

Query Optimisation Part 2

Brendan Tierney

1

Understanding the Workload

- For each query in the workload
 - What relations does it access ?
 - What attributes are retrieved ?
 - What attributes are involved in selection / join conditions ?
 - How selective are these conditions likely to be
- For each update in the workload
 - What attributes are involved in the selection / join condition
 - How selective are these conditions likely to be
 - The type of update (INSERT/UPDATE/DELETE) and the attributes that are affected
 - The restrictiveness of the update

2

2

Choices of Indexes

- What indexes should be create ?
 - What tables should have indexes ?
 - What attributes should be the search keys ?
 - How many indexes should be built ?
 - What type of index should it be ?
- One approach
 - Consider the most important queries in turn.
 - Consider the best plan using the current indexes and see if a better plan is possible with an additional index
 - If so create it, but this implies understanding of how a DB evaluates queries and creates query evaluation plans
 - Before creating an index, must also consider the impact on updates in the workload
 - Trade-off: Indexes can make queries go faster, updates slower, require disk space.

3

3

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions
 - Order of attributes are important
- Try to choose indexes that benefit as many queries as possible

```
SELECT      e.employee_id, e.salary, e.commission_pct
FROM        employees e, departments d
WHERE       e.job_id = 'SA_REP'
AND         e.department_id = d.department_id
AND         d.location_id = 2500
```

4

4

Cost Based Optimisation

- Cost-Based Optimisation
 - estimating and comparing the costs of executing a query using different execution strategies
 - should choose the strategy with the lowest cost estimate.
- Accurate cost estimates are required so that different strategies are compared fairly and realistically.
 - But the number of strategies to be considered must be limited
 - much time will be spent on making cost estimates for the different execution strategies.
- Cost functions are based on estimates
 - not on the exact cost functions
 - the query optimiser may select a query execution strategy that is not the optimal one.
 - examining the execution strategy chosen and providing **hints** to the query optimiser to process the query in a certain way.

5

5

Cost Based Optimisation

- Aim of CBO is to choose most efficient approach.
- Use formulae to estimate costs for a number of options
 - Aim is to select one with lowest cost.
- Cost of disk access is usually the dominant cost
- Many estimates are based on cardinality of the relation, so need to be able to estimate this.
 - Some DBs look at data distributions/histograms for each attribute
- Success of estimation depends on amount and currency of statistical information DBMS holds.
- Keeping statistics current can be problematic.
- If statistics updated every time a tuple is changed, this would impact performance.
- DBMS could update statistics on a periodic basis, for example nightly, or whenever the system is idle

6

6

Costs

- Costs of executing a query includes:
 - Access cost to secondary storage
 - Searching, reading and writing data blocks to/from files. Depends on indexing, data blocks allocated contiguously or scattered.
 - Storage cost
 - Cost of storing intermediate information generated by query execution strategy.
 - Computation cost
 - In-memory costs, calculations, searches, joins, sorting.
 - Memory usage cost
 - How many buffers required, memory management.
 - Communication cost
 - Time to move data from location to location

7

7

QO Aims

- For a large database the main emphasis is on *minimising* the access costs to *secondary storage*
- For small databases, where most of the data can be completely stored in memory, the emphasis is on minimising computation costs.
- In distributed databases, the emphasis is on minimising the communication costs.
- Therefore the architecture and configuration of your database is critical for the optimal performance of your queries.

8

8

Example

- For every project located in Stafford retrieve the project number, controlling department number, the department manager's last name, address and birth date.

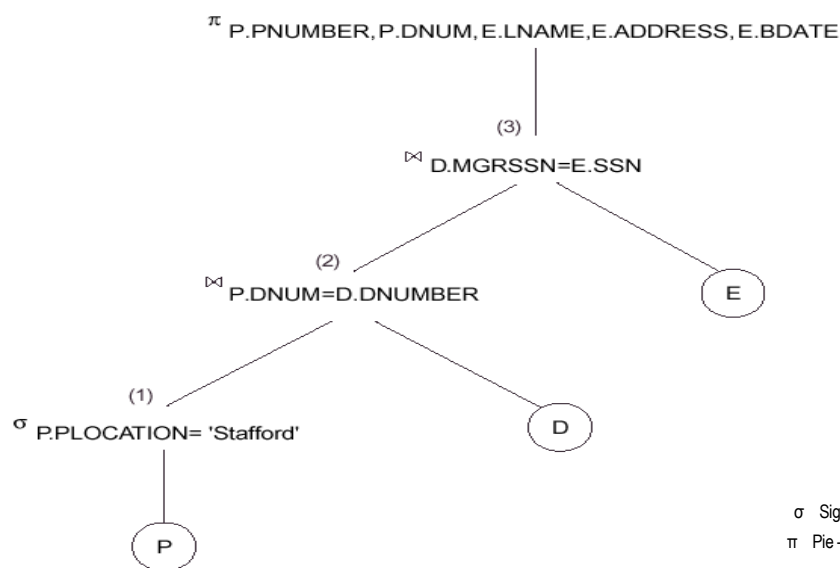
```
SELECT  pnumber, dnum, lname, address, bdate
FROM    PROJECT p, DEPARTMENT d, EMPLOYEE e
WHERE   p.dnum = d.dnumber
AND     d.mdrssn = e.ssn
AND     p.plocation = 'Stafford'
```

- Draw the query tree

9

9

Example



10

Example

- We need to gather the statistics of the tables involved
 - analyse table <table name> compute statistics
 - analyse index <index name> compute statistics
 - Dbms_stats pl/sql package

```
EXEC DBMS_STATS.gather_table_stats(USER, 'user_data', cascade => TRUE);
```

- In example the analysis of the data we obtain the following information which can be used in cost-based optimisation.

TABLE_NAME	COLUMN_NAME	NUM_DISTINCT	LOW_VALUE	HIGH_VALUE
PROJECT	PLOCATION	200	1	200
PROJECT	PNUMBER	2000	1	2000
PROJECT	DNUM	50	1	50
DEPARTMENT	DNUMBER	50	1	50
DEPARTMENT	MGRSSN	50	1	50
EMPLOYEE	SSN	10000	1	10000
EMPLOYEE	DNO	50	1	50
EMPLOYEE	SALARY	500	1	500

TABLE_NAME	NUM_ROWS	BLOCKS	INDEX_NAME	UNIQUENES	BLEVEL*	LEAF_BLOCKS	DISTINCT_KEYS
PROJECT	2000	100	PROJ_PLOC	NONUNIQUE	1	4	200
DEPARTMENT	50	5	EMP_SSN	UNIQUE	1	50	10000
EMPLOYEE	10000	2000	EMP_SAL	NONUNIQUE	1	50	500

11

Example

- Using Cost-Based Optimisation there are 4 possible combinations of join ordering, which do not cause Cartesian products
 - Project – Department – Employee
 - Department – Project – Employee
 - Department – Employee – Project
 - Employee – Department - Project
- We will take the first combination and calculate the costs
 - But the DBMS must do all 4 of them

12

12

Example

- Project – Department – Employee
 - Determine the access methods of Project and Department.
 - Department has no index so the only access method is a full table scan.
 - Project - can either do a full table scan or user the Proj_Ploc index.
 - The costs of each of these must be calculated
 - Because the Proj_Ploc index is non-unique we can estimate the possible number of data blocks to be accessed by assuming a uniform distribution of the data in the table.
 - This gives approx. $(\text{num records}/\text{num distinct}) = (2000/200) = 10$.
 - But we will do a scan of the index, giving 2 block reads (assume you will only scan half the index – 4 leaf blocks)
 - Gives a total of 12 block reads.
 - This is less than a full table scan on the Project table (100)
 - We will perform a full table scan on Department which is 5 block reads
 - plus the results from the Project table (small size – 1 block).
 - These are combined to give a temporary relation which is used with the Employee part of the query.

13

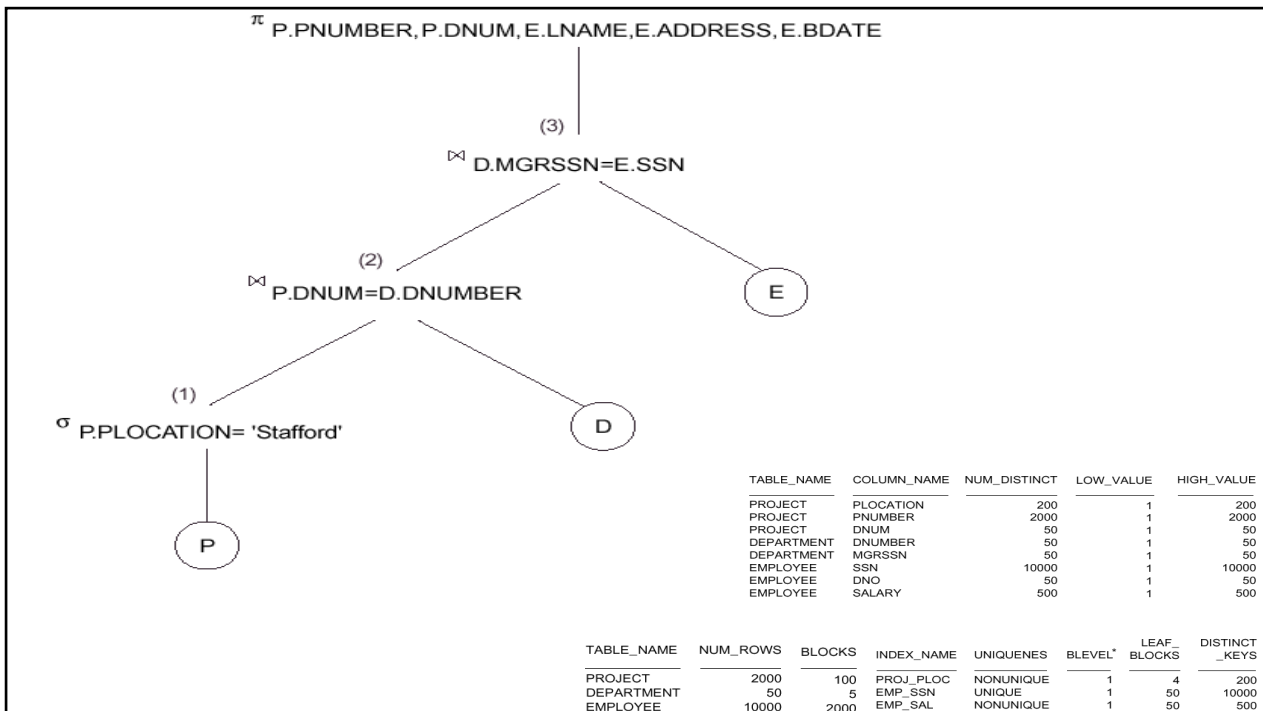
13

Example

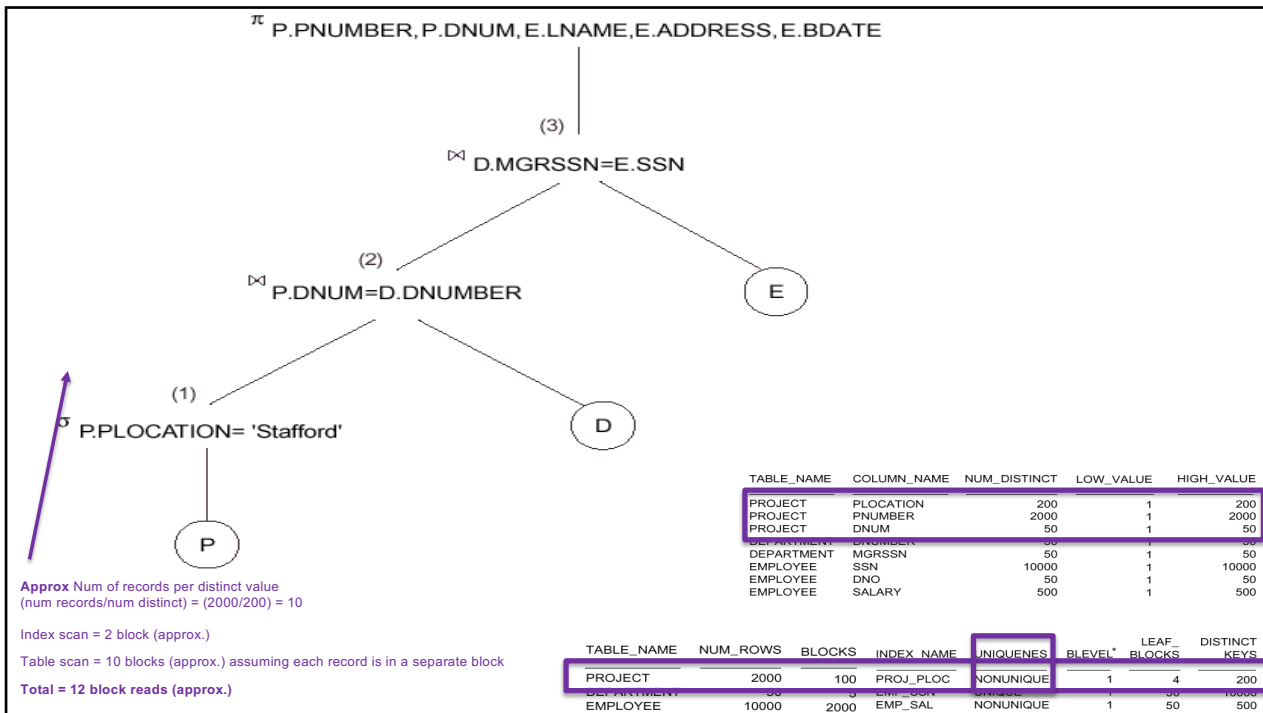
- We can join the result from the previous step to Employee
 - using the index Emp_Ssn.
 - Similarly we can work out how many block reads are required and therefore calculate the cost.
- The final part of the cost of the cost calculation is the merging of the results from the previous 2 steps to produce the total cost of the query.
 - Adding up the costs associated with each of the above steps will give us the total cost of using this particular query join combination.
- Which combination is most cost efficient
 - Need to work through the other 3 options calculating the costs
 - When you have the 4 costs you can then decide which one has the lowest
 - Only use the option that has the lowest cost
 - Use the access paths & query tree developed from calculations

14

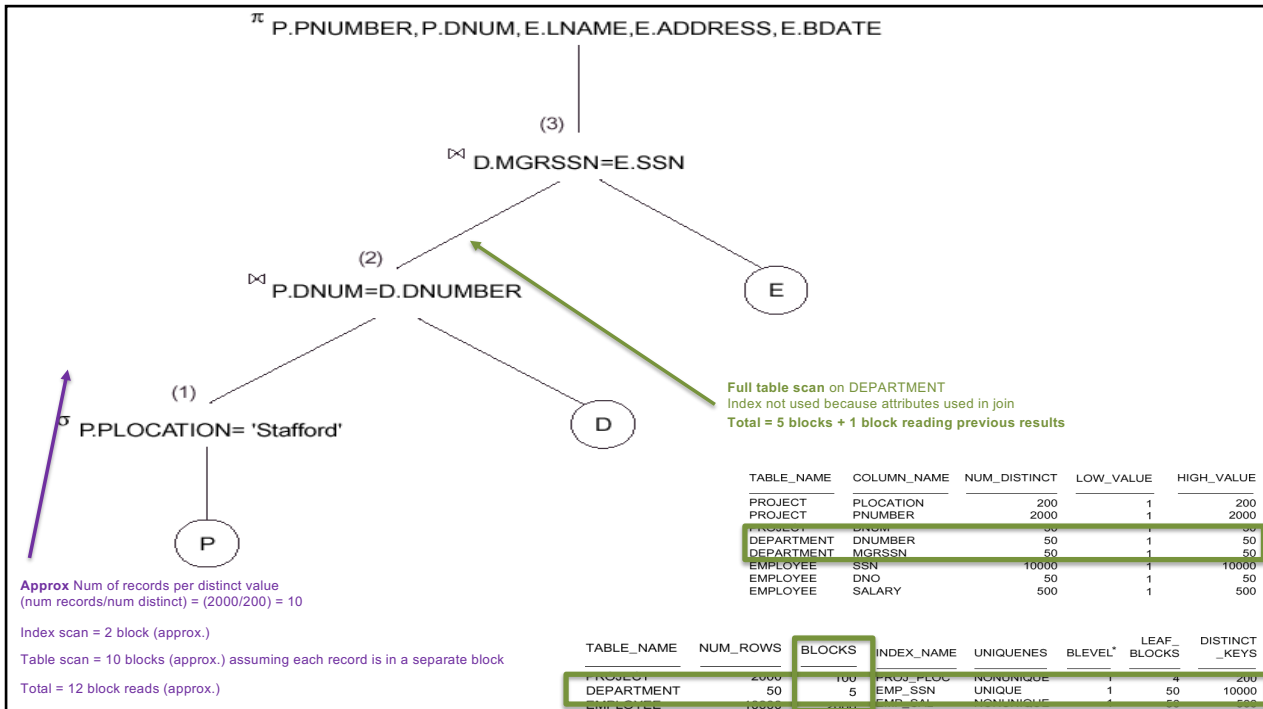
14



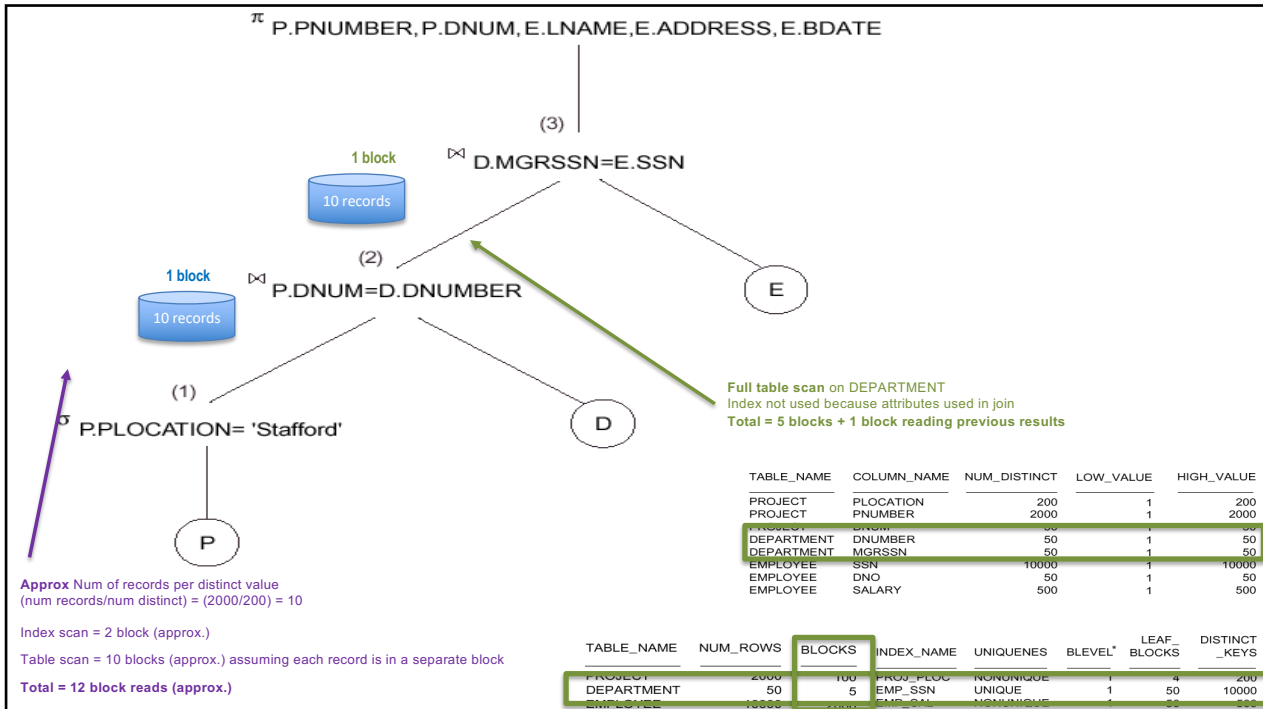
15



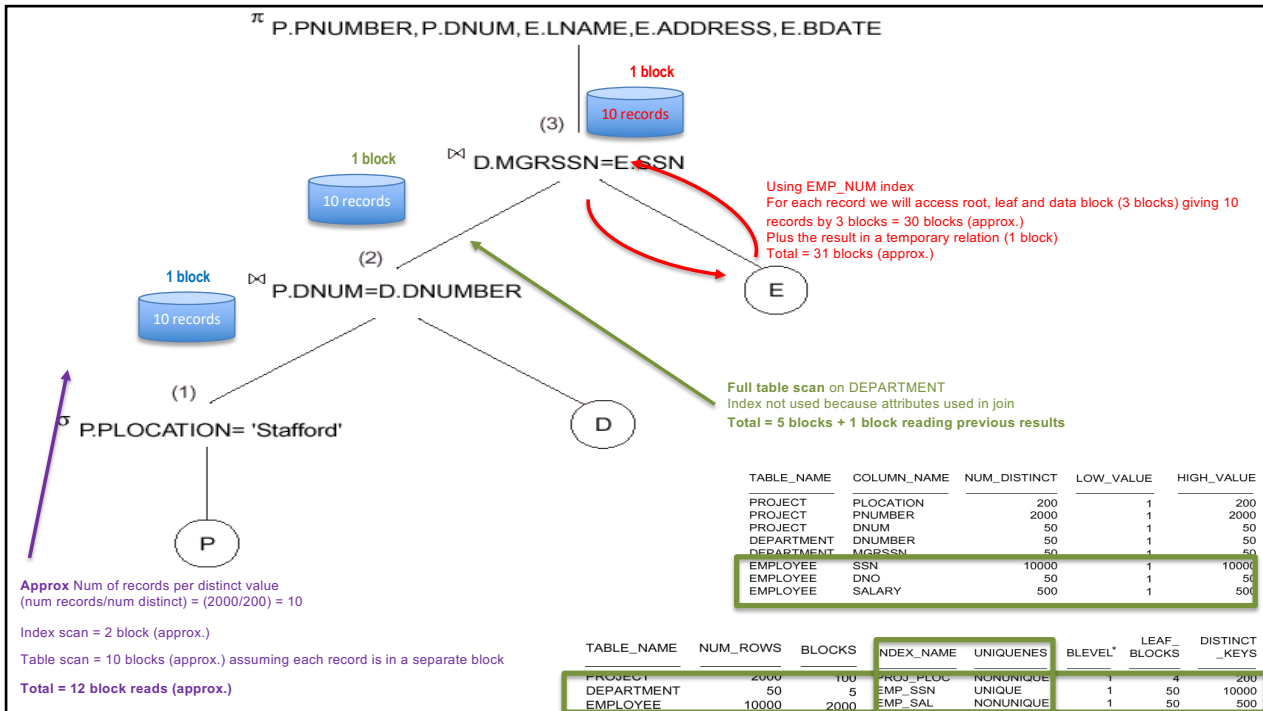
16



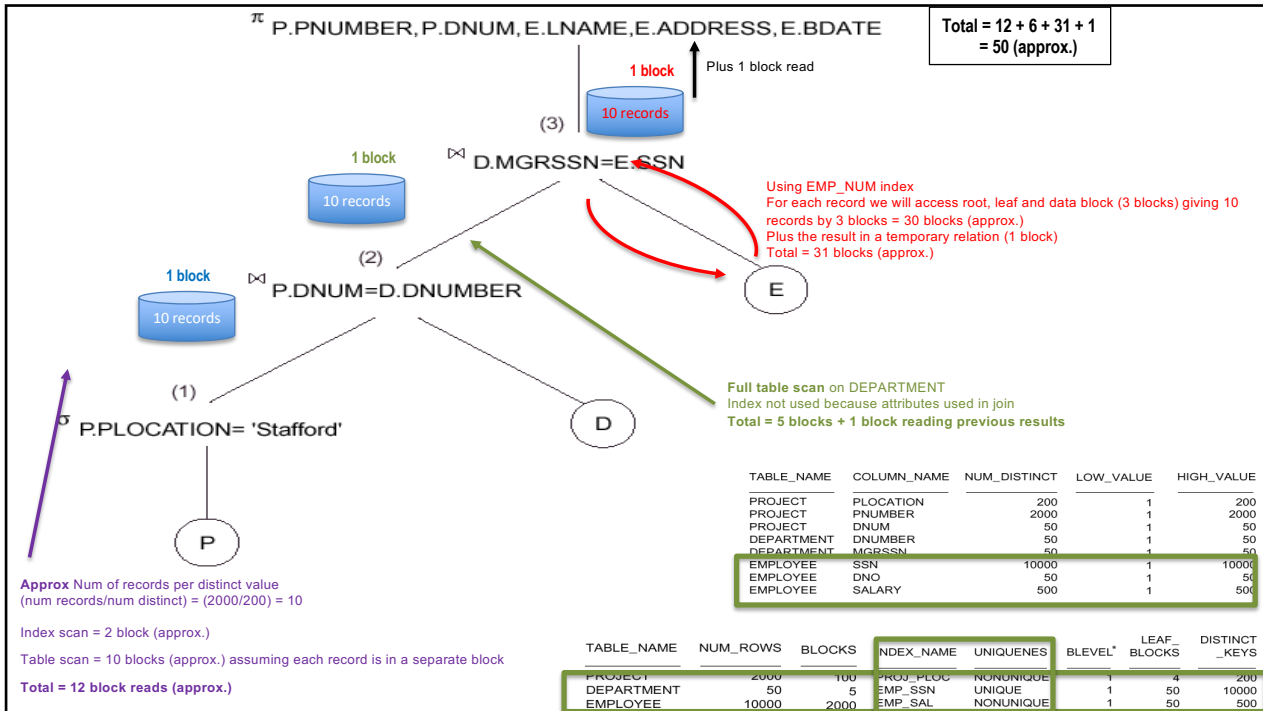
17



18



19



20

Adaptive Query Optimization

- The cost-based optimizer uses database statistics to determine the optimal execution plan for a SQL statement.
- If those statistics are not representative of the data, or if the query uses complex predicates, operators or joins the estimated cardinality of the operations may be incorrect and therefore the selected plan is likely to be less than optimal.
- Typically in most Databases, once the execution plan was determined there was no possible deviation from it at runtime.
- But in some Databases you can adapt or alter the execution plan as you are executing the query

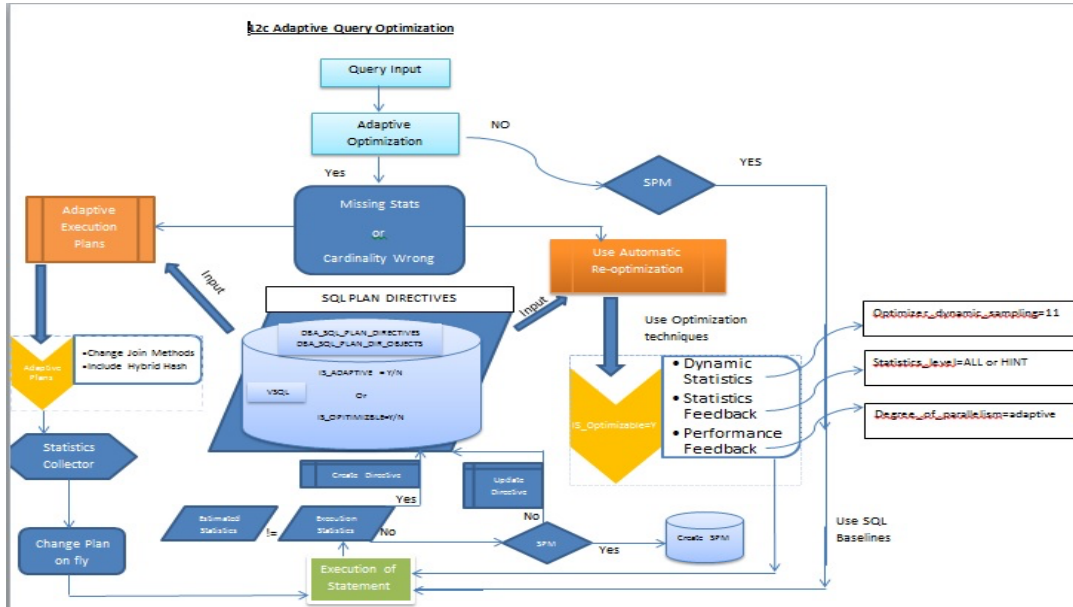
21

Adaptive Query Optimization

- Adaptive Plans in Oracle Database allow
 - runtime changes to execution plans.
 - Rather than selecting a single "best" plan, the optimizer will determine the default plan, and can include alternative subplans for each major join operation in the plan.
 - At runtime the cardinality of operations is checked using statistics collectors and compared to the cardinality estimates used to generate the execution plan.
 - If the cardinality of the operation is not as expected, an alternative subplan can be used.
 - For example, if the statistics suggest two small sets are to be joined, it is likely the optimizer will choose a nested loops join. At runtime, if the fetch operation of the first set returns more than the expected number of rows, the optimizer can switch to a subplan using a hash join instead.
- Once the query has run to completion and the optimal plan is determined, the final plan is fixed until it is aged out of the shared pool or re-optimized for some other reason.

22

Adaptive Query Optimization



23

Adaptive Query Optimization

```

SET LINESIZE 200 PAGESIZE 100
SELECT * FROM TABLE(DBMS_XPLAN.display_cursor(format => 'adaptive allstats last'));
    
```

PLAN_TABLE_OUTPUT

SQL_ID 1km5kczgr0fr, child number 0

```

SELECT /*+ GATHER_PLAN_STATISTICS */
a.data AS tab1_data,
b.data AS tab2_data FROM tab1 a
a.id WHERE a.code = 'ONE'
JOIN tab2 b ON b.tab1_id =
    
```

Plan hash value: 2672205743

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
	0	SELECT STATEMENT		1		25	00:00:00.01	8
- *	1	HASH JOIN		1	25	25	00:00:00.01	8
	2	NESTED LOOPS		1	25	25	00:00:00.01	8
	3	NESTED LOOPS		1	25	25	00:00:00.01	5
-	4	STATISTICS COLLECTOR		1		1	00:00:00.01	2
	5	TABLE ACCESS BY INDEX ROWID BATCHED	TAB1	1	1	1	00:00:00.01	2
*	6	INDEX RANGE SCAN	TAB1_CODE	1	1	1	00:00:00.01	1
	7	INDEX RANGE SCAN	TAB2_TAB1_FKI	1	25	25	00:00:00.01	3
	8	TABLE ACCESS BY INDEX ROWID	TAB2	25	25	25	00:00:00.01	3
-	9	TABLE ACCESS FULL	TAB2	0	25	0	00:00:00.01	0

Predicate Information (identified by operation id):

```

1 - access("B"."TAB1_ID"="A"."ID")
6 - access("A"."CODE"='ONE')
7 - access("B"."TAB1_ID"="A"."ID")
    
```

Note

- this is an adaptive plan (rows marked '-' are inactive)

24

Pipelining & Materialisation

- Materialisation
 - Temporary files are created at each stage to store results
 - Each temporary file acts as input to next stage etc until tree is processed
 - Time consuming creating temporary files
 - Excessive overhead
- Pipelining
 - Also called on-the-fly processing
 - Proposed to overcome the limitations of materialisation
 - Feed the results of one operation into the another operation without creating a temporary relation
 - Can save cost of creating temporary files or relations and reading them again
 - Implemented as a separate process or thread within the DBMS
 - Takes a stream of tuples from input and creates a stream of output
 - Buffers are used for adjacent operations to pass the tuples between operations
 - Drawback is that it is not always possible to do due to structure of query

25

25

Semantic Query Optimisation (SQO)

- Some Databases do this
 - Uses constraints specified on the database schema
 - Unique attributes etc
 - Modifies the query restructuring it into another one which is more efficient
- If query was to find “employees who earn more than their managers”
- Had constraint that no employee could earn more than their manager
- Then the SQO would **not** perform the query as result would be empty
- But needs efficient constraint checking
 - Very time consuming finding applicable constraints
 - Semantic optimising can also be time consuming
- May be used in combination with other techniques

Illustrates the importance of having all data rules defined in the Database

Some Databases do this

26

26

Goals of Tuning

- Objective of tuning a system is
 - to reduce the response time for end users of the system,
 - or to reduce the resources used to process the same work.
- You can accomplish both of these objectives in several ways:
 - Reduce the Workload
 - Commonly constitutes SQL tuning:
 - finding more efficient ways to process the same workload.
 - possible to change the execution plan of the statement without altering the functionality to reduce the resource consumption
 - Example : If a commonly executed query needs to access a small percentage of data in the table, then it can be executed more efficiently by using an index. By creating such an index, you reduce the amount of resources used.
 - Example : If a user is looking at the first twenty rows of the 10,000 rows returned in a specific sort order, and if the query (and sort order) can be satisfied by an index, then the user does not need to access and sort the 10,000 rows to see the first 20 rows.

27

27

Goals of Tuning

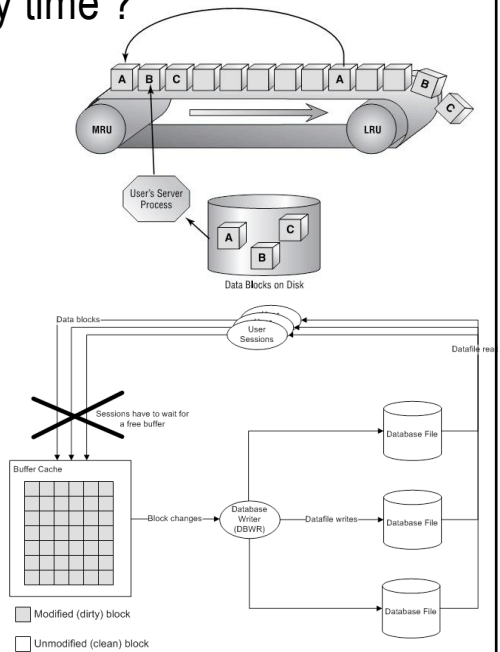
- Balance the Workload
 - Systems often tend to have peak usage in the daytime when real users are connected to the system, and low usage in the night time.
 - If non-critical reports and batch jobs can be scheduled to run in the night time, then it frees up resources for the more critical programs in the day.
- Parallelise the Workload
 - Queries that access large amounts of data (typical data warehouse queries) often can be parallelised.
 - This is extremely useful for reducing the response time in low concurrency data warehouse.

28

28

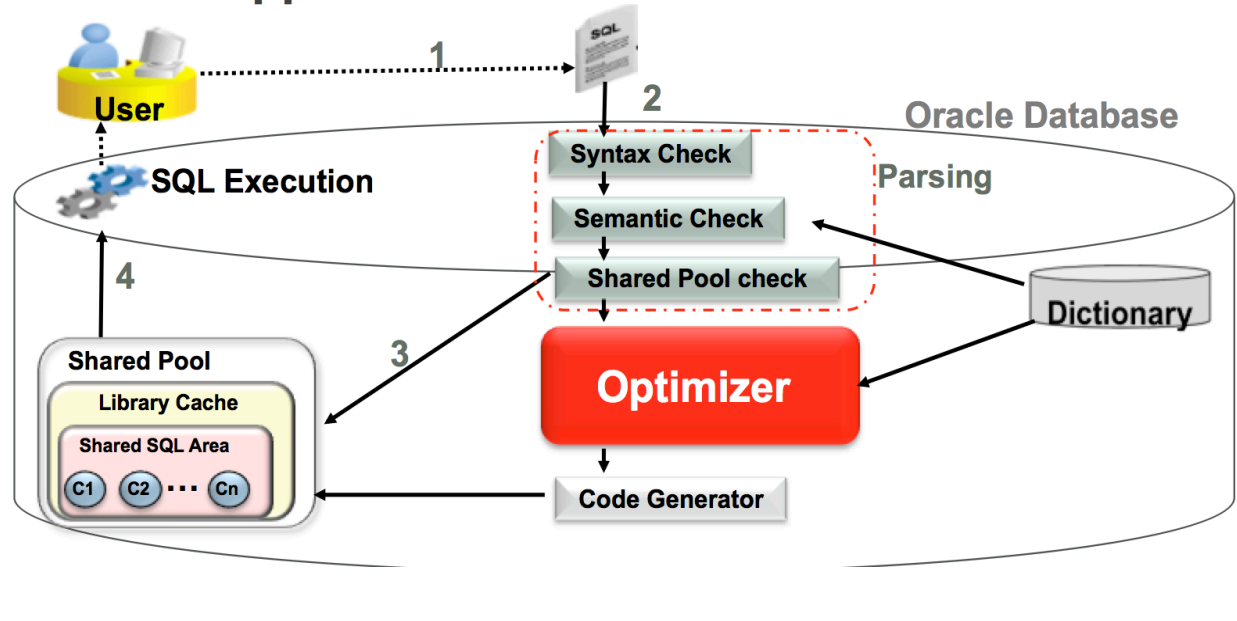
Do I have to do it every time ?

- Query Execution Cache
 - Cache of previous execution plans
 - A plan may get moved out of cache over time
 - Use Bind variables where possible :n instead of ...
=||vValForN
- Query Results Cache
 - If data has not been changed since the last time the query has run
 - Display the same data
 - If it still exists in the cache



29

What happens when a SQL statement is issued?



30

Exercises

- How can you calculate the statistics about database objects in Oracle ? What are the different types of information that is calculated ?
- When optimising DB applications what are the different components that you need to look at and in what order ?
- Discuss the differences between materialisation & pipelining. Illustrate your discussion with an example
- Read up on how you can use hints in a Oracle SQL query to alter the execution of a SQL query.
- Investigate using examples how you can use the following Oracle feature to improve query and database optimization using
 - Parallel Queries
 - Table Partitioning
 - Materialised Views
 - Clustering - no example code required

31

31

Oracle Database Features

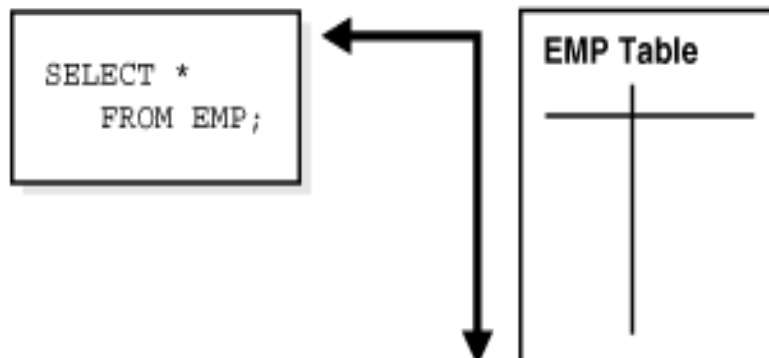
- In this section of the course will look at some of the features available in Oracle to help you implement your DW efficiently
 - Indexes
 - Parallel Queries
 - Partitioning
 - Materialised Views
 - Clustering
 - In-Memory

32

Parallel Queries

For a normal query in a database the server process performs all necessary processing for the sequential execution of a SQL statement.

For example, to perform a full table scan (such as `SELECT * FROM emp`), one process performs the entire operation



33

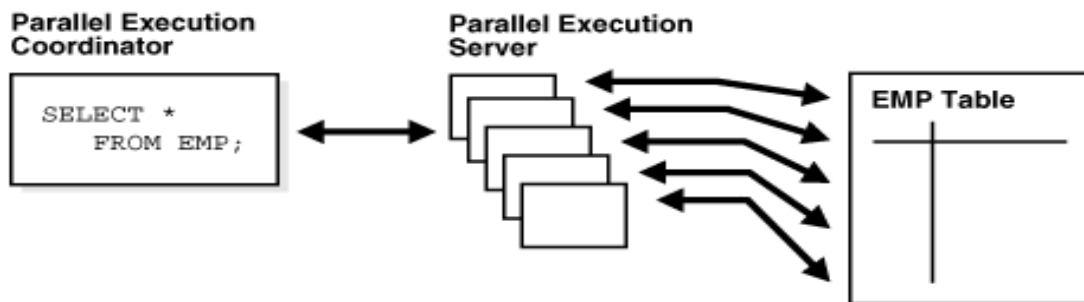
Parallel Queries

- Parallel execution dramatically reduces response time for data-intensive operations on large databases
- Parallelism is the idea of breaking down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time.
 - The statement being processed can be split up among many CPUs on a single Oracle system
 - An example of this is when four processes handle four different quarters in a year instead of one process handling all four quarters by itself.

34

Parallel Queries

- If we perform the same query using Parallel execution, the database server breaks the query into a number of processes and executes these processes to get the data.
 - The table is divided dynamically based on the number of blocks for the table



- The Degree of Parallelism (DOP) is the number of parallel execution servers assigned to a single operation

35

Parallel Queries – Select statement

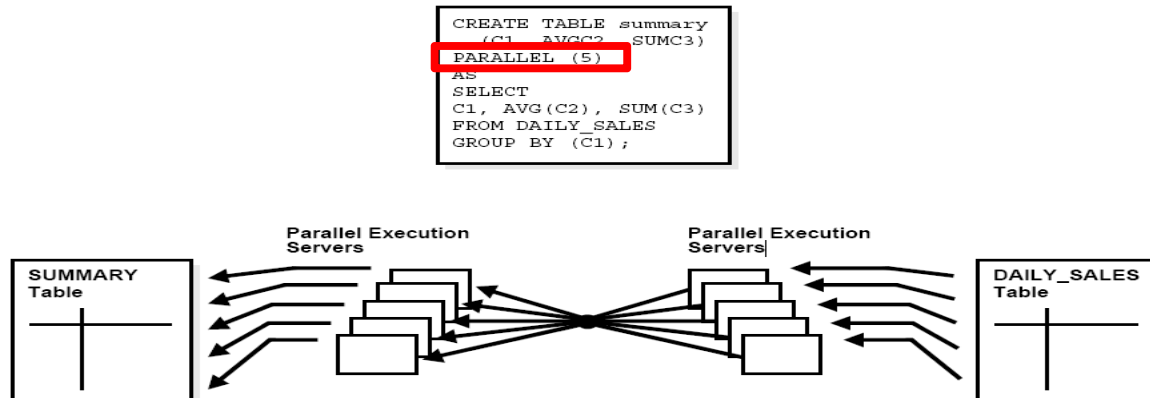
- Specified as a query hint

```
SELECT /*+ PARALLEL(orders, 4) */ COUNT(*)  
FROM orders;
```

```
SELECT /*+ PARALLEL(employees 4) PARALLEL(departments 2) */  
      MAX(salary), AVG(salary)  
FROM   employees, departments  
WHERE  employees.department_id = departments.department_id  
GROUP BY employees.department_id;
```

36

Parallel Query – Create Table



37

Parallel Queries

- Parallel execution improves processing for:
 - Large table scans and joins
 - Creation of large indexes
 - Partitioned index scans
 - Bulk inserts, updates, and deletes
 - Aggregations and copying
- Ideally suited when
 - Queries on large volumes of data
 - Extracting data from the source systems
 - Creating tables
 - Creating indexes
 - Loading data from files (SQL*Loader)
 - Bulk updates

Then I should use them
all the time.

Right?

They will consume all the
resources.

So should only be used
when suitable !

38

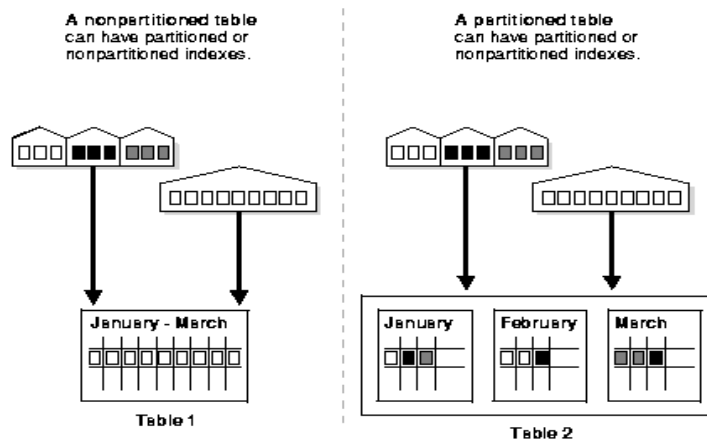
Parallel Queries

- Degree of Parallelism (DOP)
 - Lowest value is 2 : Default is 1
 - Max is determined by the number of CPUs etc available
 - You will need to test to find the appropriate value for DOP
 - It also depends on what else is running on the server at the same time
 - Particularly during the extraction process from the source systems
- Exercise to test Parallel Query
 - Create a table with 500,000 records
 - Create a table with 1M records, 2M records, ... up to 10M records
 - Issues a Select statement without using parallelism on each table
 - SQL> set time on - to get the start and end times
 - Then issue the same query with
 - Parallel (<table name>, 2)
 - Parallel (<table name>, 4)
 - Parallel (<table name>, 6)
 - Parallel (<table name>, 8)

39

Data / Table Partitioning

- Partitioning is useful for many different types of applications, particularly applications that manage large volumes of data. OLTP systems often benefit from improvements in manageability and availability, while data warehousing systems benefit from performance and manageability



40

40

Partitioning

- **Partitioning** supports the management of large tables and indexes by decomposing them into smaller and more manageable pieces called **partitions**.
 - Range & List
 - Range-List
 - Hash
- SQL queries do not need to be modified in order to access partitioned tables.
 - Queries can access and manipulate individual partitions rather than entire tables or indexes.
 - Partitioning is entirely transparent to applications.
- Each partition of a table or index must have the same logical attributes, such as column names, datatypes, and constraints, but each partition can have separate physical attributes such as pctfree, pctused, and tablespaces.

41

Data / Table Partitioning

- Partitioning offers these advantages:
 - Partitioning enables data management operations such as data loads, index creation and rebuilding, and backup/recovery at the partition level, rather than on the entire table. This results in significantly reduced times for these operations.
 - Partitioning improves query performance. In many cases, the results of a query can be achieved by accessing a subset of partitions, rather than the entire table. For some queries, this technique (called **partition pruning**) can provide order-of-magnitude gains in performance.
 - Partitioning can significantly reduce the impact of scheduled downtime for maintenance operations.
 - Partitioning increases the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance windows, recovery times, and impact of failures.
 - Partitioning can be implemented without requiring any modifications to your applications.
 - For example, you could convert a non-partitioned table to a partitioned table without needing to modify any of the SELECT statements or DML statements which access that table. You do not need to rewrite your application code to take advantage of partitioning.

42

42

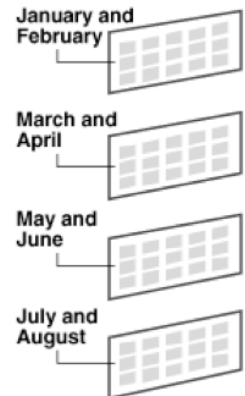
Data / Table Partitioning

Range Partitioning

- Range partitioning maps data to partitions based on ranges of partition key values that you establish for each partition.
- It is the most common type of partitioning and is often used with dates.
 - For example, you might want to partition sales data into monthly partitions

```
CREATE TABLE sales_range (  
  salesman_id NUMBER(5),  
  salesman_name VARCHAR2(30),  
  sales_amount NUMBER(10),  
  sales_date DATE)  
PARTITION BY RANGE(sales_date)  
( PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),  
  PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),  
  PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),  
  PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))  
);
```

Range Partitioning



43

Data / Table Partitioning

List Partitioning

- Enables you to explicitly control how rows map to partitions.
- Done by specifying a list of discrete values for the partitioning key in the description for each partition.
- This is different from range partitioning, where a range of values is associated with a partition.
- The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way.

```
CREATE TABLE sales_list (  
  salesman_id NUMBER(5),  
  salesman_name VARCHAR2(30),  
  sales_state VARCHAR2(20),  
  sales_amount NUMBER(10),  
  sales_date DATE)  
PARTITION BY LIST(sales_state)  
( PARTITION sales_west VALUES ('California', 'Hawaii'),  
  PARTITION sales_east VALUES ('New York', 'Virginia', 'Florida'),  
  PARTITION sales_central VALUES ('Texas', 'Illinois')  
  PARTITION sales_other VALUES(DEFAULT) );
```

List Partitioning



44

Data / Table Partitioning

■ Hash Partitioning

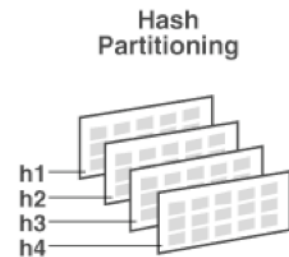
- Based on an internal Hashing algorithm
- Distributes the data randomly throughout all the partitions

```
CREATE TABLE invoices
(invoice_no NUMBER NOT NULL,
invoice_date DATE NOT NULL,
comments VARCHAR2(500))
PARTITION BY HASH (invoice_no)
PARTITIONS 4 STORE IN (users, users, users, users);
```

The Database will automatically create 4 partitions and assign them a system generated name

```
CREATE TABLE invoices
(invoice_no NUMBER NOT NULL,
invoice_date DATE NOT NULL,
comments VARCHAR2(500))
PARTITION BY HASH (invoice_no)
(PARTITION invoices_q1 TABLESPACE users,
PARTITION invoices_q2 TABLESPACE users,
PARTITION invoices_q3 TABLESPACE users,
PARTITION invoices_q4 TABLESPACE users);
```

Here we explicitly name the partition.
Better to do this.
Good Database design and management



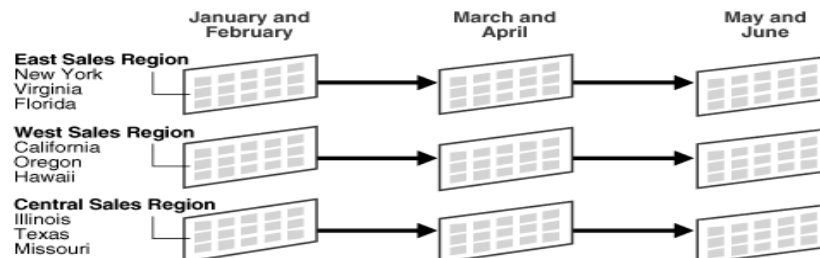
45

45

Composite Partitioning – Range - List

```
CREATE TABLE bimonthly_regional_sales (
deptno NUMBER,
item_no VARCHAR2(20),
txn_date DATE,
txn_amount NUMBER,
state VARCHAR2(2))
PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)
SUBPARTITION TEMPLATE (
SUBPARTITION east VALUES ('NY', 'VA', 'FL') TABLESPACE ts1,
SUBPARTITION west VALUES ('CA', 'OR', 'HI') TABLESPACE ts2,
SUBPARTITION central VALUES ('IL', 'TX', 'MO') TABLESPACE ts3)
( PARTITION janfeb_2000 VALUES LESS THAN (TO_DATE('1-MAR-2000','DD-MON-YYYY')),
PARTITION marapr_2000 VALUES LESS THAN (TO_DATE('1-MAY-2000','DD-MON-YYYY')),
PARTITION mayjun_2000 VALUES LESS THAN (TO_DATE('1-JUL-2000','DD-MON-YYYY')) );
```

You can use any combination of partitioning methods when creating a composite



46

Partitioning - Indexes

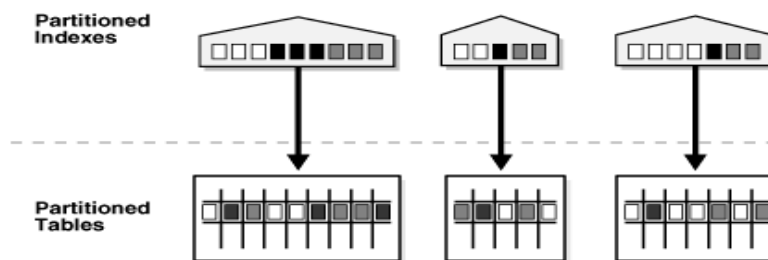
- Indexes can be created using partitioning

```
CREATE INDEX cost_ix ON sales (amount_sold)
GLOBAL PARTITION BY RANGE (amount_sold)
( PARTITION p1 VALUES LESS THAN (1000),
  PARTITION p2 VALUES LESS THAN (2500),
  PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

The same partitioning methods can be applied when creating indexes

Maybe use the same partitioning method on table and index.

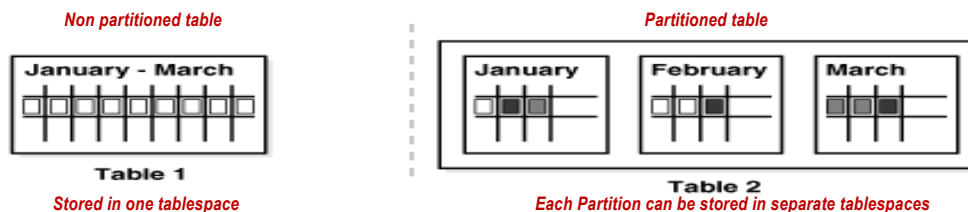
But you don't have too.



47

Partitioning

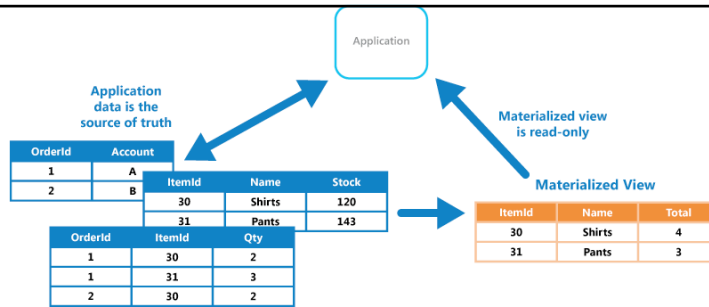
- Very suited to a data warehouse environment as it can greatly increase performance.
- Suited to tables (and indexes) where >2M records and there is a natural (even-ish) distribution of data
- Typically you would partition the Fact table in a DW
 - Based on the Time Dimension as most of the queries will be accessing the data based on dates



48

Materialised Views

Results Regular views are computed each time the view is accessed



- Materialized views are query results that have been stored in advance so long-running calculations are not necessary when queries are executed.
- It is not untypical for DW queries to take >5, 10, 30, 60 minutes
- For standardised queries that are run frequently during the day, MVs can give significant performance improvement
- MVs are typically used summary tables/views
- In a real-time environment they can be used to give current information
 - You can set a Refresh frequency

With MVs the data is cached a predefined times
The query is run once.
All users select from the cached results

49

Materialised Views

```
CREATE MATERIALIZED VIEW all_customers
PCTFREE 5 PCTUSED 60
TABLESPACE example
STORAGE (INITIAL 50K NEXT 50K)
USING INDEX STORAGE (INITIAL 25K NEXT 25K)
REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
NEXT NEXT_DAY(TRUNC(SYSDATE), 'MONDAY') + 15/24
AS SELECT * FROM sh.customers@remote
UNION
SELECT * FROM sh.customers@local;
```

- Automatically refreshes this materialized view tomorrow at 11:00 a.m. and subsequently every Monday at 3:00 p.m

50

Clustering

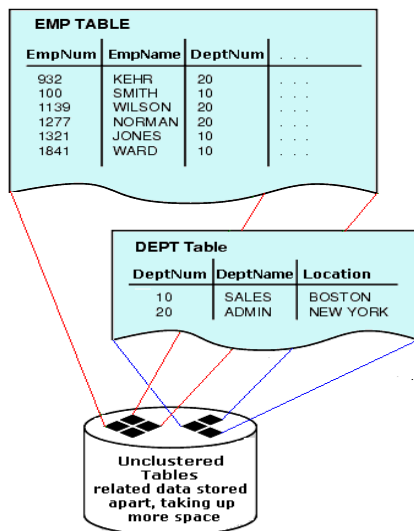


figure 1

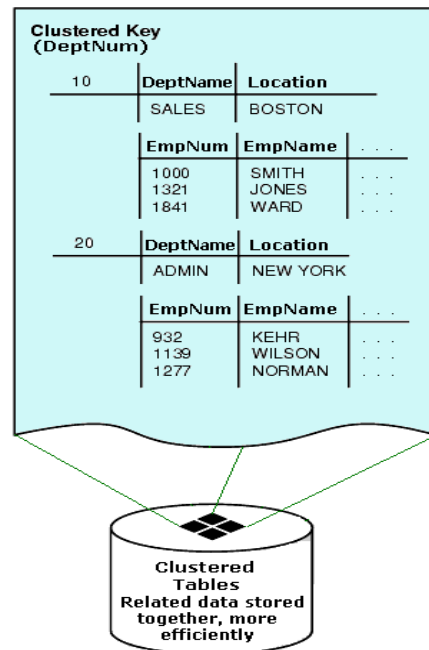


figure 2

51

Clustering

- A cluster is a group of tables that share the same data blocks
 - They share common columns and are often used together.
 - Example
 - Employees and Departments table share the department_id column.
 - When you cluster the employees and departments tables the DB physically stores all rows for each department from both the employees and departments tables in the same data blocks.
- clusters offers these benefits:
 - Disk I/O is reduced for joins of clustered tables.
 - Access time improves for joins of clustered tables.
- In a cluster, a **cluster key value** is the value of the cluster key columns for a particular row.
 - Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value.
 - Therefore, less storage is required to store related table and index data in a cluster than is necessary in non-clustered table format.
 - Example shows how each cluster key (each department_id) is stored just once for many rows that contain the same value in both the employees and departments tables

52

52

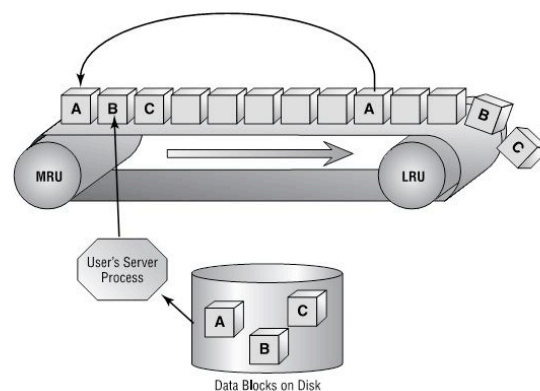
Clustering

- Records can be randomly distributed
- If they were grouped logically together then
 - Easier to locate the required data & number of disk accesses would be lower
 - Records to be stored (clustered) together in the same or adjacent block
 - This approach is most suitable to bulk/batch processing on specific segments of the data
 - Example
 - Suppose we have a Student table for all students in Ireland
 - Types of queries/processing might be on location, year of study
 - Location = Dublin 8
 - All students who live in Dublin 8 will have their records physically stored beside each other
- Can have a large database management overhead
 - May involve database reorganisation when new data is inserted
 - Because new records needs to be located beside existing data
 - This may require data or blocks being moved to another location
- Requires insight into expected use and frequency of different types of request
 - This applies to all approaches
 - Need to work out appropriate methods to implement
 - Requires skill and experience
 - Need to monitor continually and make changes if necessary

53

53

In-Memory : Column Storage



Small cache and objects have a limited life time in it

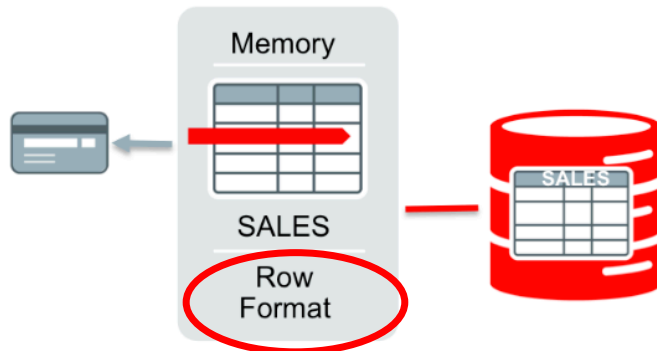
But using the Cache gives us super fast results.

Could we have semi-permanent ~~Cache~~ In-Memory storage where object will always persist?

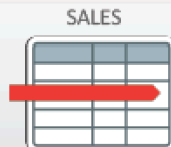
54

54

In-Memory : Column Storage



Rows Stored
Contiguously



- **Transactions** run faster on row format
 - Example: Query or Insert a sales order
 - Fast processing few rows, many columns

55

In-Memory : Column Storage



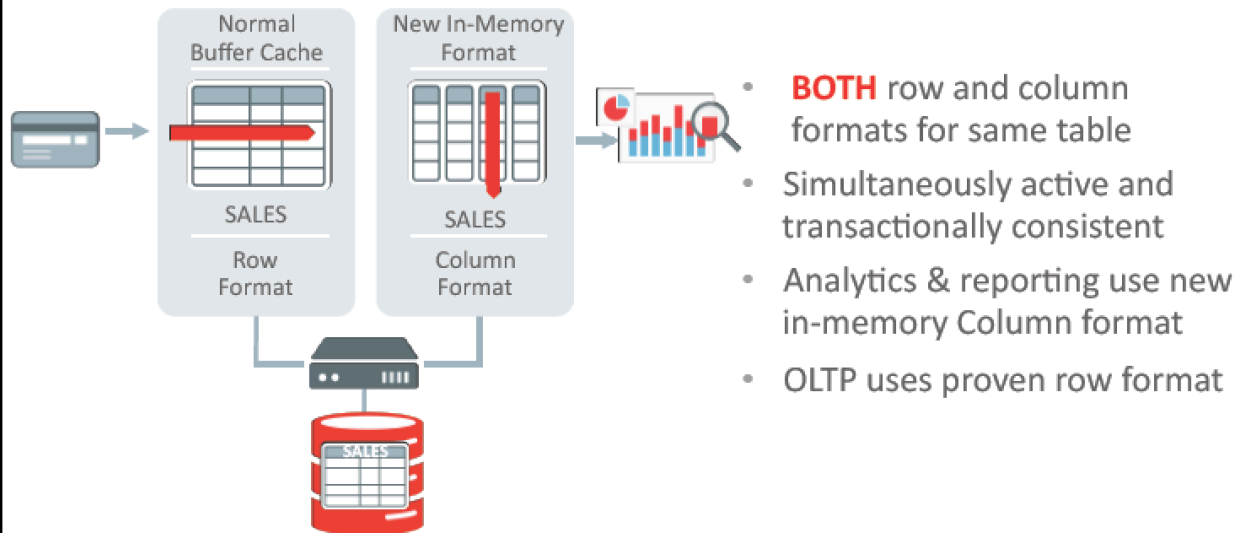
Columns
Stored
Contiguously



- **Analytics** run faster on column format
 - Example : Report on sales totals by region
 - Fast accessing few columns, many rows

56

Benefit & the Challenge

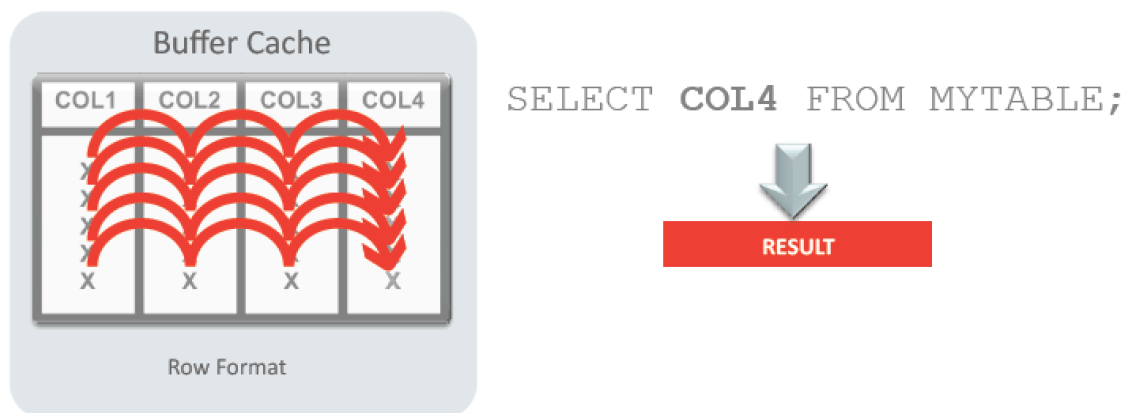


57

57

OLTP Processing of Query

Why is an In-Memory scan faster than the buffer cache?

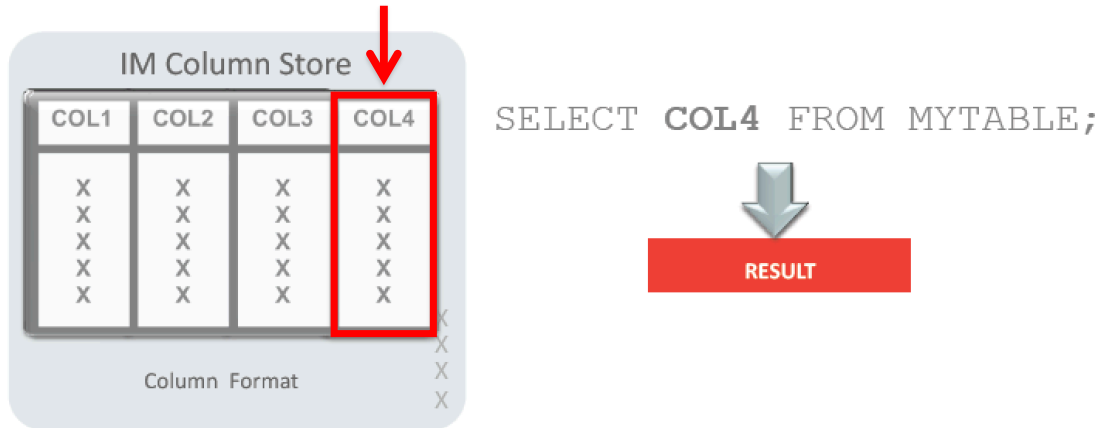


58

58

In-Memory processing of Quer

Why is an In-Memory scan faster than the buffer cache?



59

59

In-Memory – How do you do it?

- In Container

```
SQL> ALTER SYSTEM SET inmemory_size = 512M scope=spfile;  
SQL> shutdown immediate;  
SQL> startup;
```

- The Pluggables inherit this

- Unless you want to specify specific values for each

60

60

In-Memory – How do you do it?

- Add a whole table to in-memory

```
SQL> ALTER TABLE customers INMEMORY;  
table CUSTOMERS altered.
```

```
SELECT table_name, inmemory, inmemory_priority, inmemory_compression  
FROM user_tables  
WHERE table_name='CUSTOMERS';
```

TABLE_NAME	INMEMORY	INMEMORY_PRIORITY	INMEMORY_COMPRESSION
CUSTOMERS	ENABLED	NONE	FOR QUERY LOW

61

61

In-Memory – How do you do it?

- Enabling the In-Memory attribute on the EXAMPLE tablespace by specifying the INMEMORY attribute

```
SQL> ALTER TABLESPACE example INMEMORY;
```

- Enabling the In-Memory attribute on the sales table but excluding the “prod_id” column

```
SQL> ALTER TABLE sales INMEMORY NO INMEMORY(prod_id);
```

- Disabling the In-Memory attribute on one partition of the sales table by specifying the NO INMEMORY clause

```
SQL> ALTER TABLE sales MODIFY PARTITION SALES_Q1_1998 NO INMEMORY;
```

- Enabling the In-Memory attribute on the customers table with a priority level of critical

```
SQL> ALTER TABLE customers INMEMORY PRIORITY CRITICAL;
```

62

62

In-Memory – How do you do it?

PRIORITY	DESCRIPTION
CRITICAL	Object is populated immediately after the database is opened
HIGH	Object is populated after all CRITICAL objects have been populated, if space remains available in the IM column store
MEDIUM	Object is populated after all CRITICAL and HIGH objects have been populated, and space remains available in the IM column store
LOW	Object is populated after all CRITICAL, HIGH, and MEDIUM objects have been populated, if space remains available in the IM column store
NONE	Objects only populated after they are scanned for the first time (Default), if space is available in the IM column store

Figure 7. Different priority levels controlled by the PRIORITY sub clause of the INMEMORY clause

63

63

In-Memory Example

- Create table with ~11K records (this is really small data size to test IM)
- Select a count of the records + measure the costs

```
select count(*) from test_inmemory;
```

Script Output x Autotrace x Query Result x Autotrace 1 x
SQL HotSpot | 1.313 seconds

OPERATION	OBJECT_NAME	CARDINALITY	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT				47
SORT (AGGREGATE)			1	159
TABLE ACCESS (FULL)	TEST_INMEMORY	10500		47
				159

- Add the table to In-Memory

```
alter table test_inmemory inmemory PRIORITY critical;
select count(*) from test_inmemory; -- again
```

Script Output x Autotrace x Query Result x Autotrace 1 x
SQL HotSpot | 0.223 seconds

OPERATION	OBJECT_NAME	CARDINALITY	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT				7
SORT (AGGREGATE)			3	3
TABLE ACCESS (INMEMORY FULL)	TEST_INMEMORY	10500		7
				3

64

In-Memory Example

```
SELECT table_name, inmemory, inmemory_priority,  
       inmemory_distribute, inmemory_compression, inmemory_duplicate  
FROM user_tables  
WHERE inmemory = 'ENABLED'  
ORDER BY table_name;
```

SQL | All Rows Fetched: 1 in 0.345 seconds

TABLE_NAME	INMEMORY	INMEMORY_PRIORITY	INMEMORY_DISTRIBUTE	INMEMORY_COMPRESSION	INMEMORY_DUPLICATE
1 TEST_INMEMORY	ENABLED	CRITICAL	AUTO	FOR QUERY LOW	NO DUPLICATE

SQL > alter table test_inmemory no inmemory; -- remove the table from in-memory

SQL | All Rows Fetched: 0 in 0.741 seconds

TABLE_N...	INMEMORY	INMEMOR...	INMEMOR...	INMEMOR...	INMEMOR...
------------	----------	------------	------------	------------	------------

65

65

Many things to consider when writing your SQL

- We have looked at a number ways and things to consider when writing your SQL
- But is there anything else that I need to consider ?
- What about how the results are presented ?
- Remember all the data/results are sent over the network
- Can the database and network work together so that I can get my data quicker

66

Networking Compression of Query Results

- Oracle Example – Sorry !!!
- When you array fetch data from the database, SQL*Net will write the first row in its entirety on the network.
- When it goes to write the second row however, it will only transmit column values that differ from the first row.
- The third row written on the network will similarly be only the changed values from the second row, and so on.
- This compression therefore works well with repetitive data - of which we have a lot typically!
 - This compression works even better with data that is sorted by these repeating values so that the repeating values are near each other in the result set.

67

The key to this is

- How can we ensure this repetition will happen?
- We need a ORDER BY on our SELECT statements
- The compression works even better with data that is sorted by these repeating values so that the repeating values are near each other in the result set.

68

Example to illustrate

- We need to set up some data

```
ops$tkyte%ORA11GR2> create table t
2 as
3 select *
4   from all_objects;
Table created.

ops$tkyte%ORA11GR2> begin
2   dbms_stats.gather_table_stats( user, 'T' );
3 end;
4 /
PL/SQL procedure successfully completed.
```

- This newly created table is about 8MB in size and consists of 1,031 blocks in my database – number will differ from Env to Env
- Additionally, this table stores about 70 rows per block – there are about 72,000 rows on 1,031 blocks.

69

Setting up the baseline test

- Select the entire contents of the table over the network using SQL*Plus

```
ops$tkyte%ORA11GR2> select * from t;
72228 rows selected.

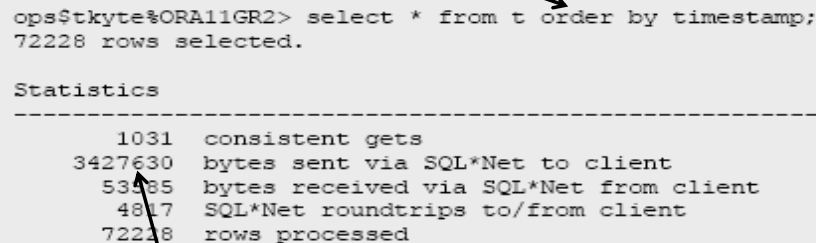
Statistics
-----
      5794 consistent gets
    8015033 bytes sent via SQL*Net to client
     53385 bytes received via SQL*Net from client
      4817 SQL*Net roundtrips to/from client
      72228 rows processed
```

- 8Mb of data was transferred from the DB server to the client machine
- The array size is enough for 15 records.
- So this equate to approx. 5 reads to get one block of data

70

Now the modified test

- Let's add an ORDER BY to the query



```
ops$tkyte@ORA11GR2> select * from t order by timestamp;
72228 rows selected.

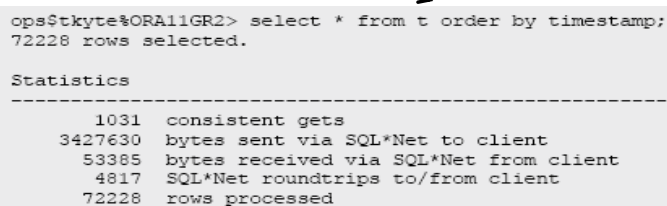
Statistics
-----
      1031  consistent gets
    3427630  bytes sent via SQL*Net to client
    533385  bytes received via SQL*Net from client
     4817  SQL*Net roundtrips to/from client
     72228  rows processed
```

- We went from 8MB of data down to about 3.4MB of data! (4.5MB saving)
- This difference is due to the repeating TIMESTAMP attribute. Every time we array fetched 15 rows, we sent the TIMESTAMP column value approximately once.

71

Now the modified test

- Let's add an ORDER BY to the query



```
ops$tkyte@ORA11GR2> select * from t order by timestamp;
72228 rows selected.

Statistics
-----
      1031  consistent gets
    3427630  bytes sent via SQL*Net to client
    533385  bytes received via SQL*Net from client
     4817  SQL*Net roundtrips to/from client
     72228  rows processed
```

- The consistent gets also dropped considerably - nothing to do with SQL*Net, but rather the way this query had to be processed.
- In order to get the first row out of this result set, the database had to have read the entire table and sorted it.
- This means that, in order to get the first row, all 1,031 blocks were read and sorted in temporary space. Then, to retrieve rows from this result set, we would be reading from temporary space, not from the buffer cache. A read from Temp is not a logical IO. It is not a consistent get. Hence the consistent gets stop at 1,031 as the entire query is read from Temp space.

72