Basic Optimization

DB Workshop
for
LHC online/offline developers

CERN January 24-28 2005

Overview

- 09:00-10:00 Basic Optimization
  Dirk Geppert, IT/ADC

- 10:30-12:00 Tuning
  Bjørn Engsig, Oracle

- 14:00-16:00+ Hands-on exercises
  and Further documentations, tutorials

Contents

- General remarks
- Execution plan
- Bind variables
- Indexes
- Optimizer
- Analyze data/Statistics
- Good SQL
- Hints
- Example
- Conclusion

Overview

- Applications must scale to many users
- Many performance problems are not seen in small environments
- Very good performance can be achieved by good application coding practice

⇒Try to make the application performant from the beginning → Basic Optimization
⇒If too slow later → Performance Tuning

DBA tuning and User tuning

- DBA tuning is at the instance / OS level,
  - by looking at ratios (old school) or wait events (new trend),
  - by inspecting the memory structure (caches) of the database
  - the effect of the database at the operating system level
- User tuning is at the session / statement level
  - Most of the gain can be achieved here!
  - No administrative privilege required
- Where is the performance gain?
  - Based on experience at CERN
    - Bad SQL statements 75%
    - Bad application design 20%
    - Problems with database parameters 5%

    $\Rightarrow$concentrate on *user* tuning

Sources of performance problems

- Using too many resources, such as CPU or disk I/O
  - Potential cause of poor response time (my SQL statement takes too long to execute)
- Waiting for others holding a single resource, such as a latch
  - Potential cause of poor scalability (adding more CPU doesn't allow me to run more concurrent users)
  - Causes *contention* for the resource

⇒ Want to avoid these from the beginning!

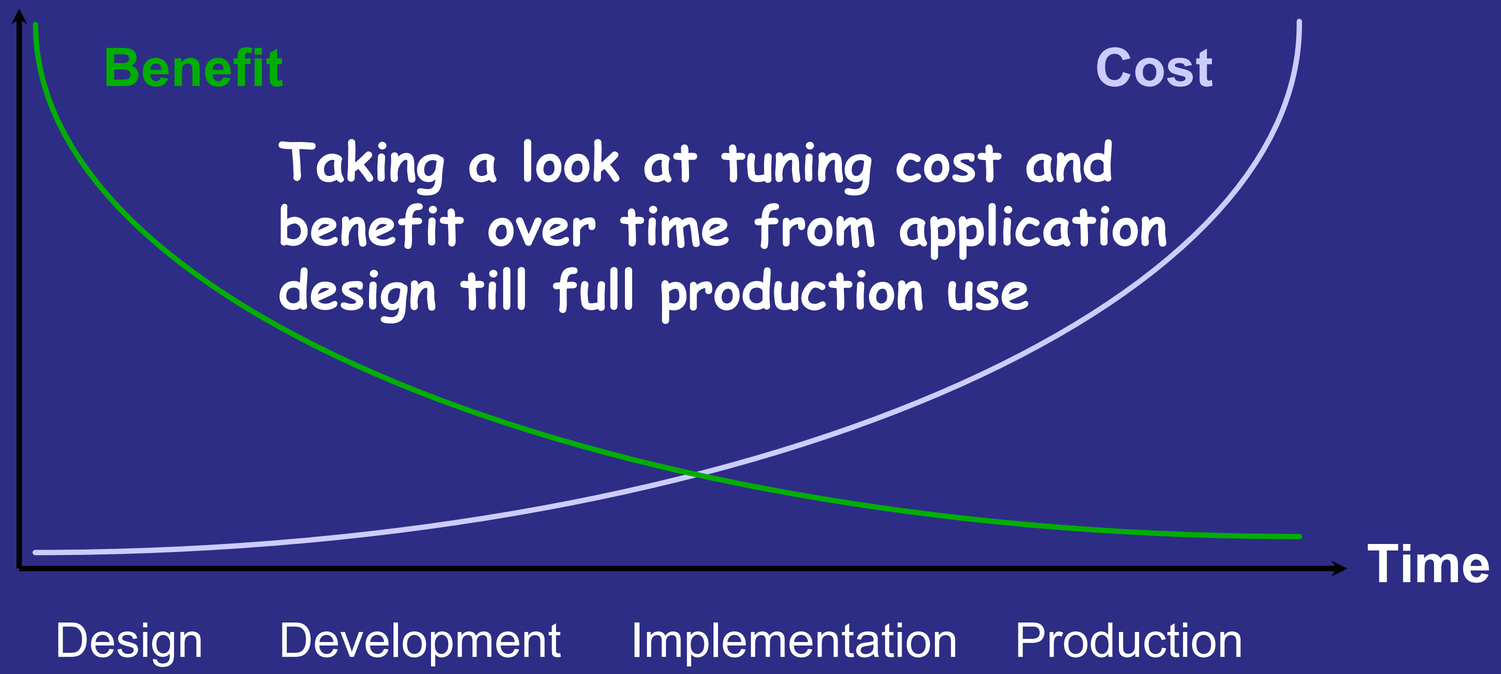
The steps for Tuning/Optimization

- Identify what is slow: an application step is often thousands of lines of code -> intuition, code instrument, profiling
- Understand what happens in this step, (execution plan, traces)
- Modify application / data so that it is better, sometimes it can be as simple as
  - Adding an index
  - Removing an index
  - Changing the definition of an index
  - Change the syntax of the select statement

Tuning Cost/Benefit

Tuning cost increases in time

Tuning benefit decreases in time



Execution plan

- Series of steps that Oracle will perform to execute the SQL statement
  - Generated by the Optimizer
  - Describes the steps as meaningful operators - Access Paths

- Full Table Scans
- RowID Scans
- Index Scans
  - Index Unique Scan
  - Index Range Scan
  - Index Range Scans Descending
  - Index Skip Scans
  - Full Scans
  - Fast Full Index Scans
  - Index Joins
  - Bitmap Joins
- Cluster Scans
- Hash Scans
- Joins
  - Nested Loop Joins
  - Hash Joins
  - SortMerge Joins
  - Cartesian Joins
  - Outer Joins

Get the predicted execution plan

- SQL command that allows to find out what is the Query Plan *before* the SQL statement runs
- Need to access *plan_table* in the user schema
- **`explain plan for <statement>;`**
- Query the contents of the plan_table with
  - **`$ORACLE_HOME/rdbms/admin/utlxpls.sql`**
  - **`$ORACLE_HOME/rdbms/admin/utlxplp.sql`**
- Use a tool (e.g. Benthic Golden Ctrl-P)

Get the real Execution plan

- USE SQL*Plus
  `set autotrace traceonly explain statistics`
  for a single statement
- SQL trace is a way to get information of the execution in a session
  - Enable it using
    - `alter session set sql_trace=true`
    - `alter session set sql_trace=false`
  - Generates a trace file in the database server => usually developer has no access to file system!

Execution plan Example

```
select contact_surname,contact_firstname
  from customers
where address2='SYDNEY';
SELECT STATEMENT
  TABLE ACCESS FULL CUSTOMERS

select contact_surname,contact_firstname
  from customers
where CUSTOMER_ID=1;
SELECT STATEMENT
  TABLE ACCESS BY INDEX ROWID CUSTOMERS
    INDEX UNIQUE SCAN PK_CUSTOMERS
```

Parsing and executing SQL statements

Oracle processes SQL statements:

- *parse* to verify syntax and access rights of the SQL statement
- *optimize* using object information
- *execute* to actually process data
- *fetch* in queries to send retrieved data to the client

```
SQL> alter session set
  2  sql_trace = true;
```

```
PARSE #1:c=10000,e=128791,p=0,cr=3,c
EXEC #1:c=0,e=157,p=0,cr=0,cu=0
FETCH #1:c=0,e=479,p=0,cr=1,cu=2
```

Parsing SQL statements

The *hard parse* does syntax checking

- High CPU cost
- Very high contention for several latches
- A parse is hard when the SQL is not already in the library cache

The *soft parse* verifies access rights

- Some CPU cost
- High contention for several latches
- A parse is soft, if the SQL statement is already found in the library cache

Reduce parsing overhead, use Bind variables!

Application coding category 1

```
parse("select * from emp where empno=1234");
execute();
fetch();
```

- Developers not aware of importance of reducing number of parses
  - Uses a *literal* (1234)
  - Causes a hard parse for each SQL statement
  - Cannot use the shared SQL area
- Only recommended for DSS type applications
  - Decision Support System (such as Data Warehouse) applications: small numbers of users executing complex SQL statements with little or no repetition
  - Bind variable hides actual value: optimizer does not have all necessary information to choose best access plan

Application coding categories 2

```
eno = 1234;
parse("select * from emp where empno=:1");
bind(":1", eno);
execute();
fetch();
```

- All steps: cursor open, parse with bind variables, execute (fetch) and close for each individual SQL statement
- Opening curser once, and repeatedly subsequent steps: parse with bind variables, execute (and fetch) for each SQL statement

- Uses a bind variable (:1) instead of literal
- Causes a soft parse for each SQL statement
- Will use the shared SQL area

Application coding category 3

```
parse("select * from emp where empno=:1");
bind(":1", eno);
loop
    eno = <some value>;
    execute();
    fetch();
end loop;
```

- Opening a cursor and parsing with bind variables only once for each distinct SQL statement, followed by repeated executes (and fetches for queries)

- Only one single parse
- Efficiently uses the shared SQL area

Indexes !

- Index can be used to speed up queries and/or to enforce integrity rules (uniqueness)
- 2 sorts of indexes
  - B-tree
  - Bitmap
  - If cardinality (number of distinct values / number of rows) is low -> use bitmap indexes
- **`create [bitmap] index index_name on table_name(list of columns) tablespace tablespace_name`**
- Specify the tablespace
- Add indexes on column(s)!
  If not requiring full table scan for some reason!

Optimizer

- Part of the database kernel that analyzes the SQL statement and decides the best way to execute it. Its input are:
  - The SQL
  - The database design
  - Information about the data
  - User specific "hints"
- There are two main optimizer modes: Rule Based and **Cost Based**
- There are several optimizer targets: ALL_ROWS (maximum throughput), FIRST_ROWS (answer starts as soon as possible)

Rule versus Cost based optimizer

- Rule: Query plan selected following a set of predefined rules
  - tend to like a lot the indexes…
  - Being removed in a future release
  - not aware of the new features (partitioning, bitmap indexes…) → Old, NOT recommended
- Cost: Requires correct up-to-date statistics about your tables/indexes…
  - No statistics available → RBO is used!
  - knows about your data distribution…

  →New, the right way to go!

Statistics gathering

- Analyze operation feeds statistics data into the dictionary
- It helps the optimizer to choose good execution plans
- Mandatory if you want to use some of the new features (ex: bitmap indexes)
- Analyzing the table may solve performance problems ! (hours -> seconds)
- If the "profile" of the data changes, it is needed to re-analyze the data.
- Statistics are gathered using
  - SQL analyze command
  - dbms_stats.gather_<OBJ>_stats() ← better!
    OBJ e.g. schema, table
    scheduled e.g. with dbms_job()

Example: Analyse data, help the optimiser!

Which customers are also employees?

```
select c.contact_surname, c.contact_firstname, c.date_of_birth
from  employees e , customers c
where e.surname=c.contact_surname
and   e.firstname=c.contact_firstname
and   e.date_of_birth=c.DATE_OF_BIRTH
SELECT STATEMENT
   NESTED LOOPS
         TABLE ACCESS FULL    CUSTOMERS
         TABLE ACCESS BY INDEX ROWID EMPLOYEES
                 INDEX RANGE SCAN EMPLOYEES_SURNAME
```

7 seconds

```
exec dbms_stats.gather_table_stats(tabname=>'customers');
exec dbms_stats.gather_table_stats(tabname=>'employees');
```

```
select c.contact_surname, c.contact_firstname, c.date_of_birth
from  employees e , customers c
where e.surname=c.contact_surname
and   e.firstname=c.contact_firstname
and   e.date_of_birth=c.DATE_OF_BIRTH
SELECT STATEMENT
   HASH JOIN
         TABLE ACCESS FULL   EMPLOYEES
         TABLE ACCESS FULL   CUSTOMERS
```

0.4 seconds

Write good SQL, help the optimiser

- The optimiser will try to rewrite the logic in the statement, but as you know the data model, you often can do it better!
- For example:
  - … where sysdate-column>30 days

    is equivalent

    … where to sysdate-30>column
  - … where person_division<>'EST'

    is equivalent to …

    … where person_division in (
    select division_name from divisions minus
    select 'EST' from dual)

Example: Write good SQL, help the optimiser! (1/2)

Which customers are also employees?

```
select contact_surname, contact_firstname,
  date_of_birth
from customers c
where exists (select 1 from employees e
  where e.surname=c.contact_surname
  and   e.firstname=c.contact_firstname
  and   e.date_of_birth=c.date_of_birth)
SELECT STATEMENT
  FILTER
     TABLE ACCESS FULL CUSTOMERS
     TABLE ACCESS BY INDEX ROWID EMPLOYEES
          INDEX RANGE SCAN EMPLOYEES_SURNAME
```

9.5 seconds

Example: Write good SQL, help the optimiser! (2/2)

```
select c.contact_surname, c.contact_firstname,
  c.date_of_birth
from  customers c, employees e
where e.surname=c.contact_surname
and   e.firstname=c.contact_firstname
and   e.date_of_birth=c.date_of_birth
SELECT STATEMENT
  HASH JOIN
     TABLE ACCESS FULL   EMPLOYEES
     TABLE ACCESS FULL   CUSTOMERS

0.5 seconds (19 times faster!)
```

Hints

- Instructions that are passed to the Optimizer to favor one query plan vs. another
- **/*+ hint hint hint … hint */**
- Performance Tuning Guide and Reference manual
  - Many different types, e.g. hints for Optimization Approaches and Goals, Access Paths, Query Transformations, Join Orders, Join Operations, Parallel Execution, …
- Our advise: avoid as much as possible!
  - complex, not stable across releases
  - CBO w/hints same as RBO w/developer setting rules instead of optimizer!
- Warning: if they are wrongly set, Oracle will plainly ignore them
  - No error condition is raised
  - Need to check the query plan to be sure..

```
select /*+ USE_INDEX(mytable.indx_mix)*/  count(*)
from mytable
where mix = 10
```

Most famous – acceptable hints

- ALL_ROWS optimizes for best throughput
- FIRST_ROWS optimizes for best response time to get the first rows…
- FULL chooses a full table scan
  - It will disable the use of any index on the table
- INDEX chooses an index scan for the table
- AND_EQUAL will merge the scans on several single-column index

Most famous – acceptable hints

- USE_NL will join two tables using a nested loop, being the table specified in the hint, the inner table
  - read row on table A (inner table)
  - then scan table B to find a match for row obtained in 1.
  - back to 1
- USE_MERGE will join two tables using a sort-merging
  - Rows are first sorted, then the results are merged based on the join columns

Example: SQL, indexes and PL/SQL

```
-- Goal: highest paid persons per department
select e1.department_id, e1.employee_id,
  e1.surname, e1.firstname
  from employees e1
  where salary = (select max(e2.salary) from
  employees e2 where
  e2.department_id=e1.department_id)
```

```
Rows     Row Source Operation
-------  ---------------------------------------
    168  FILTER
   6401   TABLE ACCESS FULL BIGEMPLOYEES
   1318   SORT AGGREGATE
 210560    TABLE ACCESS FULL BIGEMPLOYEES
```

> 216000 rows read in EMPLOYEES full table scan
(table has 6400 rows!)
12.6 seconds, 67025 blocks read (table is 97 blocks!)

Example: Index

```
-- create an index on department_id
create index employee_dep_idx
  on employees (department_id);
Rows     Row Source Operation
-------  -----------------------------------
    168  FILTER
   6401   TABLE ACCESS FULL BIGEMPLOYEES
   1318   SORT AGGREGATE
 210560    TABLE ACCESS BY INDEX ROWID BIGEMPLOYEES
 211219     INDEX RANGE SCAN EMPLOYEE_DEP_IDX
> 216000 rows read in EMPLOYEES full table scan
   (table has 6400 rows!)
3.9 seconds, 64028 blocks used (table is 97 blocks!)
```

Force the usage of the non-unique index
… select /*+ index(e2)*/ max(e2.salary) …

Example: Concatenated Index

```
-- create an index on department_id,salary
create index employee_depsal_idx on employees
   (department_id, salary);
Rows      Row Source Operation
-------  -----------------------------------
    168  FILTER
   6401   TABLE ACCESS FULL BIGEMPLOYEES
   1318   SORT AGGREGATE
    659    FIRST ROW
    659     INDEX RANGE SCAN (MIN/MAX) EMPLOYEE_DEPSAL_IDX

6400 rows read in EMPLOYEES full table scan
   (table has 6400 rows!)
1.3 seconds, 774 blocks used (table is 97 blocks!)
```

Example: PL/SQL

```
DECLARE
cursor l_emp_cur is select department_id, surname from employees
   order by department_id, salary desc;
l_lastdeptid employees.department_id%TYPE;
l_counter_num number:=0;
BEGIN
   for l_emp_row in l_emp_cur
   loop
        if l_counter_num = 0 or l_emp_row.department_id !=
   l_lastdeptid
        then
                -- first department or the department has changed,
                -- this is the highest paid
                <output l_emp_row.department_id l_emp_row.surname>
                 counter := counter + 1 ;
        end if;
   -- remember the last department_id
    l_lastdeptid := l_emp_row.department_id;
   end loop;
END;
```

1 full table scan + sort, 1.1 seconds, 449 blocks used

Example: Rank

```
select department_id, employee_id, surname, firstname
from ( select department_id, employee_id, surname, firstname,
          dense_rank()
          over ( partition by department_id
          order by salary desc nulls LAST ) dr
       from employees )
     where dr <= 1;
Rows     Row Source Operation
-------  ---------------------------------
    168  VIEW
    189   WINDOW SORT PUSHED RANK
   6400    TABLE ACCESS FULL EMPLOYEES

800 rows read in EMPLOYEES full table scan
   (table has 6400 rows!)
1.3 seconds, 98 blocks used (table is 97 blocks!)
```

Example, Summary

Method	Time	Blocks
Full	12.6s	67'025
Index	3.9s (31%)	64'028 (95%)
Concat	1.3s (10%)	774 (1.1%)
PL/SQL	1.1s (9%)	449 (0.7%)
Rank	1.3s (10%)	98 (0.1%)

Conclusion

- Good application performance comes from good application design:
  - Avoid parses
    - Soft parse also causes scalability problems
  - Use bind variables
    - But use literals for non-repeated DSS queries
- The SQL rewrite / data definition is, most of the time, where performance can be acquired.
- You need to understand how the data is processed.
- There is a tradeoff between the time / effort / complexity and the gain.

References & Resources

- oradoc.cern.ch
  - *Performance Planning* manual
  - *Performance Tuning Guide and Reference* manual
- Tom Kyte *Effective Oracle by Design*
- *Efficient use of bind variables, cursor_sharing and related cursor parameters*, Oracle White Paper, August 2001 http://www.oracle.com/technology/deploy/performance/pdf/cursor.pdf

Hands-on exercises

- Execution Plan
- Statistics to help optimizer
- Use of Indexes
- Bind Variables
- Tuning example