DB Tuning

Best Practices from a Developer's Perspective

Giuseppe Lo Presti (IT/DM)

Database Developers' Workshop, CERN, July 8th, 2008

Outline

- Background and motivation
- Tools for DB tuning
  - Understanding execution plans
  - Using indexes and hints
  - Understanding statistics
- Looking for performance bottlenecks
  - Reading an AWR report
- Conclusions

Background and motivation

- CASTOR, the Cern Advanced STORage manager
  - Handles all physics data at CERN and in 3 Tier 1s (10s of PetaBytes)
  - Deals with magnetic tapes and a level of cache on disk
- DB centric system holding all its state in Oracle databases
  - Includes tape states, cache states, namespace, ...
- Programmed in C++, uses heavily PL/SQL and OCCI
- The CASTOR logic is PL/SQL code

Background and motivation

- As part of our development activities, we daily face database tuning issues

- What follows is a knowledge base for **developing** and **tuning** database oriented applications
  - First more theoretical, then more practical
  - The Case study will show many tricks in action
  - *Note that the developer's perspective has been largely influenced by DBAs' ones!*

The Theory: Tools for DB tuning

- Understanding Execution Plans
- Using Indexes and Hints
- Understanding Statistics

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Understanding Execution Plans

- Synopsis: exec plans are characterized by
  - Access Path
    - Full-table scan
    - Row ID scan
    - Cluster scan
    - Index scan
  - Join Order
  - Join Method
    - Nested-loop
    - Hash
    - Sort-merge
    - Anti or Semi
    - Cartesian

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Access Paths (1)

- Full Table Scan (FTS)
  - Small table (< 5K rows), no indexes, most rows to be accessed anyway
  - Oracle optimizes FTSs using multiblock I/O
  - *Hint(!):* `FULL(Table)`
- Row ID Scan
  - Usually after index lookup
    - Not always! If index already contains requested data, no table access is performed at all
  - Using the rowID is the fastest way to retrieve a single row
    - But **not necessarily the fastest** to retrieve multiple rows!
  - *Hint:* `ROWID(Table)`
- Cluster scan
  - For *clustered* tables
    - Pairs of tables stored as permanently joined, *replicating* data where needed

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Access Paths (2)

- Index scan, when index(es) avaiable
- Indexes contain rowIDs, which are used afterwards to access the data via rowID scan
  - Unique scan
    - When `UNIQUE` or `PRIMARY KEY` constraints
  - Range scan [descending]
    - Standard traversal of an index: data is returned in ascending [descending] order of index columns
    - NOT NULL constraints help choosing an index to satisfy an ORDER BY clause, thus avoiding further sorting
    - *Hint:* *`INDEX(Table Index_name)`*
  - Skip scan
    - For composite (or concatenated) indexes – more later
    - *Hint:* *`INDEX_SS(Table Index_name)`*

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Access Paths (3)

- Index scan (continued)
  - Fast full scan (FFS)
    - Using multiblock I/O (fast), **not** in order
    - *Hint:* `INDEX_FFS(Table Index_name)`
  - Full scan
    - Preserves order, less efficient w.r.t. I/O than FFS
  - Index join
    - Hash join of several indexes that together contain all the table columns referenced in the query
    - *Hint:* `INDEX_JOIN`
  - Bitmap join
    - In case Bitmap indexes are defined (more later), or when complex boolean operations are required: in such a case, Oracle may build bitmaps on the fly
    - *Hint:* `INDEX_COMBINE` *(only to force usage of a bitmap index)*

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Index Scan vs. Full Table Scan

- Index Scan = index access + table access via rowID
- Full Table Scan = table access via multiblock I/O
- Which one is the fastest access?
  - Very selective query vs. non-selective one
- Imagine you have a table where a lot of DML activity occurs - and the indexes become very fragmented
- Index Clustering factor
  - When too high, index access performances may drop
  - And FTS may outperform index access!
  - You need to rebuild the indexes, or use different techniques…

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

DM

Join Order

- Rule 1
  - A *single-row predicate* (e.g. `T.value = :1`) forces its row source to be placed **first** in the join order
- Rule 2
  - For outer joins, the table with the outer-joined table must come **after** the other table in the join order for processing the join
- Ordering can be overridden
  - `LEADING` hint allows specifying a complete join order
    - Example at the Case study session
  - If the suggested order violates rule 2, the hint is ignored

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Join Methods (1)

- Nested Loop Joins
  - Outer (or driving) table, inner table
  - Basically:
    for each (out table)   // *O*(*n*) access
      for each (in table)   // *O*(*n**m*) accesses
        check for a match
  - Usually for joining a small number of rows that have a good (= selective) driving condition
  - Most powerful (and most expensive)
  - *Hint:* `USE_NL(Table1 Table2)`

Join Methods (2)

- Hash Joins
  - **Only** on equijoins
  - Used when most data from a table need to be joined
  - The smaller of the two tables is scanned (FTS) to build a hash table on the join key
  - Then the larger one is scanned (FTS) probing the hash table to find the joined rows
  - Better than sort-merge and NLs, but more expensive in memory (PGA)
  - *Hint:* `USE_HASH(Table1 Table2)`

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Join Methods (3)

- Sort-Merge joins
  - The rows from each table are sorted on the join predicate columns
  - The two sorted sources are then merged and returned
  - It may be expensive due to the sorting operation, especially if it is not performed all in memory
  - Used if no equijoin, or if sorts are required for subsequent operations
  - *Hint:* `USE_MERGE(Table1 Table2)`

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Join Methods (4)

- Antijoins
  - Queries including a `NOT IN` subquery
- Semijoins
  - Queries with an `EXISTS` subquery
- Cartesian joins
  - Joins without condition
  - Normally a programming mistake…

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Digression: sorting

CERN IT Department

- Sorts are common operations in execution plans.
  We can find the following sorts in execution plans:
  - SORT UNIQUE: if query specifies a `DISTINCT` clause or if next step requires unique values
  - SORT AGGREGATE: ***not*** a real sort, it's used when aggregates are computed across the whole set of rows (e.g. `MIN()`)
  - SORT GROUP BY: used on `GROUP BY` queries. The sort is required to separate the rows in groups
  - SORT JOIN: during sort-merge joins
  - SORT ORDER BY: if query specifies an `ORDER BY` clause
- Other clauses which require sorting: `UNION, MINUS, INTERSECT`
  - These are expensive operations!

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Determining exec plans

- EXPLAIN PLAN command
  - **Theoretical** plan that can be used by a stmt
  - ```
    EXPLAIN PLAN SET statement_id = 'myStmt' FOR (<any
    SQL query>);
    SELECT PLAN_TABLE_OUTPUT FROM
      TABLE(DBMS_XPLAN.DISPLAY());
    ```
- VSQL_PLAN , VSQL_PLAN_STATISTICS_ALL views
  - **Actual** plan being used by a running cursor
  - ```
    SELECT PLAN_TABLE_OUTPUT FROM
      TABLE(DBMS_XPLAN.DISPLAY_CURSOR(
        <sql_id> [, <format>]));
    ```
  - AWR reports can help here, as they provide the sql_id and usage statistics of top activity cursors/queries
    - *More later*
- SQL*Plus autotrace
  - ```
    set autotrace on | traceonly [explain]
    ```
    ```

The Theory: Tools for DB tuning

CERN IT Department

- Understanding Execution Plans
- Using Indexes and Hints
- Understanding Statistics

Indexes and Hints

- Synopsis: indexes can be
  - Unique vs. nonunique
  - Composite
  - Bitmap
  - Bitmap join
  - Function based
- Storage: B*tree
  - Normal
  - Reverse key
  - Function based
- Index data is usually separated from table data
  - Index-organized tables (IOT) have data stored *within* an index

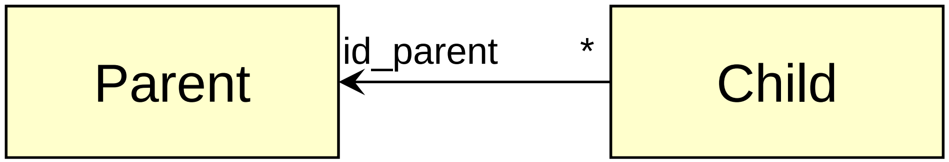
CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Effect of DML queries

- *INSERT, UPDATE, DELETE* clauses
- *Inserts* result in the insertion of an index entry in the appropriate block
  - Block splits might occur
- *Deletes* result in a deletion of the index entry
  - Empty blocks become available
- *Updates* to the key columns result in a logical delete + insert to the index
- After heavy DML activity, it is adviced to reorganize (rebuild) B*tree indexes

Indexes and constraints

- Primary or Unique key constraints implicitly create an index
- FKs don't have implicit indexes
  - But are welcome…
- With FKs, when deleting or updating parent rows
  - All matching child rows need to be located to make sure there are no dependents (otherwise => FK violation)
  - Without an index, this results in a FTS of the child table



CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Composite indexes

- Indexes on more than one column
  - Better selectivity
  - If all columns selected by a query are in a composite index, no access is performed on the table (cf. IOTs later)
- Guidelines
  - Column order should match WHERE clauses
  - Most queried columns -> leading part of the index
    - Partial match on the **leading** part fine as well
  - Most restrictive column -> leading part of the index?
    - Oracle can use *Index Skip Scanning* access on a composite index when the index prefix column is not part of the predicate
    - …but this common sense guideline is actually a myth!
    - Index compressibility arguments make the opposite choice preferable – and performance-wise there's no difference

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Bitmap indexes

- Designed for low cardinality columns
  - For each distinct value of the column, a bitmap “stripe” is created, with size = #rows in the table
  - Very storage efficient, each stripe is compressed and stored in a B*tree structure
- Pros
  - Complex WHERE clauses and group functions (e.g. COUNT and SUM) are resolved with bitwise operations
  - Large tables benefit wrt standard index
    - Breakeven point: #different values <= 1% #rows
- Cons
  - Adding/removing values in the indexed column(s) makes new stripes to be built/old ones to be dropped
  - Hence high DML activity kills performances…

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Table Partitioning

CERN IT Department

- Not really an indexing technique…
- When you have a low cardinality column
  - And Bitmap indexes are out of the game because of the DML activity
  - Plus you want to be able to shrink online your table
- Then consider list partitioning on that column
  - You can choose to make indexes on other columns *local* to the partitions, or global (default)
- Pros
  - Queries accessing one or few values will concentrate only on the involved partition(s)
  - The underlying table can have `ROW MOVEMENT` enabled for shrinking
- Cons
  - A bit more complex to handle?

Bitmap join indexes

- Bitmap index on the join of two or more tables
  - Kind of denormalization (cf. Clustered tables) but at index level: key in one table, value (= rowID) on another one
- Pros
  - Queries on that join often don't need to access the table data
  - Space efficient
- Cons
  - Only one table can be updated concurrently by different transactions: a table update effectively takes a lock on the indexed values
  - Cannot be (re)built online

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Function-based indexes

- Index on expressions (virtual columns)
- Can be created as bitmap index
- Pros
  - Queries with complex expressions as conditions may benefit from a FB index on that expression
- Cons
  - The underlying table cannot have `ROW MOVEMENT` enabled, thus online shrinking not permitted

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Index Organized Tables

- Equivalent to a table with a Composite index on *all* of its columns
  - Based on a B*tree on the PK of the table
  - Index values directly contain all other data, not rowIDs
  - Large rows (e.g. when LOB fields are present) may be stored in other segments, to preserve the dense storage of the B*tree structure
  - Fragmentation may occur as result of incremental updates. `ALTER TABLE TabName MOVE [OVERFLOW]` rebuilds the IOT (cf. index rebuilding)
- Pros
  - Fast, key-based access for queries involving exact match or range searches on the PK
- Cons
  - Not suitable for queries that do not use the PK in a predicate

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

DM

Other miscellaneous hints

CERN IT Department

- On top of the mentioned hints to suggest access paths / indexes, other recognized hints are:
  - For access paths
    - `NO_INDEX`: disallows using (a set of) indexes
    - `AND_EQUAL(Table Idx1..IdxN)`: merges the scans on several single-column indexes; 2 <= N <= 5
  - For query transformations
    - `USE_CONCAT`: expands/rewrites `OR` into `UNION ALL`, and OR-expands all `IN`-lists.
    - `NO_EXPAND`: prevents this expansion
  - Others
    - `ALL_ROWS | FIRST_ROWS(n)`: for overall query optimization
    - `APPEND | NO_APPEND`: for direct-path INSERTs
    - `ORDERED_PREDICATES`: forces predicate evaluation order
    - `DYNAMIC_SAMPLING(n)`: *more on Statistics*

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

The Theory: Tools for DB tuning

- Understanding Execution Plans
- Using Indexes and Hints
- Understanding Statistics

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Understanding Statistics

- *Statistics*: information used by the Optimizer to estimate
  - Selectivity of predicates
  - Cost of each execution plan
  - Access and join method
  - CPU and I/O costs
- Types of statistics
  - Objects: Table (e.g. avg row length), Column (# of distinct values, histogram), Index
  - System: I/O performance, CPU performance
- Object (not System) stats automatically gathered
  - Scheduled job 'GATHER_STATS_JOB'
  - Manual gathering possible via DBMS_STATS package

Gathering statistics

- Oracle uses a DML monitoring facility to track objects for stale or missing statistics
  - Enabled by default when `STATISTICS_LEVEL` is set to `TYPICAL` or `ALL`
  - The `user_tab_modifications` view can be used to see information about changes to tables
  - To force regathering of stale statistics:

```
DBMS_STATS.GATHER_DATABASE_STATS
  (options => GATHER_STALE);
```

- Statistics gathering relies on *sampling*
  - `estimate_percent` is an argument of `GATHER_DATABASE_STATS()` to help steering the sampling percentage
    - `AUTO_SAMPLE_SIZE` value maximizes performance while achieving necessary statistical accuracy
- Statistics can be *locked*

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Histograms

- Provide improved selectivity estimates in the presence of data skew
  - Values with large variations in the number of duplicates
- Can be created on demand
  - ```
    DBMS_STATS.GATHER_TABLE_STATS
       (userName, tableName, method_opt =>
         'for columns size g column_name');
    ```
  - `g` is the granularity, i.e. the number of buckets
    - Default value is 75, max is 254, `auto` may be specified too
    - Oracle never creates more buckets than # of distinct values
- Guidelines
  - Do **not** use them unless they substantially improve performances
    - Storage and CPU costs

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it
    ```

Dynamic sampling

- Used to automatically collect statistics when
  - Cost of collecting stats is minimal compared to exec time
  - Query is executed many times
- The `OPTIMIZER_DYNAMIC_SAMPLING` parameter enables dynamic sampling. Values:
  - 0: disabled
  - 1: enabled when the optimizer determines that a Full Table Scan is required due to non-existent statistics
  - 2..10: any value in this range increases the likelihood that dynamic sampling is an option
  - *Hint:* *`DYNAMIC_SAMPLING(n)`*

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

System statistics

- Statistics on CPU and I/O costs
- Only for DBAs
- When generated, already existing execution plans don't get invalidated
- Automatic gathering controlled by `DBMS_STATS.GATHER_SYSTEM_STATS()`

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Outline

- Background and motivation
- Tools for DB tuning
  - Understanding execution plans
  - Using indexes and hints
  - Understanding statistics
- Looking for performance bottlenecks
  - Reading an AWR report
- Conclusions

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

DM

Looking for performance bottlenecks

CERN IT Department

- A practical method is proposed here, based on the AWR, to help finding *high-load* SQL queries
  - See also Luca's presentation
- The AWR (Automatic Workload Repository) is a repository of statistics gathered by Oracle
  - Automatically, e.g. every 20 minutes
  - On demand:
    `DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT('ALL');`
- Detailed reports can be extracted about the database activity and workload between two snapshots
  - **This is the whole activity**: if more users share an Oracle instance, they will all appear in the report
  - `SQL> @awrrpt`
  - Give at least 5 minutes between the two snapshots

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Reading an AWR Report

CERN IT Department

- Header
  - Db time vs clock time
  - When the ratio is > (or >>) 1, there may be a problem. E.g.:

WORKLOAD REPOSITORY report for

DB Name	DB Id	Instance	Inst num	Release	RAC	Host
C2ALISTG	276138430	C2ALISTG	1	10.2.0.2.0	NO	lxfsrk4302.cern.ch

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	6027	02-Oct-06 15:00:31	90	36.4
End Snap:	6028	02-Oct-06 16:00:53	90	34.8
Elapsed:		60.38 (mins)		
DB Time:		210.62 (mins)		

- Main Report
  - Look for SQL Statistics, in particular SQL ordered by Elapsed Time table
  - Then look for the most consuming query

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

DM

Reading an AWR Report

CERN IT Department

- Reading info about cache hits/misses
  - *Db block gets* and *Physical reads* are 'cache miss', real disk I/O operations
  - *Consistent gets* include all gets both from memory cache and from disk
- E.g. first two queries are reading a lot from disk:

SQL ordered by Reads

- Total Disk Reads: 46,826,641
- Captured SQL account for 99.7% of Total

Physical Reads	Executions	Reads per Exec	% Total	CPU Time (s)	Elapsed Time (s)	SQL Id	SQL Module	SQL Text
44,982,460	5,235	8,592.64	96.06	3239.73	4569.90	2hyxdv4kwp6gb	stager@c2alicesrv03.cern.ch (TNS V1-V3)	BEGIN putStart(:1, :2, :3, ...
44,981,449	5,236	8,590.80	96.06	3234.03	4550.41	g45yf6u0x9zx2	stager@c2alicesrv03.cern.ch (TNS V1-V3)	UPDATE CASTORFILE SET LASTKNOW...
992,875	693	1,432.72	2.12	157.71	265.94	d9t4kmuzfymv5	stager@c2alicesrv03.cern.ch (TNS V1-V3)	SELECT

  - Why?

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Finding what's wrong

CERN IT Department

- In this case, the `putStart()` PL/SQL procedure contains a query which is badly performing:
  - ```
    UPDATE CastorFile
       SET lastKnownFileName = ...
     WHERE <some nested criteria with joins>;
    ```

#	Operation	Options	Object name	Mode	Cost	Bytes	Cardinality
DML	UPDATE CASTORFILE SET LASTKNOWNFILENAME ...						
0	UPDATE STATEMENT			ALL_ROWS	5647	2079506	17186
1	UPDATE		CASTORFILE				
2	HASH JOIN	RIGHT SEMI			5647	2079506	17186
3	VIEW		VW_NSO_1		2052	223418	17186
|4	NESTED LOOPS				2052	4124640	17186
| 5	NESTED LOOPS				5	126	1
| |6	TABLE ACCESS	BY INDEX ROWID	SUBREQUEST	ANALYZED	3	12	1
| ||7	INDEX	UNIQUE SCAN	SYS_C003155	ANALYZED	2		1
| |8	TABLE ACCESS	BY INDEX ROWID	CASTORFILE	ANALYZED	2	38447982	337263
| | 9	INDEX	UNIQUE SCAN	SYS_C003165	ANALYZED	1		1
| 10	TABLE ACCESS	BY INDEX ROWID	CASTORFILE	ANALYZED	2047	1959204	17186
| 11	INDEX	RANGE SCAN	I_CASTORFILE_LASTKNOWNFILENAME	ANALYZED	55		[illegible]
12	TABLE ACCESS	FULL	CASTORFILE	ANALYZED	3588	36424404	337263
    ```

Explaining the problem

- The execution plan indicates that a Full Table Scan is performed on CastorFile, with *O*(800K) rows
  - The problem is that Oracle thinks **337K** entries must be retrieved from CastorFile (the *cardinality* value on the right)
    - But we know from the application perspective that it might be **1**, if any!
    - So an index access on the lastKnownFileName field is sufficient here!
  - In fact, statistics were getting stale on the relevant index…
- Possible action
  - Update statistics:

```
exec dbms_stats.gather_table_stats(
  ownname=>'castor_stager', tabname=>'CastorFile');
```

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Going further

CERN IT Department

- It might be not enough to recompute statistics
  - E.g. all indexes involved in the query are properly updated
- The *theoretical* plan may look good
  - But you want to know why your query is following a bad plan *at runtime* (i.e. on the real data)
- Then you can use the v$sql_plan_statistics_all system view
  - You first need to enable full statistics for a while:

    ```
    SQL> alter system set statistics_level='ALL' scope=memory;
    ```
  - The view contains data about **expected** vs. **actual** #rows read by each step of the execution plan
  - Usually gives good hints about unexpected data distributions, which may have led to the bad plan
    - Again, YOU know the "good" data distribution!

CERN IT Department
CH-1211 Genève 23
Switzerland
www.cern.ch/it

Conclusions

CERN IT Department

- We have shown a number of tools and techniques for DB tuning
  - Indexes and hints
  - Usage of AWR report

- But don't forget that no matter what Oracle provides, **the best optimizer is the developer!**

Acknowledgments

- Most of the presented material comes from an Oracle course on advanced SQL tuning
- Many thanks to IT/DM and IT/DES DBAs for their advices

- Questions?