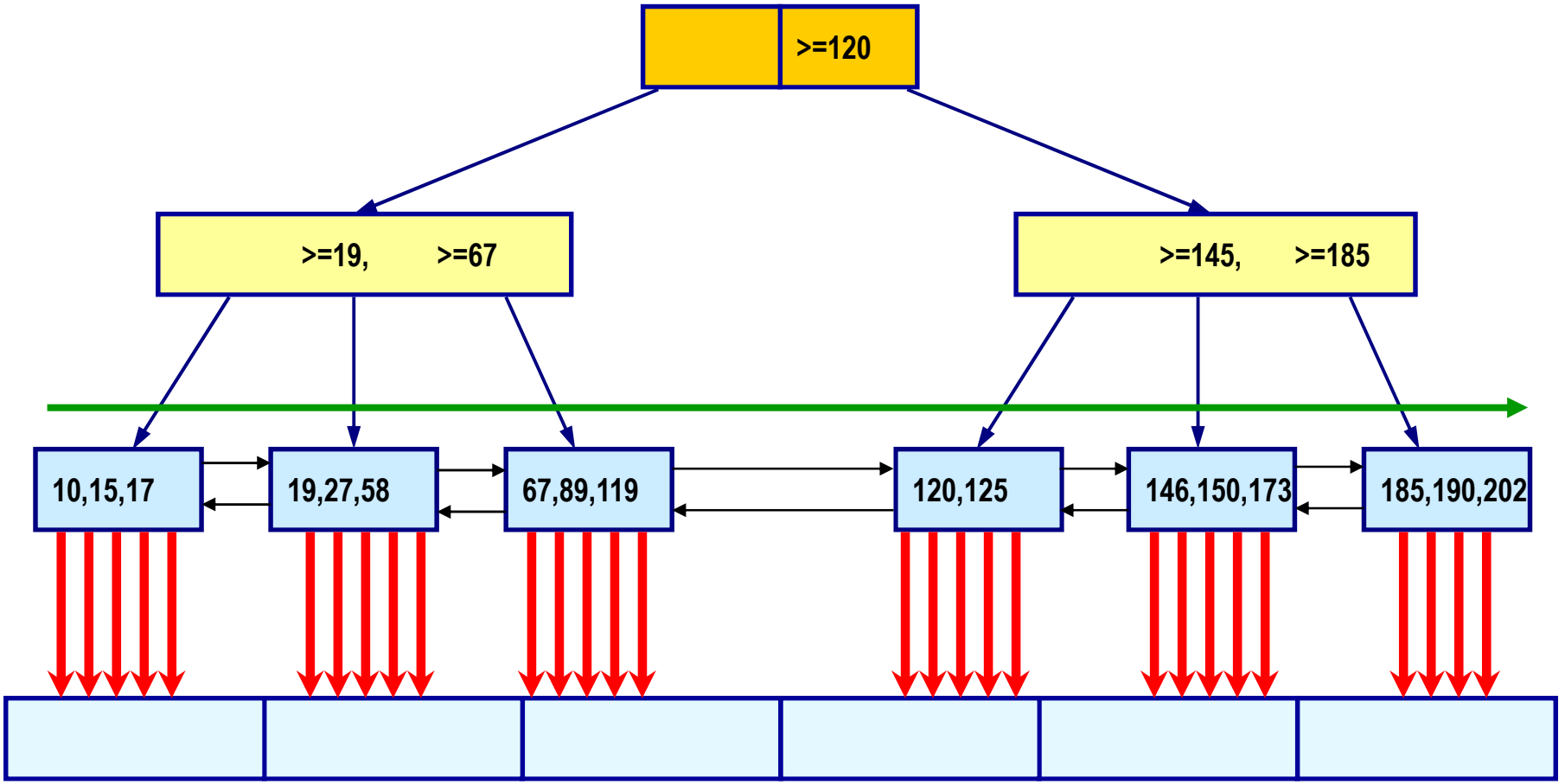


- Rows clustered according to the primary key
- Secondary indexes can be used for other queries but they could become stale

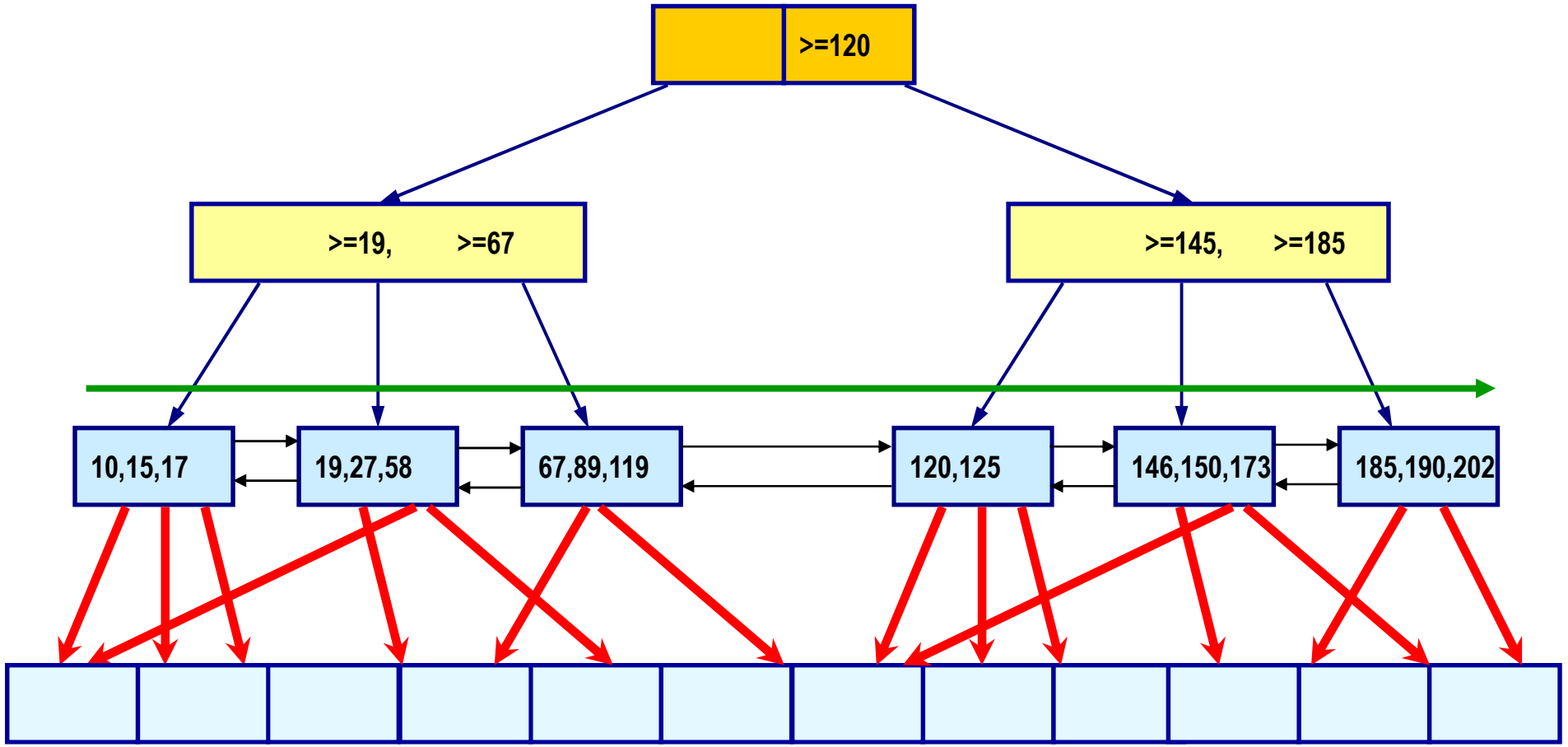
Clustering of Data(2)

- The CBO uses Clustering Factor (part of index statistics) to find out how many blocks should be visited while performing index range scan.
- High clustering factor means higher cost and the CBO can potentially choose another index which will give lower cost.
- Clustering factor influence:
  - No influence on clustering factor for single column indexes!!
  - If the order of columns in a concatenated index key is not important one can lower the clustering factor by putting the column with the lowest clustering factor in the first place. Other columns still have some impact on the clustering factor when there are many rows sharing the same value for the first column.

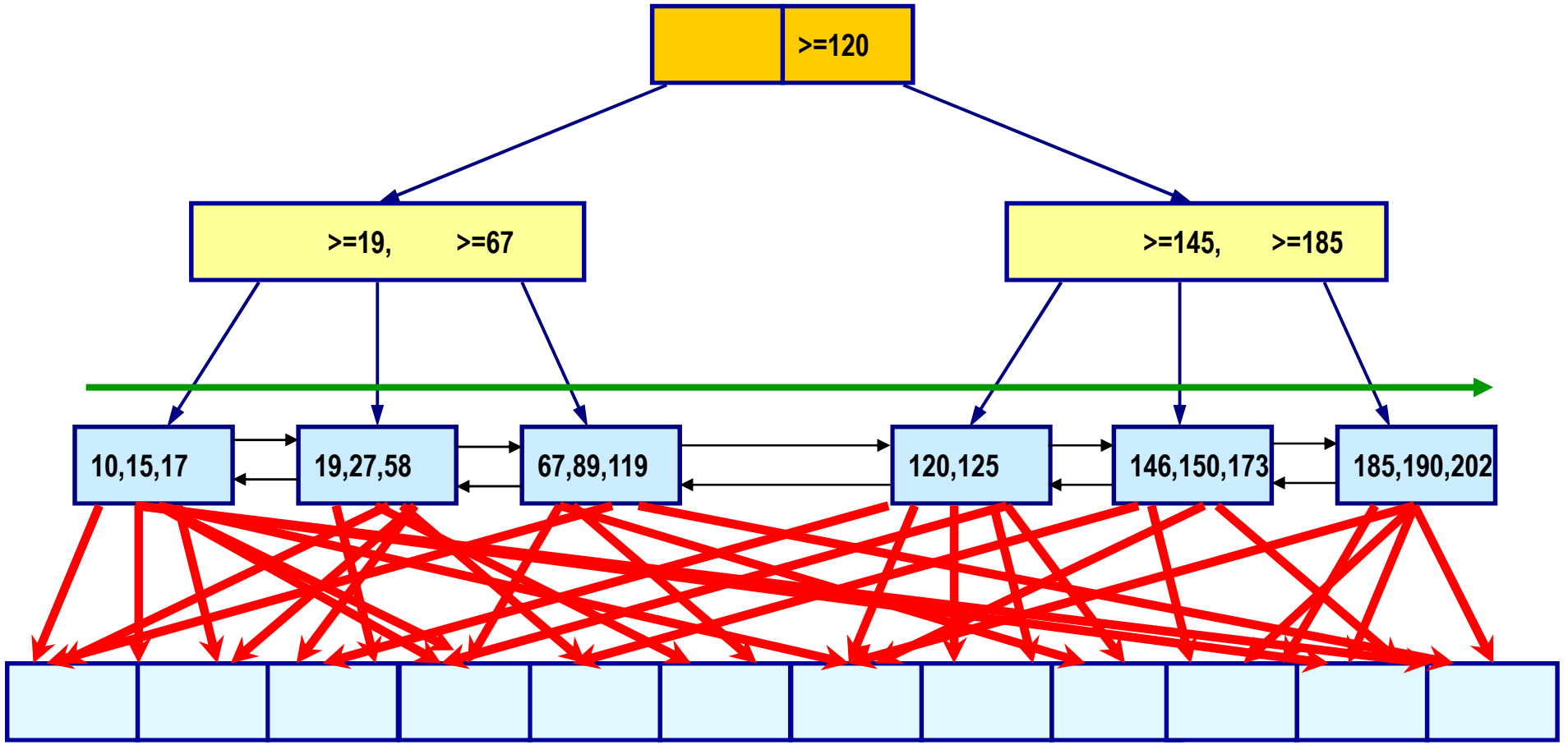
Clustering factor (low)



Clustering factor (medium)

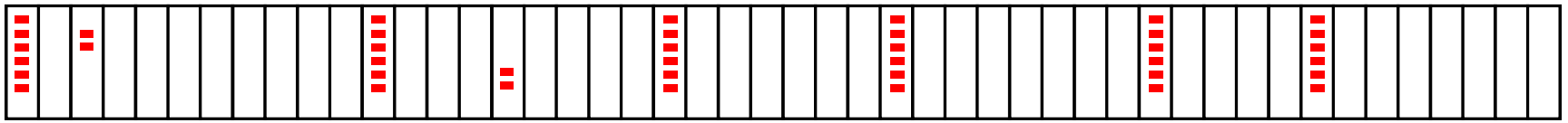


Clustering factor (high)

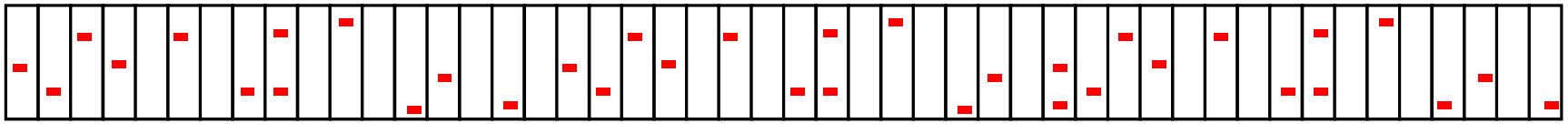


Problem of Data Distribution

Rows are clustered



Rows are spread across the table



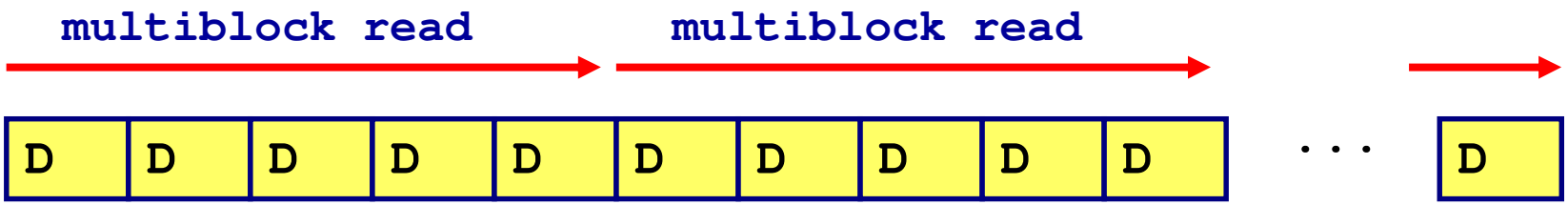
When CBO detects that it would perform more IO using an index then by performing a FTS it decides to use full table scan (FTS)

Full Table Scans

- Full Table Scans (FTS) are not always evil !
- FTS could be the best access path
  - FTS is done by multi block reads (number of blocks is OS dependent)
  - FTS does not flood the buffer cache with table data. Frequently used data blocks will remain in buffer cache.
  - Due to low data clustering we have to read almost all table blocks and therefore a FTS is faster (less I/O operations – they are multi block) than first accessing index with a single block I/O and subsequent access to table blocks again with a single block I/O.
- Alternate method to get data is FAST FULL INDEX SCAN – used when all rows are present in index and the index key contains the value(s) required.

Full Table Scan

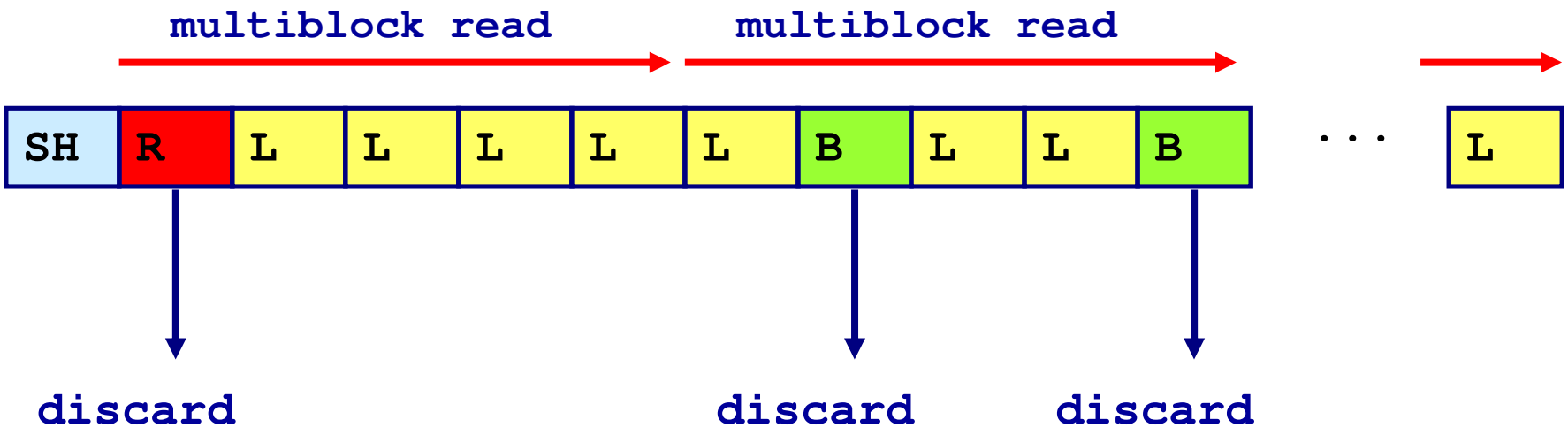
```
db_file_multiblock_read_count = 5
```



Index Fast Full Scan

```
db_file_multiblock_read_count=5
```

```
LEGEND:
SH=segment header
R=root block
B=branch block
L=leaf block
```



Index Table Access

- Many times only PK/FK indexes are created without any "performance indexes"
- PK indexes have low clustering factor, FK usually very high.
- Do we need to create another index?
  - Sometimes yes, but many times we can just add some additional columns to the index to become more 'selective'.
  - Oracle supports using non-unique indexes to enforce uniqueness!
  - Don't do something like this (I have seen this too many times):
    - Index1 on col1 (PK index)
    - Index2 on col1, col2
    - Index3 on col1, col2, col3 (this one can satisfy all requirements)
  - Be aware that every index adds about 110% more logical I/O during DML operations because it has to be maintained
- Index blocks are cached better in buffer cache than table block (only because they are used more frequently). Hence traversing them will infer mostly logical I/O and only small physical I/O.
- OPTIMIZER_INDEX_CACHING init parameter defines the percentage of cached index blocks that could be expected (NL joins, IN-list operations).

How Much of Index Key Can Be Used?

	C1	C2	C3	C4
Row1	**100**	**20**	**1**	**P**
Row2	**100**	**20**	**2**	**D**
Row3	**100**	**20**	**3**	**G**

- Index1 key is C1,C2,C3 (let us assume this is a FK index)
- The query is

```
select c3,c4
from T
where c1 = 100
and   c2 = 20
```

- Do we need a subsequent table access?
- What if we create index with index key C1,C2,C3,C4
- No table access at all, index key contains all required data!
- The foreign key index can have additional column(s)

Special cases

- Summing (when this is a very frequent operation)
  - INDEX RANGE SCAN can be used for summing if the value to be summed is present in the index key.
  - Usually one can just add value to the most appropriate index (PK index)
  - Result: faster execution due to lower LIO, for such query no table blocks are visited hence they don't need to be in the buffer cache.
- ...

Increasing the Index Selectivity

- Making index more selective
  - Add column(s) to the index in order to reduce the number of rows which are returned by the index range scan - filter out as many rows as possible already at the index level.
  - Reasons:
    - Table blocks are not cached so well, therefore likely more PIO – PIO is slow in comparison with LIO.
    - Index range scan scans whole index blocks, while table access reads just one or several rows from a table block.
- When this should be used:
  - Huge index range scans which turn out to have low cardinality after filtering out the rows at table level.
- Do this only if you have very frequent queries which can benefit from this optimization.
  - Drawback – index is bigger what really increases the amount of workload – trade off process

Index Compression

- Making index smaller – more rows will be cached

Regular index

```
Key values
Column1 Column2 Column3
      1       1 'ABCDH'
      1       1 'GHSLL'
      1       2 'SHHKK'
      1       2 'SZGHL'
      1       2 'HKKKS'
 ....
```

Compressed index

```
Prefix #0 row#0 1,1  #2
Prefix #1 row#0 1,2  #3

Entry #0: 'ABCDH'  prefix#: 0
Entry #1: 'GHSLL'  prefix#: 0
Entry #2: 'SHHKK'  prefix#: 1
Entry #3: 'SZGHL'  prefix#: 1
Entry #4: 'HKKKS'  prefix#: 1
```

```
create unique index XXX on YYY( c1, c2, c3 ) compress 2;
```

Join Elimination And Constraints

- Eliminate unnecessary joins if there are constraints defined on join columns. If join has no impact on query results it can be eliminated.
  - e.departmens_id is foreign key and joined to primary key d.department_id
- Eliminate unnecessary outer joins – doesn't even require primary key – foreign key relationship to be defined.

```
SQL> select e.first_name, e.last_name, e.salary
     from employees e,
          departments d
     where e.department_id = d.department_id;
```

```
-------------------------------------+-----------------------------------+
| Id  | Operation          | Name     | Rows  | Bytes | Cost  | Time      |
-------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT   |          |       |       |     3 |           |
| 1   |  TABLE ACCESS FULL | EMPLOYEES|   106 |  2332 |     3 |  00:00:01 |
-------------------------------------+-----------------------------------+
Predicate Information:
----------------------
1 - filter("E"."DEPARTMENT_ID" IS NOT NULL)
```

JE - Join Elimination (2)

- **Purpose of join elimination**
  - Such situations are very common when a view is used which contains a join and only a subset of columns is used.
  - In 11gR1 the optimization became available also for ANSI compliant joins.

- **Known Limitations** (Source: Optimizer group blog)
  - Multi-column primary key-foreign key constraints are not supported.
  - Referring to the join key elsewhere in the query will prevent table elimination. For an inner join, the join keys on each side of the join are equivalent, but if the query contains other references to the join key from the table that could otherwise be eliminated, this prevents elimination. A workaround is to rewrite the query to refer to the join key from the other table.

Partition Pruning

- CBO ability to eliminate partitions that do not need to be scanned, and only access those which are concerned by predicates
- Used in
  - Range, LIKE, equality, and IN-list predicates on the range or list partitioning columns
  - Equality and IN-list predicates on the hash partitioning columns.
  - Range partition level and at the hash or list subpartition level using on composite partitioned objects
  - Reduces the amount of data retrieved from disk and shortens the response time
- Can be applied on tables and indexes
- With a global partitioned index, partition pruning also eliminates index partitions even when the partitions of the underlying table cannot be eliminated.

Types of Partition Pruning

- **Static pruning**
  - Occurs at statement compile-time - the partitions to be accessed are known in advance
  - Constant literal on the partition key column is used
- **Dynamic pruning**
  - Occurs at run-time - the exact partitions to be accessed by a statement are not known in advance
  - example: bind variables, partitions determined by a subquery,...

How to Identify Pruning

- PLAN_TABLE columns for partition pruning:
  - PARTITION_ID: Step where pruning occurs
  - PARTITION_START: First partition accessed
  - PARTITION_STOP: Last partition accessed
  - FILTER_PREDICATES: The predicate applied
  - PARTITION operation with an OPTION associated depending on the number of partitions accessed
    - All: All partitions must be accessed
    - Single: Only one partition is accessed
    - Iterator: When accessing many partitions
    - Subquery: When a subquery is used
    - OR: When OR is used
    - Inlist: Same as iterator but for IN-List predicate
    - Invalid: No partition matches the predicate
    - Bloom filter: used for partition pruning instead of cost based subquery pruning (11g, 10gR2 as well)

Static Partition Pruning

- The partitions to be accessed are determined at compile time.
- Occurs when:
  - the predicates on the partitioning columns use un equality or a range predicate.
  - the predicates must only use constants to enable CBO to determine the start and stop partition at compile time.
- If static partition pruning is used the actual partition numbers show up in the **Pstart** and **Pstop** columns of the explain plan.

Partition Pruning Techniques

- Basic Partition Pruning
  - equality, range, and IN-list are the most commonly used cases of partition pruning.
- Advanced partition pruning techniques
  - presence of more complex predicates or SQL statements
    - bind variables – depends on the current value of bind variable - **KEY**
    - IN-list partition pruning - **KEY(I)**
    - Partition pruning based on subquery - cost-based decision - **KEY(SQ)**
    - Partition pruning based on OR - **KEY(OR)**
    - **Bloom filtering** used in latest versions of 10g and in 11g – not documented yet!!! - **:BF0000**

Some examples of Dynamic Partition Pruning

- The actual partitions are determined in runtime.
- For IN-list predicate the plan output will show KEY(I) in the partition start and stop column.

```
SQL> explain plan for select * from sales where time_id in ('23.11.01','25.11.01');

Explained.

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name           | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                |  1259 | 36511 |   186   (0)| 00:00:03 |       |       |
|   1 |  INLIST ITERATOR                   |                |       |       |            |          |       |       |
|   2 |   PARTITION RANGE ITERATOR         |                |  1259 | 36511 |   186   (0)| 00:00:03 |KEY(I) |KEY(I) |
|   3 |    TABLE ACCESS BY LOCAL INDEX ROWID| SALES          |  1259 | 36511 |   186   (0)| 00:00:03 |KEY(I) |KEY(I) |
|   4 |     BITMAP CONVERSION TO ROWIDS    |                |       |       |            |          |       |       |
|*  5 |      BITMAP INDEX SINGLE VALUE     | SALES_TIME_BIX |       |       |            |          |KEY(I) |KEY(I) |
-----------------------------------------------------------------------------------------------------------

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

   5 - access("TIME_ID"='23.11.01' OR "TIME_ID"='25.11.01')
```

PARTITION RANGE SUBQUERY (10g)

```
SQL> explain plan for
  2  SELECT /*+ use_hash(s t) */ t.day_number_in_month, SUM(s.amount_sold)
  3  FROM sales s, times t
  4  WHERE s.time_id = t.time_id
  5  AND t.calendar_month_desc='2000-12'
  6  GROUP BY t.day_number_in_month;
-------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  |Cost | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |    20 |  645|       |       |
|   1 |  HASH GROUP BY             |       |    20 |  645|       |       |
|*  2 |   HASH JOIN                |       | 19153 |  637|       |       |
|*  3 |    TABLE ACCESS FULL       | TIMES |    30 |   17|       |       |
|   4 |    PARTITION RANGE SUBQUERY|       |   918K|  574|KEY(SQ)|KEY(SQ)|
|   5 |     TABLE ACCESS FULL      | SALES |   918K|  574|KEY(SQ)|KEY(SQ)|
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("S"."TIME_ID"="T"."TIME_ID")
   3 - filter("T"."CALENDAR_MONTH_DESC"='2000-12')
```

Same Statement with NL Join

```
SQL> explain plan for
  2  SELECT t.day_number_in_month, SUM(s.amount_sold)
  3  FROM sales s, times t
  4  WHERE s.time_id = t.time_id
  5  AND t.calendar_month_desc='2000-12'
  6  GROUP BY t.day_number_in_month;


-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name           | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                |    20 |   640 |   232   (1)| 00:00:03 |       |       |
|   1 |  HASH GROUP BY                      |                |    20 |   640 |   232   (1)| 00:00:03 |       |       |
|   2 |   NESTED LOOPS                      |                |       |       |            |          |       |       |
|   3 |    NESTED LOOPS                     |                | 19153 |   598K|   231   (0)| 00:00:03 |       |       |
|*  4 |     TABLE ACCESS FULL               | TIMES          |    30 |   570 |    18   (0)| 00:00:01 |       |       |
|   5 |     PARTITION RANGE ITERATOR        |                |       |       |            |          |   KEY |   KEY |
|   6 |      BITMAP CONVERSION TO ROWIDS    |                |       |       |            |          |       |       |
|*  7 |       BITMAP INDEX SINGLE VALUE     | SALES_TIME_BIX |       |       |            |          |   KEY |   KEY |
|   8 |    TABLE ACCESS BY LOCAL INDEX ROWID| SALES          |   629 |  8177 |   231   (0)| 00:00:03 |     1 |     1 |
-------------------------------------------------------------------------------------------------------------------

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

   4 - filter("T"."CALENDAR_MONTH_DESC"='2000-12')
   7 - access("S"."TIME_ID"="T"."TIME_ID")
```

Same Statement with Bloom Filter (11g)

```
SQL> explain plan for
  2  SELECT /*+ use_hash(s t) */ t.day_number_in_month, SUM(s.amount_sold)
  3  FROM sales s, times t
  4  WHERE s.time_id = t.time_id
  5  AND t.calendar_month_desc='2000-12'
  6  GROUP BY t.day_number_in_month;

Explained.
-------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |          |    20 |   640 |   517   (4)| 00:00:07 |       |       |
|   1 |  HASH GROUP BY                 |          |    20 |   640 |   517   (4)| 00:00:07 |       |       |
|*  2 |   HASH JOIN                    |          | 19153 |   598K|   515   (4)| 00:00:07 |       |       |
|   3 |    PART JOIN FILTER CREATE     | :BF0000  |    30 |   570 |    18   (0)| 00:00:01 |       |       |
|*  4 |     TABLE ACCESS FULL          | TIMES    |    30 |   570 |    18   (0)| 00:00:01 |       |       |
|   5 |    PARTITION RANGE JOIN-FILTER |          |   918K|    11M|   493   (3)| 00:00:06 |:BF0000|:BF0000|
|   6 |     TABLE ACCESS FULL          | SALES    |   918K|    11M|   493   (3)| 00:00:06 |:BF0000|:BF0000|
-------------------------------------------------------------------------------------------------------

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

   2 - access("S"."TIME_ID"="T"."TIME_ID")
   4 - filter("T"."CALENDAR_MONTH_DESC"='2000-12')
```

- :BF0000 stands for Bloom filter

How Bloom Filter Works?

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name    | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |         |    20 |   640 |   517   (4)| 00:00:07 |       |       |
|   1 |  HASH GROUP BY                 |         |    20 |   640 |   517   (4)| 00:00:07 |       |       |
|*  2 |   HASH JOIN                    |         | 19153 |   598K|   515   (4)| 00:00:07 |       |       |
|   3 |    PART JOIN FILTER CREATE     | :BF0000 |    30 |   570 |    18   (0)| 00:00:01 |       |       |
|*  4 |     TABLE ACCESS FULL          | TIMES   |    30 |   570 |    18   (0)| 00:00:01 |       |       |
|   5 |    PARTITION RANGE JOIN-FILTER|         |   918K|    11M|   493   (3)| 00:00:06 |:BF0000|:BF0000|
|   6 |     TABLE ACCESS FULL          | SALES   |   918K|    11M|   493   (3)| 00:00:06 |:BF0000|:BF0000|
-----------------------------------------------------------------------------------------------------------

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

   2 - access("S"."TIME_ID"="T"."TIME_ID")
   4 - filter("T"."CALENDAR_MONTH_DESC"='2000-12')
```

- Step 4 – table TIMES is accessed and rows passing the filter CALENDAR_MONTH_DESC"='2000-12` are returned
- STEP 3 – a bloom filter is created with the information which partitions contain the relevant rows; at the same time a memory hash table is built for the outer data source
- Step 5 and 6 – only partitions which contain relevant data according to the bloom filter :BF0000 are accessed and the rows are probed against memory hash table and if they match `("S"."TIME_ID"="T"."TIME_ID")` a join is preformed

Parallel Query Servers (Slaves)

- Controlled by the QC process
- Slaves are allocated in slave sets, which act as either ***producers*** or ***consumers.***
- A slave set may act as both producer and consumer at different stages within a query.
- Do most of the work for a parallel query

Parallel Execution Example

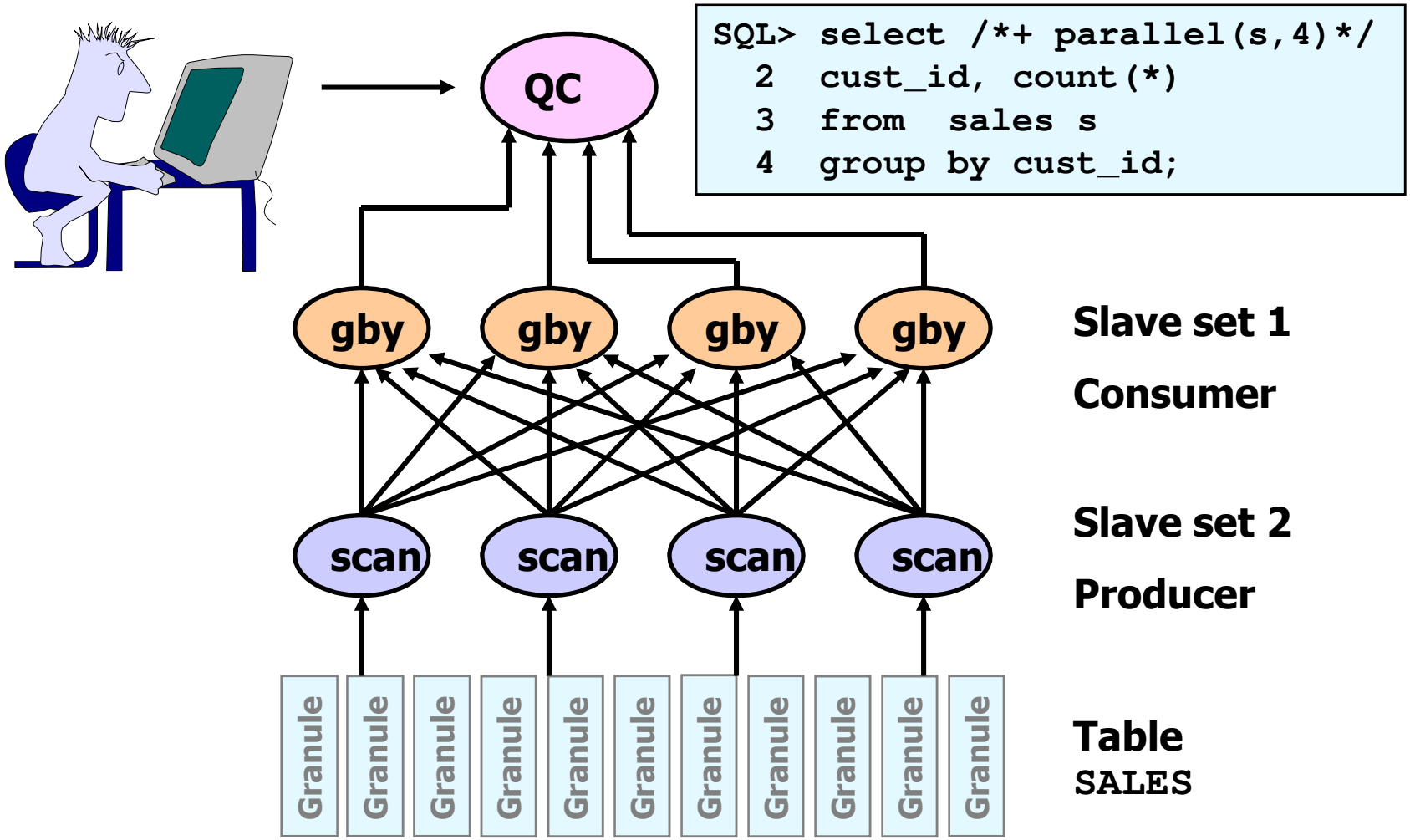
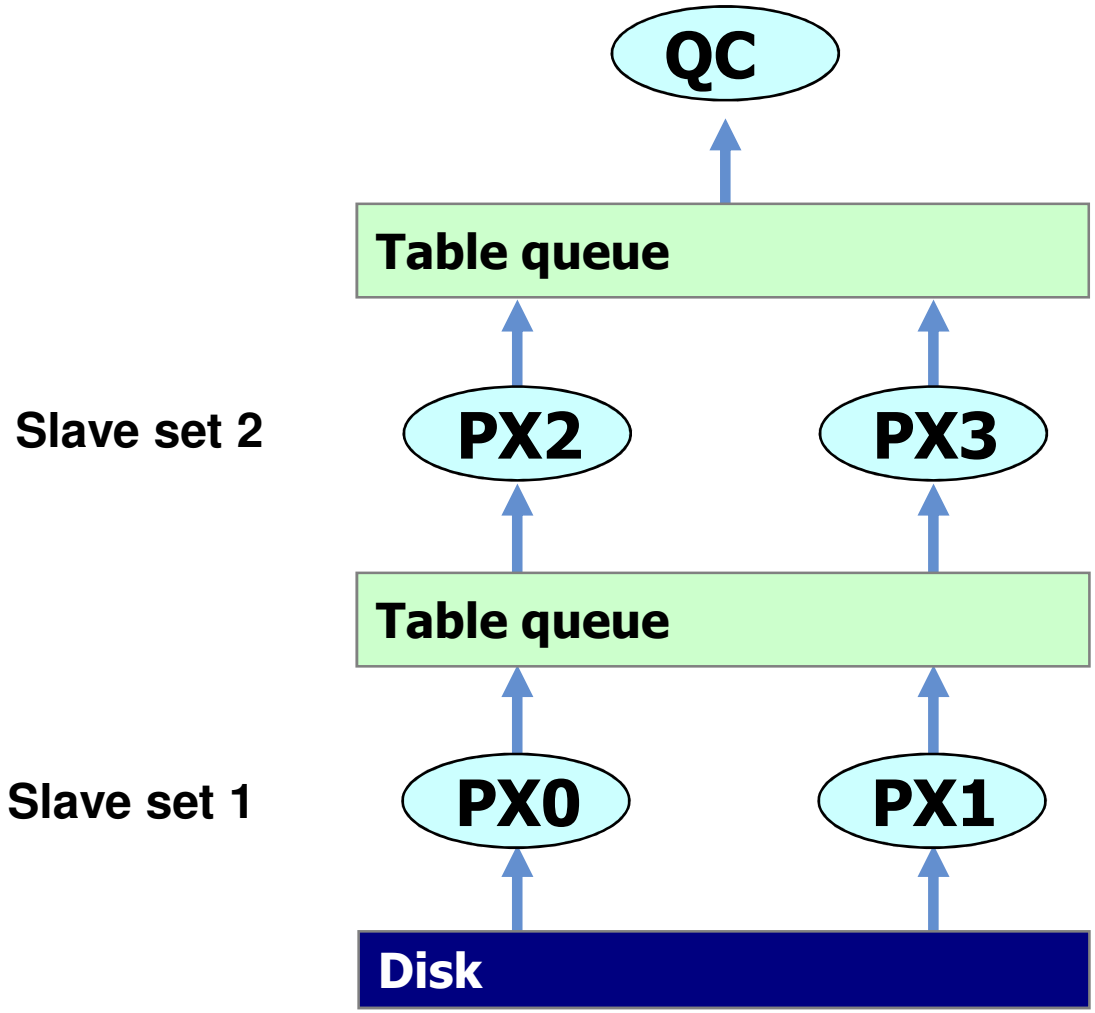


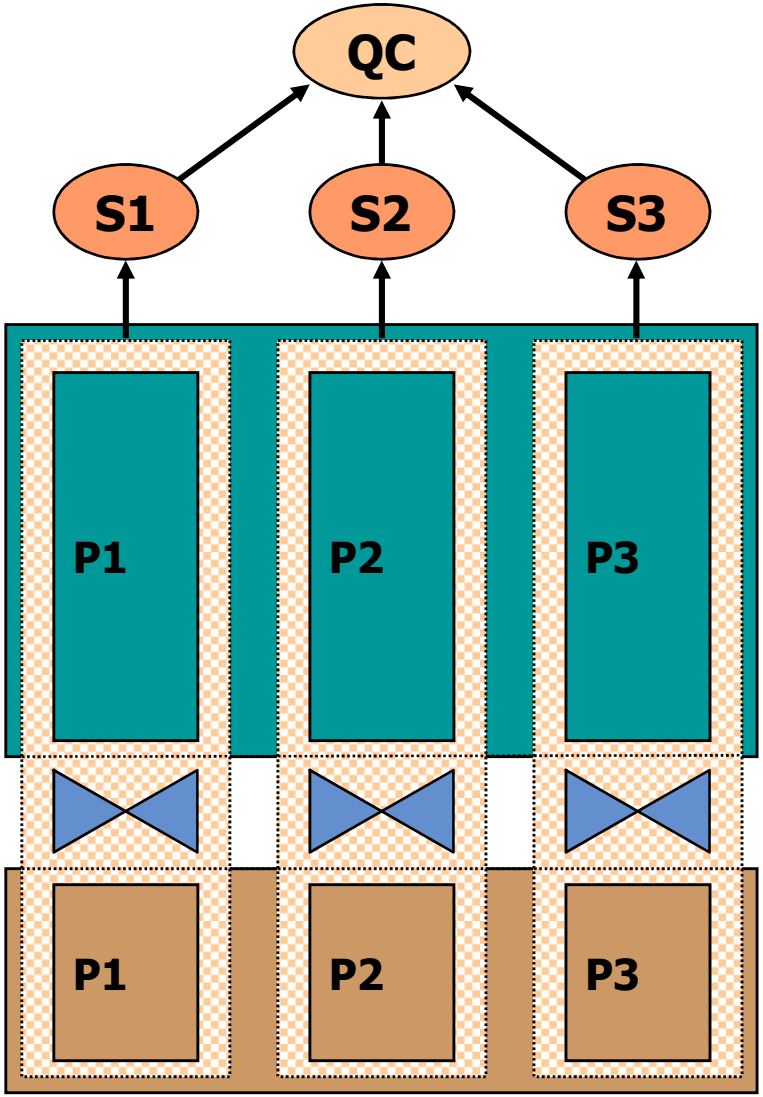
Table Queues (TQ)

- Table Queues (TQ) are abstract communication mechanism
- Slave sets are interconnected by TQ.
- TQ are numbered uniquely based on the sequence **SYS.ORA_TQ_BASE$**

Table Queues



Full Partition-Wise Join



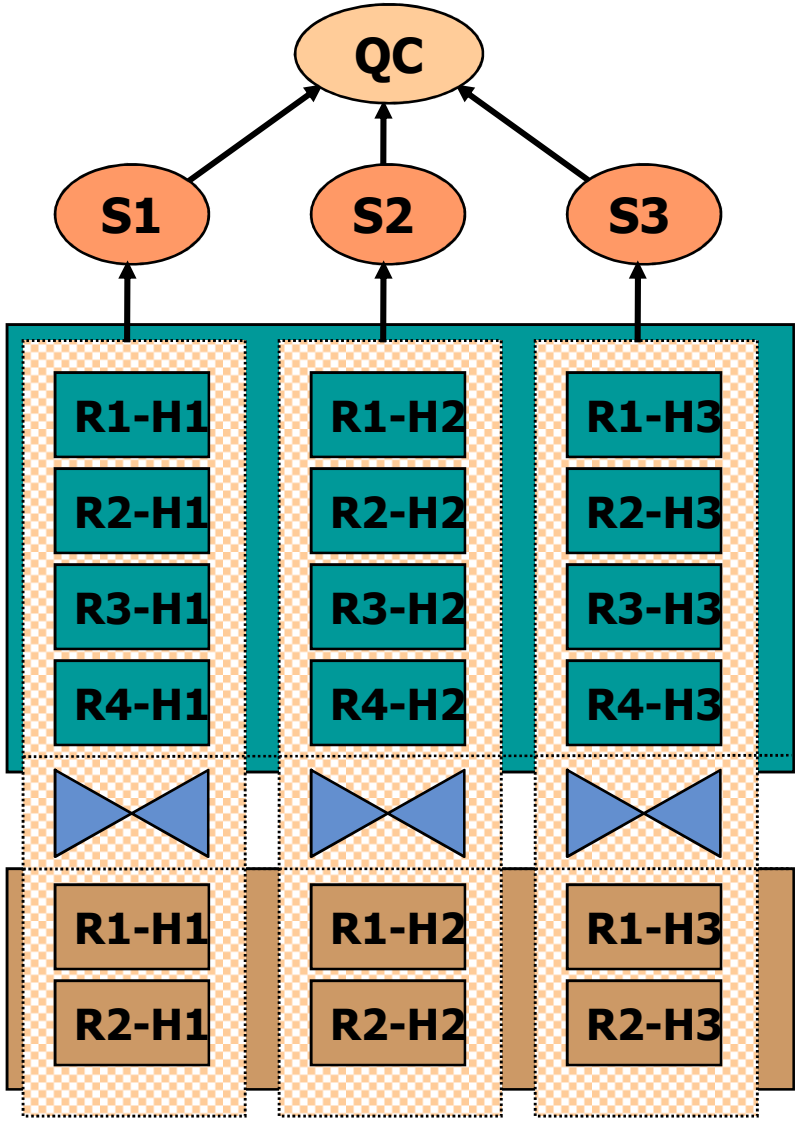

```
select  sum(amount_sold)
from    sales_hash s,
        customers_hash c
where   s.cust_id = c.cust_id
group   by cust_last_name;
```

Partitioned tables:

`customers_hash (cust_id)`

`sales_hash (cust_id)`

Partial Partition-Wise Join



Composite partitioned tables:

```
sales (
sale_date range 4,
department_id hash 3)
```

```
departments(
manager_id range 2,
department_id hash 3)
```

Conclusions

- When joining tables avoid unnecessary work – reduce as much the number of blocks visited (LIO/PIO).
- Prepare carefully your index strategy to make your indexes more selective in order to reduce the amount of LIO/PIO.
- By rewriting your SQL statements you can achieve things which the CBO currently can't do.
- To be successful you should know your data and also some internals how Oracle (CBO) works.

Thank you for your interest!

Q&A