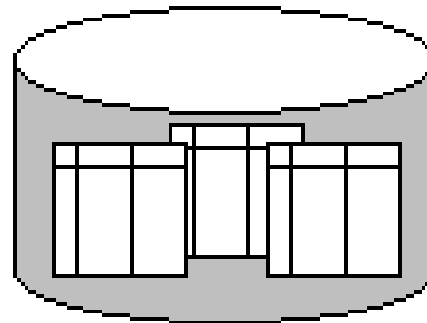


QUERY OPTIMIZATION AND TUNING IN ORACLE



Query Tuning Hints

- Avoid redundant DISTINCT
- Change nested queries to join
- Avoid unnecessary temp tables
- Avoid complicated correlation subqueries
- Join on clustering and integer attributes
- Avoid HAVING when WHERE is enough
- Avoid views with unnecessary joins
- Maintain frequently used aggregates
- Avoid external loops
- Avoid cursors
- Retrieve needed columns only
- Use direct path for bulk loading

Avoid Redundant DISTINCT

```
SELECT DISTINCT ssnum  
FROM Employee  
WHERE dept = 'information systems'
```

- **DISTINCT** usually entails a **sort operation**
 - ◆ Slow down query optimization because one more “interesting” order to consider
- Remove if you know the result has no duplicates (or duplicates are acceptable) or if answer contains a key

Avoid HAVING when WHERE is enough

```
SELECT  MIN (E.age)
FROM    Employee E
GROUP BY E.dno
HAVING  E.dno=102
```



```
SELECT  MIN (E.age)
FROM    Employee E
WHERE   E.dno=102
```

- May first perform grouping for *all* departments!
- Consider DBMS use of index when writing arithmetic expressions:
 $E.age=2*D.age$ will benefit from index on $E.age$, but might not benefit from index on $D.age$!

Avoid Using Intermediate Relations

```
SELECT * INTO Temp
FROM Emp E, Dept D
WHERE E.dno=D.dno
AND D.mgrname='Joe'
```

and

```
SELECT T.dno, AVG(T.sal)
FROM Temp T
GROUP BY T.dno
```

vs.

```
SELECT E.dno, AVG(E.sal)
FROM Emp E, Dept D
WHERE E.dno=D.dno
      AND D.mgrname='Joe'
GROUP BY E.dno
```

- Creating **Temp** table causes update to catalog
 - Cannot use any index on original table
- Does not materialize the intermediate relation **Temp**

Optimizing Set Difference Queries

Suppose you have to select all of the employee's that are not account representatives:

Table1:

s_emp	soc_number	last_name	first_name	salary
-------	------------	-----------	------------	--------

Table2:

soc_number	last_name	first_name	region
------------	-----------	------------	--------

- This query is slower:

```
SELECT soc_number FROM s_emp  
MINUS  
SELECT soc_number FROM s_account_rep;
```

- because the minus has to select distinct values from both tables

Optimizing Set Difference Queries

```
SELECT soc_number
FROM s_emp
WHERE soc_number NOT IN
      (SELECT soc_number
       FROM s_account_rep);
```

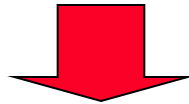
- Faster, but still not as fast because we are not joining and are not using indexes, so the following query is faster

```
SELECT /*+ index(t1) */ soc_number
FROM s_emp t1
WHERE NOT EXISTS
      (SELECT /*+ index(t1) index(t2) */ *
       FROM s_account_rep t2
       WHERE T1.soc_number = t2.soc_number);
```

Change Nested Queries to Join

```
SELECT ssn  
FROM Employee  
WHERE dept IN (SELECT dept FROM Techdept)
```

- Might not use index on Employee.dept

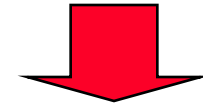


```
SELECT ssn  
FROM Employee, Techdept  
WHERE Employee.dept = Techdept.dept
```

- Need DISTINCT if an employee might belong to multiple departments

Avoid Complicated Correlation Subqueries

```
SELECT snum
FROM Employee e1
WHERE salary =
      (SELECT MAX(salary)
       FROM Employee e2
       WHERE e2.dept = e1.dept)
```

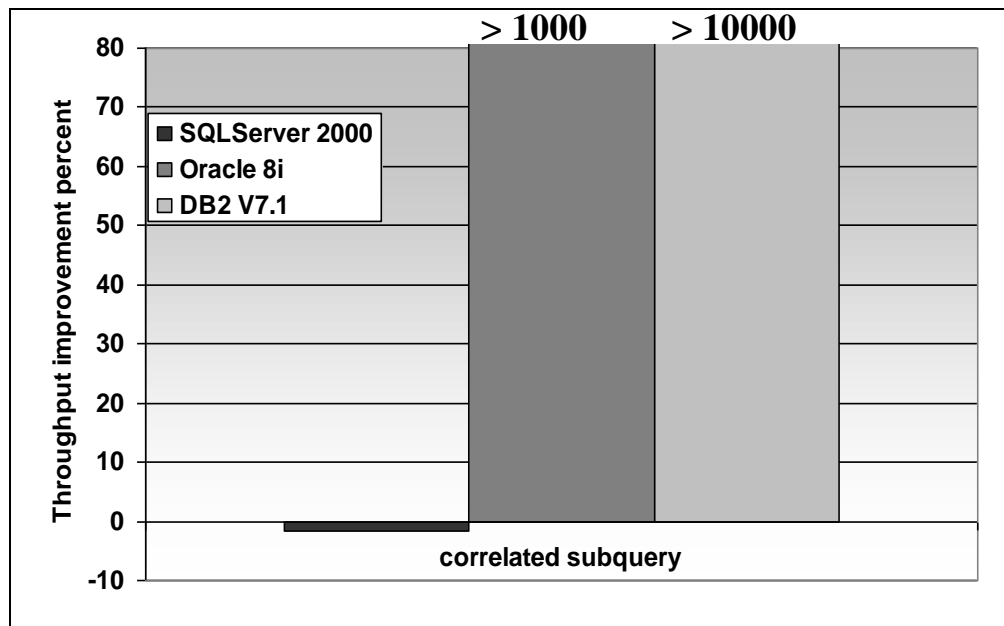


- Search all of e2 for each e1 record!

```
SELECT MAX(salary) as bigsalary, dept INTO Temp
FROM Employee
GROUP BY dept
```

```
SELECT snum
FROM Employee, Temp
WHERE salary = bigsalary
AND Employee.dept = Temp.dept
```

Avoid Complicated Correlation Subqueries

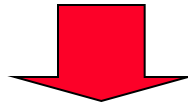


- SQL Server 2000 does a good job at handling the correlated subqueries (a hash join is used as opposed to a nested loop between query blocks)
 - ◆ The techniques implemented in SQL Server 2000 are described in “Orthogonal Optimization of Subqueries and Aggregates” by C.Galindo-Legaria and M.Joshi, SIGMOD 2001

Join on Clustering and Integer Attributes

```
SELECT Employee.ssnnum  
FROM Employee, Student  
WHERE Employee.name = Student.name
```

- Employee is clustered on ssnnum
- ssnnum is an integer

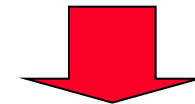


```
SELECT Employee.ssnnum  
FROM Employee, Student  
WHERE Employee.ssnnum = Student.ssnnum
```

Avoid Views with Unnecessary Joins

```
CREATE VIEW Techlocation  
AS SELECT ssnnum, Techdept.dept, location  
FROM Employee, Techdept  
WHERE Employee.dept = Techdept.dept  
  
SELECT dept  
FROM Techlocation  
WHERE ssnnum = 4444
```

- Join with Techdept unnecessarily



```
SELECT dept  
FROM Employee  
WHERE ssnnum = 4444
```

Aggregate Maintenance

- Materialize an aggregate if needed “frequently”
- Use trigger to update

```
create trigger updateVendorOutstanding on
                                orders for insert as
update vendorOutstanding
set amount =
    (select vendorOutstanding.amount+sum(
                                inserted.quantity*item.price)
    from inserted,item
    where inserted.itemnum = item.itemnum
    )
where vendor = (select vendor from inserted) ;
```

Avoid External Loops

- No loop:

```
sqlStmt = "select * from lineitem where l_partkey <= 200;"
```

```
odbc->prepareStmt(sqlStmt);
```

```
odbc->execPrepared(sqlStmt);
```

- Loop:

```
sqlStmt="select * from lineitem where l_partkey = ?;"
```

```
odbc->prepareStmt(sqlStmt);
```

```
for (int i=1; i<200; i++)
```

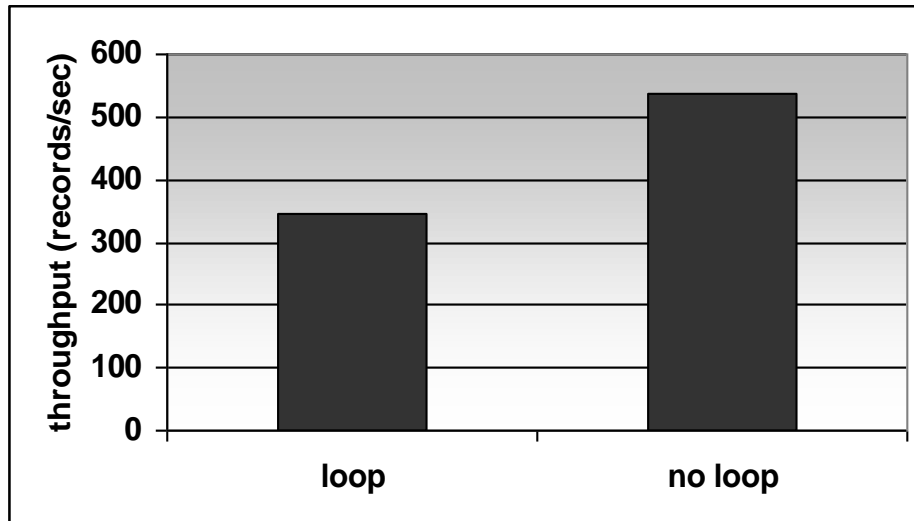
```
{
```

```
    odbc->bindParameter(1, SQL_INTEGER, i);
```

```
    odbc->execPrepared(sqlStmt);
```

```
}
```

Avoid External Loops



- SQL Server 2000 on Windows 2000
- Crossing the application interface has a significant impact on performance

• Let the DBMS optimize set operations

Avoid Cursors

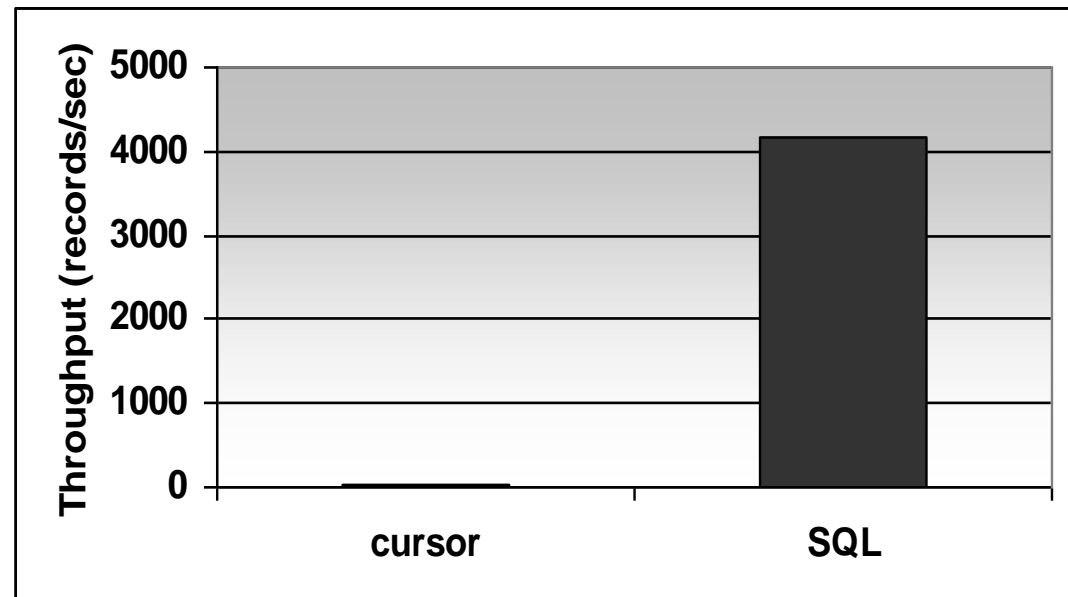
- No cursor

```
select * from  
employees;
```

- Cursor

```
DECLARE d_cursor CURSOR  
FOR select * from  
employees;  
OPEN d_cursor  
while(@@FETCH_STATUS=0)  
BEGIN  
    FETCH NEXT from  
    d_cursor  
END  
CLOSE d_cursor  
go
```

- SQL Server 2000 on Windows 2000
- Response time is a few seconds with a SQL query and more than an hour iterating over a cursor

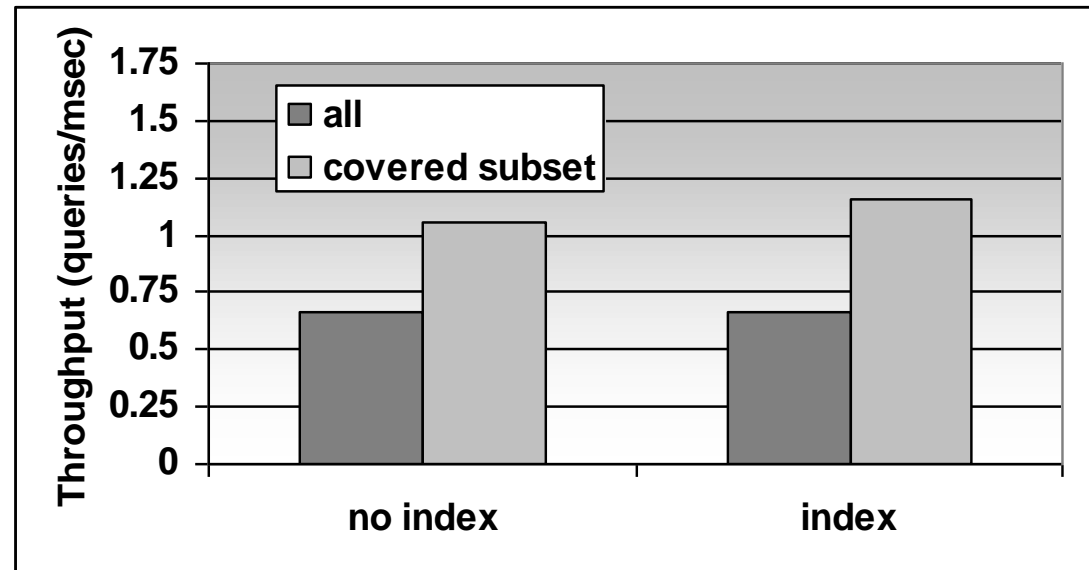


Retrieve Needed Columns Only

- All
 - select * from lineitem;
- Avoid transferring unnecessary data
- May enable use of a covering index

- Covered subset

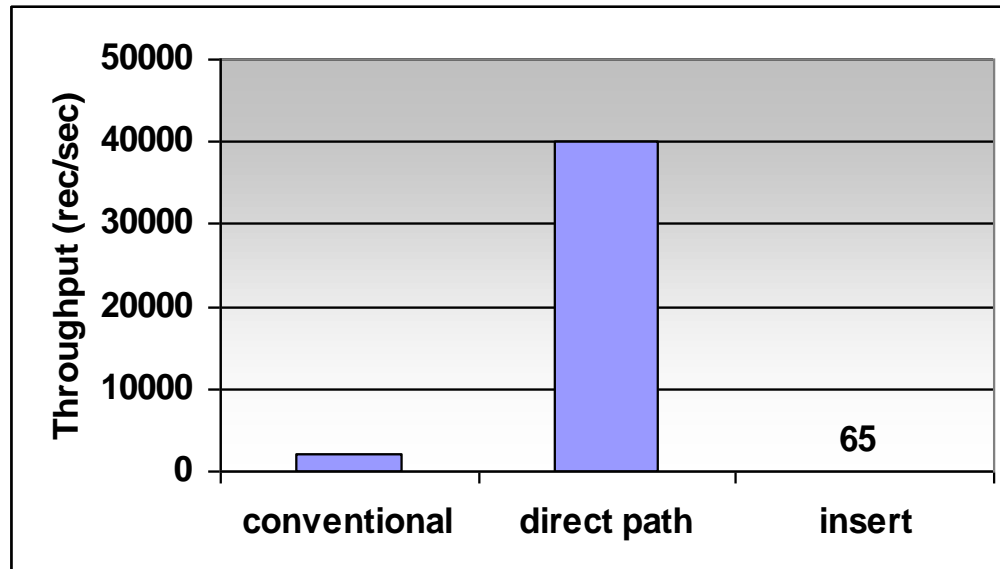
```
select l_orderkey,  
       l_partkey, l_suppkey,  
       l_shipdate,  
       l_commitdate from  
lineitem;
```



Use Direct Path for Bulk Loading

```
sqlldr directpath=true control=load_lineitem.ctl
data=E:\Data\lineitem.tbl
load data
infile "lineitem.tbl"
into table LINEITEM append
fields terminated by '|'
(
  L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER,
  L_QUANTITY, L_EXTENDEDPRICE, L_DISCOUNT, L_TAX,
  L_RETURNFLAG, L_LINESTATUS, L_SHIPDATE DATE "YYYY-
MM-DD", L_COMMITDATE DATE "YYYY-MM-DD",
  L_RECEIPTDATE DATE "YYYY-MM-DD", L_SHIPINSTRUCT,
  L_SHIPMODE, L_COMMENT
)
```

Use Direct Path for Bulk Loading



- Direct path loading bypasses the query engine and the storage manager
- It is orders of magnitude faster than for conventional bulk load (commit every 100 records) and inserts (commit for each record)

ORACLE Query Optimization Approaches

- Oracle supports two approaches for query optimization: rule-based and cost-based, which was introduced in Oracle 7 in order to improve query optimization
- **Rule-based**: The optimizer ignores statistics
- **Cost-based**: Three different goals
 - ◆ **All_Rows**: The optimizer optimizes with a goal of best throughput (minimum resource use to complete the entire statement)
 - ◆ **First_Rows_n**: The optimizer optimizes with a goal of best response time to return the first n number of rows; n can equal 1, 10, 100 or 1000
 - ◆ **First_Rows**: The optimizer uses a mix of cost and heuristics to find a best plan for fast delivery of the first few rows

ORACLE Query Optimization Approaches

- Note: using **heuristics** sometimes leads the CBO to generate a plan with a cost that is significantly larger than the cost of a plan without applying the heuristic
- For a specific statement the goal to be used by the optimizer can be stated using a hint
- To specify the optimizer's goal for an entire session, use the following statement:

```
alter session set optimizer_mode = <MODE_VALUE>,  
where MODE_VALUE = {rule, all_rows, first_rows,  
first_rows_n, choose}
```

ORACLE Query Optimization Approaches

- The “choose” mode states that:
 - ◆ The optimizer chooses between a cost-based approach and a rule-based approach, depending on whether statistics are available. This is the default value
 - ◆ If the data dictionary contains statistics for at least one of the accessed tables, then the optimizer uses a cost-based approach and optimizes with a goal of best throughput
 - ◆ If the data dictionary contains only some statistics, then the cost-based approach is still used, but the optimizer must guess the statistics for the subjects without any statistics. This can result in suboptimal execution plans

Rule-Based Approach

- When ignoring statistics & heuristics, there should be a way to choose between possible access paths suggested by different execution plans
- Thus, 15 rules were ranked in order of efficiency. An access path for a table is chosen if the statement contains a predicate or other construct that makes that access path available
- Score assigned to each execution strategy (plan) using these rankings and strategy with best (lowest) score selected
- When two strategies produce the same score, tie-break resolved by making decision based on order in which tables occur in the SQL statement

Understanding the RBO

Table 20.4 Rule-based optimization rankings.

Rank	Access path
1	Single row by ROWID (row identifier)
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite key
9	Single-column indexes
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort–merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed columns
15	Full table scan

RBO: An Example

- Suppose there is a table “PropertyForRent” with indexed attributes: propertyNo, rooms and city. Consider the query:
SELECT propertyNo
FROM PropertyForRent
WHERE rooms > 7 AND city = ‘Sydney’
 - ◆ Single-column access path using index on city from WHERE condition (city = ‘Sydney’): rank 9
 - ◆ Unbounded range scan using index on rooms from WHERE condition (rooms > 7): rank 11.
 - ◆ Full table scan: rank 15
 - ◆ Although there is an index on propertyNo, the column does not appear in the WHERE clause and so is not considered by the optimizer
- Based on these paths, rule-based optimizer will choose to use the index on the “city” column

Cost-Based Approach

- Cost-based optimizer depends on statistics for all tables, clusters, and indexes accessed by query
 - ◆ Users' responsibility to generate statistics and keep them up-to-date
- Two ways for generating and managing statistics:
 - ◆ By using package DBMS_STATS, for example:
EXECUTE
DBMS_STATS.GATHER_SCHEMA_STATS(' *schema_name* ');
Schema_name: name of user that owns tables
 - ◆ By issuing the ANALYZE statement, for example:
 - ANALYZE TABLE <table_name> COMPUTE/ESTIMATE STATISTICS;
 - ANALYZE TABLE <table_name> COMPUTE/ESTIMATE STATISTICS FOR TABLE;
 - ANALYZE TABLE <table_name> COMPUTE/ESTIMATE STATISTICS FOR ALL INDEXES;

Understanding the CBO

- **Functionality**

- ◆ Parse the statement
- ◆ Generate a list of all potential execution plans
- ◆ Calculate (estimate) the cost of each execution plan
- ◆ Select the plan with the lowest cost

- **Parameters**

- ◆ Primary Key – Unique Index
- ◆ Non-Unique Index
- ◆ Range evaluation (with bind variables)
- ◆ Histograms
- ◆ System Resource Usage
- ◆ Current Stats

Query Tuning -- What to do?

- Problematic SQL statements usually have:
 - ◆ Excessive number of buffer gets
 - ◆ Excessive number of physical reads
- So, if we consume less resources, we save time
 - ◆ Reduce buffer gets (more efficient access paths)
 - Avoid (most) full table scans
 - Check selectivity of index access paths
 - Stay away from Nested Loop joins on large row sources
 - ◆ Avoid physical I/O
 - Avoid (most) full table scans
 - Try to avoid sorts that write to disk, such as order by, group by, merge joins (set adequate sort_area_size)
 - Try to avoid hash joins writing to disk (hash_area_size)

Access Paths

- Next, some of the triples **operation-option-description** (corresponding to **access paths**) that can be found in execution plans are being described
 - ◆ Not all of them are available with the rule-based optimizer
- For more details, check Table 9-4 in chapter 9 of Database Performance Tuning Guide and Reference

B*-Tree Indexes

- Excellent performance for highly selective columns
 - ◆ Not effective for low selectivity columns
- Unique scan is most efficient, equality predicate on unique index
- Range scan can be quite efficient, but be careful of the size of the range specified
- Excellent for FIRST_ROWS access, particularly with queries returning a small number of results
- Index access paths
 - ◆ INDEX UNIQUE SCAN
 - ◆ INDEX RANGE SCAN
 - ◆ INDEX FULL SCAN
 - ◆ INDEX FAST FULL SCAN
 - ◆ INDEX SKIP SCAN (9i only)

B*-Tree Index Access Paths

- INDEX UNIQUE SCAN

- ◆ Equality predicate on unique or primary key column(s)
- ◆ Generally considered most efficient access path
- ◆ usually no more than 3-4 buffer gets
- ◆ If table is “small”, FULL TABLE SCAN could be cheaper

- INDEX RANGE SCAN

- ◆ Equality predicate on non-unique index, incompletely specified unique index, or range predicate on unique index
- ◆ Be careful of the size of the range
 - Large ranges could amount to huge # of buffer gets
 - If so, consider a FAST FULL INDEX SCAN or FULL TABLE SCAN

B*-Tree Index Access Paths

● INDEX FULL SCAN

- ◆ Will scan entire index by walking tree, in index order
- ◆ Provides ordered output, can be used to avoid sorts for ORDER BY clauses that specify index column order
- ◆ Slower than INDEX FAST FULL SCAN, if there is no ORDER BY requirement

● INDEX FAST FULL SCAN

- ◆ Will read index, in disk block order, and discard root and branch blocks
- ◆ Will do *db file scattered read*, reading *db_file_multiblock_read_count* blocks at a time
- ◆ Equivalent to FULL TABLE SCAN for an index
- ◆ Fastest way to read entire contents of an index

B*Tree Index Access Path

- INDEX SKIP SCAN (Oracle 9i only)
 - ◆ Allows some benefits of multi-column index, even without specifying the leading edge
 - ◆ Oracle will “skip scan”, starting with root block, skipping through B*-tree structure, masking sections of tree that cannot have applicable data
 - ◆ Could be costly, depending on size of index, distribution of data, and bind variable values

Bitmap Indexes

- Are most often implemented in a Data Warehouse environment
- Are useful for columns which:
 - ◆ have relatively low cardinality, where B*-Tree indexes will fail to provide any benefit
 - ◆ are often specified along with other columns in WHERE clauses of SQL statements, optimizer will BITMAP AND the results of many single column bitmap indexes together
- Are most efficient when doing COUNT(*) operations, where optimizer can utilize the BITMAP CONVERSION COUNT access path
- Index Access Paths
 - ◆ BITMAP INDEX SINGLE VALUE
 - ◆ BITMAP INDEX RANGE SCAN
 - ◆ BITMAP INDEX FULL SCAN
 - ◆ BITMAP AND
 - ◆ BITMAP OR
 - ◆ BITMAP NOT
 - ◆ BITMAP CONVERSION COUNT
 - ◆ BITMAP CONVERSION TO ROWIDs

Bitmap Index Access Paths

- BITMAP INDEX SINGLE VALUE
 - ◆ Used to satisfy equality predicate
- BITMAP INDEX RANGE SCAN
 - ◆ Used to satisfy range operations such as BETWEEN
 - ◆ Unlike range scans on B*-Tree, is very efficient even for very large ranges
- BITMAP INDEX FULL SCAN
 - ◆ Used to satisfy NOT predicate
 - ◆ Scan of entire index to identify rows NOT matching
- BITMAP INDEX AND, OR, NOT
 - ◆ Used for bitwise combinations of multiple bitmap indexes

Bitmap Conversions

- **BITMAP CONVERSION COUNT**

- ◆ Used to evaluate COUNT(*) operation for queries whose where clause predicates only specify columns having bitmap indexes
- ◆ Very fast, very efficient

- **BITMAP CONVERSION TO ROWIDS**

- ◆ Used in cases where row source produced by bitmap index operations needs to be joined to other row sources, i.e., join to another table, group by operation, to satisfy a TABLE ACCESS BY ROWID operation
- ◆ More resource intensive than BITMAP CONVERSION COUNT
- ◆ Can be quite expensive if number of ROWIDs is large

Other Miscellaneous Access Paths

- UNION, UNION-ALL, MINUS, INTERSECTION

- ◆ Directly correspond to the SQL set operators
- ◆ UNION-ALL is cheapest, since no SORT(UNIQUE) is required

- TABLE FULL SCAN

- ◆ Reads all blocks allocated to table
- ◆ Can be most efficient access path for “small” tables
- ◆ Can cause significant physical I/O, particularly on larger tables
 - Consider ALTER TABLE table_name CACHE or putting table into KEEP buffer pool

Other Miscellaneous Access Paths

- TABLE ACCESS BY ROWID

- ◆ Generally used in conjunction with an index access path, where rowid has been identified, but Oracle needs access to a column not in the index
- ◆ Consider whether adding a column to an existing index will provide substantial benefit
- ◆ Cost is directly proportional to number of rowid lookups that are required

- TABLE (HASH)

- ◆ More efficient than index access
- ◆ Requires creation of hash cluster, more administrative overhead

Join Methods

- Nested Loops

- ◆ Generally geared towards FIRST_ROWS access
- ◆ Ideal for B*-Tree index driven access, small row sources
- ◆ When this is the case, always best for first row response time
- ◆ Can get very costly very quickly if no index path exists or index path is inefficient

- Sort Merge

- ◆ Generally geared towards ALL_ROWS access
- ◆ Can be useful for joining small to medium size row sources, particularly if viable index path is not available or if cartesian join is desired
- ◆ Be wary of sort_area_size, if it's too small, sorts will write to disk, performance will plummet

- Hash

- ◆ Most controversial (and misunderstood) join method
- ◆ Can be very powerful, when applied correctly
- ◆ Useful for joining small to medium sized to a large row source
- ◆ Can be sensitive to instance parameters such as hash_area_size, hash_multiblock_io_count, db_block_size

Optimize Joins

- Pick the best join method

- ◆ Nested loops joins are best for indexed joins of subsets
- ◆ Hash joins are usually the best choice for “big” joins
- ◆ Hash Join can only be used with equality
- ◆ Merge joins work on inequality
- ◆ If all index columns are in the where clause a merge join will be faster

- Pick the best join order

- ◆ Pick the best “driving” table
- ◆ Eliminate rows as early as possible in the join order

- Optimize “special” joins when appropriate

- ◆ STAR joins for data-warehousing applications
- ◆ STAR_TRANSFORMATION if you have bitmap indexes
- ◆ ANTI-JOIN methods for NOT IN sub-queries
- ◆ SEMI-JOIN methods for EXISTS sub-queries

Using ORACLE Optimization Modes

- When will the RBO be used?
 - ◆ `OPTIMIZER_MODE = RULE`
 - ◆ `=CHOOSE` & statistics are not present for all tables in SQL statement
 - ◆ Alter session has been issued
 - ◆ `RULE` hint is present
- When will the CBO be used?
 - ◆ `OPTIMIZER_MODE = CHOOSE`
 - ◆ `=CHOOSE` & statistics are not present for any tables in SQL statement
 - ◆ Alter session set optimizer_mode = (choose, first_rows or all_rows)
 - ◆ `CHOOSE`, `ALL_ROWS` or `FIRST_ROWS` hint is present

Tuning Tools

- A significant portion of SQL that performs poorly in production was originally crafted against empty or nearly empty tables
- Make sure you establish a reasonable sub-set of production data that is used during development and tuning of SQL
- In order to monitor execution plans and tune queries, Oracle 9i (and higher) provides the following three tools:
 - ◆ **Explain Plan** command
 - ◆ **TkProf** trace file formatter
 - ◆ The **SQLTrace** (or **AutoTrace**) facility
- These tools, mainly, allow the user to the **verify which access paths are considered by an execution plan**
 - ◆ Some of them provide, also, information about the number of buffers used, physical reads from buffers, rows returned from each step, etc
- Effective SQL tuning requires either familiarity with these tools or the use of commercial alternatives such as **SQLAB**

Explain Plan

- The **EXPLAIN PLAN** reveals the execution plan for an SQL statement
 - ◆ The execution plan reveals the exact sequence of steps that the Oracle optimizer has chosen to employ to process the SQL
- The execution plan is stored in an Oracle table called the **PLAN_TABLE**
 - ◆ Suitably formatted queries can be used to extract the execution plan from the **PLAN_TABLE**
 - ◆ **Create PLAN_TABLE command:**
`@$ORACLE_HOME/rdbms/admin/utlxplan.sql`
 - ◆ **Issue explain plan command:**
`Explain plan set statement_id = 'MJB' for
select * from dual;`
 - ◆ **Issue query to retrieve execution plan:**
`@$ORACLE_HOME/rdbms/admin/utlxpls.sql`
- The more heavily indented an access path is, the earlier it is executed
 - ◆ If two steps are indented at the same level, the uppermost statement is executed first
 - ◆ Some access paths are “joined” – such as an index access that is followed by a table lookup

Plan_Table

```
create table PLAN_TABLE
(
  statement_id varchar2(30),
  timestamp    date,
  remarks      varchar2(80),
  operation    varchar2(30),
  options      varchar2(30),
  object_node  varchar2(128),
  object_owner varchar2(30),
  object_name  varchar2(30),
  object_instance      numeric,
  object_type  varchar2(30),
  optimizer    varchar2(255),
  search_columns
  id
  parent_id
  position
  cost
  cardinality
  bytes
  other_tag
  partition_start
  partition_stop
  partition_id
  other
  distribution
  number,
  numeric,
  numeric,
  numeric,
  numeric,
  numeric,
  numeric,
  varchar2(255),
  varchar2(255),
  varchar2(255),
  numeric,
  long,
  varchar2(30)
);
```

Explain Plan

- **Sample Query:** Explain plan set statement_id = 'MJB' for

```
select doc_title  
from documents doc, page_collections pc  
where pc.pc_issue_date = '01-JAN-2002'  
and pc.pc_id = doc.pc_id;
```

- **Sample Explain plan output**

Operation	Object_Name	Rows	Bytes	Cardinality	Pstart	Pstop
SELECT STATEMENT		61K	3M	328		
NESTED LOOPS		61K	3M	328		
TABLE ACCESS BY INDEX ROWID	PAGE_COLL	834	9K	78		
INDEX RANGE SCAN	PC_PC2_UK	834		6		
INDEX RANGE SCAN	DOC_DOC2_	86M	4G	3		

Viewing the Execution Plan of a Query in Oracle

```
SQL> EXPLAIN PLAN
  2 SET STATEMENT_ID = 'PB'
  3 FOR SELECT b.branchNo, b.city, propertyNo
  4 FROM Branch b, PropertyForRent p
  5 WHERE b.branchNo = p.branchNo
  6 ORDER BY b.city;

Explained.

SQL> SELECT ID||' '||PARENT_ID||' '||LPAD(' ', 2*(LEVEL - 1))||OPERATION||' '||OPTIONS||
  2 ' '||OBJECT_NAME "Query Plan"
  3 FROM Plan_Table
  4 START WITH ID = 0 AND STATEMENT_ID = 'PB'
  5 CONNECT BY PRIOR ID = PARENT_ID AND STATEMENT_ID = 'PB';
```

Query Plan

```
-----
0  SELECT STATEMENT
1 0  SORT ORDER BY
2 1  NESTED LOOPS
3 2  TABLE ACCESS FULL PROPERTYFORRENT
4 2  TABLE ACCESS BY INDEX ROWID BRANCH
5 4  INDEX UNIQUE SCAN SYS_C007455

6 rows selected.
```

Plan_Table – an SQL Table

Statement_id – plan identifier

Id – a number assigned to each step

Parent_id – id of next step which operates on output of this step

Operation – eg internal operation select, insert etc

Options – name of internal operation

A More Complex EXPLAIN PLAN

```
Oracle SQL*Plus
File Edit Search Options Help
QUERY_PLAN
-----
SELECT STATEMENT
  SORT AGGREGATE
    TABLE ACCESS FULL DEPT
      NESTED LOOPS
        NESTED LOOPS
          NESTED LOOPS
            TABLE ACCESS FULL SODA_SHIPMENT_DETAIL
            TABLE ACCESS BY INDEX ROWID SODA
              INDEX UNIQUE SCAN SODA_PK
            TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
              INDEX UNIQUE SCAN SHIPMENT_PK
          TABLE ACCESS BY INDEX ROWID BEVERAGE_DISTRIBUTOR
            INDEX UNIQUE SCAN DISTRIBUTOR_ID_UK
        TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
          INDEX UNIQUE SCAN SHIPMENT_PK
      NESTED LOOPS
        NESTED LOOPS
          NESTED LOOPS
            TABLE ACCESS FULL SODA_SHIPMENT_DETAIL
            TABLE ACCESS BY INDEX ROWID SODA
              INDEX UNIQUE SCAN SODA_PK
            TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
              INDEX UNIQUE SCAN SHIPMENT_PK
          TABLE ACCESS BY INDEX ROWID BEVERAGE_DISTRIBUTOR
            INDEX UNIQUE SCAN DISTRIBUTOR_ID_UK
        TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
          INDEX UNIQUE SCAN SHIPMENT_PK
  SORT ORDER BY
    TABLE ACCESS FULL DEPT
      NESTED LOOPS
        NESTED LOOPS
          NESTED LOOPS
            TABLE ACCESS FULL SODA_SHIPMENT_DETAIL
            TABLE ACCESS BY INDEX ROWID SODA
              INDEX UNIQUE SCAN SODA_PK
            TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
              INDEX UNIQUE SCAN SHIPMENT_PK
          TABLE ACCESS BY INDEX ROWID BEVERAGE_DISTRIBUTOR
            INDEX UNIQUE SCAN DISTRIBUTOR_ID_UK
        TABLE ACCESS BY INDEX ROWID SODA_SHIPMENTS
          INDEX UNIQUE SCAN SHIPMENT_PK
      NESTED LOOPS
        NESTED LOOPS
          NESTED LOOPS
```

TkProf

- More details provided than **Autotrace** or **Explain Plan**
- For more useful information:

```
alter session set timed_statistics = true;
```
- To enable tracing:

```
alter session set sql_trace = true;
```
- Trace file written to *user_dump_dest*
- Use:

```
tkprof <trace_file> <output_file>
```


TkProf Sample Output

count = number of times OCI procedure was executed
cpu = cpu time in seconds executing
elapsed = elapsed time in seconds executing
disk = number of physical reads of buffers from disk
query = number of buffers gotten for consistent read
current = number of buffers gotten in current mode (usually for update)
rows = number of rows processed by the fetch or execute call

<some text deleted>

```
select doc_title
  from documents doc,
       page_collections pc
 where pc.pc_issue_date = '01-JAN-2002'
       and pc.pc_id = doc.pc_id
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1415	0.07	0.09	0	1853	0	21206
total	1417	0.07	0.10	0	1853	0	21206

TkProf Sample Output

Rows Row Source Operation

```
-----  
21206 NESTED LOOPS  
      31 TABLE ACCESS BY INDEX ROWID PAGE_COLLECTIONS  
      31 INDEX RANGE SCAN (object id 22993)  
21206 INDEX RANGE SCAN (object id 22873)
```

Rows Execution Plan

```
-----  
      0 SELECT STATEMENT    GOAL: CHOOSE  
21206 NESTED LOOPS  
      31 TABLE ACCESS    GOAL: ANALYZED (BY INDEX ROWID) OF 'PAGE_COLLECTIONS'  
      31 INDEX            GOAL: ANALYZED (RANGE SCAN) OF 'PC_PC2_UK' (UNIQUE)  
21206 INDEX           GOAL: ANALYZED (RANGE SCAN) OF 'DOC_DOC2_UK' (UNIQUE)
```

SQL_TRACE and tkprof

- ALTER SESSION SET SQL_TRACE TRUE causes a trace of SQL execution to be generated
- The TKPROF utility formats the resulting output
- Tkprof output contains breakdown of execution statistics execution plan and rows returned for each step
 - ◆ These stats are not available from any other source
- Tkprof is the most powerful tool, but requires a significant learning curve

Tkprof output

TKPROF - Tuning Scenario

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	57	0.22	0.22	0	28	6	849
total	59	0.23	0.23	0	28	6	849

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 5 (SYSTEM)

Rows Execution Plan

0	SELECT STATEMENT GOAL: CHOOSE
849	MERGE JOIN
5113	SORT (JOIN)
5113	TABLE ACCESS (FULL) OF 'TKP_EXAMPLE2'

Using SQLab

- Because EXPLAIN PLAN and tkprof are unwieldy and hard to interpret, third party tools that automate the process and provide expert advice improve SQL tuning efficiency
- The Quest SQLab product:
 - ◆ Identifies SQL your database that could benefit from tuning
 - ◆ Provides a sophisticated tuning environment to examine, compare and evaluate execution plans
 - ◆ Incorporates an expert system to advise on indexing and SQL statement changes
- Features
 - ◆ Display execution plan in a variety of intuitive ways
 - ◆ Provide easy access to statistics and other useful data
 - ◆ Model changes to SQL and immediately see the results

SQLab SQL tuning lab

SQLab Vision - [Tuning Session (FROM QUEST_SPC_DEMO_USER) [connected to MAUI.WORLD as QUEST(14)]]

File Edit Action Hints Data Tools Window Help

MAUI.WORLD as QUEST(14)

Statement View Comparison View

Original SQL

SQL Advise AutoTune Execute Explain

User: QUEST MAUI

DA SHIPMENT_DETAIL Not Analyzed

Rows N/A

Last DDL Date 30-Sep-2003 10:41:17

Type Normal

Average Row N/A

Index Name Column Names Unique

Column Name Index Participation Distinct

Shipment_Id None

Soda_Id None

Quantity None

SELECT STATEMENT CHOOSE

14 SORT ORDER BY

13 NESTED LOOPS

10 NESTED LOOPS

7 NESTED LOOPS

4 NESTED LOOPS

1 TABLE ACCESS FULL
QUEST_SPC_DEMO_USER.SODA_SHIPMENT_DETAIL in
QUEST_SPC_DEMO_TS_01 tablespace

3 TABLE ACCESS BY INDEX ROWID
QUEST_SPC_DEMO_USER.SODA in QUEST_SPC_DEMO_TS_01
tablespace

2 UNIQUE INDEX UNIQUE SCAN
QUEST_SPC_DEMO_USER.SODA_PK

6 TABLE ACCESS BY INDEX ROWID
QUEST_SPC_DEMO_USER.SODA_SHIPMENTS in
QUEST_SPC_DEMO_TS_01 tablespace

1 Every row in the table SODA_SHIPMENT_DETAIL is read.

2 Rows were retrieved using the unique index SODA_PK.

3 Rows from table SODA were accessed using rowid got from an index.

4 For each row retrieved by step 1, the operation in step 3 was performed to find a matching row.

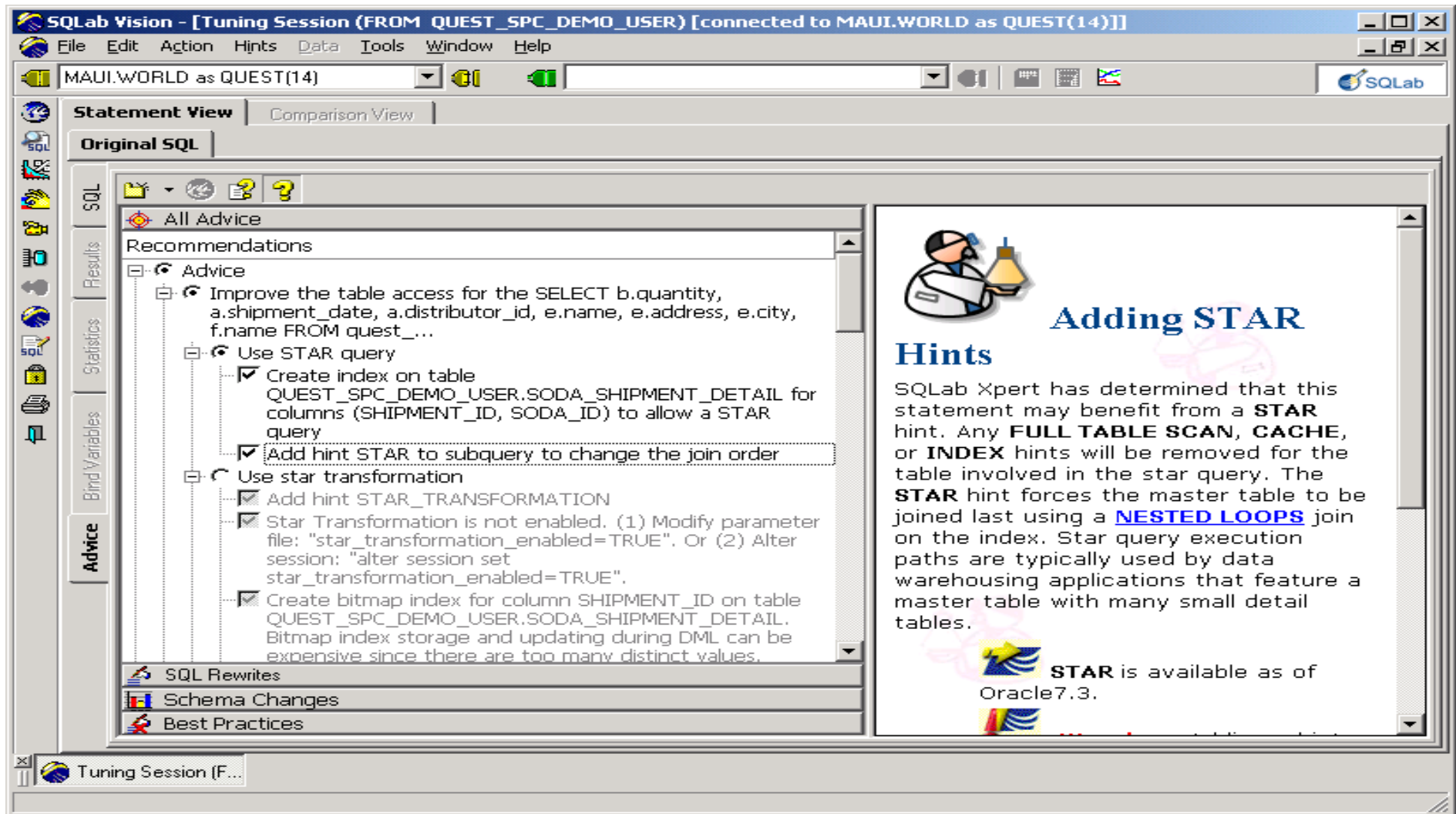
5 Rows were retrieved using the unique index SHIPMENT_PK.

6 Rows from table SODA_SHIPMENTS were accessed using rowid not from an index.

Tuning Session (F...

SQLab Expert Advice

- SQLab provides specific advice on how to tune an SQL statement



The screenshot displays the SQLab Vision application window, titled "SQLab Vision - [Tuning Session (FROM QUEST_SPC_DEMO_USER) [connected to MAUI.WORLD as QUEST(14)]]". The interface includes a menu bar (File, Edit, Action, Hints, Data, Tools, Window, Help) and a toolbar. The main window is divided into several panes:

- Statement View:** Shows the "Original SQL" and a list of "All Advice" recommendations.
- Advice Pane:** Contains a tree view of recommendations. The selected recommendation is: "Improve the table access for the SELECT b.quantity, a.shipment_date, a.distributor_id, e.name, e.address, e.city, f.name FROM quest_...". Under this, there are three sub-recommendations:
 - ☒ Use STAR query
 - ☒ Create index on table QUEST_SPC_DEMO_USER.SODA_SHIPMENT_DETAIL for columns (SHIPMENT_ID, SODA_ID) to allow a STAR query
 - ☒ Add hint STAR to subquery to change the join order
 - ☒ Use star transformation
 - ☒ Add hint STAR_TRANSFORMATION
 - ☒ Star Transformation is not enabled. (1) Modify parameter file: "star_transformation_enabled=TRUE". Or (2) Alter session: "alter session set star_transformation_enabled=TRUE".
 - ☒ Create bitmap index for column SHIPMENT_ID on table QUEST_SPC_DEMO_USER.SODA_SHIPMENT_DETAIL. Bitmap index storage and updating during DML can be expensive since there are too many distinct values.
- SQL Rewrites:** A pane for viewing alternative SQL statements.
- Schema Changes:** A pane for viewing schema modifications.
- Best Practices:** A pane for viewing best practice recommendations.

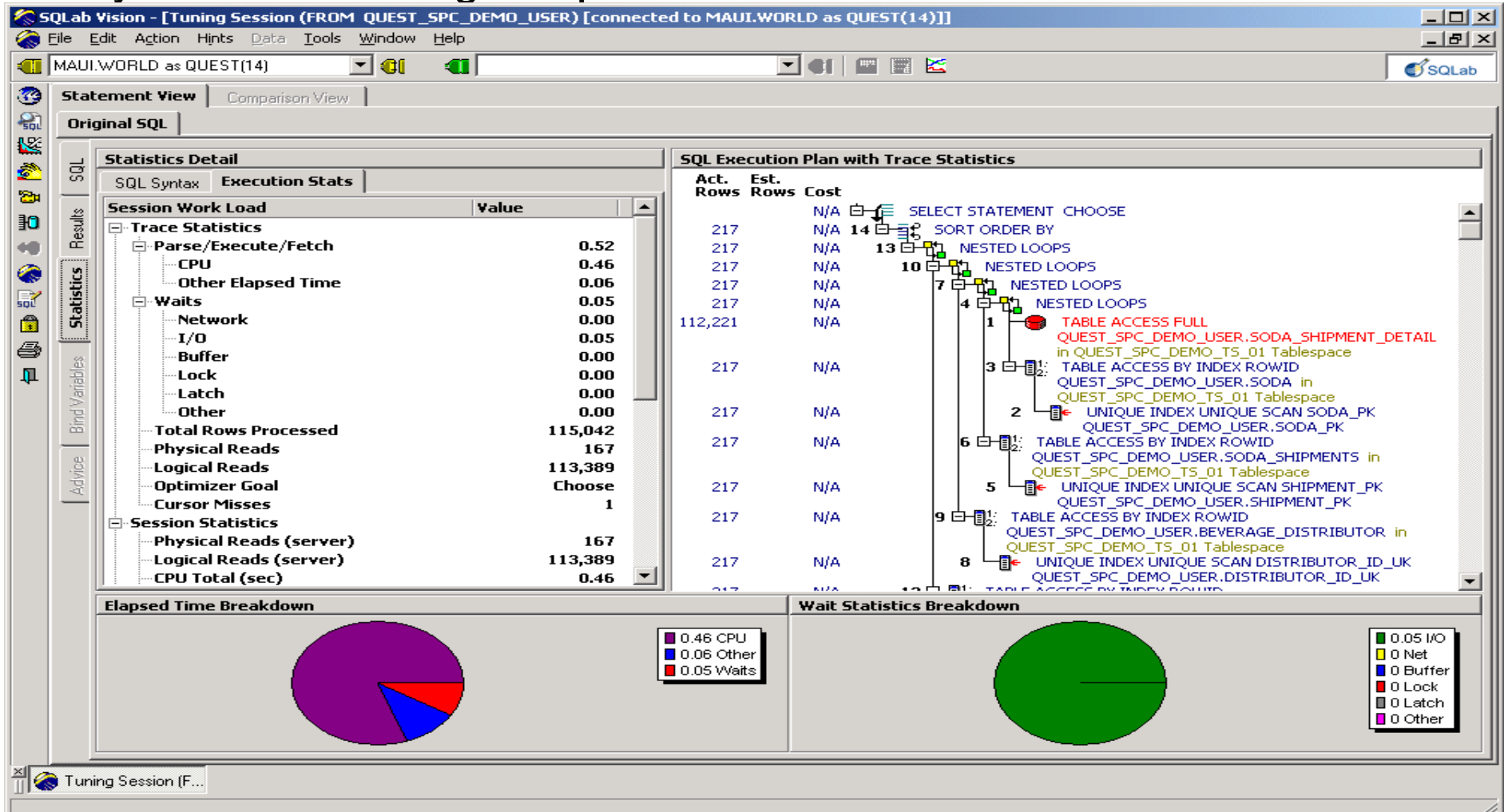
On the right side of the interface, there is a large panel titled "Adding STAR Hints" with an illustration of a scientist. The text in this panel reads:

SQLab Xpert has determined that this statement may benefit from a **STAR** hint. Any **FULL TABLE SCAN**, **CACHE**, or **INDEX** hints will be removed for the table involved in the star query. The **STAR** hint forces the master table to be joined last using a **NESTED LOOPS** join on the index. Star query execution paths are typically used by data warehousing applications that feature a master table with many small detail tables.

Below this text, it states: "STAR is available as of Oracle7.3."

SQLab SQL trace integration

- SQLab can also retrieve the execution statistics that are otherwise only available through tkprof



Choosing a Driving Table

- The **driving table** is the table that is first used by Oracle in processing the query
 - ◆ Choosing the correct driving table is critical
- Driving table should be **the table that returns the smallest number of rows** and do the smallest number of buffer gets
 - ◆ Driving table should not necessarily be the table with the smallest number of rows
- In the case of cost-based optimization, the driving table is first after the FROM clause. Thus, **place smallest table first after FROM, and list tables from smallest to largest**
 - ◆ The table order still makes a difference in execution time, even when using the cost-based optimizer

Choosing a Driving Table

- Example:

```
select doc_title
from documents doc, page_collections pc
where pc.pc_issue_date = '01-JAN-2002'
      and pc.pc_id = doc.pc_id
```

- Which table should be driving?

- ◆ DOCUMENTS has 110+ million rows

- No filtering predicates in where clause, all rows will be in row source

- ◆ PAGE_COLLECTIONS has 1.4+ million rows

- PC_ISSUE_DATE predicate will filter down to 30 rows

Using Hints

- Hints are used to convey your tuning suggestions to the optimizer
 - ◆ Misspelled or malformed hints are quietly ignored
- Commonly used hints include:
 - ◆ `ORDERED`
 - ◆ `INDEX(table_alias index_name)`
 - ◆ `FULL(table_alias)`
 - ◆ `INDEX_FFS(table_alias index_name)`
 - ◆ `INDEX_COMBINE(table_alias index_name1 .. index_name_n)`
 - ◆ `And_EQUAL(table_alias index_name1 index_name2 .. Index_name5)`
 - ◆ `USE_NL(table_alias)`
 - ◆ `USE_MERGE(table_alias)`
 - ◆ `USE_HASH(table_alias)`
- Hints should be specified as: `/*+ hint */`
 - ◆ Hints should immediately follow the 'SELECT' keyword
 - ◆ The space following the '+' can be significant inside of PL/SQL, due to bug in Oracle parser (see bug #697121)
- Driving table will never have a join method hint, since there is no row source to join it to

Simple Example of Tuning with Hints

- Initial SQL

```
select doc_id, doc_title, pc_issue_date
from documents doc, page_collections pc
where doc.pc_id = pc.pc_id and doc.doc_id = 9572422;
```

- Initial Execution Plan

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2575 Card=50 Bytes=3600)
1      0      MERGE JOIN (Cost=2575 Card=50 Bytes=3600)
2      1      TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_COLLECTIONS' (Cost=2571
                                Card=1442348)
3      2      INDEX (FULL SCAN) OF 'PC_PK' (UNIQUE) (Cost=3084 Card=1442348)
4      1      SORT (JOIN) (Cost=3 Card=1 Bytes=60)
5      4      TABLE ACCESS (BY INDEX ROWID) OF 'DOCUMENTS' (Cost=1 Card=1
                                Bytes=60)
6      5      INDEX (UNIQUE SCAN) OF 'DOC_PK' (UNIQUE) (Cost=2 Card=1)
```

- Initial number of buffer gets: 444

Simple Example of Tuning with Hints

- First Tuning attempt

```
select /*+ FULL(pc)*/ doc_id, doc_title, pc_issue_date
from documents doc, page_collections pc
where doc.pc_id = pc.pc_id and doc.doc_id = 9572422;
```

- Tuned Execution Plan

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3281 Card=50 Bytes=3600)
1    0      HASH JOIN (Cost=3281 Card=50 Bytes=3600)
2    1        TABLE ACCESS (FULL) OF 'PAGE_COLLECTIONS' (Cost=1675 Card=1442348
                                           Bytes=17308176)
3    1        TABLE ACCESS (BY INDEX ROWID) OF 'DOCUMENTS' (Cost=1 Card=1
                                           Bytes=60)
4    3          INDEX (UNIQUE SCAN) OF 'DOC_PK' (UNIQUE) (Cost=2 Card=1)
```

- Number of buffer gets: 364

Simple Example of Tuning with Hints

- Second Tuning attempt

```
select /*+ ORDERED USE_NL(pc)*/ doc_id, doc_title,  
pc_issue_date  
from documents doc, page_collections pc  
where doc.pc_id = pc.pc_id and doc.doc_id = 9572422;
```

- Second Tuned Execution Plan

Execution Plan

```
-----  
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=50 Bytes=3600)  
1      0      NESTED LOOPS (Cost=2 Card=50 Bytes=3600)  
2      1      TABLE ACCESS (BY INDEX ROWID) OF 'DOCUMENTS' (Cost=1 Card=1  
              Bytes=60)  
3      2      INDEX (UNIQUE SCAN) OF 'DOC_PK' (UNIQUE) (Cost=2 Card=2)  
4      1      TABLE ACCESS (BY INDEX ROWID) OF 'PAGE_COLLECTIONS' (Cost=1  
              Card=1442348)  
5      4      INDEX (UNIQUE SCAN) OF 'PC_PK' (UNIQUE) (Cost=1 Card=1442348)
```

- Number of buffer gets: 7

Considerations and Cautions

- Fundamental changes to the query structure allow the optimizer different options
- Using select in the select list allowed for a GROUP BY result without a GROUP BY operation, thus avoiding costly BITMAP CONVERSION TO ROWIDS
- Other places where re-writing query can have benefits:
 - ◆ Rewrite sub-select as join, allows optimizer more options
 - ◆ consider EXISTS/NOT EXISTS and IN/NOT IN operations
- Adding hints to a large number of your SQL statements?
 - ◆ Take a step back, consider whether you need to tune your CBO params
 - ◆ Hand tuning a majority of SQL in an application will complicate code, and add a lot of time to development effort
 - ◆ As new access paths are introduced in Oracle, statements that use hints will not utilize them, and continue using the old access paths
- When individual statement tuning is necessary, a solid understanding of access paths, join order and join methods is the key to success

Considerations and Cautions

- Use hints sparingly
 - ◆ If you have the opportunity, tune via CBO parameters first
 - ◆ Don't over-specify hints
 - ◆ SQL Tuning is as important as ever:
 - Need to understand the access paths, join orders, and join methods, even if only to evaluate what the CBO is doing
 - CBO gets better with each release, but it will never know as much about the application and data model as a well-trained developer

Myths

- SQL tuned for RBO will run well in the CBO
- SQL developers do not need to be retrained to write SQL for the CBO
- 8i and 9i do not support the RBO
- You can't run RULE and COST together
- Oracle says the CBO is unreliable and you should use RULE
- Hints can't be used in RULE

Top 9 Oracle SQL Tuning Tips

1. Design and develop with performance in mind
2. Index wisely
3. Reduce parsing
4. Take advantage of Cost Based Optimizer
5. Avoid accidental table scans
6. Optimize necessary table scans
7. Optimize joins
8. Use array processing
9. Consider PL/SQL for “tricky” SQL

Design and Develop with Performance in Mind

- Explicitly identify performance targets
- Focus on critical transactions
 - ◆ Test the SQL for these transactions against simulations of production data
- Measure performance as early as possible
- Consider prototyping critical portions of the applications
- Consider de-normalization and other performance by design features early on

De-Normalization

- If normalizing your OLTP database forces you to create queries with many multiple joins (4 or more)
- De-normalization is the process of selectively taking normalized tables and re-combining the data in them in order to reduce the number of joins needed them to produce the necessary query results
- Sometimes the addition of a single column of redundant data to a table from another table can reduce a 4-way join into a 2-way join, significantly boosting performance by reducing the time it takes to perform the join

De-Normalization

- Example: We have the following schema:

Similarities:

user1	user2	similarity
-------	-------	------------

Averages:

user	average
------	---------

- **Similarities** table contains the similarity measure for all the possible pairs of users and **Averages** table the average ratings of all users in Database
- In order to update all similarity measures we need the average value for each user
- Suppose we have over 1.000.000 users stored in our Database (about 500 billions of user-pairs!)
- To avoid joining we should consider of the following schema:

Similarities:

user1	user2	similarity	average1	average2
-------	-------	------------	----------	----------

De-Normalization

- While de-normalization can boost join performance, it can also have negative effects. For example, by adding redundant data to tables, you risk the following problems:
 - ◆ More data means reading more data pages than otherwise needed, hurting performance
 - ◆ Redundant data can lead to data anomalies and bad data
 - ◆ In many cases, extra code will have to be written to keep redundant data in separate tables in synch, which adds to database overhead
 - ◆ As you consider whether to de-normalize a database to speed joins, be sure you first consider if you have the proper indexes on the tables to be joined
 - It is possible that your join performance problem is more of a problem with a lack of appropriate indexes than it is of joining too many tables

Index Wisely

- Index to support selective WHERE clauses and join conditions
- Use concatenated indexes where appropriate
- Consider over-indexing to avoid table lookups
- Consider advanced indexing options
 - ◆ Hash Clusters
 - When a table is queried frequently with equality queries
 - You can avoid using the ORDER BY clause, as well as sort operations
 - More administrative overhead
 - ◆ Bit mapped indexes
 - Can use large amounts of memory
 - Use sparingly
 - ◆ Index only tables

Index Wisely

- Do not index columns that are modified frequently
 - ◆ UPDATE statements that modify indexed columns and INSERT and DELETE statements that modify indexed tables take longer than if there were no index
 - must modify data in indexes as well as data in tables
- Do not index keys that appear only with functions or operators
 - ◆ A WHERE clause that uses a function (other than MIN or MAX) or an operator with an indexed key does not make available the access path that uses the index (except with function-based indexes)
- When choosing to index a key, consider whether the performance gain for queries is worth the performance loss for INSERTs, UPDATEs, and DELETEs and the use of the space required to store the index
 - ◆ You might want to experiment by comparing the processing times of the SQL statements with and without indexes
 - You can measure processing time with the SQL trace facility

Reduce Parsing

- Use Bind variables
 - ◆ Bind variables are key to application scalability
 - ◆ If necessary set cursor `CURSOR_SHARING` to `FORCE`
- Reuse cursors in your application code
 - ◆ How to do this depends on your development languages
- Use a cursor cache
 - ◆ Setting `SESSION_CACHED_CURSORS` can help applications that are not re-using cursors

Bind Values

- Use bind variables rather than literals in SQL statements whenever possible
- For example, the following two statements cannot use the same shared area because they do not match character for character:

```
SELECT employee_id FROM employees  
WHERE department_id = 10;  
SELECT employee_id FROM employees  
WHERE department_id = 20;
```

- By replacing the literals with a bind variable, only one SQL statement would result, which could be executed twice:

```
SELECT employee_id FROM employees  
WHERE department_id = :dept_id;
```

Bind Values

- In SQL*Plus you can use bind variables as follows:

```
SQL> variable dept_id number  
SQL> exec :dept_id := 10  
SQL> SELECT employee_id FROM employees  
        WHERE department_id = :dept_id;
```

- What we've done to the SELECT statement now is take the literal value out of it, and replace it with a placeholder (our bind variable), with SQL*Plus passing the value of the bind variable to Oracle when the statement is processed.

Cursors

Instead of:

```
select count(*) into tot from s_emp  
where emp_id = v_emp_id;
```

Declare a cursor for the count:

```
cursor cnt_emp_cur(v_emp_id number) is  
select count(*) emp_total from s_emp  
emp_id = v_emp_id;  
cnt_emp_rec cnt_emp%rowtype;
```

Or if just checking for existence

```
cursor cnt_emp_cur(v_emp_id  
number) is  
select emp_id from s_emp where  
where  
emp_id= v_emp_id and rownum = 1;
```

And then do the fetch from this cursor:

```
...  
open cnt_emp(v_emp_id);  
fetch cnt_emp into cnt_emp_rec;  
...  
close cnt_emp;
```

Take Advantage of the Cost Based Optimizer

- The older rule based optimizer is inferior in almost every respect to the modern cost based optimizer; basic RBO problems
 - ◆ Incorrect driving table 40%
 - ◆ Incorrect index 40%
 - ◆ Incorrect driving index 10%
- Using the **cost based optimizer effectively** involves:
 - ◆ Regular collection of table statistics using the ANALYZE or DBMS_STATS command
 - ◆ Understand hints and how they can be used to influence SQL statement execution
 - ◆ Choose the appropriate optimizer mode;
 - FIRST_ROWS is best for OLTP applications
 - ALL_ROWS suits reporting and OLAP jobs

Analyze – Wrong Data

- Tables were analyzed with incorrect data volumes
- When does this occur?
 - ◆ Table rebuilt
 - ◆ Index added
 - ◆ Migrate schema to production
 - ◆ Analyze before a bulk load
- Missing Stats:
 - ◆ Oracle will estimate the stats for you
 - ◆ These stats are for this execution only
 - ◆ Stats on Indexes

Avoid Accidental Table Scans

- Table scans that occur unintentionally are a major source of poorly performing SQL
- Causes include:
 - ◆ Missing Index
 - ◆ Using “!=“ , “<>” or NOT
 - Use inclusive range conditions or IN lists
 - ◆ Looking for values that are NULL
 - Use NOT NULL values with a default value
 - ◆ Using function on indexed columns

Factors that can Cause an Index not to be Used

1) Using a function on the left side

```
SELECT * FROM s_emp  
WHERE substr(title,1,3) = 'Man';  
SELECT * FROM s_emp  
WHERE  
trunc(hire_date)=trunc(sysdate);
```

→ Since there is a function around this column the index will not be used. This includes Oracle functions to_char, to_number, ltrim, rtrim, instr, trunc, rpad, lpad.

Solution:

Use 'like' :

```
SELECT * FROM s_emp  
WHERE title LIKE 'Man%';
```

Use >, < :

```
SELECT * FROM s_emp  
WHERE hire_date >= sysdate  
AND hire_date < sysdate + 1;
```


Factors that can Cause an Index not to be Used

2) Comparing incompatible data types

```
SELECT * FROM s_emp  
WHERE employee_number = '3';  
  
SELECT * FROM s_emp  
WHERE hire_date = '12-jan-01';
```

→ There will be an implicit
to_char conversion used

→ There will be an implicit
to_date conversion used

Solution:

```
SELECT * FROM s_emp  
WHERE employee_number = 3;  
  
SELECT * FROM s_emp  
WHERE hire_date = to_date('12-jan-01');
```

Factors that can Cause an Index not to be Used

3) Using null and not null

```
SELECT * FROM s_emp  
WHERE title IS NOT NULL;  
SELECT * FROM s_emp  
WHERE title IS NULL;
```

→ Since the column title has null values, and is compared to a null value, the index can not be used

Solution:

```
SELECT * FROM s_emp  
WHERE title >= ' ';  
Use an Oracle hint:  
SELECT /*+ index (s_emp) */ *  
FROM s_emp WHERE title IS NULL;
```

→ Oracle hints are always enclosed in /*+ */ and must come directly after the select clause
The index hint causes indexes to be used

Factors that can Cause an Index not to be Used

- 4) Adding additional criteria in the where clause for a column name that is of a different index

```
SELECT * FROM s_emp  
WHERE title= 'Manager'  
AND salary = 100000;
```

→ Column title and salary have separate indexes on these columns

Solution:

(Use an Oracle hint)

```
SELECT /*+ index (s_emp) */  
FROM s_emp  
WHERE title= 'Manager'  
AND salary = 100000;
```

→ Oracle hints are always enclosed in `/*+ */` and must come directly after the select clause
The index hint causes indexes to be used
S_EMP is the Oracle table

Make sure most Restrictive Indexes are being Used by using Oracle hints

```
SELECT COUNT(*) FROM vehicle
WHERE
assembly_location_code = 'AP24A'
AND production_date = '06-apr-01';
```

```
COUNT(*)
```

```
-----
```

```
787
```

```
Elapsed: 00:00:10.00
```

→ This does not use an index

→ Notice it is 10 seconds

Make sure most Restrictive Indexes are being Used by using Oracle hints

```
SELECT
/*+ index (vehicle FKI_VEHICLE_1)
*/ COUNT(*)
FROM vehicle
WHERE
assembly_location_code = 'AP24A'
AND production_date = '06-apr-01';

COUNT(*)
-----
          787

Elapsed: 00:00:00.88
```

→ This does use an index





→ Notice it is less than 1 second. USE THE MOST SELECTIVE INDEX that will return the fewest records

Some Idiosyncrasies

- Condition Order:
 - ◆ The order of your where clause will effect the performance
- OR may stop the index being used
 - ◆ break the query and use UNION

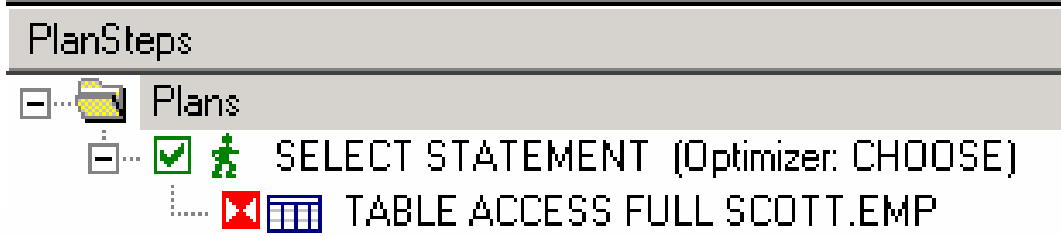
```
SELECT /*+ CHOOSE */
      *
FROM emp
WHERE  ename = 'smith'
      OR  deptno = 1
```

PlanSteps

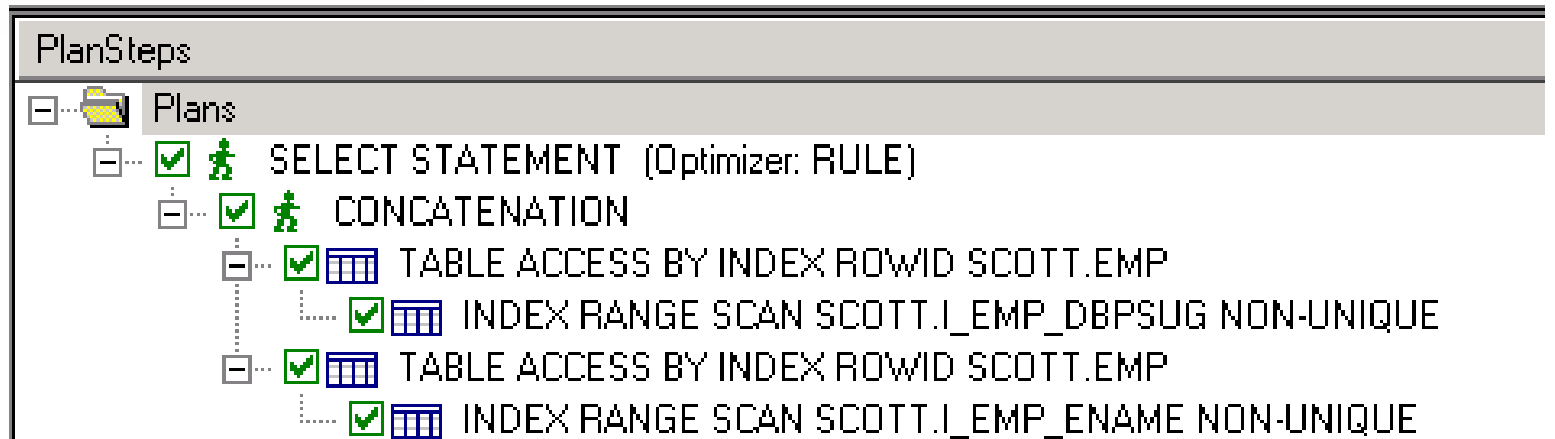
- [-] Plans
 - [-]   SELECT STATEMENT (Optimizer: CHOOSE)
 -   TABLE ACCESS FULL SCOTT.EMP

OR

```
SELECT /*+ INDEX(emp, I_EMP_DBPSUG) INDEX(emp, I_EMP_ENAME) */
      *
FROM emp
WHERE  ename = 'smith'
      OR  deptno = 1
```



```
SELECT *
FROM emp
WHERE  ename = 'smith'
      OR  deptno = 1
```



Optimize Necessary Table scans

- There are many occasions where a table scan is the only option, If so:
 - ◆ Consider parallel query option
- Try to reduce size of the table
 - ◆ Adjust PCTFREE and PCTUSED
 - ◆ Relocate infrequently used long columns
- Improve the caching of the table
 - ◆ Use the CACHE hint or table property
 - ◆ Implement KEEP and RECYCLE pools
- Partition the table
- Consider the fast full index scan

IN Lists

```
SELECT empno FROM emp WHERE  
deptno IN (10,20,30)
```

- Rewritten as:

```
SELECT empno FROM emp  
WHERE deptno = 10  
UNION ALL  
SELECT empno FROM emp  
WHERE deptno = 20  
UNION ALL  
SELECT empno FROM emp  
WHERE deptno = 30
```

Data Partitioning

- If you are designing a database that potentially could be very large, holding millions or billions of rows, consider the option of horizontally partitioning your large tables
 - ◆ Horizontal partitioning divides what would typically be a single table into multiple tables, creating many smaller tables instead of a single, large table
 - ◆ The advantage of this is that is generally is much faster to query a single small table than a single large table
- For example, if you expect to add 10 million rows a year to a transaction table, after five years it will contain 50 million rows
 - ◆ In most cases, you may find that most queries (although not all) queries on the table will be for data from a single year
 - ◆ If this is the case, if you partition the table into a separate table for each year of transactions, then you can significantly reduce the overhead of the most common of queries

When Joining...

- Make sure everything that can be joined is joined (for 3 or more tables)

Instead of:

```
SELECT * FROM t1, t2, t3
WHERE t1.emp_id = t2.emp_id
AND t2.emp_id = t3.emp_id
```

add:

```
SELECT * FROM t1, t2, t3
WHERE t1.emp_id = t2.emp_id
AND t2.emp_id = t3.emp_id
AND t1.emp_id = t3.temp_id;
```

- Make sure smaller table is first in the from clause

Joining too Many Tables

- The more tables the more work for the optimizer
- Best plan may not be achievable

Tables	Permutations
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880

Tables	Permutations
10	3628800
11	39916800
12	479001600
13	6226020800
14	87178291200
15	1307674368000

Use ARRAY Processing

- Retrieve or insert rows in batches, rather than one at a time
- Methods of doing this are language specific
- Suppose that a new user registers in our Database
- We have to create an entry in Similarities table for each pair of new user and the existing users
- Instead of selecting all the existing users and make the insertion individually we should use the following statement:

```
insert into
similarities (user1, user2, similarity)
'new_user_id' as user1,
select user_id from users as user2,
0 as similarity;
```

Consider PL/SQL for “Tricky” SQL

- With SQL you specify the data you want, not how to get it
 - ◆ Sometime you need to specifically dictate your retrieval algorithms
- For example:
 - ◆ Getting the second highest value
 - ◆ Correlated updates
 - ◆ SQL with multiple complex correlated subqueries
 - ◆ SQL that seems to hard to optimize unless it is broken into multiple queries linked in PL/SQL
- Using explicit instead of implicit cursors
 - ◆ Implicit cursors always take longer than explicit cursors because they are doing an extra to make sure that there is no more data
- Eliminating cursors where ever possible

When your SQL is Tuned, Look to your Oracle Configuration

- When SQL is inefficient there is limited benefit in investing in Oracle server or operating system tuning
- However, once SQL is tuned, the limiting factor for performance will be Oracle and operating system configuration
- In particular, check for internal Oracle contention that typically shows up as latch contention or unusual wait conditions (buffer busy, free buffer, etc)

Other Parameters

- OPTIMIZER_MAX_PERMUTATIONS

- ◆ Remember the too many joins?
- ◆ Default is 80,000
- ◆ Can lead to large Parse times
- ◆ Altering can lead to non optimal plan selection

- OPTIMIZER_INDEX_CACHING

- ◆ Represents # of blocks that can be found in the cache
- ◆ Range 0 - 99
- ◆ Default is 0 – implies that index access will require a *physical read*
- ◆ Should be set to 90

Other Parameters

- **OPTIMIZER_INDEX_COST_ADJ**
 - ◆ Represents cost of index access to full table scans
 - ◆ Range 1 – 10000
 - ◆ Default is 100 – Means index access is as costly as Full Table Scan
 - ◆ Should be between 10 – 50 for OLTP and approx 50 for DSS
- **DB_FILE_MULTIBLOCK_READ_COUNT**
 - ◆ Setting too high can cause Full Table Scans
 - ◆ Can adjust for this by setting OPTIMIZER_INDEX_COST_ADJ
- **DB_KEEP_CACHE_SIZE**
- **DB_RECYCLE_CACHE_SIZE**
- **DB_BLOCK_HASH_BUCKETS**

References

- Dennis Shasha and Phillipe Bonnet *Database Tuning : Principles Experiments and Troubleshooting Techniques*, Morgan Kaufmann Publishers 2002.
- Mark Levis *Common tuning pitfalls of Oracle's optimizers*, Compuware
- Duane Spencer *TOP tips for ORACLE SQL tuning*, Quest Software, Inc.