ORACLE SQL PERFORMANCE TUNING AND OPTIMIZATION

By Kevin Meade

Oracle SQL Performance Tuning and Optimization

km133688@sbcglobal.net

Contents

About the Author 19
How this Book is Different..... 21
Those who came before..... 23
Getting the FREE STUFF 25
Reviews on Amazon 27
WWW.ORAFAQ.COM 29

John Watson 29
Jim Irvine..... 29
Lalit Kumar 29
Mark Kilgour 29
Ross Leishman 30
Saša Dominković 30

Soaring with Eagles 31

Barry Ward..... 31
Dennis Deluzio..... 31

Dheeraj Madadi 31
Jack McGuirk 31
Nirav Kathrani 31
Robert Romanowski 31
Subrahmanyam Jannalagadda 32

Chapter 1: DRIVING TABLE and JOIN ORDER 33

The Four Parts of a Query 33
The Filtered Rows Percentage Method 34

First Look at a Query 35
Query Diagrams 36
Driving Table and Join Order 37
COUNT QUERIES and FILTER QUERIES 38
FRP Spreadsheet 39

A Filtered Rows Percentage Example 40

1. Format the PROBLEM QUERY 41
2. Familiarize yourself with the problem query 45

Get a Description 45
Look for Mistakes 45
Check for the Unusual 46

3. Create a spreadsheet 47

4. List tables from the query in the spreadsheet 47
5. Note the row count for each table 49
6. Build and run FILTER QUERIES for each table 50
7. Note the filtered row counts 52
8. Compute FILTERED ROWS PERCENTAGE for each table . 52
9. Determine PREFERRED JOIN ORDER 53
10. Construct a QUERY DIAGRAM 54
11. Determine INITIAL JOIN ORDER 55
12. Build and run RECONSTRUCTION QUERIES 57

ATT_EMP_ORG Reconstruction Query 58
CBE_EMP Reconstruction Query 58
V_CBE_LV_RQST Reconstruction Query 58
V_PLCY_DIM Reconstruction Query 59
V_LV_PLN_USGE_FACT Reconstruction Query 60
ATT_LV_TYP (LV_SEG_LV_TYP) Reconstruction Query 61
ATT_LV_PLN_TYP Reconstruction Query 62
ATT_LV_PLN Reconstruction Query 63

13. Note reconstruction row counts 65
14. Use CARDINALITY FEEDBACK to adjust join order 65
15. Repeat (11) thru (12) once if join order changed 65
16. Determine if further action is necessary 66

Summary #1 69
How to use FILTERED ROWS PERCENTAGE Method 69
Backtracking on CARDINALITY FEEDBACK 70

14. Use CARDINALITY FEEDBACK to adjust join order 70
15. Repeat (11) through (12) once if join order changed 71

The Short Cut (Brain over Brawn) 73

Scripts used in the Chapter 79
Chapter Summary 79

Chapter 2: Ways to Use a Query Execution Plan .....**Error! Bookmark not defined.**

EXPLAIN PLAN Conventions Used ..... **Error! Bookmark not defined.**
Ways to Use a QEP .....**Error! Bookmark not defined.**

Identifying where the CBO thinks it will be spending most of its time and resources .....**Error! Bookmark not defined.**
Locating mistakes in Cardinality Estimates. **Error! Bookmark not defined.**
Observing how Oracle has modified the Query..**Error! Bookmark not defined.**

QUERY REWRITE (Implicit Data Type Conversion).....**Error! Bookmark not defined.**
QUERY REWRITE (Meaningless Expression Removal and Predicate Ordering) .....**Error! Bookmark not defined.**
QUERY REWRITE (Imaginary Predicates) .....**Error! Bookmark not defined.**

Checking Predicate Efficiencies....**Error! Bookmark not defined.**

FILTER = WASTE: Evaluating Efficiency in Fetching Rows.....**Error! Bookmark not defined.**
FILTER = WASTE: Evaluating Efficiency in Joining Rows .....**Error! Bookmark not defined.**
FILTER = WASTE: Partial Use of a Concatenated Index.....**Error! Bookmark not defined.**

Comparing Estimates to Actuals... **Error! Bookmark not defined.**
Viewing text of REMOTE queries in distributed transactions **Error! Bookmark not defined.**
Learning about Database Features from the OUTLINE ..... **Error! Bookmark not defined.**

Chapter Summary..... **Error! Bookmark not defined.**

Chapter 3: The Best Indexes for a Query... **Error! Bookmark not defined.**

What a Simple Query has to Say..... **Error! Bookmark not defined.**

Poor Use of Data Types ..... **Error! Bookmark not defined.**
Effect of Functions on Indexing..... **Error! Bookmark not defined.**
WHERE Clause Organization..... **Error! Bookmark not defined.**

ACCESS vs. FILTER vs. COVERAGE... **Error! Bookmark not defined.**

ACCESS..... **Error! Bookmark not defined.**
FILTER (Pre-Table Filtering) ..... **Error! Bookmark not defined.**
COVERAGE..... **Error! Bookmark not defined.**

Example of BEST INDEX..... **Error! Bookmark not defined.**

Example of Code Yielding ACCESS..... **Error! Bookmark not defined.**
Example of Code Yielding FILTER**Error! Bookmark not defined.**

Quick Summary #1.....**Error! Bookmark not defined.**
Creating Indexes for Join Queries ..... **Error! Bookmark not defined.**

The Join Query.....**Error! Bookmark not defined.**
ACCESS and FILTER in a Join Query..... **Error! Bookmark not defined.**
Reversing the Join Order.....**Error! Bookmark not defined.**

Quick Summary #2.....**Error! Bookmark not defined.**
How Predicates Use Indexes**Error! Bookmark not defined.**

Inequality Predicates stop ACCESS and start FILTER ..... **Error! Bookmark not defined.**
DUNSEL COLUMNS in an Index stop ACCESS and start FILTER .....**Error! Bookmark not defined.**

COVERAGE....**Error! Bookmark not defined.**

Example of a COVERING INDEX. **Error! Bookmark not defined.**
Limitations of COVERAGE ..... **Error! Bookmark not defined.**

Quick Summary #3 ..... **Error! Bookmark not defined.**
Our versions of CREATE INDEX..... **Error! Bookmark not defined.**

New CREATE INDEX command Syntax Variations..... **Error! Bookmark not defined.**

A Larger Indexing Example in Action..... **Error! Bookmark not defined.**

Making Friends with the Query ..... **Error! Bookmark not defined.**
The Indexing Process..... **Error! Bookmark not defined.**

1) Determine the DRIVING TABLE and JOIN ORDER for the query..... **Error! Bookmark not defined.**

Indexing the Driving Table..... **Error! Bookmark not defined.**

2) Reduce the query down to only those elements related to the table..... **Error! Bookmark not defined.**
3) Select the appropriate variation of our own CREATE INDEX command syntax (DRIVING TABLE or INNER TABLE OF A JOIN) ..... **Error! Bookmark not defined.**
4) Walk the REDUCED QUERY for each rule in the selected CREATE INDEX command syntax and pop columns into the index as they satisfy a rule. ..... **Error! Bookmark not defined.**

Indexing an INNER TABLE of a join ..... **Error! Bookmark not defined.**

2) Reduce the query down to only those elements related to the table. ..... **Error! Bookmark not defined.**
3) Select the appropriate variation of our own CREATE INDEX command syntax (DRIVING TABLE or INNER TABLE OF A JOIN) ..... **Error! Bookmark not defined.**
4) Walk the REDUCED QUERY for each rule in the selected CREATE INDEX command syntax and pop columns into the index as they satisfy a rule. ..... **Error! Bookmark not defined.**

Rest of the Larger Example ..... **Error! Bookmark not defined.**

Index for table SNAPTSHOT_DATE (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table COVERAGE_DIM (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table CLAIM_STATUS_DIM (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table CLAIM_DETAIL_DIM (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table CLAIM_DISAB (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table COVERAGE_PLAN_CURRENT (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table BEN (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table CLAIMANT_DIM (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table DIAGNOSIS_DIM (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table RELATED_CLAIMS_FLAG_DIM (inner table in a join) ..... **Error! Bookmark not defined.**
Index for table CLAIM_BENEFIT_EFF (inner table in a join) ..... **Error! Bookmark not defined.**

Larger Example Summary ..... **Error! Bookmark not defined.**
A Closer Look at Filtering ..... **Error! Bookmark not defined.**

Query Plan Variation #1 (nothing) ..... **Error! Bookmark not defined.**
Query Plan Variation #2 (post-table filtering) ..... **Error! Bookmark not defined.**
Query Plan Variation #3 (filter) ..... **Error! Bookmark not defined.**
Query Plan Variation #4 (filter / post-table filtering) ..... **Error! Bookmark not defined.**
Query Plan Variation #5 (access) ..... **Error! Bookmark not defined.**
Query Plan Variation #6 (access / post-table filtering) ..... **Error! Bookmark not defined.**
Query Plan Variation #7 (access / filter) ..... **Error! Bookmark not defined.**
Query Plan Variation #8 (access / filter / post-table filtering) ..... **Error! Bookmark not defined.**

Quick Summary #4 ..... **Error! Bookmark not defined.**
Quick Summary #5 ..... **Error! Bookmark not defined.**

Complicating Issues..... **Error! Bookmark not defined.**

Poor Predicate Selectivity and Column Cardinality ..... **Error! Bookmark not defined.**

Scripts used in the Chapter .. **Error! Bookmark not defined.**
Chapter Summary..... **Error! Bookmark not defined.**

Chapter 4: JOINS..... **Error! Bookmark not defined.**

The Join Game **Error! Bookmark not defined.**
TYPES OF QUERIES .... **Error! Bookmark not defined.**

Example of PRECISION style..... **Error! Bookmark not defined.**
Example of WAREHOUSE style ... **Error! Bookmark not defined.**
Example of typical optimizations... **Error! Bookmark not defined.**

SORT MERGE JOIN..... **Error! Bookmark not defined.**

JOIN CHARACTERISTICS .. **Error! Bookmark not defined.**
THE 2% RULE **Error! Bookmark not defined.**
Using the 2% RULE ..... **Error! Bookmark not defined.**
NESTED LOOPS JOIN .. **Error! Bookmark not defined.**

Nested Loops Join Sweet Spot..... **Error! Bookmark not defined.**
BAD NESTED LOOPS JOIN: CARDINALITY ERROR ..... **Error! Bookmark not defined.**
BAD NESTED LOOPS JOIN: INDEX SELECTIVITY ..... **Error! Bookmark not defined.**

HASH JOIN ..... **Error! Bookmark not defined.**

Hash Join Sweet Spot ..... **Error! Bookmark not defined.**
Hash Join Tune-ability ..... **Error! Bookmark not defined.**
Two Most Common Hash Join Failures ..... **Error! Bookmark not defined.**

The Hash Join that Never Was..... **Error! Bookmark not defined.**
Hash Join across a Partial Join Key..... **Error! Bookmark not defined.**

Hash Join Summary #1. ..... **Error! Bookmark not defined.**
Hash Join Memory Management .. **Error! Bookmark not defined.**

INNER Table does not fit..... **Error! Bookmark not defined.**
Watching Work Areas Change..... **Error! Bookmark not defined.**

Making a Hash Join go faster .....**Error! Bookmark not defined.**

Exploit the DUNSEL JOIN feature and drop the join altogether ...**Error! Bookmark not defined.**
Reduce the amount of data hashed in memory .....**Error! Bookmark not defined.**
Increase memory allocated for hash joins.....**Error! Bookmark not defined.**
Use Partitioning and Parallel Query to turn big hash joins into lots of small hash joins .....**Error! Bookmark not defined.**

Complexity of Managing Work Area Size.... **Error! Bookmark not defined.**

Fiddle with parameters that affect automatic memory management.....**Error! Bookmark not defined.**
Go OLD SCHOOL and use manual memory management.....**Error! Bookmark not defined.**
Use partitioning to create lots of way smaller joins.....**Error! Bookmark not defined.**

Do you really need it.....**Error! Bookmark not defined.**

Scripts used in the Chapter ..**Error! Bookmark not defined.**
Chapter Summary**.....Error! Bookmark not defined.**

Chapter 5: HINTS**..... Error! Bookmark not defined.**

Hints are a Discovery Tool and we only need Three**..... Error! Bookmark not defined.**

CARDINALITY / OPT_ESTIMATE Hint..... **Error! Bookmark not defined.**

CARDINALITY Hint..... **Error! Bookmark not defined.**
OPT_ESTIMATE Hint..... **Error! Bookmark not defined.**
Comparing CARDINALITY / OPT_ESTIMATE..... **Error! Bookmark not defined.**
Exploiting CARDINALITY / OPT_ESTIMATE ..... **Error! Bookmark not defined.**

ORDERED / LEADING Hint..... **Error! Bookmark not defined.**

ORDERED Hint..... **Error! Bookmark not defined.**
LEADING Hint ..... **Error! Bookmark not defined.**
Exploiting ORDERED / LEADING ..... **Error! Bookmark not defined.**

NO_INDEX Hint..... **Error! Bookmark not defined.**

Exploiting NO_INDEX..... **Error! Bookmark not defined.**

Other Useful Hints ..... **Error! Bookmark not defined.**

NO_PARALLEL..... **Error! Bookmark not defined.**
OPTIMIZER_FEATURES_ENABLE..... **Error! Bookmark not defined.**
GATHER_PLAN_STATISTICS..... **Error! Bookmark not defined.**
DYNAMIC_SAMPLING..... **Error! Bookmark not defined.**

Scripts used in the Chapter.. **Error! Bookmark not defined.**
Chapter Summary ..... **Error! Bookmark not defined.**

Chapter 6: BASICS ..... **Error! Bookmark not defined.**

What are the Basics..... **Error! Bookmark not defined.**

Modeling Paradigm, 3rd Normal Form, SQL Workload (OLTP, DSS, ANALYTIC) ..... **Error! Bookmark not defined.**
The Value of Third Normal Form to SQL Performance ..... **Error! Bookmark not defined.**
Data Types (Date=Date, Number=Number, String=String) .. **Error! Bookmark not defined.**
NOT NULL..... **Error! Bookmark not defined.**
Constraints (Primary Key, Unique Key, Foreign Key, Check) ..... **Error! Bookmark not defined.**

DUNSEL Join Removal (aka. Join Elimination)..... **Error! Bookmark not defined.**
RELY Constraints..... **Error! Bookmark not defined.**
Constraints vs. Statistics..... **Error! Bookmark not defined.**
Constraints vs. Indexes ..... **Error! Bookmark not defined.**

INDEXES..... **Error! Bookmark not defined.**

Types of Indexes and Workload Scenarios ..... **Error! Bookmark not defined.**
Strategy ..... **Error! Bookmark not defined.**
BITMAP Indexes ..... **Error! Bookmark not defined.**
Summarizing Indexes ..... **Error! Bookmark not defined.**

Statistics..... **Error! Bookmark not defined.**

Fundamentals of Statistics ..... **Error! Bookmark not defined.**
Where statistics can go wrong..... **Error! Bookmark not defined.**
What 11gR2 offers ..... **Error! Bookmark not defined.**

Good Elementary SQL ..... **Error! Bookmark not defined.**

Modifying indexed columns..... **Error! Bookmark not defined.**
Specifying incomplete outer-joins..... **Error! Bookmark not defined.**

Use of NOT IN when faced with NULLS ..... **Error! Bookmark not defined.**
Using non-deterministic PL/SQL functions from SQL..... **Error! Bookmark not defined.**
Failure to use the WITH clause ..... **Error! Bookmark not defined.**

Scripts used in the Chapter.. **Error! Bookmark not defined.**
Chapter Summary ..... **Error! Bookmark not defined.**

Chapter 7: ROW COUNTS and RUN TIMES .. **Error! Bookmark not defined.**

The Meeting .... **Error! Bookmark not defined.**
Analyzing the Analysis ... **Error! Bookmark not defined.**

Dissecting PLAN_OUTPUT ..... **Error! Bookmark not defined.**
Three (3) Kinds of Row Counts..... **Error! Bookmark not defined.**
Asking Questions about a Query Plan and FRP Data..... **Error! Bookmark not defined.**
Asking Questions about Runtimes of Query Plan Steps ..... **Error! Bookmark not defined.**
Asking Questions about Errors in Query Plan Steps..... **Error! Bookmark not defined.**
Asking Questions about EFFICIENCY of Query Plan Steps **Error! Bookmark not defined.**

Asking Questions about WASTE ROWS (FILTER Predicates) .....**Error! Bookmark not defined.**
How to Fix Problems .....**Error! Bookmark not defined.**

Environment.....**Error! Bookmark not defined.**
Semantics.....**Error! Bookmark not defined.**
Process.....**Error! Bookmark not defined.**

Scripts used in the Chapter ..**Error! Bookmark not defined.**
Chapter Summary.....**Error! Bookmark not defined.**

Chapter 8: EXADATA..... **Error! Bookmark not defined.**

Not an Expert..**Error! Bookmark not defined.**
Why EXADATA**Error! Bookmark not defined.**
Reasons NOT to go EXADATA ..... **Error! Bookmark not defined.**
How Much Faster Will My Apps Run ..... **Error! Bookmark not defined.**
Workload Characteristics.....**Error! Bookmark not defined.**
Famed Ten Times to Forty Times Speed Up.....**Error! Bookmark not defined.**

SMARTSCAN..**Error! Bookmark not defined.**

Column Projection .....**Error! Bookmark not defined.**
Row Filtering .....**Error! Bookmark not defined.**
iDB Messaging .....**Error! Bookmark not defined.**
Other Significant SMARTSCAN Optimizations ..**Error! Bookmark not defined.**

Storage Indexes .....**Error! Bookmark not defined.**
PRE-Join Filtering with Bloom Filters.....**Error! Bookmark not defined.**

Less Common SMARTSCAN Optimizations**Error! Bookmark not defined.**
Things that Sour the Secret Sauce ..... **Error! Bookmark not defined.**

Reconstructing a Consistent Block (block is being updated) .....**Error! Bookmark not defined.**
Chained Rows .....**Error! Bookmark not defined.**
Modified Columns in the WHERE clause.....**Error! Bookmark not defined.**

Partitioning and SMARTSCAN .....**Error! Bookmark not defined.**
Proving SMARTSCAN Happened.**Error! Bookmark not defined.**

2X to 4X Speed Up .....**Error! Bookmark not defined.**

Increased Hardware Power .....**Error! Bookmark not defined.**
Database Software Enhancements ..... **Error! Bookmark not defined.**
SMART FLASH CACHE.....**Error! Bookmark not defined.**
Effects on Applications .....**Error! Bookmark not defined.**

OLTP .....**Error! Bookmark not defined.**

Row by Row (Slow by Slow) Processing..... **Error! Bookmark not defined.**
Star Schema Analytics..... **Error! Bookmark not defined.**
Star Schema Analytics using SMARTSCAN ..... **Error! Bookmark not defined.**

Mapping Applications to Speed Category ..... **Error! Bookmark not defined.**
Do we still need Indexes **Error! Bookmark not defined.**

New 2% RULE ..... **Error! Bookmark not defined.**
Optimizer does not know SMARTSCAN Math ... **Error! Bookmark not defined.**

Do we still need Traditional Tuning..... **Error! Bookmark not defined.**
EXADATA Miscellaneous ..... **Error! Bookmark not defined.**

SQL has costs on the database server too . **Error! Bookmark not defined.**
HCC is a Specialized Feature..... **Error! Bookmark not defined.**

Why..... **Error! Bookmark not defined.**
Minding the Details of HCC..... **Error! Bookmark not defined.**
Compression Levels ..... **Error! Bookmark not defined.**
READ ONLY Data..... **Error! Bookmark not defined.**
Different Compression Strategies for Different Data Processing Scenarios **Error! Bookmark not defined.**

How to Become Better at EXADATA..... **Error! Bookmark not defined.**

Chapter Summary .....**Error! Bookmark not defined.**

LAB: Reverse Engineering the QEP..... **Error! Bookmark not defined.**

LAB Time = 4 hours .....**Error! Bookmark not defined.**
Summary of the LAB .....**Error! Bookmark not defined.**

Part 1 (query analysis).....**Error! Bookmark not defined.**
Break.....**Error! Bookmark not defined.**
Part 2 (cardinality analysis).....**Error! Bookmark not defined.**
Break.....**Error! Bookmark not defined.**
Part 3: (rebuilding the original query)..... **Error! Bookmark not defined.**
NOTES:.....**Error! Bookmark not defined.**

Part 1 (query analysis) ...**Error! Bookmark not defined.**

Step #1: Dump the QEP for the query to get a query plan to work with.....**Error! Bookmark not defined.**

PLAN_TABLE_OUTPUT ..... **Error! Bookmark not defined.**
Query Block Name / Object Alias (identified by operation id):..... **Error! Bookmark not defined.**
Outline Data ..... **Error! Bookmark not defined.**
Predicate Information (identified by operation id): ..... **Error! Bookmark not defined.**
Column Projection Information (identified by operation id):..... **Error! Bookmark not defined.**

Step #2: Identify the DRIVING TABLE for the query plan .... **Error! Bookmark not defined.**
Step #3: Locate references to database row source objects in the query plan..... **Error! Bookmark not defined.**
Step #4: Construct the list of tables accessed by the query plan ..... **Error! Bookmark not defined.**
Step #5: Construct the data model for the query plan..... **Error! Bookmark not defined.**
Step #6: Construct the join order (join sentence) for the query plan ..... **Error! Bookmark not defined.**
Step #7: Construct the join hierarchy for the query plan ..... **Error! Bookmark not defined.**
Step #8: Construct the query diagram for the query plan..... **Error! Bookmark not defined.**

Summary #1.... **Error! Bookmark not defined.**
Break..... **Error! Bookmark not defined.**
Part 2 (cardinality analysis)...**Error! Bookmark not defined.**

Step #9: Construct NUM_ROWS query for the query plan .. **Error! Bookmark not defined.**
Step #10: Construct COUNT QUERIES for the query plan.. **Error! Bookmark not defined.**
Step #11: Construct FILTER QUERIES for the query plan .. **Error! Bookmark not defined.**

Step #12: Construct the FRP spreadsheet for the query plan (NUM_ROWS/row count/rows/filtered row count) ..... **Error! Bookmark not defined.**

Summary #2 .... **Error! Bookmark not defined.**
Part 3: (rebuilding the original query) ..... **Error! Bookmark not defined.**

What is Column Projection ..... **Error! Bookmark not defined.**
Column Projection with Joins ..... **Error! Bookmark not defined.**
Reconstructing the Simple Query . **Error! Bookmark not defined.**
Negative Practices affecting Performance .. **Error! Bookmark not defined.**
Rebuilding the Lab Query ..... **Error! Bookmark not defined.**

Scripts used in the Chapter .. **Error! Bookmark not defined.**
LAB SUMMARY ..... **Error! Bookmark not defined.**

Appendix: Know Your Scripts **Error! Bookmark not defined.**

Cowardly Disclaimer ..... **Error! Bookmark not defined.**

Scripts for analyzing queries and plans .. **Error! Bookmark not defined.**
Scripts for examining an active database **Error! Bookmark not defined.**
Scripts for looking at metadata and trying to fix a problem query ..... **Error! Bookmark not defined.**
Script Walkthrough..... **Error! Bookmark not defined.**

showplan11g ..... **Error! Bookmark not defined.**
showplan11gshort ..... **Error! Bookmark not defined.**
showplanconstraints11g ..... **Error! Bookmark not defined.**
showplancountqueries11g ..... **Error! Bookmark not defined.**
showplandatamodel11g..... **Error! Bookmark not defined.**
showplandrivingtable11g ..... **Error! Bookmark not defined.**
showplanfilterqueries11g ..... **Error! Bookmark not defined.**
showplanfrpspreadsheetcode11g . **Error! Bookmark not defined.**
showplanindexes11g ..... **Error! Bookmark not defined.**
showplannumrows11g..... **Error! Bookmark not defined.**
showplanquerydiagram11g..... **Error! Bookmark not defined.**
showplantables11g..... **Error! Bookmark not defined.**
showplantablesunique11g ..... **Error! Bookmark not defined.**
loadplanfromcache11g ..... **Error! Bookmark not defined.**
loadplanfromhist11g ..... **Error! Bookmark not defined.**
showtopcpu11g ..... **Error! Bookmark not defined.**
showowner ..... **Error! Bookmark not defined.**
showindexes..... **Error! Bookmark not defined.**
showconstraints..... **Error! Bookmark not defined.**

showcolstats.....**Error! Bookmark not defined.**
showhistograms.....**Error! Bookmark not defined.**
showallscanrates.....**Error! Bookmark not defined.**
showallworkareas.....**Error! Bookmark not defined.**
showgencardinalitycheckcode.....**Error! Bookmark not defined.**

Script Summary.....**Error! Bookmark not defined.**

Index ..... **Error! Bookmark not defined.**

About the Author

Thanks for buying this book. I hope you enjoy it and I hope I am able to offer you something new in the world of SQL Tuning. I am Kevin Meade.

In 1984 I typed *SELECT 8 FROM EMP;* into a SQL*Plus prompt, and after a brief delay was presented with a green screen full of scrolling 8s. Following five minutes of confusion and certainty that this thing called Oracle was broken, I realized I had fat-fingered the shift key on my very first SQL statement, and thus began my career as an Oracle RDBMS Professional with a mistake.

In retrospect, *IT WAS MY FAULT* might have been the best possible first lesson to learn, for someone just starting out in the Oracle space. There would be many more mistakes over the next thirty years, and many more times when my ego had me jumping to the conclusion that it was an Oracle bug. But there was always a nagging reminder of a screen full of 8s giving cause to keep at it a little while longer to find some yet elusive better understanding. Though there were bugs to be found, there were many more occasions where it was my fault.

My career path is not anything particularly special. It is probably a lot like yours. Just out of school, I started in Connecticut in the scientific industry as a COBOL Developer supporting a clunky DSS system, and landed my first Oracle work by accident. Three months on the first real job that Mommy and Daddy didn't get me, my Boss at the time John was hunched over his desk looking at pages full of SELECT statements and things called DEPT and EMP. "Oh, Structured Query Language eh?" I said. "You know this stuff?" he inquired. "Well …" but I never got a chance to finish. "Good, you are now our resident Oracle expert Kev. Here is what I need", pushing a boxed set of three 7x9 books in my direction (yeah that was it for documentation in those days). I took notes while he talked, like maybe that might impress him, said thanks, and went back to my office to start reading (yeah we had real offices in those days too). I didn't have the heart to tell him that there was only one class I had taken related to this stuff, and that class itself actually had nothing to do with SQL, but rather traditional Data Processing. My experience was limited to 20 pages of appendix material in the back of the class book that I only read because I was waiting outside the computer lab at college to ask a girl out for margaritas and chips at the Margarita Ville right next door. This was a commonly used watering hole for college students at CCRI for as we all know, assembler language makes much more sense when buzzed. The girl never materialized, but six months later the Oracle job did. Not a bad trade I guess.

So I was having a good time at this Oracle stuff. It was way easier than writing COBOL code. A couple years went by, a couple of Oracle conferences attended, and I learned there was this job called ORACLE DBA that paid 50% more than my job called ORACLE DEVELOPER. I want me one of those jobs, I thought. So I got out the resume, made liberal use of WHITE-OUT and after six minutes of blowing across the page, I typed over what used to say Oracle Developer, the words Oracle DBA. What did I have to lose? Near as I could tell it was pretty much the same job anyway cause in those days you did everything. A local head-hunter assured me that if I was willing to travel then it would be no problem at all. True to his word, in three weeks' time I was on my way to Georgia to a place that did configuration management for Nuclear Power Plants. I hated to leave my first job. I knew I was going to miss the wit and humor of the team, and their passion for quotes from old movies, but John said he was glad for me. He was starting to wonder why I had waited so long to make my next move.

The job was great too. Besides a lot of cool and unique problems to solve, I remember Quan Hong Lin (we called him Henry for short (I miss Henry)) teaching me the right way to eat soft shell crab, and Friday's with the gang of six down at a local diner where if you bought a pitcher of beer you got the box of hot wings for free. Six of us, six pitchers of Bud, and six boxes of free hot wings. Good times. But it was not to last. My girl at the time Elaine, (these days my wife) said "this long distance thing is not working out. Either come back or don't bother to come back". So I updated my resume again, made calls to head-hunters back home, and said my goodbyes; for it was off to Massachusetts to another Oracle job in the engineering industry.

This DBA thing is awesome I thought. Two new jobs with two 50% raises, both in one year. But this job I didn't like. An hour drive and an hour on the T with people who looked like they had the life sucked out of them (not hard to figure out why), to work in a company where tension was high since the place was going out of business (the head-hunter forgot to mention that). Fortunately I was saved six months later by a friend from my first job who was heading up an Ingres database project. He offered me a spot as a Senior DBA. Wow I thought, do they make those? The job was of course more then I should have tried for, but how could I say no. I wanted out of where I was and this was yet another raise, and Bob said he would give me all the support I needed to get up to speed.

So without a lot of fanfare I made my way back to Connecticut to an Insurance company and started learning that business. It wasn't Oracle but how different could it be? Turned out there were a lot of transferable skills. And the product Ingres at that time was actually a technically superior product to Oracle, and this company I was now working for was throwing money around like crazy. So I got in with a really good team: Peter and Paul and Gary and Rick and Rochelle and Sandy to name a few. They taught me how to play Hearts which we did every day at lunch. And I had access to the greatest minds of that era in Relational Database Technology. I even met Michael Stronebraker and got an autographed copy of his book "The Ingres Papers" (how geeky is that?). I learned a lot. But as with every BE ALL TO END ALL project, this one spent way too much and produced nothing that anyone actually wanted, which someone finally noticed, so after a few years of creative book keeping, it got shut down and I was looking for another place.

Once more to the head-hunters and this time down the street to another insurance company. It was Oracle work again, the biggest user of Oracle in the state. Three years in I said, I like it here, plenty of work, they let me do whatever I want, and they keep telling me I am great. I think I'll stay. That was 1995 and I have been here ever since. It was a very good decision too, for today I work remote out of my house, four days a week working from a recliner (WFR) tuning SQL and in general providing whatever support my team asks of me. A great team it is, full of people every bit as smart as me. You benefit from their brain power too as about twenty of them have reviewed chapters of this book ahead of you. **Let me tell you, *it's nice to soar with eagles.***

How this Book is Different

This book is different in two ways from its competition. First because I created rules for what this book should be like, which I followed. And second I had the advantage of an intense unique experience to draw upon and from which this book arose.

There is in fact a lot of competition for this book. This year alone there are two books published that directly compete with this one. We are in a good time for writing about Oracle topics. Turns out I am not the only Oracle Professional with multiple decades of Oracle experiences on hand to talk from. There are plenty of us, and we are starting to get vocal.

After several false starts I finally finished the first chapter of this book. At this point it became clear that it was not going to be easy writing a book on SQL Tuning. I also realized that there were several things about other books I had read in the past which I really did not like, and it would take effort on my part, to keep my book from doing the same things. So I created the following rules to write by and threw away the first chapter to start over yet one more time.

My rules for this book.

- **Only write about what I know**. For some reason, maybe the need to feel like they covered all the right topics, some authors talk about topics they do not really know. This does two bad things. First it creates a topic for which there is only weak coverage. Second it takes away precious space in a book that could have been used to talk in more depth about something they really did know about. Neither of these two things does the reader any service. Having seen this too many times in other technical books about Oracle, I was adamant that I would not do that. One ramification of this is that there may be topics that I should have covered but did not. That's the breaks. I make no excuse for it. There is no such thing as a complete or definitive reference on anything Oracle anyway. But the good news is that this one rule forced me to evaluate every topic in this book against other topics I could have used. So you got the best of what there was for the five hundred pages I wanted to fill.
- **Focus on teaching the skill of SQL Tuning**. I always liked college texts because they actually taught something. Not that it is bad, there is clearly a market for them, but many books in the Oracle space try to cover a large area and so have hundreds of little points of interest, but in the end lack an overall goal. I wanted to teach SQL TUNING as a skill so that is what I do. I provide the necessary critical mass of knowledge to understand the topics discussed, and a process in how to apply it, making certain to focus on the skill of SQL Tuning. Every topic covered in this book was subjected to the question "How is this related to SQL Tuning?".

- **Offer something Unique**. I do not enjoy books that are mostly a recapitulation of the manuals. If I want to read the manuals I can do that online. Today a good book should not only offer technical fact, but should also offer the author's unique experiences and perspectives that only they can give. That is where the real value-added proposition is. What is the point of having decades of experience if you don't share it? So I am offering up my unique approaches and perspectives, along with the technical details. I think you will find ideas presented here that you won't read anywhere else.

In addition to a set of rules that guided me in writing it, this book has the advantage of a special time in my SQL Tuning life. Four years ago I was put on a three month gig. I was supposed to go into a place that had a shortage of people, and fix some critical performance problems they had, while backfilling for lost staff. I did such a good job at it that my now boss Sandeep would not give me back when the three months were up. He wanted to keep me permanently. Seems he had the juice to do it too so I was moved to his Division and the three month tuning job became a three year thing that covered hundreds of databases and problem SQL from each.

I was not long into it when I figured out what a singularly unique opportunity had been put in front of me. So much wrong. So much to be fixed. Opportunities Savoir Faire (they were everywhere). If I put real effort into it, if I paid attention, and if I kept good notes, then I would have a fabulous experience to tell people about. Hundreds and hundreds of problem SQL tuned, Sixty, maybe more Crisis Meetings attended. In the end, it yielded a total revelation for me in how to tune SQL, with plenty of examples doing it. I draw heavily from what I learned during this period, putting it into this book. What you get here is based on technique that works in the trenches.

To summarize it all, this book is based on deep experience, stays true to its goal of teaching you SQL Tuning, presents unique perspectives you won't find anywhere else, and is backed by a massive reservoir of successful tuning events and the learnings that came from them. That is how this book is different.

Those who came before

There are a lot of good books written in the last decade about SQL Tuning. But of all these books, there is one that stands out among them. This book is Dan Tow's SQL TUNING published in 2003. This book is by many, considered the greatest book on SQL Tuning ever written. I hope now the second greatest book on SQL Tuning ever written.

On the surface Dan's book did not appear to be much different from most of the other material on the subject. But it had one chapter that pushed it to the front of the class, and which to this day still makes it relevant. Though at the time, there were many of us who practiced similar processes to what Dan presents, Dan was the first to put into print in an easily digestible manner, a process which could be duplicated, and which could reliably be used to tune SQL. Thus Dan was the first to make SQL Tuning accessible to the masses. Though he wrote it for the RULES BASED OPTIMIZER, the technique he demonstrates is CARDINALITY based and so, still relevant even today.

I kept waiting for Dan to revise his book to encompass the advances of Oracle in the last decade for things like STATISTICS and HASH JOIN and DYNAMIC SAMPLING and STAR SCHEMAS and so on. But it never came. Eventually I realized my expectations were unfair. Dan had done his job. He wrote the book that needed to be written for that time. It was up to someone else to write the next one. It may sound pompous of me, but I like to think I took that up with this book. You will find my first chapter to be similar to Dan's work. How could it not be? The basic idea of cardinality as a driver of performance in query plans is the same.

I do not know Dan. I have never met him, and have only visited his LinkedIn page twice. But I can still recognize Dan for his contribution to our Art.

Thanks Dan.

Getting the FREE STUFF

You many have noticed that I have chosen to self-publish this book. The big publishers have a business to run and need to make choices between going with a new author, and providing capital for existing successful authors instead. I am a new author and that means a risk so no one was ready to take a chance on me yet. Since AMAZON is the major player in self-publishing, that is where I went. But there are two issues that result from this choice. First is that I do not have paid editors reviewing my work or layout of the pages of the book. So you may find some things missing; for example, none of my exhibits have labels (too much work for a lazy IT guy), and you may find my grammar less than stellar in a few places since no professional has edited it.

The other problem I have encountered is difficulty in figuring out how to add additional downloads to this book or even to create a free downloadable document, and this prevents me from offering the scripts of this book in electronic form through Amazon. It is doubtful Amazon will change how this works by the time the book goes to print so instead, you can use these other two mechanisms to get the free stuff. This is essentially the list of scripts I provide as part of the book.

Use one of these two methods to download electronic versions of the scripts from this book, ready to use.

- WWW.ORAFAQ.COM. This is a great website on all things Oracle, and I am a registered member and moderator. So I posted the material on this site as an article for you to download. Search on my name, or the book title, to find it. While you are there, why don't you become a member and stay a while. Membership is also free and it offers a chance to ask questions of other Oracle experts better than me.
- KM133688@SBCGLOBAL.NET. This is my personal email. I should be so lucky that the book sells so many copies it floods my in box. I look forward to hearing about what you think. I would also be happy to provide additional items as they become available, including sending you an email with all the scripts.

Reviews on Amazon

I have come to be a believer in the peer review system. Whether your review would be favorable or not, I encourage you to write a review on this book on Amazon. People need to know what others think about the stuff they buy before they buy it, and Amazon reviews offer that opportunity.

There is a plethora of books on related topics to this book that people will have to choose between when they start looking for new knowledge in this subject area. Your review can help them make better choices and that is what sharing is about. Don't worry, I won't sue you for leaving a negative feedback if you really think my book was bad.

On the other hand, if you really think this book needs something it does not have, drop me a note about it so I can add it to the text. One advantage of self-publishing is the ability to dynamically change subject matter without a long delayed process to publish it.

If you do decide to write an Amazon review, please consider providing some of the following information.

- Your role in the Oracle space. Are you a Developer, or DBA, or something else. If you are a Developer then other Developers will pay more attention to your review, and so on.
- Your experience level in the Oracle space. Consider noting how long have you been in IT, worked with relational databases, and worked with Oracle. Others with similar time spent in the business will understand how your opinion relates to their experience level.
- The PROS and CONS of the book. No book is perfect. It gives more weight to your opinion if you cover both sides of the fence. People are looking for reasons to buy the book and reasons to avoid it. This is where it helps to be a bit detailed, even noting specific pages or wordings from the book if you are so inclined.
- Your final BUY OR BUST decision. Do you recommend it or not, for others like yourself.

The purpose of these specifics are so that those reading your comments can get some idea of how well your situation matches their own and therefore how likely your review is relevant to their situation.

Thanks very much if you write a review for this book.

WWW.ORAFAQ.COM

This website is a special place to find help with Oracle. Membership is free and there are many Oracle specialists who participate in this site providing their knowledge at no charge. From this pool of specialists there were several who took time out of their lives to review chapters of this book. Each person mentioned was assigned one of the chapters, though several helped me with more than one. Please find a description of each as they describe themselves.

John Watson

Oracle Certified Master DBA

Director of Database Services

Skillbuilders Inc.

Jim Irvine

Jim is a Contract IT consultant working both in Oracle development and Testing functions, busy father of twins and Oracle pedant.

Lalit Kumar

I started my IT career in ORDBMS technology as Database application developer, worked for couple of organizations. Currently working in Oracle Corporation. My favorite topic has always been performance tuning. It needs skills and expertise, always a difficult task, as mostly it is done manually, there are no shortcuts. I am privileged to be one among the early reviewers of the "Basics of performance tuning" chapter. Honestly, I never came across any other study material that talks about topics at such basic level. Now I know the secret of Kevin's quote and it has become my favorite now:

"Performance tuning is all about cardinality". Read the book to reveal the secret

Mark Kilgour

Currently a production DBA at BSKYB* specializing in database performance tuning and troubleshooting. Mainly I keep myself to myself, but I find the world of performance fascinating and rewarding and I like to pass on as much as I can.

When I'm not doing that, you'll find me throwing a mountain bike down a hill somewhere.

Ross Leishman

Principal Consultant, DWS Limited, Melbourne, Australia.

Saša Dominković

Croatia, Oracle developer.

Soaring with Eagles

I work with a team of experts in the Oracle space. Most of them have decades of experience behind them. Several of them also have a specialty which interests them which to me is another indication of professionalism. Several of my teammates took time from their lives to review chapters of this book. They are listed here using descriptions of themselves as they provided them.

Barry Ward

Senior DBA, Data Warehousing specialty, and Connecticut College Men's Head Squash Coach.

Dennis Deluzio

Senior DBA, 20 years.

Dheeraj Madadi

Senior Database Manager, 15 years.

Jack McGuirk

Senior DBA, Oracle Advanced Security Option specialty, 36 years in IT, 22 years doing Oracle.

Experience covers Unix System Administration, Networking, and Web Development from notepad to ASP to VB.net.

Nirav Kathrani

Three years in Software Engineering, two in IT. Currently a Data Innovation Consultant.

Robert Romanowski

DBA, Enterprise Data Management, Technology Leadership Development Program.

Subrahmanyam Jannalagadda

Senior DBA, Enterprise Data Management, Technology and Solutions.

Chapter 1: DRIVING TABLE and JOIN ORDER

Shopping analogies are often used to explain ideas in IT, so why not one more. Let us suppose you have a shopping list with several items, and you must visit five different stores to buy all the items. How will the shopping trip unfold?

If you view your shopping trip as a recreational experience, then you probably ignore the list of stores someone else had in mind, and head to the Mall where upon your shopping trip becomes an outing. You get something to eat from the Food Court, spend a half hour in the game room racking up extra balls on Silver Ball Mania, do some comparison shopping revisiting several stores to get the best value for items that are not actually on your list so you can justify why it is OK to get them, sit in a massage chair for ten minutes because your feet hurt, and so on. Eventually you get the items you need whilst managing to overpay for them. For some this is fun, though it will never be fast.

But if you are a gopher for some company (you go for this and you go for that), then you are not on vacation; you are working. As such you want your shopping trip to be just that, a shopping trip. Your primary concern is to get the stuff you need as fast as possible because other people are waiting on these items so they can do their jobs. For this kind of shopping a smart gopher will grab a road map. The gopher will locate on the map, each of the five stores designated by the Boss as valid shopping spots, and make two decisions: 1) the gopher will decide on which store to got to first, and then 2) decide on what order to visit the rest of the stores. The goal of the gopher is to minimize the amount of time spent on the road by optimizing the travel route and this is accomplished by making good choices for which store to start at and what order to visit stores.

An experienced gopher will consider far more than just mileage between stores. The gopher may ask questions like: What alternative routes exist between stores? What are the traffic volumes at different times of the day for each route? How many stop signs and traffic signals are there on each route? Are any of the alternative routes also school bus routes? Does it make sense to park the truck somewhere and take a Trolley or Subway or other public transportation? The gopher may end up picking several legs of the journey for factors other than simple shortest miles traveled, if these other factors make for shorter overall travel time. In the end, through some kind of magic, the gopher figures out the best place to start, and the best order in which to visit other places to be visited, and the gopher figures it out fast.

Executing a SQL query is much like gopher shopping, it is not a recreational experience. We want whatever we do to be not only correct, but also efficient in the way it is done. Given a query that joins five tables, a database optimizer must make the same two decisions as the gopher. It must decide on which table to visit first (called DRIVING TABLE), and the order in which to visit tables during query execution (called JOIN ORDER).

The Four Parts of a Query

There are in fact four parts to every efficient query:

- **Driving Table:** Table from which we access all other tables in the query.
- **Join Order:** Order in which we access tables in the query.
- **Access Method:** How we get rows from a table.
- **Join Method:** How we put rows together from two tables.

Although a poor choice with any of these four items can result in a poorly performing query, the first two (DRIVING TABLE and JOIN ORDER) are by far the more important of the four. If the optimizer picks a good DRIVING TABLE and a good JOIN ORDER for a query, then 99% of the time the optimizer will also make good choices for ACCESS METHOD and JOIN METHOD everywhere else in the query.

It is somehow sad to think that SQL query tuning can be reduced in most cases to the act of getting the right choices made for DRIVING TABLE and JOIN ORDER. But if we push the noise generated by various neat theories and special cases into lane 2 and reserve Pole Position for our observations of what actually happens in every day SQL workloads, then we are forced to conclude that this is just the way it is. And if it is going to be this way, then we might as well exploit such a strong truth to our advantage.

Therefore we start our work in learning how to discover what is wrong with a problem query, by learning how to determine what should be the DRIVING TABLE and JOIN ORDER of choice for a query. To do this, we are going to use a technique called *Filtered Rows Percentage.*

The Filtered Rows Percentage Method

During execution of a query, keep the number of rows in transit as small as possible, at all times.

Said another way

Remove as many rows as possible as early as possible in query execution.

Yes, this is pretty much it. The truth, back in the 80's when relational databases first made the scene, about doing the least amount of work necessary in manipulating data to get an answer, is still the main query performance truth of today. The CBO (Cost Based Optimizer) is going to apply lots of fancy mathematics and look at potentially thousands of different query plans in its efforts to find the best way to get data for a query. And if we did our jobs as DBAs, then most of the time the CBO will give us a well performing query we do not need to worry about.

We will observe that ninety nine times out of one hundred, the best query plan is going to be the one that does what was just said; keeps the size of intermediary row-sets as small as possible by eliminating as much data as possible as early as possible. It is not a coincidence that a query plan which manipulates the least amount of data in answering its query, is also most often the least expensive query plan. And this is the goal of Filtered Rows Percentage; to move the least number of rows around during query execution by eliminating as many rows as possible as early as possible in query execution.

First Look at a Query

The first rule of tuning SQL is not to tune the SQL but instead to get to learn the query. Consider this simple example:

```
select count(*)
         from T1
,T2
   ,T3
 where t1.pk = t2.fk and
 T2.pk = 2 and t3.c = 3 and t2.pk
 = t3.fk;
```

Boy that is an ugly query. Let us format it so we can more readily work it. Neatness counts.

```
SELECT count(*)
FROM   t1,
       t2,
       t3
WHERE  t1.pk = t2.fk
   AND T2.pk = 2
   AND t3.c = 3
   AND t2.pk = t3.fk;
```

That is better. Look at the query for a moment to familiarize yourself with it and see what can be learned off hand.

Observations on a query.

- First, there are three tables in the query. And if the names of the columns and the joins they participate in are any indication of reality, we can presume that these three tables are actually a hierarchy of tables (PARENT / CHILD / GRANDCHILD). Naturally at some point if it becomes relevant, we should be consulting the data dictionary and its constraint metadata to validate what we think is true about these tables. But for now we will go with what we see.
- Additionally, we can see that the query seeks only one row from table T2 (T2.PK = 2). And we want only some of the child rows of this row from T3 (T3.C = 3). So it appears that we are looking for a small subset of the total data available. Small of course is relative but we offer a more formal definition of small in Chapter 4: Joins.
- Lastly, we might even question the need to reference table T1 at all in this query. If we make some additional assumptions about our table: that T2.FK is defined as NOT NULL and the foreign key is TRUSTED (it is enabled from the beginning, or we have explicitly told Oracle to trust this constraint via RELY), then of what value is the join back to T1? We cannot duplicate or lose rows joining to T1 under these stated conditions, and we don't want any data from T1 in our answer, so it is what we can call a DUNSEL JOIN. It is a join which serves no useful purpose. We don't even need to do it.

Well, that is a lot to see in a cursory examination of a query. Is any of this information useful? Maybe. What is really important is to take our time and be neat because neatness counts. The first two steps in evaluating any query are to 1) Format it, and 2) Look at it before we do anything else.

Query Diagrams

Having completed our initial familiarization with the query, more detailed analysis can begin. One tool for understanding queries is the query diagram. If we build a query diagram for our query, we can get a visual representation of what it looks like. The basic query diagram for the above query looks like this:

```
 T1  -----  T2  -----  T3
            |          |
            |          |
           (c)        (c)
```

Note we have listed tables in order of the FROM clause, and indicated joins by connecting lines, and indicated constant test predicates via a sub-line and indicator. Already one can begin to see the natural join progressions that might be possible.

Remembering the query's WHERE clause, it is reasonable to assume that the filter criteria T2.PK = 2 would yield 1 row (it is after all a primary key check), whereas T3.C = 3 could yield multiple rows (though we do not know how many).

We can label the query diagram with some made up row counts for discussion. Referencing the numbers below, the filter criteria as applied to T2 without regard to joins in the query, will yield one row (this we know from the PK). Also, the filter criteria as applied to T3 without regard to joins in the query, will yield 100 rows (this we just made up so we can talk about what it might mean). Again, we made these numbers up so we could have a meaningful discussion on join order. Given these expected row counts based on the query WHERE clause criteria, we can evaluate different join orders to see what happens. For example:

```
 T1  -----  T2  -----  T3
            |          |
            |          |
           (c)        (c)
           (1)        (100)   FILTERED ROW COUNT
```

If we start with T2 as our driving table, we start with pushing one row into query execution. This means we start with one row in our intermediary row-set. We could then go up the table chain to T1 if we want. That one row would lead us to one probe into T1 which assuming we are going up a foreign key to a parent table, would yield again, only one row and thus we still have only one row in our intermediary row-set. We can then join that one row to T3. This will result in one probe of T3 returning one-hundred rows (based on T.C = 3) so that our final row-set has one-hundred rows. Given this join order, we carried one row in transit most of the time, getting one hundred rows in the final step.

Or we could start with table T3 as the driving table. This choice of driving table would push one-hundred rows into the query. Execution would proceed to join with T2 where we would do one-hundred probes into T2 resulting in one-hundred rows in our intermediary row-set. These are then followed by one-hundred probes into T1 which would result in one-hundred rows in our final row-set. In this situation, the query carried forward one-hundred rows all the time which is though not a lot, still more than the previous driving table and join order.

As possibly a really bad choice, we could start with table T1 as our driving table. To understand why T1 might be a poor choice for driving table, let us add the number of rows in each table in our query to our query diagram so we have more to talk about. Again, these numbers are made up to serve our discussion. T1 has one million rows in it; we note this in our query diagram. T2 has ten million rows and T3 has one-hundred million rows. We note these in our query diagram.

```
 T1  -----  T2  -----  T3
            |          |
```

```
             |           |
            (c)         (c)
            (1)         (100)     FILTERED ROW COUNT
(1M)        (10M)       (100M)    rows in table
```

Given the total rows in each table as noted above, starting with table T1, we would push one-million rows into query execution. T1 gives one million rows because it has no filter criteria and thus all rows must be fetched and retained in this step. Once fetched these one million rows would lead to one-million probes of table T2 that would yield all rows in T2, which by our diagram is ten-million rows. We would then filter these ten-million rows to return only one row (T2.PK = 2). Execution would then do one probe of T3 to return one-hundred rows. Though the final answer is the same as before, we see that this choice of using T1 as the driving table and the subsequent join order this driving table leads to, had us holding at two points during query execution, millions of rows in our intermediary row-set.

These examples show that even for a simple query, not all tables are equal when it comes to their behavior as a driving table. In one join order case, we needed to access all rows in a one-million row table and all rows in a ten-million row table and throw away a lot of junk, which would require lots of I/O and lots of page gets and lots of comparisons during joins and filtering. Yet in another case we had only one row in transit with the final row-set of one-hundred rows being dumped directly to the caller when retrieved in the last step of query execution.

Clearly, where you choose to start a query can have a significant impact on the work required to do the query. A query diagram provides a visual of the alternatives when different driving tables are considered.

Driving Table and Join Order

The DRIVING TABLE is the first table from which other tables in a query can be joined. You have to start by getting your first set of rows from some table in your query so where you start is the DRIVING TABLE

Two things make the driving table important:

- It determines which tables can be accessed next during query execution, effectively deciding on a roadmap for the query.
- It determines the initial number of rows that the query will start with and thus drives the amount of work the query will have to do in subsequent steps.

JOIN ORDER determines ongoing workload for the query. Assuming we do not allow CARTESIAN JOINS in our query plans, the possible join orders (also called a join sequence) for this query are those shown here. Only VALID join orders are considered. This means joins must follow the join paths made available by the query, which can easily be seen by looking at the query diagram.

```
DRIVING  POSSIBLE VALID
TABLE    JOIN SEQUENCE
-------  --------------------
T1       T1  -->  T2  -->  T3
T3       T3  -->  T2  -->  T1
T2       T2  -->  T1  -->  T3
T2       T2  -->  T3  -->  T1
```

We also saw that if we use NUMBER OF ROWS MANIPULATED as an estimate of the workload of a specific JOIN SEQUENCE, then the cost of each join sequence is potentially different, sometimes very different, particularly when we consider the SIZE OF INTERMEDIARY ROW-SETS generated as the outcome of some of the joins.

The key to correct join order for our purposes, is to keep the intermediary row-set sizes as small as possible throughout query execution. There are two ways to do this, each yielding a different cost.

Two strategies for keeping intermediary row-set sizes small:

- LEAST NUMBER OF ROWS: Pick a join sequence that is likely to return the least number of rows after each join. This is what we saw in the example above when we determined that the best join sequence was most likely T2, T1, and T3.
- LEAST PERCENTAGE OF ROWS: Pick a join sequence that is likely to remove the highest percentage of rows from consideration as soon as possible. This is what FILTERED ROWS PERCENTAGE method will attempt to do.

As an quick thought, if we actually used FRP (FILTERED ROWS PERCENTAGE) which is the strategy this book wants you to learn and use, as the method to determine join sequence in the above example, we would have concluded that T2, T3, T1 is likely the best join order, not the order T2, T1, T3 that we think is the best order. This is just the way FRP works. Some might say this shows a flaw in the method. However the opposite is actually true. If a strategy was to use LEAST NUMBER OF ROWS rather than LEAST PERCENTAGE OF ROWS as the method for reducing workload, then it would tend to pick smaller tables over larger tables as driving tables regardless of filtering criteria. It is common for queries to filter away significant percentages of rows from large tables yet still have row-sets from those tables that have more remaining rows than small lookup tables. A large table with one-hundred million rows in it but for which a query filters away 99.9% of the rows, still sends one-hundred thousand rows into the query. This is far more rows than a simple fifty row reference table. In such a situation, LEAST NUMBER OF ROWS would pick the small table over the large table when considering driving table and placement in the join order, even though no rows had been filtered from the small table. This could be a disaster for performance. Going with the small table that does not remove any rows means keeping 100% of the workload. After access to such a table, no reduction in workload would be achieved. Thus LEAST NUMBER OF ROWS misses opportunities for workload reduction.

What this means is, we want to use the LEAST PERCENTAGE OF ROWS strategy when deciding on DRIVING TABLE and JOIN ORDER.

COUNT QUERIES and FILTER QUERIES

Since cardinalities (row counts) are central to SQL tuning, we will often run pieces of a query in order to validate ideas and collect information about intermediary row-set sizes and things like that. In particular there are two types of these kinds of queries that the FILTERED ROWS PERCENTAGE technique uses. They are called COUNT QUERIES and FILTER QUERIES. The next section will go into detail about these. For now here is a simple example.

```
select *
from dept,emp,project
where project.emp_id = emp.emp_id
and emp.dept_id = dept.dept_id
and project.group_type = 'PT1'
and emp.ssn = '12345';
```

There are three tables in this query. From this we can construct three COUNT QUERIES.

```
select count(*) from dept;
select count(*) from emp;
select count(*) from project;
```

The idea is simple of course. These will count the number of rows in the table. Thus they are called COUNT QUERIES.

But looking more closely, we see that there are predicates in this query too. Some are join predicates and some are constant test predicates. We are interested in the constant test predicates. Since there are three tables in the query we could create as many as three FILTER QUERIES as well. A filter query is the count query but with all the predicates from the parent query that are based on constants. Note we are not interested in the join predicates. Thus the parent query would produce these three filter queries.

```
select count(*) from dept;
select count(*) from emp where emp.ssn = '12345'
select count(*) from project where project.group_type = 'PT1';
```

Notice that these queries contain only the constant test predicates for the associated table. Also note that the DEPT filter query is the same as its count query since there are no constant test predicates.

These filter queries tell us about the size of the workload the associated table brings into the query. This is crucial information used in FRP. Indeed, the major purpose of FRP as a strategy is to determine a good choice of DRIVING TABLE and JOIN ORDER, and to learn if your query got off to a good start. A query that gets off to a good start usually has few performance problems. We will refine what we mean by "good start" as we progress through our examples.

FRP Spreadsheet

Ultimately one of your goals in this FRP process is to produce a spreadsheet that looks like this.

```
                                                  Plan    Filtered Actual
ID    TABLE_NAME      NUM_ROWS   ROWCOUNT Cardinality Cardinality    FRP
----- ------------- ---------- ---------- ----------- ----------- ------
    8 EMP_DIM          6243035    6243035      240117      215414    3.5
    9 EMP_LOC_DIM       329699     329699      296337      329699  100.0
   15 EMPLR_LOC_DIM       8874       8874        8874        8872  100.0
   19 EMP_DIM          6243035    6243035      240117      236469    3.8
   21 EMP_LOC_DIM       329699     329699      251761      212993   64.6

5 rows selected.
```

This spreadsheet collects cardinality information using COUNT QUERIES and FILTER QUERIES and metadata from the data dictionary, and even data from the PLAN_TABLE, and does some simple math with it. It is powerful. At the end of this chapter is a short cut example of how to use a special script that generates this spreadsheet for you (less a few bugs you might need to work around).

Cardinality simply means row count, but there are many different kinds of row counts. We see in the FRP Spreadsheet different cardinalities of interest. They can be classified as either estimated or actual, and can also be classified as either unfiltered (full table) or filtered (rows removed based on constant test predicates in the WHERE clause).

The four Cardinalities in the FRP Spreadsheet have these classification combinations:

- NUM_ROWS: Estimated, Unfiltered.
- ROWCOUNT: Actual, Unfiltered.
- PLAN CARDINALITY: Estimated, Filtered.
- FILTERED CARDINALITY: Actual, Filtered.

This FRP spreadsheet will allow you to consider many facets of the query. You can consider possible DRIVING TABLE and JOIN ORDER for a query. You can consider the quality of statistics on your tables. You can consider the ACCESS METHOD and JOIN METHOD for each table. And you can get a very close idea of where the overall query workload is coming from. All of which you can compare to the current query plan details for the problem query to see how different the current query plan may be from what the FRP spreadsheet suggests it should be. You will learn all this in this chapter.

A Filtered Rows Percentage Example

We are going to walk through a real world example of using FRP (Filtered Rows Percentage). There are 16 steps in the process. Yes this is some work but nobody said tuning SQL was going to be easy. Again there is a short cut script at the end of this chapter that does the work for you.

In this process about to be demonstrated, we are going to build a query diagram for our problem query, and a FRP spreadsheet one piece at a time. When we are done the FRP spreadsheet will show us the DRIVING TABLE and JOIN ORDER that is most likely the best choice for this query. It will also let us make other determinations about the query and its query plan and the state of the environment it lives in.

Filtered Rows Percentage Method (FRP) helps us understand several crucial aspects about a query.

- Determine Driving Table and Join Order
- Determine basic query style (PRECISION QUERY or WAREHOUSE QUERY)
- Determine if the query got off to a good start (Stats and Cardinality Accuracy)

Once you have gone through this process successfully, a SQL script will be provided that does most of this work automatically. Although the script may have some limitations, it makes using FRP much more efficient. So please work your way through the FRP process here to understand its parts.

The process of FILTERED ROWS PERCENTAGE is as follows:

1. Format the PROBLEM QUERY
2. Familiarize yourself with the problem query
3. Create a spreadsheet
4. List tables from the query in the spreadsheet
5. Build COUNT QUERIES and note the row count for each table
6. Build and run FILTER QUERIES for each table
7. Note the filtered row counts
8. Compute FILTERED ROWS PERCENTAGE for each table
9. Determine PREFERED JOIN ORDER
10. Construct a QUERY DIAGRAM
11. Determine INITIAL JOIN ORDER

12. Build and run RECONSTRUCTION QUERIES
13. Note reconstruction row counts
14. Use CARDINALITY FEEDBACK to adjust join order
15. Repeat (11) thru (12) once if join order changed
16. Determine if further action is necessary

The FRP process may seem like a lot of work. The reality is this is a lot of work. But since we are only applying this method for those one in one-thousand queries that is a problem, using the method is well worth the effort. Additionally as you get better at this process, you will skip steps that have obvious answers to you. Also, the magical script employed later will do this work for you.

The subsections that follow will walk through a real-life example. You will see how the method can take you systematically towards a solution. Please notice that I have selected for teaching a rather large query. This is for several reasons. One is that it allows for some repetition in the process to give plenty of opportunities to do certain steps, and another is that I want to impress upon you that size of queries no longer matters. When you use the FRP process, the big queries fall just as easily as the small ones. They just take a little more time.

1. Format the PROBLEM QUERY

This first step of formatting the problem query is easy, and may seem so trivial as to be unnecessary. Don't omit it though. The act of formatting the query forces you to begin to read it, and you'll find yourself catching on to how the query was written to produce the desired business result. You'll be surprised what your subconscious can pick up while you are rearranging a problem query to make it more readable.

For example, here is a poorly-performing query in its original state. This query has in fact already been formatted once using a tool (via Toad or that free online SQL formatting website). But this kind of formatting is only part of what we want.

```
SELECT v_cbe_lv_rqst.lv_rqst_cd,
       att_lv_pln.be_name,
       att_lv_pln.lv_pln_cd,
       att_lv_pln_typ.be_name,
       CASE
         WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN ( 'Not classified at this level'
                                             ) THEN
         LV_SEG_LV_TYP.be_name
       END seg_be_name,
       cbe_emp.emp_natl_id,
       SUM (Nvl (v_lv_pln_usge_fact.hrs_used_diff, 0)),
       SUM (Nvl (v_lv_pln_usge_fact.hrs_rmn_diff, 0))
FROM   v_cbe_lv_rqst,
       att_lv_pln,
       att_lv_pln_typ,
       att_lv_typ LV_SEG_LV_TYP,
       cbe_emp,
       v_lv_pln_usge_fact,
       v_plcy_dim,
```

```
       att_emp_org
WHERE  ( cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
         AND cbe_emp.start_date <= current_date
         AND cbe_emp.end_date > current_date )
   AND ( att_emp_org.be_id = cbe_emp.emp_org_parent
         AND att_emp_org.start_date <= current_date
         AND att_emp_org.end_date > current_date )
   AND ( att_lv_pln_typ.be_id = v_lv_pln_usge_fact.lv_pln_typ
         AND att_lv_pln_typ.start_date <= current_date
         AND att_lv_pln_typ.end_date > current_date )
   AND ( att_lv_pln.be_id = v_lv_pln_usge_fact.lv_pln
         AND att_lv_pln.start_date <= current_date
         AND att_lv_pln.end_date > current_date )
   AND ( LV_SEG_LV_TYP.be_id = v_lv_pln_usge_fact.lv_typ
         AND LV_SEG_LV_TYP.start_date <= current_date
         AND LV_SEG_LV_TYP.end_date > current_date )
   AND ( v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
         AND v_plcy_dim.start_date <= current_date
         AND v_plcy_dim.end_date > current_date )
   AND ( v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
         AND v_cbe_lv_rqst.start_date <= current_date
         AND v_cbe_lv_rqst.end_date > current_date )
   AND ( v_cbe_lv_rqst.scrty_cnstr_cd = '1' )
   AND ( v_plcy_dim.scrty_cnstr_cd = '1' )
   AND ( ( ( v_plcy_dim.case_code ) = '807915' )
         AND ( v_lv_pln_usge_fact.lv_pln_usge_dt <= '08-aug-2012' )
         AND Trim(Substr (att_emp_org.emp_org_cd,
                  Instr (att_emp_org.emp_org_cd, '-')
                  + 1))
             IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       )
GROUP  BY v_cbe_lv_rqst.lv_rqst_cd,
          att_lv_pln.be_name,
          att_lv_pln.lv_pln_cd,
          att_lv_pln_typ.be_name,
          CASE
            WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN (
                 'Not classified at this level' ) THEN
            LV_SEG_LV_TYP.be_name
          END,
          cbe_emp.emp_natl_id;
```

This query was identified by an application team as taking too long and causing backlog in their batch reporting system. It does some stuff with leave calculations, whatever that is, summing up used and remaining leave days by some key. It is always nice to know in business terms what the query does, particularly when you start looking at rewrites.

As just noted, if the query is not already formatted, then begin the process of getting to know the query by formatting it for neatness. Everyone has their own approach to formatting. I recommend something simple such as TOAD formatter, or an online web site that will format SQL for you.

Next, follow up with some organization of the SQL in addition to simple formatting.

```
SELECT v_cbe_lv_rqst.lv_rqst_cd,
       att_lv_pln.be_name,
       att_lv_pln.lv_pln_cd,
       att_lv_pln_typ.be_name,
       CASE
         WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN ( 'Not classified at this level'
                                             ) THEN
         LV_SEG_LV_TYP.be_name
       END seg_be_name,
       cbe_emp.emp_natl_id,
       SUM (Nvl (v_lv_pln_usge_fact.hrs_used_diff, 0)),
       SUM (Nvl (v_lv_pln_usge_fact.hrs_rmn_diff, 0))
FROM   v_cbe_lv_rqst,
       att_lv_pln,
       att_lv_pln_typ,
       att_lv_typ LV_SEG_LV_TYP,
       cbe_emp,
       v_lv_pln_usge_fact,
       v_plcy_dim,
       att_emp_org
  WHERE 1 = 1
--
       AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND att_lv_pln_typ.be_id = v_lv_pln_usge_fact.lv_pln_typ
       AND att_lv_pln.be_id = v_lv_pln_usge_fact.lv_pln
       AND lv_seg_lv_typ.be_id = v_lv_pln_usge_fact.lv_typ
       AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
       AND v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
--
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND att_lv_pln_typ.start_date <= current_date
       AND att_lv_pln_typ.end_date > current_date
       AND att_lv_pln.start_date <= current_date
       AND att_lv_pln.end_date > current_date
       AND lv_seg_lv_typ.start_date <= current_date
       AND lv_seg_lv_typ.end_date > current_date
       AND v_plcy_dim.start_date <= current_date
       AND v_plcy_dim.end_date > current_date
       AND v_cbe_lv_rqst.start_date <= current_date
       AND v_cbe_lv_rqst.end_date > current_date
--
       AND v_cbe_lv_rqst.scrty_cnstr_cd = '1'
       AND v_plcy_dim.scrty_cnstr_cd = '1'
       AND v_plcy_dim.case_code = '807915'
       AND v_lv_pln_usge_fact.lv_pln_usge_dt <= '08-aug-2012'
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') + 1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
```

```
GROUP  BY v_cbe_lv_rqst.lv_rqst_cd,
          att_lv_pln.be_name,
          att_lv_pln.lv_pln_cd,
          att_lv_pln_typ.be_name,
          CASE
            WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN (
                 'Not classified at this level' ) THEN
            LV_SEG_LV_TYP.be_name
          END,
          cbe_emp.emp_natl_id;
```

Formatting makes things neat, and neatness counts in this business. If you want to communicate ideas, you do a better job when the page is pretty to look at.

Formatting concepts that can be applied to a query.

- **Liberal use of white space.** White space makes things easier to read so exploit this fact.
- **One thing, one line.** Notice the CASE expression in the select list. It was simple enough that it can fit on one line. So it gets put on one line. Same goes for WHERE clause predicates, and everything else. Having to move across multiple lines to read a single idea is more difficult than reading that same idea from one line. (Yes you see things on multiple lines here, blame that on page size of the book).
- **Organized WHERE clause.** Maybe the most important change to the formatting of the query is that the WHERE clause has been reorganized. This organization makes it easier to see what lines are JOINS and what lines are CONSTANT TESTS.

Notice the expression WHERE 1 = 1. This is a do nothing line but it makes it possible to move all the other lines around and this freedom to move stuff around makes organizing the rest of the WHERE clause easier. It will also make doing additional steps in FRP easier as we will soon see.

Notice that meaningless parenthesis have been removed. These get in the way of easy comprehension of the code when they exist but do nothing. It is presumed that the existence of open/close parenthesis pairs means that there is some logic that requires them. This in turn makes us look for that logic. But this is a waste of time for this query, since there is no such logic that requires these parenthesis pairs. The entire query is CONJUNCTIVE FORM (ALL AND) so there is no need for them.

Notice we have grouped the WHERE clause into three sections. This is very convenient for understanding what is happening in the WHERE clause, and it will make additional steps in FRP easier to do. We can see a section for joins, and two sections for CONSTANT TESTS against the tables. Normally we would only need one such section of each type, but because of the repeating nature of the tests, two seemed appropriate in this case.

Notice also that the lines in the WHERE clause have been ordered by table name (more or less). Another convenience that we will be able to exploit in a moment.

It may seem that we have spent considerable time formatting this query, and we have, twenty minutes in fact. But it will pay dividends. This in itself is an important lesson to learn: if you are not willing to spend the time to do things like this, then maybe you are not meant to be a SQL Tuner. Tuning takes time. If you don't have the time or patience to do something like format a query BEFORE you work with it, then you should consider going into a different line of work.

The purpose of formatting is to aid in comprehension and communication. Anything you can do to improve your ability to comprehend what you are working with is a good thing. Anything you can do to improve your

ability to communicate what you are thinking to someone else is even better. Good formatting helps with both. A formatted query will also make other steps in the FRP process much easier to do so even if we are thinking selfishly, we want to format the query before we work with it.

2. Familiarize yourself with the problem query

After the query is formatted, familiarization is the next step, so please familiarize yourself with the problem query. Do this by getting a description of the query. You want the description in business terms. You know what the code does. But what is the business problem being solved? Then look for mistakes. Many a query bug has been found on the way towards optimizing a too-slow SQL statement. Finally, take note of anything else unusual that happens to catch your eye.

Get a Description

Get a description in business terms of what the query is supposed to be doing. You won't use this description now, but it does make for a better presentation later when you have to explain to others what you did, and it will be useful if you should reach the point where you have to start considering various rewrites of the query. For the example query, we have already obtained a description from the developer of what the query does. The developer remarked that the query sums up used and remaining leave days by employee and plan and other keys. This description is a bit sparse but good enough for now I suppose.

Additionally we need at least some statistics on how much work this problem query is doing. Usually the best you can hope for is an ELAPSED TIME metric from a recent problem run. In this case we were told that this query was taking 22 minutes to complete. This will be very important information because any alternatives we offer must be faster and ideally much faster than 22 minutes.

Look for Mistakes

Look for mistakes in the query. There is no sense in spending a lot of time trying to tune a query with obvious mistakes. The sad truth is about one in ten queries you will be looking at have significant mistakes; mistakes that make the answer they are giving suspect. The owners of these queries will not like being told their query has an error but you are justified in not spending time in doing a job that you will only have to re-do later once they fix their mistake. Who knows, they might fix their performance problem at the same time as fixing the mistake.

Our example query does not appear to have any mistakes, but it does seem a little sloppy in places. For example, look at the following portions of the query:

```
  SELECT
       CASE
         WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN ( 'Not classified at this level'
                                             ) THEN
         LV_SEG_LV_TYP.be_name
       END seg_be_name,
…
       SUM (Nvl (v_lv_pln_usge_fact.hrs_used_diff, 0)),
       SUM (Nvl (v_lv_pln_usge_fact.hrs_rmn_diff, 0));
```

The case expression has been given an alias. The sum expressions have not. It does make me wonder how the query is actually used if there are no column names that can be referenced once the final result is produced. Is this query view text that was extracted from the database? Is it a query passed back to a java app that uses ordinal placement instead of column names to retrieve values? Or was the developer just

sloppy? We do not know. We may not care either. But this apparent sloppiness does make me pay a little more attention suspecting that this query may have been put together somewhat hastily or maybe by mindless cut/paste coding.

Check for the Unusual

Look for other things that seem unusual. This would be anything you fancy from your experience; User Defined Functions, Use of Views, Interesting Sub-Queries, etc.

In the example query we see what appears to be at least three views being referenced. This could prove to be a complicating factor later if we determine that the cause of a slowdown is in a view somewhere. We note the views and keep looking.

```
FROM   v_cbe_lv_rqst,
       att_lv_pln,
       att_lv_pln_typ,
       att_lv_typ LV_SEG_LV_TYP,
       cbe_emp,
       v_lv_pln_usge_fact,
       v_plcy_dim,
       att_emp_org
```

We also see that there are eight objects referenced in the FROM clause, but only five objects from which we ultimately select data in our SELECT list. This means that three of the objects in the FROM clause are either pass through objects, or existential objects. We must either go through an object in order to join to another object or we are using a join to an object only to filter rows from (or possibly duplicate rows in) the answer. Some of this information could be useful should we need to look at indexing strategies for this query.

```
SELECT v_cbe_lv_rqst.lv_rqst_cd,
       att_lv_pln.be_name,
       att_lv_pln.lv_pln_cd,
       att_lv_pln_typ.be_name,
       CASE
         WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN ( 'Not classified at this level'
                                             ) THEN
         LV_SEG_LV_TYP.be_name
       END seg_be_name,
       cbe_emp.emp_natl_id,
       SUM (Nvl (v_lv_pln_usge_fact.hrs_used_diff, 0)),
       SUM (Nvl (v_lv_pln_usge_fact.hrs_rmn_diff, 0))
```

We notice that the query has a GROUP BY clause, but no ORDER BY clause. I am pretty sure this is a query from a 9i database. And I am also pretty sure the application system this query comes from is headed for an EXADATA platform (11gR2). So I have to ask, is this another case of sloppy development? Is the lack of an ORDER BY clause reflective of the fact that ordering is not needed because the receiver of the data really does not care about the final ordering of rows from this query? Or will this application be calling me in six months to debug why their ordered data is suddenly no longer ordered after they migrated to their new platform, because Oracle 11g has group by strategies that use hashing instead of sorting. These do not exist in 9i, so reliance on GROUP BY to order data while working in 9i often won't work in 11g.

Lastly we notice some suspicious constant values in the WHERE clause:

```
  WHERE 1 = 1
…
```

```
       AND v_plcy_dim.case_code = '807915'
…
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') + 1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
```

Are those quoted strings really business data? They look awfully like surrogate key values from a sequence number generator. Has someone gone and started using surrogate keys as real data? Maybe this is just a query that was generated as an intermediary step by some transformation process. In fact that is exactly what this query is but it was scary for a moment, and tells me choice of column names was maybe not great.

But enough of this. The purpose of looking at the query, BEFORE you do anything with it, is to gain some appreciation for what it is and what it has suffered through to become the problem that it is. Remember our analogy of gopher shopping wherein our gopher was taking advantage of knowledge of the environment to get the best travel route (is this alternative route a bus route, etc.). Reviewing a query is your chance to acquire similar knowledge. It is an opportunity to start thinking about traffic patterns so to speak and bus routes, and stop signs, etc. so take advantage of it and review the query.

3. Create a spreadsheet

Now that we have dispensed with formatting and familiarization with the query at hand, we can begin the process of creating the actual FRP spreadsheet. The FRP spreadsheet will eventually list out various types of row counts for all tables in the problem query. The first step is to create an empty spreadsheet. If not using the supplied script (and at the moment we are not) then we normally would use Excel for this. But for various "production" reasons of this book we are going to simplify this and use simple text reports as our spreadsheet. This is OK as eventually that is what the automated script will give us anyway.

```
OWNER           TABLE_NAME
--------------- ---------------
```

That was easy. This spreadsheet is one of the two main artifacts that is going to help us determine our DRIVING TABLE and JOIN ORDER. We will be adding various columns in steps we are about to see, and updating and sorting rows as we go. The final result will be a spreadsheet that gives us our DRIVING TABLE and JOIN ORDER based on how rows are filtered out of the result set. The first step of course is to start an empty spreadsheet.

4. List tables from the query in the spreadsheet

Next we create a listing of tables. Notice that the spreadsheet has two columns so far, one for the table owner and one for the table name. We list tables at the moment in alphabetical order for convenience. This will change as we add more columns to the spreadsheet.

```
    FROM   V_CBE_LV_RQST,
           ATT_LV_PLN,
           ATT_LV_PLN_TYP,
           ATT_LV_TYP LV_SEG_LV_TYP,
           CBE_EMP,
           V_LV_PLN_USGE_FACT,
           V_PLCY_DIM,
           ATT_EMP_ORG

OWNER  TABLE_NAME
```

```
------ ------------------
WHSUSR ATT_EMP_ORG
WHSUSR ATT_LV_PLN
WHSUSR ATT_LV_PLN_TYP
WHSUSR ATT_LV_TYP
WHSUSR CBE_EMP
WHSUSR V_CBE_LV_RQST
WHSUSR V_LV_PLN_USGE_FACT
WHSUSR V_PLCY_DIM
```

We get a list of tables used by the query and drop these on our spreadsheet. We might also be interested in who owns these objects and whether these objects are tables or views or synonyms, and if an ALIAS was used in the query for any of them. For brevity I usually keep this information out of the working spreadsheet but here is a query that will garner some of this information for you. From it we will learn that PUBLIC SYNONYMS are in use (not a good idea if you are moving to a shared platform), and that user WHSUSR is the owner for these objects.

```
col object_name format a30

select owner,object_type,object_name
from dba_objects
where object_name = upper('&&1')
order by 1,2,3
/

17:29:41 SQL> @showowner V_CBE_LV_RQST

OWNER                          OBJECT_TYPE         OBJECT_NAME
------------------------------ ------------------- --------------------------
PUBLIC                         SYNONYM             V_CBE_LV_RQST
WHSUSR                         VIEW                V_CBE_LV_RQST

2 rows selected.

17:29:50 SQL> @showowner ATT_LV_PLN

OWNER                          OBJECT_TYPE         OBJECT_NAME
------------------------------ ------------------- --------------------------
PUBLIC                         SYNONYM             ATT_LV_PLN
WHSUSR                         TABLE               ATT_LV_PLN
WHS_VIEWER                     SYNONYM             ATT_LV_PLN

3 rows selected.

… remaining omitted.
```

Note use of the SHOWOWNER script used to obtain the existential and ownership information on an object. We have skipped many of the objects used in the query to save space. Adding these additional details, the FRP spreadsheet now contains four columns. Not all columns are always needed.

```
OWNER  TABLE_NAME         OBJECT_TYPE ALIAS
------ ------------------ ----------- --------------
WHSUSR ATT_EMP_ORG        TABLE
WHSUSR ATT_LV_PLN         TABLE
```

```
WHSUSR ATT_LV_PLN_TYP     TABLE
WHSUSR ATT_LV_TYP         TABLE       LV_SEG_LV_TYPE
WHSUSR CBE_EMP            TABLE
WHSUSR V_CBE_LV_RQST      VIEW
WHSUSR V_LV_PLN_USGE_FACT VIEW
WHSUSR V_PLCY_DIM         VIEW
```

5. Note the row count for each table

We are going to add a ROWS column to our spreadsheet and note in it the number of rows in each table.

We can get this information two ways:

- From NUM_ROWS in DBA_TABLES
- By counting the rows in each table

NUM_ROWS is easier, but counting is more accurate and in the case of views or other composite objects may be the only way to get a row count. The optimizer will normally be using NUM_ROWS when estimating CARDINALITY of plan steps (though dynamic sampling changes this). If we count rows, we need to construct COUNT QUERIES for each table. These are the COUNT QUERIES we need to get row counts for tables in the problem query.

```
select count(*) rowcount,'ATT_EMP_ORG' from ATT_EMP_ORG;
select count(*) rowcount,'ATT_LV_PLN' from ATT_LV_PLN;
select count(*) rowcount,'ATT_LV_PLN_TYP' from ATT_LV_PLN_TYP;
select count(*) rowcount,'LV_SEG_LV_TYP' from ATT_LV_TYP;
select count(*) rowcount,'CBE_EMP' from CBE_EMP;
select count(*) rowcount,'V_CBE_LV_RQST' from V_CBE_LV_RQST;
select count(*) rowcount,'V_LV_PLN_USGE_FACT' from V_LV_PLN_USGE_FACT;
select count(*) rowcount,'V_PLCY_DIM' from V_PLCY_DIM;

13:28:00 SQL> select count(*) rowcount,'ATT_EMP_ORG' from ATT_EMP_ORG;

  ROWCOUNT 'ATT_EMP_OR
---------- -----------
     10875 ATT_EMP_ORG

1 row selected.
```

Above you see the COUNT QUERIES for each table in our query. You also see the result of executing the first count query. For brevity, the rest of the count executions have been omitted and I filled in the results in the spreadsheet after running each count query.

Notice the FRP spreadsheet now contains a filled in ROWS column so we now know the number of rows in each table. As a hint, if we sort the spreadsheet by TABLE_NAME so that the order on the spreadsheet matches the order in which we will run our generated COUNT QUERIES, it is easier to plug the resulting numbers into the spreadsheet.

```
OWNER  TABLE_NAME         OBJECT_TYPE ALIAS          ROWS
------ ------------------ ----------- -------------- -------
WHSUSR ATT_EMP_ORG        TABLE                       10875
WHSUSR ATT_LV_PLN         TABLE                         844
WHSUSR ATT_LV_PLN_TYP     TABLE                          15
```

```
WHSUSR ATT_LV_TYP          TABLE       LV_SEG_LV_TYPE       8
WHSUSR CBE_EMP             TABLE                       8079309
WHSUSR V_CBE_LV_RQST       VIEW                         613431
WHSUSR V_LV_PLN_USGE_FACT  VIEW                        4387560
WHSUSR V_PLCY_DIM          VIEW                        3657382
```

At this point we have two things to consider:

- We have a moderately useful piece of information. From the spreadsheet we have a clear idea of which tables in the query are big and which ones are small (relative to each other). We are starting to learn about where resources may be spent on this query.
- We have ignored the potential complexity that VIEWS could throw into our process. Most of the time when building a FRP spreadsheet, we can treat views as tables. Each is just a source of rows. But there may be times when it becomes necessary to tune individual views in order to successfully tune a query. At this moment, it would be a mistake to assume that any of these views was the root of a performance issue with this query. This also demonstrates how NUM_ROWS from the dictionary is not sufficient. There is no NUM_ROWS for views so we are required to run a count query to get the row count we need.

6. Build and run FILTER QUERIES for each table

Now that we know how many rows are in each table, we next want to know how many rows actually feed into query processing from each table. We use what are called FILTER QUERIES to figure this out. Filter Queries are queries that show us for some specific table, how many rows will remain AFTER FILTERING, if all we did was use the filtering criteria found in the WHERE clause for that table. We learn how many rows remain after filtering. For a table in the query, we examine the WHERE clause looking for CONSTANT TESTS against the table. We then construct a COUNT (*) query against the table and add to it only these constant tests. This gives us a Filter Query.

Since every table in this query has at least one filter expression that references a constant, we will need one filter query for each table for a total of eight filter queries. If there are no filter criteria in the WHERE clause for a table in the query, then there is no need to construct a filter query for that table since the filtered row count will be the same as the number of rows in the table and we already have that information.

Here are the eight filter queries for our problem query. Notice there is a filter query for each table that has associated filtering predicates in the WHERE clause that use constants. We are able to build a filter query by examining the WHERE clause looking for those filtering predicates based on constants that are applied to a specific table. You can see for example, how for table ATT_EMP_ORG, we took the three predicates from the WHERE clause such that the predicates referenced the table, and they are based on CONSTANT TEST filtering. For those who do not know it, CURRENT_DATE is an Oracle environment variable like SYSDATE.

```
SELECT Count(*) rowcount,
       'ATT_EMP_ORG'
FROM   att_emp_org
WHERE  1 = 1
   AND att_emp_org.start_date <= current_date
   AND att_emp_org.end_date > current_date
   AND Trim(Substr (att_emp_org.emp_org_cd,
            Instr (att_emp_org.emp_org_cd, '-') + 1))
       IN ( '4167781', '4167779', '4167777', '4167783', '4167785' );
```

```
SELECT Count(*) rowcount,
       'ATT_LV_PLN'
FROM   att_lv_pln
WHERE  1 = 1
   AND att_lv_pln.start_date <= current_date
   AND att_lv_pln.end_date > current_date;

SELECT Count(*) rowcount,
       'ATT_LV_PLN_TYP'
FROM   att_lv_pln_typ
WHERE  1 = 1
   AND att_lv_pln_typ.start_date <= current_date
   AND att_lv_pln_typ.end_date > current_date;

SELECT Count(*) rowcount,
       'LV_SEG_LV_TYP,'
FROM   att_lv_typ lv_seg_lv_typ
WHERE  1 = 1
   AND lv_seg_lv_typ.start_date <= current_date
   AND lv_seg_lv_typ.end_date > current_date;

SELECT Count(*) rowcount,
       'CBE_EMP'
FROM   cbe_emp
WHERE  1 = 1
   AND cbe_emp.start_date <= current_date
   AND cbe_emp.end_date > current_date;

SELECT Count(*) rowcount,
       'V_CBE_LV_RQST'
FROM   v_cbe_lv_rqst
WHERE  1 = 1
   AND v_cbe_lv_rqst.start_date <= current_date
   AND v_cbe_lv_rqst.end_date > current_date
   AND v_cbe_lv_rqst.scrty_cnstr_cd = '1';

SELECT Count(*) rowcount,
       'V_LV_PLN_USGE_FACT'
FROM   v_lv_pln_usge_fact
WHERE  v_lv_pln_usge_fact.lv_pln_usge_dt <= '08-aug-2012';

SELECT Count(*) rowcount,
       'V_PLCY_DIM'
FROM   v_plcy_dim
WHERE  1 = 1
   AND v_plcy_dim.start_date <= current_date
   AND v_plcy_dim.end_date > current_date
   AND v_plcy_dim.case_code = '807915';
```

Running all eight filtering queries yields filtered row counts for each table. A filtered row count is the number of rows that remain after filtering predicates have been applied. We see below that the filter query for table ATT_EMP_ORG tells us that after filtering using only the CONSTANT TEST predicates applied to the table, we will get only five rows from ATT_EMP_ORG feeding into the query.

```
11:00:52 SQL> SELECT Count(*) rowcount,
11:00:53   2         'ATT_EMP_ORG'
11:00:53   3  FROM   att_emp_org
11:00:53   4  WHERE  1 = 1
11:00:53   5     AND att_emp_org.start_date <= current_date
11:00:53   6     AND att_emp_org.end_date > current_date
11:00:53   7     AND Trim(Substr (att_emp_org.emp_org_cd,'
                          Instr (att_emp_org.emp_org_cd, '-') + 1)) IN (
11:00:53   8         '4167781', '4167779', '4167777', '4167783', '4167785' );

  ROWCOUNT 'ATT_EMP_OR
---------- -----------
         5 ATT_EMP_ORG

1 row selected.
```

For brevity, the rest of the Filter Query executions have been omitted. I ran them all. They generated the FILTERED ROWS column of the FILTERED ROWS PERCENTAGE spreadsheet and we will add these filtered row counts in the next step.

7. Note the filtered row counts

Having executed all the filter queries, we take the resulting row count and put each into the spreadsheet by table. Notice we have added the additional column FILTERED_ROWS and dropped our filtered row count for each table into this column.

OWNER	TABLE_NAME	OBJECT TYPE	ALIAS	ROWS	FILTERED ROWS
WHSUSR	ATT_EMP_ORG	TABLE		10875	5
WHSUSR	ATT_LV_PLN	TABLE		844	841
WHSUSR	ATT_LV_PLN_TYP	TABLE		15	15
WHSUSR	ATT_LV_TYP	TABLE	LV_SEG_LV_TYPE	8	7
WHSUSR	CBE_EMP	TABLE		8079309	657370
WHSUSR	V_CBE_LV_RQST	VIEW		613431	158041
WHSUSR	V_LV_PLN_USGE_FACT	VIEW		4387560	4091046
WHSUSR	V_PLCY_DIM	VIEW		3657382	5

This starts to look interesting, but let us do the next step before commentary.

8. Compute FILTERED ROWS PERCENTAGE for each table

Now that we know for each table, how many rows there are in the table, and how many rows are filtered out of the result based on INITIAL FILTER CRITERIA found in the WHERE clause, we can generate a statistic that gives us a good idea of the EFFICIENCY of the filtering done against each table. We can compute the FILTERED ROWS PERCENTAGE statistic for each table in this query. This is the magic number that tells us how well our WHERE clause is filtering data from tables.

To do this we add a column to the spreadsheet called FILTERED ROWS % and we fill it in using the simple Excel formula ROUND ((1-Y/X)*100, 0). So for table ATT_EMP_ORG, this is (1-5/10875)*100 = 99.95% which when we round to the nearest integer gives us 100%. For the table ATT_EMP_ORG, 100% of the data has been removed from consideration by the query's filtering criteria. 100% is not an exact number because

of our rounding. We know from the spreadsheet that we will be getting five (5) rows from this table. But this number is so small compared to the total number of rows in the table that the effective filtering is near 100%.

An alternative calculation and the one that the automated script uses, is to calculate the % of rows remaining after filtering. It is the same basic info, but just how the script does it. The advantage of the alternative is that it emphasizes the idea of eliminating rows as quickly as possible. But for now we use this first calculation.

OWNER	TABLE_NAME	OBJECT TYPE	ALIAS	ROWS	FILTERED ROWS	FILTERED ROWS %
WHSUSR	ATT_EMP_ORG	TABLE		10875	5	100
WHSUSR	ATT_LV_PLN	TABLE		844	841	0
WHSUSR	ATT_LV_PLN_TYP	TABLE		15	15	0
WHSUSR	ATT_LV_TYP	TABLE	LV_SEG_LV_TYPE	8	7	13
WHSUSR	CBE_EMP	TABLE		8079309	657370	92
WHSUSR	V_CBE_LV_RQST	VIEW		613431	158041	74
WHSUSR	V_LV_PLN_USGE_FACT	VIEW		4387560	4091046	7
WHSUSR	V_PLCY_DIM	VIEW		3657382	5	100

At this point, we now have very interesting data to work with. We can see from the FILTERED ROWS PERCENTAGE column that there are some tables for which initial filtering criteria will remove almost all the rows from consideration, and some tables from which we eliminate almost none of the rows from consideration. Recall from our discussion of DRIVING TABLE that one of the ramifications of DRIVING TABLE selection was the initial number of rows feeding into query execution, and that these rows drove the amount of work for the rest of the query. As such, it makes sense for us to select a DRIVING TABLE for which most of the data in the table is removed.

Based on the FILTERED ROWS PERCENTAGE data, we see there are two tables that look promising as DRIVING TABLES: (ATT_EMP_ORG and V_PLCY_DIM). It may be easier to see this if we sort the rows in our spreadsheet by FILTERED ROWS PERCENTAGE descending. Again please recall that for the time being we are treating VIEWS as TABLES; they both are row sources for this query.

9. Determine PREFERRED JOIN ORDER

Sorting by FILTERED ROWS % column descending (and then ROWS ascending if we like) we get our PREFERRED JOIN ORDER.

OWNER	TABLE_NAME	OBJECT TYPE	ALIAS	ROWS	FILTERED ROWS	FILTERED ROWS %
WHSUSR	ATT_EMP_ORG	TABLE		10875	5	100
WHSUSR	V_PLCY_DIM	VIEW		3657382	5	100
WHSUSR	CBE_EMP	TABLE		8079309	657370	92
WHSUSR	V_CBE_LV_RQST	VIEW		613431	158041	74
WHSUSR	ATT_LV_TYP	TABLE	LV_SEG_LV_TYPE	8	7	13
WHSUSR	V_LV_PLN_USGE_FACT	VIEW		4387560	4091046	7
WHSUSR	ATT_LV_PLN	TABLE		844	841	0
WHSUSR	ATT_LV_PLN_TYP	TABLE		15	15	0

Recall from our discussion of JOIN ORDER that our goal for picking a join order was to keep the size of intermediary row-sets as small as possible throughout query execution, and that there were two ways to achieve this. Also recall that by using FILTERED ROWS PERCENTAGE we were opting for choice #2.

2) Pick a join sequence that is likely to remove the highest percentage of rows from consideration as soon as possible.

If we can process our tables in the order listed above in our spreadsheet, we are highly likely to be removing the maximum amount of data from consideration as soon as is possible during query execution. By using this join order, we will be reducing the number of rows joined, and number of key comparisons and data comparisons required in subsequent steps, and reducing the number of rows floated by the query as the query progresses in joining to each table in turn. So ATT_EMP_ORG looks to be our choice for DRIVING TABLE because it is first in JOIN ORDER and the JOIN ORDER shown above is our PREFERRED JOIN ORDER for all tables in the query.

However, it is very likely that we cannot simply access tables in the order listed above. This is because the joins in the query determine what join sequences are valid for this query and our PREFERRED JOIN ORDER may not be a valid join sequence for this query. So although the above ordering may well be our PREFERRED JOIN ORDER, it will likely need to be modified to account for the valid join sequences the query allows. We have to somehow account for the joins in our query when considering our join order.

10. Construct a QUERY DIAGRAM

To assist us in determining what join sequences are valid and which are not and thus help us modify our PREFERRED JOIN ORDER so that it reflects a valid join sequence, we can construct a QUERY DIAGRAM that visually shows us what we need to know.

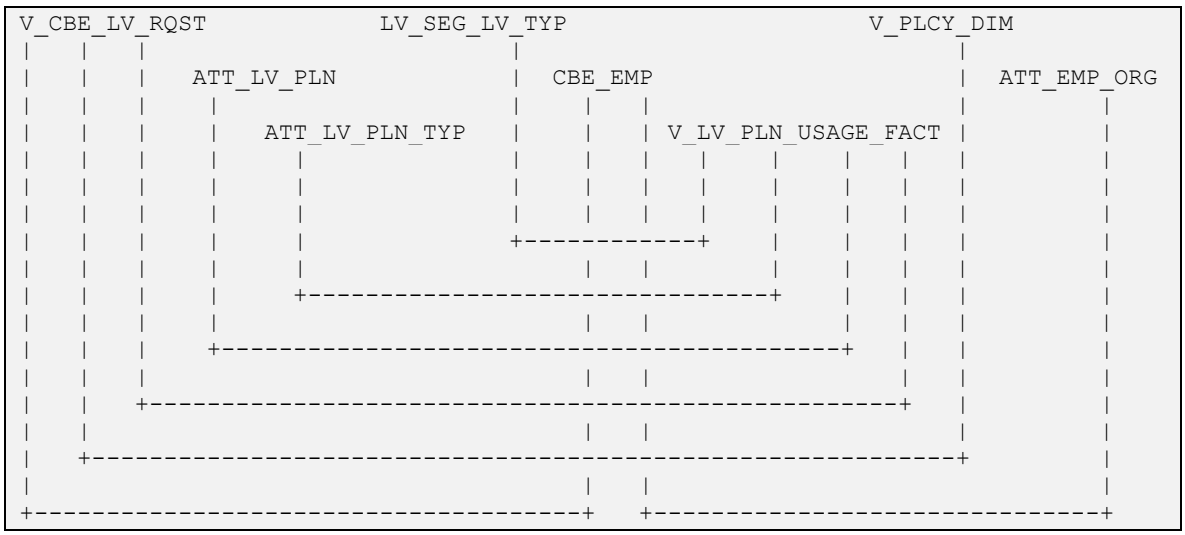
To construct a query diagram, follow these steps.

- List tables in a straight line across the page, with even space between them, in the top down order as seen in the FROM clause
- Examine the WHERE clause for joins between tables and draw lines to connect the tables that are joined
- If desired, examine the WHERE clause for constant tests against tables and draw a vertical line from a filtered table down and label it with (c)
- If desired, note outer-joins using (+) syntax
- If desired, annotate the diagram with spreadsheet data (PREFERRED JOIN ORDER, ROWS, FILTERED ROWS, FILTERED ROWS %, and whatever makes sense to you)
- Use the alias for a table name on the query diagram if you need to save space or to eliminate ambiguity.

This is one place where our SQL formatting will start to pay dividends. Recall that we organized the WHERE clause such that all the joins between tables were in one place.

```
AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
AND att_emp_org.be_id = cbe_emp.emp_org_parent
AND att_lv_pln_typ.be_id = v_lv_pln_usge_fact.lv_pln_typ
AND att_lv_pln.be_id = v_lv_pln_usge_fact.lv_pln
AND lv_seg_lv_typ.be_id = v_lv_pln_usge_fact.lv_typ
AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
AND v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
```

We can exploit this when constructing our query diagram. How much information we put on the diagram depends upon our use for it. At this point we only need the join information so we will limit our query diagram to containing only the join information. The diagram gives us a visual representation of the joins between tables in our query.



Using the QUERY DIAGRAM, it is easy to see that when positioned at table X, only a subset of other tables are immediately accessible via a join path. For example, if we position ourselves to table ATT_LV_PLN (second from the left) then we can only move to table V_LV_PLN_USGE_FACT (just follow the line coming out of ATT_LV_PLN). But if we did the opposite and position ourselves on V_LV_PLN_USGE_FACT (third from the right) then we could go to any of LV_SEG_LV_TYP or ATT_LV_PLN or ATT_LV_PLN_TYP or V_CBE_LV_RQST (again, just follow the lines).

Please note that I had to offset the table names in the diagram to make it easier to fit on the page without reducing font sizes. This has no meaning to the data.

11. Determine INITIAL JOIN ORDER

With this QUERY DIAGRAM, it is now possible to merge our PREFERRED JOIN ORDER with the valid join sequences as shown on the diagram, to determine a valid join order that most closely aligns with our PREFERRED JOIN ORDER. We will call this our INITIAL JOIN ORDER and we will use it to start runtime testing.

The FILTERED ROWS PERCENTAGE spreadsheet tells us that our DRIVING TABLE will be ATT_EMP_ORG. It also tells us that the next table we want to visit is V_PLCY_DIM. But by examining the query diagram, it is clear there is no direct path to V_PLCY_DIM from ATT_EMP_ORG. We would like to join from ATT_EMP_ORG directly to V_PLCY_DIM but this is not a valid join sequence based on the query diagram. Instead we must go through other tables to get from ATT_EMP_ORG to V_PLCY_DIM. Specifically, we must follow the joins in the diagram such that we do this (this is a JOIN SENTENCE):

ATT_EMP_ORG → CBE_EMP → V_CBE_LV_RQST → V_PLCY_DIM.

To reach V_PLCY_DIM from ATT_EMP_ORG we must first visit ATT_EMP_ORG, then visit CBE_EMP, then visit V_CBE_LV_RQST, which will then allow us to visit V_PLCY_DIM. This is what we mean when we say "valid". The PREFERRED JOIN ORDER does not present a valid join sequence based on the valid joins in the query, which we can easily see using the query diagram.

In a similar fashion, we go through the motions and follow our PREFERRED JOIN ORDER, one table at a time till we have accounted for all tables. So after making our way to V_PLCY_DIM, we check our spreadsheet to see that the next table in our PREFERRED JOIN ORDER is CBE_EMP, but we have already visited this table as a result of seeking V_PLCY_DIM, so we can move on. The next preferred table is V_CBE_LV_RQST but we have already visited this table so we can move on. The next preferred table is ATT_LV_TYP (alias LV_SEG_LV_TYP). We have not visited this table yet so we need to add it to our valid join sequence. However once again there is no direct link between any of the tables we have already visited and LV_SEG_LV_TYP so we have to follow the joins to get to it. Extending our join sequence accordingly we come to:

ATT_EMP_ORG → CBE_EMP → V_CBE_LV_RQST → V_PLCY_DIM → V_LV_PLN_USGE_FACT → LV_SEG_LV_TYP.

So from V_PLCY_DIM we visit V_LV_PLN_USGE_FACT and then we are allowed to visit ATT_LV_TYP (alias LV_SEG_LV_TYP).

After visiting ATT_LV_TYP (alias LV_SEG_LV_TYP), the next preferred table is V_LV_PLN_USGE_FACT but we have already visited this table so we can move on. The next preferred table is ATT_LV_PLN_TYP which is directly accessible from one of the tables we have already visited so we extend our join sequence again with:

ATT_EMP_ORG → CBE_EMP → V_CBE_LV_RQST → V_PLCY_DIM → V_LV_PLN_USGE_FACT → LV_SEG_LV_TYP → ATT_LV_PLN_TYP.

After visiting ATT_LV_PLN_TYPE, the next preferred table is ATT_LV_PLN which is also now accessible from at least one table we have already visited so we extend our join sequence with that table giving us:

ATT_EMP_ORG → CBE_EMP → V_CBE_LV_RQST → V_PLCY_DIM → V_LV_PLN_USGE_FACT → LV_SEG_LV_TYP → ATT_LV_PLN_TYP → ATTT_LV_PLN.

This was the last table in our PREFERRED JOIN ORDER so we have now a JOIN SENTENCE that we arrived at by following our PREFERRED JOIN ORDER as closely as possible but which we modified to account for valid join paths as seen in our query diagram. This gives us our INITIAL JOIN ORDER. After going through all that work, we do not want to lose this information so, we add a column to our spreadsheet to show the INITIAL JOIN ORDER.

We see that the INITIAL JOIN ORDER is not the same as the PREFERRED JOIN ORDER but it is not too far off either. A few tables have switched positions. We can reorder the spreadsheet by INITIAL JOIN ORDER to aid the next step. In interest of space, I have started dropping columns we no longer need from the report.

TABLE_NAME	OBJECT ALIAS	ROWS	FILTERED ROWS	FILTERED ROWS %	PREFERRED JOIN ORDER	INITIAL JOIN ORDER
ATT_EMP_ORG		10875	5	100	1	1
CBE_EMP		8079309	657370	92	3	2
V_CBE_LV_RQST		613431	158041	74	4	3
V_PLCY_DIM		3657382	5	100	2	4
V_LV_PLN_USGE_FACT		4387560	4091046	7	6	5
ATT_LV_TYP	LV_SEG_LV_TYPE	8	7	13	5	6
ATT_LV_PLN_TYP		15	15	0	7	7
ATT_LV_PLN		844	841	0	8	8

12. Build and run RECONSTRUCTION QUERIES

Reconstruction queries are a series of queries which are used to incrementally reconstruct our initial query by incrementally adding one table at a time from our join order. Each successive query will introduce one new table. Given that we have a join order to use from above, we can rewrite our beginning query into a series of such queries. These queries will allow us to see how each new table adds to the cost of the query. It will confirm for us that we have a good join sequence and after consideration of CARDINALITY FEEDBACK, if there is potentially a better join order.

Below are the reconstruction queries based on our INITIAL JOIN ORDER for our problem query. Notice a few things about them.

- **ORDERED** hint is used to ensure Oracle accesses tables in the order we want so that we can explicitly determine DRIVING TABLE and JOIN ORDER. In 11g we could also use LEADING hint but you will have to modify the hint with every new reconstruction query. I am not suggesting that this hint should be used as the final solution, only that it is a quick and easy way to validate our results.
- **COUNT (*)** is used to get the row count after the join to the new table and thus gives us an indication of workload for the new table. By using only COUNT (*) we potentially invalidate runtime metrics other than row count, but for this step in the process we want the row count returned by each reconstruction query and are not interested in its runtime. The row count will be useful in learning if a better join order might exist than the INITIAL JOIN ORDER we are currently using.
- **GROUP BY** clause has been removed. We are interested in the number of raw rows we visit because of each join, not aggregate rows.

We will have to undo these differences when we do our final confirmation test. But these changes are necessary to build our reconstruction queries. A review of these queries will show that each new reconstruction query adds one new table to the prior reconstruction query. The table added will be the next table in our INITIAL JOIN ORDER from whatever the last table was that we built a reconstruction query for. Obviously when we add a new table to make our next reconstruction query, we also need to add the join criteria and filter criteria that is relevant.

I have not abbreviated the list of reconstruction queries that follows. I want you to see the full reconstruction process and this requires all queries. These could have been put into an appendix but I feel the process example loses something when this detail is removed.

The first table in our INITIAL JOIN ORDER is ATT_EMP_ORG, so this is the first reconstruction query we build.

ATT_EMP_ORG Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') + 1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' );
```

Since CBE_EMP is the second table in our INITIAL JOIN ORDER this reconstruction query is next. We build this reconstruction query by taking the prior reconstruction query and doing whatever we need to do to add the next table into it. We add CBE_EMP to the FROM clause. We add to the WHERE clause, the join from ATT_EMP_ORG to CBE_EMP, and add any constant tests related to the new table that are valid for the totality of tables we have in this reconstruction query (currently only two tables).

CBE_EMP Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org,
       cbe_emp
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') +
                1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       --
       -- CBE_EMP
       --
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date;
```

We do the same for all tables in our problem query (eight in this case). You can see now that once again our formatting of the problem query pays another dividend. These reconstruction queries are much easier to write given that we have organized the WHERE clause by section and table.

V_CBE_LV_RQST Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org,
       cbe_emp,
       v_cbe_lv_rqst
```

```
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') +
                1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       --
       -- CBE_EMP
       --
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date
       --
       -- V_CBE_LV_RQST
       --
       AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
       AND v_cbe_lv_rqst.start_date <= current_date
       AND v_cbe_lv_rqst.end_date > current_date
       AND v_cbe_lv_rqst.scrty_cnstr_cd = '1';
```

V_PLCY_DIM Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org,
       cbe_emp,
       v_cbe_lv_rqst,
       v_plcy_dim
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') +
                1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       --
       -- CBE_EMP
       --
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date
       --
       -- V_CBE_LV_RQST
       --
       AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
       AND v_cbe_lv_rqst.start_date <= current_date
```

```
       AND v_cbe_lv_rqst.end_date > current_date
       AND v_cbe_lv_rqst.scrty_cnstr_cd = '1'
       --
       -- V_PLCY_DIM
       --
       AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
       AND v_plcy_dim.start_date <= current_date
       AND v_plcy_dim.end_date > current_date
       AND v_plcy_dim.scrty_cnstr_cd = '1'
       AND v_plcy_dim.case_code = '807915';
```

V_LV_PLN_USGE_FACT Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org,
       cbe_emp,
       v_cbe_lv_rqst,
       v_plcy_dim,
       v_lv_pln_usge_fact
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') +
                1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       --
       -- CBE_EMP
       --
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date
       --
       -- V_CBE_LV_RQST
       --
       AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
       AND v_cbe_lv_rqst.start_date <= current_date
       AND v_cbe_lv_rqst.end_date > current_date
       AND v_cbe_lv_rqst.scrty_cnstr_cd = '1'
       --
       -- V_PLCY_DIM
       --
       AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
       AND v_plcy_dim.start_date <= current_date
       AND v_plcy_dim.end_date > current_date
       AND v_plcy_dim.scrty_cnstr_cd = '1'
       AND v_plcy_dim.case_code = '807915'
       --
       --
```

```
       AND v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
       AND v_lv_pln_usge_fact.lv_pln_usge_dt <=
           To_date('08-aug-2012', 'dd-mon-rrrr');
```

ATT_LV_TYP (LV_SEG_LV_TYP) Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org,
       cbe_emp,
       v_cbe_lv_rqst,
       v_plcy_dim,
       v_lv_pln_usge_fact,
       att_lv_typ lv_seg_lv_typ
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') +
                1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       --
       -- CBE_EMP
       --
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date
       --
       -- V_CBE_LV_RQST
       --
       AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
       AND v_cbe_lv_rqst.start_date <= current_date
       AND v_cbe_lv_rqst.end_date > current_date
       AND v_cbe_lv_rqst.scrty_cnstr_cd = '1'
       --
       -- V_PLCY_DIM
       --
       AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
       AND v_plcy_dim.start_date <= current_date
       AND v_plcy_dim.end_date > current_date
       AND v_plcy_dim.scrty_cnstr_cd = '1'
       AND v_plcy_dim.case_code = '807915'
       --
       -- V_LV_PLN_USGE_FACT
       --
       AND v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
       AND v_lv_pln_usge_fact.lv_pln_usge_dt <=
           To_date('08-aug-2012', 'dd-mon-rrrr')
       --
       -- ATT_LV_TYP
```

```
       --
       AND lv_seg_lv_typ.be_id = v_lv_pln_usge_fact.lv_typ
       AND lv_seg_lv_typ.start_date <= current_date
       AND lv_seg_lv_typ.end_date > current_date;
```

ATT_LV_PLN_TYP Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org,
       cbe_emp,
       v_cbe_lv_rqst,
       v_plcy_dim,
       v_lv_pln_usge_fact,
       att_lv_typ lv_seg_lv_typ,
       att_lv_pln_typ
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') +
                1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       --
       -- CBE_EMP
       --
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date
       --
       -- V_CBE_LV_RQST
       --
       AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
       AND v_cbe_lv_rqst.start_date <= current_date
       AND v_cbe_lv_rqst.end_date > current_date
       AND v_cbe_lv_rqst.scrty_cnstr_cd = '1'
       --
       -- V_PLCY_DIM
       --
       AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
       AND v_plcy_dim.start_date <= current_date
       AND v_plcy_dim.end_date > current_date
       AND v_plcy_dim.scrty_cnstr_cd = '1'
       AND v_plcy_dim.case_code = '807915'
       --
       -- V_LV_PLN_USGE_FACT
       --
       AND v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
       AND v_lv_pln_usge_fact.lv_pln_usge_dt <=
           To_date('08-aug-2012', 'dd-mon-rrrr')
```

```
       --
       -- ATT_LV_TYP
       --
       AND lv_seg_lv_typ.be_id = v_lv_pln_usge_fact.lv_typ
       AND lv_seg_lv_typ.start_date <= current_date
       AND lv_seg_lv_typ.end_date > current_date
       --
       -- ATT_LV_PLN_TYP
       --
       AND att_lv_pln_typ.be_id = v_lv_pln_usge_fact.lv_pln_typ
       AND att_lv_pln_typ.start_date <= current_date
       AND att_lv_pln_typ.end_date > current_date;
```

ATT_LV_PLN Reconstruction Query

```
SELECT /*+ ORDERED */ Count(*) ROWCOUNT
FROM   att_emp_org,
       cbe_emp,
       v_cbe_lv_rqst,
       v_plcy_dim,
       v_lv_pln_usge_fact,
       att_lv_typ lv_seg_lv_typ,
       att_lv_pln_typ,
       att_lv_pln
WHERE  1 = 1
       --
       -- ATT_EMP_ORG
       --
       AND att_emp_org.start_date <= current_date
       AND att_emp_org.end_date > current_date
       AND Trim(Substr (att_emp_org.emp_org_cd,
                Instr (att_emp_org.emp_org_cd, '-') +
                1))
           IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
       --
       -- CBE_EMP
       --
       AND att_emp_org.be_id = cbe_emp.emp_org_parent
       AND cbe_emp.start_date <= current_date
       AND cbe_emp.end_date > current_date
       --
       -- V_CBE_LV_RQST
       --
       AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
       AND v_cbe_lv_rqst.start_date <= current_date
       AND v_cbe_lv_rqst.end_date > current_date
       AND v_cbe_lv_rqst.scrty_cnstr_cd = '1'
       --
       -- V_PLCY_DIM
       --
       AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
       AND v_plcy_dim.start_date <= current_date
```

```
       AND v_plcy_dim.end_date > current_date
       AND v_plcy_dim.scrty_cnstr_cd = '1'
       AND v_plcy_dim.case_code = '807915'
       --
       -- V_LV_PLN_USGE_FACT
       --
       AND v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
       AND v_lv_pln_usge_fact.lv_pln_usge_dt <=
           To_date('08-aug-2012', 'dd-mon-rrrr')
       --
       -- ATT_LV_TYP
       --
       AND lv_seg_lv_typ.be_id = v_lv_pln_usge_fact.lv_typ
       AND lv_seg_lv_typ.start_date <= current_date
       AND lv_seg_lv_typ.end_date > current_date
       --
       -- ATT_LV_PLN_TYP
       --
       AND att_lv_pln_typ.be_id = v_lv_pln_usge_fact.lv_pln_typ
       AND att_lv_pln_typ.start_date <= current_date
       AND att_lv_pln_typ.end_date > current_date
       --
       -- ATT_LV_PLN
       --
       AND att_lv_pln.be_id = v_lv_pln_usge_fact.lv_pln
       AND att_lv_pln.start_date <= current_date
       AND att_lv_pln.end_date > current_date;
```

We run all the reconstruction queries and note the row count.

```
SELECT /*+ ORDERED */
             COUNT(*) ROWCOUNT,'ATT_EMP_ORG' table_name
  FROM
         ATT_EMP_ORG
 WHERE 1 = 1
--
-- ATT_EMP_ORG
--
   AND att_emp_org.start_date <= current_date
   AND att_emp_org.end_date > current_date
   AND Trim(Substr (att_emp_org.emp_org_cd,
              Instr (att_emp_org.emp_org_cd, '-') + 1))
         IN ( '4167781', '4167779', '4167777', '4167783', '4167785' );

  ROWCOUNT TABLE_NAME
---------- -----------
         5 ATT_EMP_ORG

1 row selected.
```

For brevity, the rest of the SQL executions have been omitted. We can see results in our spreadsheet. Note I have included the elapsed seconds as well though this is not usually necessary and is often misleading because the reconstruction queries are only doing a COUNT(*).

The column JOIN ROWS tells us that these reconstruction queries are returning row counts after a join is done, not just based on filtering criteria.

13. Note reconstruction row counts

We have added two additional columns, JOIN ROWS and ELAPSED SECONDS to our FRP spreadsheet. The later EPLASED SECONDS is how long the reconstruction query took to run. The former (JOIN ROWS) is the row count returned from the reconstruction query and is thus the row count resulting from filtering criteria applied to all tables in the reconstruction query plus any row count change resulting from the actual join to the new table. It is the number of rows after the join to the indicated table.

TABLE_NAME	OBJECT ALIAS	ROWS	FILTERED ROWS	FILTERED ROWS %	INITIAL JOIN ORDER	JOIN ROWS	ELP SEC
ATT_EMP_ORG		10875	5	100	1	5	0
CBE_EMP		8079309	657370	92	2	3270	273
V_CBE_LV_RQST		613431	158041	74	3	598	273
V_PLCY_DIM		3657382	5	100	4	598	273
V_LV_PLN_USGE_FACT		4387560	4091046	7	5	8625	418
ATT_LV_TYP	LV_SEG_LV_TYPE	8	7	13	6	8625	418
ATT_LV_PLN_TYP		15	15	0	7	8625	418
ATT_LV_PLN		844	841	0	8	8625	418

14. Use CARDINALITY FEEDBACK to adjust join order

CARDINALITY FEEDBACK is the use of the count of rows retained after a join (our JOIN ROWS column) to determine if a query plan should be changed. For our purposes in FRP, we are interested in knowing if row counts go down as we progress, and if so, if there is a different join order that would allow us to get to the reduced number of rows faster. For example, if we were able to access the table on line 3 (V_CBE_LV_RQST) before accessing the table from line 2 (CBE_EMP), then doing so would likely be a good idea because the JOIN ROWS for V_CBE_LV_RQST is smaller than that of CBE_EMP. We would likely see further workload reduction by changing the join order accordingly, assuming the JOIN ROWS remained 598 (or at least something smaller than 3270) after we made the switch. For this to be possible, the query diagram has to show a path between tables that makes the alternative join sequence valid. However, for this problem query, there are no alternative join sequences that are valid and for which we also see a reduced row count in a later step. Though the JOIN ROWS column suggests possible join sequence changes, the query diagram for this query clearly indicates that resulting alternatives are not valid join sequences for this problem query. We will see later another example of where we can make changes based on cardinality feedback, and what effect doing so can have. So right now we do nothing with CARDINALITY FEEDBACK except to note what the JOIN ROWS numbers are.

15. Repeat (11) thru (12) once if join order changed

If we had changed the INITIAL JOIN ORDER based on the CARDINALITY FEEDBACK we gained from our RECONSTRUCTION QUERIES, then we would need to build new RECONSTRUCTION QUERIES to account for the change in join sequence and run those queries to see if new results beat the results we just got. But since we are not changing the INITIAL JOIN ORDER, we skip this step and go forward.

16. Determine if further action is necessary

Further Action? That is a pretty open idea. This could mean anything. This is where you start to think about what to change to create a fast query. Without going into lots of detail that would distract us from our discussion of DRIVING TABLE and JOIN ORDER, we can say that we did in fact take further action on this query based on these numbers. It was determined that new indexes were needed. After adding these new indexes, we reran the reconstruction queries and got new numbers.

TABLE_NAME	ROWS	FILTERED ROWS	FILTERED ROWS %	INITIAL JOIN ORDER	JOIN ROWS	ELP SEC	E 2
ATT_EMP_ORG	10875	5	100	1	5	0	0
CBE_EMP	8079309	657370	92	2	3270	273	0
V_CBE_LV_RQST	613431	158041	74	3	598	273	0
V_PLCY_DIM	3657382	5	100	4	598	273	0
V_LV_PLN_USGE_FACT	4387560	4091046	7	5	8625	418	0
ATT_LV_TYP	8	7	13	6	8625	418	0
ATT_LV_PLN_TYP	15	15	0	7	8625	418	0
ATT_LV_PLN	844	841	0	8	8625	418	0

After our changes we affect the desired effect. Note that we have added a second ELAPSED SECONDS column to the spreadsheet (E2). Things are looking much better now. But we should keep in mind that all our reconstruction queries were COUNT (*) queries. We have not run the actual original query (or our version of it). Even if we make no other changes than to define a DRIVING TABLE and JOIN ORDER for the query, we always need to take the additional step of running our modified version of the original query.

Fortunately this proves easy to do. We need only take the last reconstruction query and add the original SELECT list to it and the GROUP BY clause we removed when building our reconstruction queries. We write a CREATE TABLE query to actually run and save results of this query. This will let us run the actual query modified by our changes, and retain the answer. This simulates doing something with the results once collected, and provides opportunity to review the answer for correctness against the original answer. Recall that some of the columns in the SELECT list were expressions without an alias. To do a CREATE TABLE command, we will need to name these. You will also see that there are duplicate column names in the select list. We missed that in our initial evaluation of the query. We add a column alias where necessary for that too.

Note that we have used the NOLOGGING parameter here. This SQL statement is a test of query performance. Creating a table is an easy way to ensure that the query runs to completion and may even allow us to validate results if we have changed the query, but we want to avoid logging overhead if possible.

Note also the use of the ORDERED hint. This chapter is about teaching the importance of DRIVING TABLE and JOIN ORDER. As such we have followed through here with enforcing the DRIVING TABLE and JOIN ORDER we determined was best. In the real world however, what we will really be after, is correcting performance without use of hints if possible. In this case, the hint was actually not necessary once the missing indexes were put in place. However, this chapter is not about how to index. That comes later. Instead, this query demonstrates the power of DRIVING TABLE and JOIN ORDER and how to determine them correctly. In future chapters we will see how to relax the use of the ORDERED hint by correcting issues that prevented the optimizer from selecting a good DRIVING TABLE and JOIN ORDER. For now, we enforce what we want with the hint.

```
CREATE TABLE kevtemp1
nologging AS
  SELECT /*+ ORDERED */ v_cbe_lv_rqst.lv_rqst_cd,
         att_lv_pln.be_name,
```

```
        att_lv_pln.lv_pln_cd,
        att_lv_pln_typ.be_name                          C3,
        CASE
          WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN (
               'Not classified at this level'
                                              ) THEN
          LV_SEG_LV_TYP.be_name
        END                                      seg_be_name,
        cbe_emp.emp_natl_id,
        SUM (Nvl (v_lv_pln_usge_fact.hrs_used_diff, 0)) C1,
        SUM (Nvl (v_lv_pln_usge_fact.hrs_rmn_diff, 0))  C2
FROM    att_emp_org,
        cbe_emp,
        v_cbe_lv_rqst,
        v_plcy_dim,
        v_lv_pln_usge_fact,
        att_lv_typ lv_seg_lv_typ,
        att_lv_pln_typ,
        att_lv_pln
WHERE   1 = 1
        --
        -- ATT_EMP_ORG
        --
        AND att_emp_org.start_date <= current_date
        AND att_emp_org.end_date > current_date
        AND Trim(Substr (att_emp_org.emp_org_cd,
                 Instr (att_emp_org.emp_org_cd, '-') +
                 1))
            IN ( '4167781', '4167779', '4167777', '4167783', '4167785' )
        --
        -- CBE_EMP
        --
        AND att_emp_org.be_id = cbe_emp.emp_org_parent
        AND cbe_emp.start_date <= current_date
        AND cbe_emp.end_date > current_date
        --
        -- V_CBE_LV_RQST
        --
        AND cbe_emp.object_id = v_cbe_lv_rqst.emp_parent
        AND v_cbe_lv_rqst.start_date <= current_date
        AND v_cbe_lv_rqst.end_date > current_date
        AND v_cbe_lv_rqst.scrty_cnstr_cd = '1'
        --
        -- V_PLCY_DIM
        --
        AND v_plcy_dim.be_id = v_cbe_lv_rqst.case_1_parent
        AND v_plcy_dim.start_date <= current_date
        AND v_plcy_dim.end_date > current_date
        AND v_plcy_dim.scrty_cnstr_cd = '1'
        AND v_plcy_dim.case_code = '807915'
        --
        -- V_LV_PLN_USGE_FACT
        --
```

```
         AND v_lv_pln_usge_fact.lv_rqst = v_cbe_lv_rqst.object_id
         AND v_lv_pln_usge_fact.lv_pln_usge_dt <=
             To_date('08-aug-2012', 'dd-mon-rrrr')
         --
         -- ATT_LV_TYP
         --
         AND lv_seg_lv_typ.be_id = v_lv_pln_usge_fact.lv_typ
         AND lv_seg_lv_typ.start_date <= current_date
         AND lv_seg_lv_typ.end_date > current_date
         --
         -- ATT_LV_PLN_TYP
         --
         AND att_lv_pln_typ.be_id = v_lv_pln_usge_fact.lv_pln_typ
         AND att_lv_pln_typ.start_date <= current_date
         AND att_lv_pln_typ.end_date > current_date
         --
         -- ATT_LV_PLN
         --
         AND att_lv_pln.be_id = v_lv_pln_usge_fact.lv_pln
         AND att_lv_pln.start_date <= current_date
         AND att_lv_pln.end_date > current_date
  --
  GROUP  BY v_cbe_lv_rqst.lv_rqst_cd,
            att_lv_pln.be_name,
            att_lv_pln.lv_pln_cd,
            att_lv_pln_typ.be_name,
            CASE
              WHEN LV_SEG_LV_TYP.lv_typ_cd NOT IN (
                   'Not classified at this level' ) THEN
              LV_SEG_LV_TYP.be_name
            END,
            cbe_emp.emp_natl_id;

Table created.

Elapsed: 00:02:55.40
```

This is 175 ELAPSED SECONDS for the original problem query after our modifications for DRIVING TABLE and JOIN ORDER and the new indexes. This is a substantial improvement over the original time and the application team is very happy. Recall we said the original was taking 22 minutes. After our analysis it takes 3 minutes.

We could go even further with this query. We could build more reconstruction queries. We could build reconstruction queries as CREATE TABLE statements that use SELECT list items in them rather than a COUNT(*). This would show us how much accessing the actual table data adds to each step of the query. From this we might be able to deduce if there are other optimizations from a storage perspective that we could exploit (compression being one obvious choice). However part of tuning is to know when to quit. It is not necessary for every query to go as fast as possible. It is necessary for your system to meet its Service Level Agreements. 175 seconds turns out to be good enough to satisfy those who needed satisfying.

Summary #1

Let us take a moment to reflect upon what we have discussed so far. It is not a coincidence that the query plan which manipulates the least amount of data executing a query, is also most often the least expensive query plan for that query. A strategy which seeks to eliminate the most amount of data from consideration in a query as early as possible, is a strategy that directly aims to reduce the total amount of work done by the query to its minimum amount. FILTERED ROWS PERCENTAGE method is a method that does this. It shows us how to select a DRIVING TABLE and a JOIN ORDER which eliminates the maximum number of rows from consideration as early as possible during query execution. It is thus not a coincidence that FRP yields efficient query plans.

FRP does not directly address ACCESS METHOD nor JOIN METHOD choices of a query plan (we will discuss these in later chapters). However, if good choices are made for DRIVING TABLE and JOIN ORDER for a query, then the CBO will almost always make good choices for ACCESS METHOD and JOIN METHOD for the tables and joins in the query. This means we rarely need to directly force such decisions with hints to obtain good choices for ACCESS METHOD and JOIN METHOD, and this is good. Giving the CBO information so that it can do what it knows how to do is always preferable to forcing the CBO to do what we want to do. It is true that we did use an ORDERED hint to force the join sequence of the problem query in this chapter. But the ORDERED hint only affects the join sequence, it does not force other decisions. Thus the ORDERED hint is far more tolerated by tuning specialists worldwide over most other hints. The ORDERED hint is a useful tool when used in specific problem queries to help the CBO to overcome its limitations, or a flaw in your system that you otherwise cannot (or choose not) to correct. LEADING hint would be used in 11g and 12c.

So far we have seen how to:

- Build a query diagram to give us a visual picture of our query.
- Walk a query diagram to see valid join sequences for our query.
- Build a FRP spreadsheet to reveal DRIVING TABLE and JOIN ORDER for our query.
- Build COUNT QUERIES, FILTER QUERIES, and RECONSTRUCTION QUERIES to get the row counts for the FRP spreadsheet.
- Make intelligent decisions using the information collected by FRP.

How to use FILTERED ROWS PERCENTAGE Method

It may seem that we used FRP to find an OPTIMAL JOIN ORDER for a query so that we could force that join order on the query via an ORDERED hint. This may in fact be necessary for the problem query we are tuning and we may in the end choose to employ an ORDERED or LEADING hint in our query to get the results we want. However, this is not the real purpose of FRP. The real purpose of FRP is to assist us as a Tuner to understand the problem query better so that we can make intelligent conversation about it, and ultimately figure out what is really wrong so we can fix it.

Consider this question: Given a long running query, how do you know if any given query plan for that query is a good one? Most people cannot answer this question. But you can, at least half of it anyway. Recall our discussion of A GOOD QUERY PLAN.

Every fast query plan has decided on four things correctly.

- A good choice of DRIVING TABLE for the query.
- A good choice of JOIN ORDER for all tables.
- A good choice of ACCESS METHOD for each table.
- A good choice of JOIN METHOD for row-sets being joined.

If this is our measure of a good query plan then these questions must be answerable.

- Does the query plan show a reasonable choice for DRIVING TABLE?
- Does the query plan show a reasonable choice for JOIN ORDER?
- Does the query plan show a reasonable choice for ACCESS METHOD for each table?
- Does the query plan show a reasonable choice for JOIN METHOD between row sets?

If you can answer YES or NO to these questions, then you can tell people if a query plan is a good one or not and why. This is actually quite impressive: to be able to declare that a plan is a good plan or not and to explain why it is so.

You can answer the first two questions. You can do a FRP analysis of the query and then check the analysis against the actual query plan being used. If the query plan uses the DRIVING TABLE that FRP recommends then you know the CBO has made a reasonable choice for DRIVING TABLE. Even if the DRIVING TABLE is different from that recommended by FRP, you can tell by the various row count columns in your FRP spreadsheet, if the DRIVING TABLE selected by the CBO is a reasonable alternative to the FRP recommended DRIVING TABLE.

The same goes for JOIN ORDER. If the CBO uses the JOIN ORDER you determined via FRP then the JOIN ORDER is a reasonable one. If the JOIN ORDER the CBO chooses is different, you can evaluate if the difference is significant or not.

In cases where you determine that the CBO choices are likely not reasonable, you can explain why based on the row count data. You also can have some confidence that a better alternative exists and thus there is the possibility that the problem query can be made faster. This lets you answer another common question: IS IT POSSIBLE TO MAKE THIS QUERY GO FASTER? If you can explain why the CBO has made what look like mistakes, then you can say there is some possibility that the query could go faster. It is your ability to EXPLAIN WHY that puts you ahead of others in the tuning game.

Backtracking on CARDINALITY FEEDBACK

We skipped over the use of cardinality feedback previously, because the example used did not provide any opportunity to see how a change in row counts coming out of joins might suggest a better join order. This let us concentrate on the primary process of FRP. But it sometimes does happen that joins remove rows and sometimes a significant number of rows. Cardinality Feedback can tells us this and we can use this to select a different join order to see if it offers better performance by reducing the number of rows held in transit, earlier in query execution. So for steps 14 and 15, an example is offered that demonstrates.

14. Use CARDINALITY FEEDBACK to adjust join order

No there is nothing missing here. This step and the next we talk about below together.

15. Repeat (11) through (12) once if join order changed

Recall that we skipped these two steps in our first example because that example had nothing in it for these two steps to be useful. Let us look at a different example that actually benefited from a change in join order which we were able to determine using CARDINALITY FEEDBACK from column JOIN ROWS.

A second query yields this FRP spreadsheet. We omit all the FRP work done to get to this FRP spreadsheet for brevity. Assume we did all the FRP work needed to get to this point. We see here the INITIAL JOIN ORDER that FRP recommends. We can also see that the JOIN ROWS column which we get after running all our reconstruction queries, has some interesting row count reductions that take place late in the query. Since the goal of FRP is to remove rows during query execution as soon as possible, it would be desirable to move access to the tables we see at the end of this query, forward in our join order. Assuming this is possible via some valid join sequence, our aim would be to take advantage of the row count reductions that happen when we do the related joins. Again, please observe the JOIN ROWS column to see where dramatic changes occur in a row count after a join is done. Note row 6 (based on initial join order).

ALIAS	ROWS	FILTERED ROWS	FILTERED ROWS %	PREFERRED JOIN ORDER	INITIAL JOIN ORDER	JOIN ROWS
CASE_DIM2	343487	1	100	1	1	1
LOSS_FACT	464685	464865	0	10	2	375650
CLAIM_DIM2	71109681	71109681	0	11	3	242498
POLICY_DIM2	19808250	1714354	91	2	4	237637
CLAIMANT_DIM2	3068589	1054782	66	3	5	232844
PAYMENT_DIM2	4098213	3175871	23	4	6	84855
LOSS_UNIT_DIM2	456097	421442	8	5	7	84855
BENEFIT_CODE_DIM2	322292	312449	3	6	8	84855
LOSS_BENEFIT_PROCESS	296	290	2	7	9	84855
LOSS_CHECK_END	71592	71592	0	8	10	84855
V_PT_MATRIX2	71592	71592	0	9	11	84855

Given any particular join order (in our case the INTIAL JOIN ORDER) it is possible to convert a QUERY DIAGRAM into a JOIN TREE DIAGRAM. A JOIN TREE DIAGRAM is a simple list of tables that shows by indentation the join sequence of a set of tables in a query, and the join level each table in the join sequence sits at. The join tree diagram can be used to evaluate the JOIN ROWS column to see if there might be a better ordering of tables aside from our INITIAL JOIN ORDER. This is possible because a join operation has the potential to change the number of rows floated in a query either by increasing that number as the case would be for a PARENT → CHILD join, or to decrease the number as the case would be for joins where one or more column values was null or did not match an opposing row in the join. We are looking at our JOIN ROWS column for either an increase in row count or a decrease in row count (decrease being more likely) as large changes in row count are an opportunity to further reduce the size of intermediary row-sets being floated by our problem query during query execution.

We can see from the join tree below (to which we have added our JOIN ROWS column) that indeed there is a case where changing the join order may result in further reduced workload. Given the join sequence we see, we note that at three levels in, there is a series of tables that all join back to the same prior table in the join sequence. This table is LOSS_FACT and it is valid to join to any of the tables we see at the third level of our join tree from LOSS_FACT. In the query there are joins from LOSS_FACT to each of these tables. In the query diagram these joins would be readily visible.

We selected the order we see here, because of our FRP analysis and the resultant FILTERED ROWS % column. But we can see that for table LOSS_UNIT_DIM2, there is a dramatic drop in row count after the join as compared to the tables before it in the join sequence at its level of indentation.

It thus makes sense to try and move that table forward in the join order (possibly other tables as well but let us start with one). If we change the INITIAL JOIN ORDER to be a CARDINALITY FEEDBACK JOIN ORDER, we can build a new set of reconstruction queries based on this new join order and check the resulting join cardinalities. So that is what we did here. We moved LOSS_UNIT_DIM2 forward in the join order, then built new reconstruction queries to reflect the new join order and ran these queries. As hoped for, we see in the second JOIN TREE DIAGRAM, even more reduction in row counts. This means a reduction in intermediary row-set sizes earlier in the query, because of this new join order based on CARDINALITY FEEDBACK.

```
                                                    JOIN
JOIN TREE                                           ROWS
--------------------------------------------------- ------
CASE_DIM_2                                              1
          LOSS_FACT                                375650
                    CLAIM_DIM2                     242498
                    POLICY_DIM2                    237637
                    CLAIMANT_DIM2                  232844
                    LOSS_UNIT_DIM2                  84855
                    PAYMENT_DIM2                    84855
                    BENEFIT_CODE_DIM2               84855
                    LOSS_BENEFIT_PROCESS            84855
                    LOSS_CHECK_END                  84855
                                        V_PT_MATRIX 84855


RE-ORDERED                                          JOIN
JOIN TREE                                           ROWS
--------------------------------------------------- ------
CASE_DIM_2                                              1
          LOSS_FACT                                375650
                    LOSS_UNIT_DIM2                  89701
                    CLAIM_DIM2                      86598
                    POLICY_DIM2                     84855
                    CLAIMANT_DIM2                   84855
                    PAYMENT_DIM2                    84855
                    BENEFIT_CODE_DIM2               84855
                    LOSS_BENEFIT_PROCESS            84855
                    LOSS_CHECK_END                  84855
                                        V_PT_MATRIX 84855
```

The reverse is also possible, though less common. We may see a dramatic increase in row count at which point we would try to delay access to that table until later. A word of caution: it may come to pass that moving tables around based on CARDINALITY FEEDBACK does not provide any benefit. Do not be disappointed. At least you know how to identify the situation and check for any benefit.

Building a join tree is easy; just start with the driving table and follow the joins, indenting one level each for each new table that does a join. As is observed from the join trees above, joins fall into layers because of the indentation and sometimes a layer may have many tables in it. Tables in the same layer can be joined to in any order. Since the FRP method does not take into account rows lost during a join, FRP cannot determine what the best order to join tables in any single layer would be. It can only go by the initial filtered rows percentage based on constant test filtering. Thus cardinality feedback based on joins allows us to identify cases where a different join ordering can result in fewer rows being held in-transit during query execution and that is exactly what this example shows. Moving table LOSS_UNIT_DIM2 to the front of its layer lets its join remove more rows earlier in query execution and this affects all subsequent joins. Thus it is more efficient.

It should be noted that the database query optimizer does try to do these kinds of calculations. However it can get the answers wrong which is why an understanding of this process provides the knowledge needed to correct situations when the optimizer has made an error for whatever reason.

It should also be noted that CARDINALITY FEEDBACK JOIN ORDER is really just a minor refinement of the more important INITIAL JOIN ORDER delivered by FRP. If you don't do it, that is OK.

The Short Cut (Brain over Brawn)

Even though this chapter is called Driving Table and Join Order, it has really been about the FILTERED ROWS PERCENTAGE (FRP) spreadsheet and how this leads to an overall analysis of the query plan for goodness and for determination of a good Driving Table and Join Order. To produce the FRP spreadsheet requires an understanding of many topics like Query Diagrams, Join Sentences, Join Trees, Count Queries, Filter Queries, Reconstruction Queries, and so on. The FRP also requires a significant investment of time and intellectual horse-power to complete.

My experience however has been that people in IT are basically lazy bums who do not like repetition. Indeed these qualities are positives in the IT field as they prod us forward to invent new things to simplify life and remove the dull work from our existence. Brain over Brawn is our motto. Fortunately I am as lazy as the next guy. About half way through this book I got tired of building FRP spreadsheets by hand so I took a detour into finding a way to get the database to do most of the work with automation, and I had some modest success. I figured out how to have the database generate a FRP spreadsheet for a query with significantly less work. I present this to you now with the recognition that this short cut will allow you to make the FRP spreadsheet one of the main artifacts you use going forward in your tuning adventures, which it should be. It is my main artifact and part of my SQL Tuning Opening Moves.

During any real world SQL tuning, a query plan is eventually generated using the EXPLAIN PLAN command. I assume you the reader already knows how to use this command. The EXPLAIN PLAN command as you should know, records a great deal of information about the parsed query, objects the query needs, optimizer decisions made, etc. in sufficient detail to allow a report to be produced that shows the expected query plan. This information is saved by the EXPLAIN PLAN command into the PLAN_TABLE table.

The PLAN_TABLE table is like all other database metadata objects, an object which can be queried with SQL. Why not use this data for our own purposes? Nothing says we have to be limited by just what Oracle gives us. Using the information from the PLAN_TABLE, it is possible to write a code generator script that will produce for us the COUNT QUERIES and FILTER QUERIES needed to construct a FRP spreadsheet. With more effort this code generator could combine all these mini-queries into one large query that actually produces the FRP spreadsheet. Here is such a code generator that produces the FRP spreadsheet. It does have a few flaws but they are minor and have easy workarounds should you run into them.

@GENFRPSPREADSHEETCODE411G.SQL is the current version of this generator. This name means fourth generation of the 11g version. The 11g version works on 10g and 12c as well. There is also a different version for 9i, necessitated by differences in the PLAN_TABLE table. Here is an example of how it works. This example is repeated in Chapter 7: Row Counts and Run Times where it is used to demonstrate the tuning process.

We start with a query. It references a view. No matter, the EXPLAIN PLAN command will analyze the view and save query plan details which show us the actual tables used and any filtering criteria hidden by the view. Hmm… already a significant benefit. Automatic View Decomposition is done for us by EXPLAIN PLAN.

```
SELECT COUNT(*) FROM VW_EMP_LOC_DIM;
```

We run EXPLAIN PLAN to get all the details.

```
08:42:22 SQL> explain plan for SELECT COUNT(*) FROM VW_EMP_LOC_DIM;

Explained.
```

After execution of this command, the PLAN_TABLE table has the query plan in it, including all the tables referenced, all the joins between tables, and all the filtering predicates used, parsed out into separate columns. We can view the query plan and predicate info by dumping the plan. Note how the query plan shows all table accesses, not simply a view access. The EXPLAIN PLAN command has parsed the query into the PLAN_TABLE table with all sorts of query plan details.

```
@SHOWPLAN11GSHORT.SQL

Plan hash value: 1232802935
-------------------------------------------------------------------------
| Id  | Operation                          | Name              | Rows  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                   |     1 |
|   1 |  SORT AGGREGATE                    |                   |     1 |
|   2 |   VIEW                             | VW_EMP_LOC_DIM    |   198M|
|   3 |    UNION-ALL                       |                   |       |
|*  4 |     HASH JOIN                      |                   |   296K|
|*  5 |      VIEW                          |                   |   240K|
|*  6 |       WINDOW SORT PUSHED RANK      |                   |   240K|
|*  7 |        FILTER                      |                   |       |
|*  8 |         TABLE ACCESS STORAGE FULL  | EMP_DIM           |   240K|
|*  9 |      TABLE ACCESS STORAGE FULL     | EMP_LOC_DIM       |   296K|
|  10 |     COUNT                          |                   |       |
|* 11 |      HASH JOIN                     |                   |   152M|
|  12 |       VIEW                         |                   |  8874 |
|  13 |        HASH UNIQUE                 |                   |  8874 |
|* 14 |         FILTER                     |                   |       |
|* 15 |          TABLE ACCESS STORAGE FULL | EMPLR_LOC_DIM     |  8874 |
|  16 |       VIEW                         |                   |   240K|
|  17 |        MINUS                       |                   |       |
|  18 |         SORT UNIQUE                |                   |   240K|
|* 19 |          TABLE ACCESS STORAGE FULL | EMP_DIM           |   240K|
|  20 |         SORT UNIQUE                |                   |   251K|
|* 21 |          TABLE ACCESS STORAGE FULL | EMP_LOC_DIM       |   251K|
-------------------------------------------------------------------------

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

   4 - access("EMP_GID"="EL"."EMP_LOC_GID")
   5 - filter("PRTY_REF_ID_RNK"=1)
   6 - filter(RANK() OVER ( PARTITION BY "GRP_BEN_CASE_ID","PRTY_REF_ID" ORDER
BY CASE
             "EE"."EMPLMT_STAT_CD" WHEN 'T' THEN 2 WHEN 'R' THEN 2 WHEN 'D'
THEN 2 WHEN 'I' THEN 2 ELSE 1 END
             ,INTERNAL_FUNCTION("EE"."SRCE_EFF_START_TMSP") DESC
,INTERNAL_FUNCTION("EE"."EMP_PK_ID") DESC )<=1)
   7 - filter(SYS_CONTEXT('APP1','CURRENT_SCHEMA')='CLDW_THPA_DLV1')
   8 - storage("EE"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00' AND
```

```
              "EE"."SRCE_APP_SYS_CD"='ELIG')
       filter("EE"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00' AND
              "EE"."SRCE_APP_SYS_CD"='ELIG')
   9 - storage("EL"."POPULATION_STATUS_CD"<>'D')
       filter("EL"."POPULATION_STATUS_CD"<>'D')
  11 - access("A"."GRP_BEN_CASE_ID"="B"."GRP_BEN_CASE_ID")
  14 - filter(SYS_CONTEXT('APP1','CURRENT_SCHEMA')='CLDW_THPA_DLV1')
  15 - storage("E"."EMPLR_LOC_PK_ID"<>(-1) AND "E"."EMPLR_LOC_PK_ID"<>(-2) AND
              "E"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00')
       filter("E"."EMPLR_LOC_PK_ID"<>(-1) AND "E"."EMPLR_LOC_PK_ID"<>(-2) AND
              "E"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00')
  19 - storage("EE"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00' AND
              "EE"."POPULATION_STATUS_CD"<>'D' AND "EE"."EMP_PK_ID"<>(-1) AND
"EE"."EMP_PK_ID"<>(-2))
       filter("EE"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00' AND
              "EE"."POPULATION_STATUS_CD"<>'D' AND "EE"."EMP_PK_ID"<>(-1) AND
"EE"."EMP_PK_ID"<>(-2))
  21 - storage("EL"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00' AND
              "EL"."POPULATION_STATUS_CD"<>'D')
       filter("EL"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31 00:00:00' AND
              "EL"."POPULATION_STATUS_CD"<>'D')

Note
-----
   - dynamic sampling used for this statement (level=4)
   - automatic DOP: Computed Degree of Parallelism is 1
```

Once the query has been explained and the query plan has been saved to the PLAN_TABLE table, we can use the code generator to generate a FRP spreadsheet query.

```
08:45:24 SQL> @genfrpspreadsheetcode411g.sql

SQLTEXT
--------------------------------------------------------------------------------
with
      frp_data as (
                    select '   15' id, 'SCOTT' table_owner,'EMPLR_LOC_DIM'
table_name,'E' table_alias,8874 NUM_ROWS,count(*) rowcount,8874
cardinality,count(case when "E"."EMPLR_LOC_PK_ID"<>(-1) AND
"E"."EMPLR_LOC_PK_ID"<>(-2) AND "E"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31
00:00:00' then 1 end) filtered_cardinality from EMPLR_LOC_DIM E union all
                    select '    9' id, 'SCOTT' table_owner,'EMP_LOC_DIM'
table_name,'EL' table_alias,329699 NUM_ROWS,count(*) rowcount,296337
cardinality,count(case when "EL"."POPULATION_STATUS_CD"<>'D' then 1 end)
filtered_cardinality from EMP_LOC_DIM EL union all
                    select '    8' id, 'SCOTT' table_owner,'EMP_DIM'
table_name,'EE' table_alias,6243035 NUM_ROWS,count(*) rowcount,240117
cardinality,count(case when "EE"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31
00:00:00' AND "EE"."SRCE_APP_SYS_CD"='ELIG' then 1 end) filtered_cardinality
from EMP_DIM EE union all
                    select '   19' id, 'SCOTT' table_owner,'EMP_DIM'
table_name,'EE' table_alias,6243035 NUM_ROWS,count(*) rowcount,240117
cardinality,count(case when "EE"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31
```

```
00:00:00' AND "EE"."POPULATION_STATUS_CD"<>'D' AND "EE"."EMP_PK_ID"<>(-1) AND
"EE"."EMP_PK_ID"<>(-2) then 1 end) filtered_cardinality from EMP_DIM EE union
all
                    select '   21' id, 'SCOTT' table_owner,'EMP_LOC_DIM'
table_name,'EL' table_alias,329699 NUM_ROWS,count(*) rowcount,251761
cardinality,count(case when "EL"."SRCE_EFF_END_TMSP"=TIMESTAMP' 9999-12-31
00:00:00' AND "EL"."POPULATION_STATUS_CD"<>'D' then 1 end) filtered_cardinality
from EMP_LOC_DIM EL union all
                    select null,null,null,null,null,null,null,null from dual
                  )
select frp_data.*,round(frp_data.filtered_cardinality/case when
frp_data.rowcount = 0 then cast(null as number) else frp_data.rowcount
end*100,1) actual_frp,decode(frp_data.filtered_cardinality,null,cast(null as
number),round(frp_data.cardinality/case when frp_data.NUM_ROWS = 0 then
cast(null as number) else frp_data.NUM_ROWS end*100,1)) plan_frp
from frp_data
where id is not null
order by frp_data.id
/

19 rows selected.
```

Executing the query generated by the code generator will generate the FRP spreadsheet as a simple report. Running the query, results in scanning all rows in all tables of the query but that is what COUNT QUERIES and FILTER QUERIES do. Waiting for information to be collected is a significant part of the SQL tuning experience so be patient please.

Using this script is way faster than all those time consuming steps used in this chapter to show what FRP is and how it is constructed. Using this script means changing focus slightly in tuning a problem SQL query. The focus turns slightly from trying to determine the best Driving Table and Join Order to a validation of the query plan in general. More specifically we seek to know if the query plan accurately estimated its initial workload, and if it did, if the Driving Table and Join Order selected make sense as well as the specific access methods used to access each table given consideration of the 2% RULE. We introduce the 2% RULE in Chapter 4: Joins.

Is it beautiful or what?

```
                                                    Plan    Filtered Actual
ID    TABLE_NAME      NUM_ROWS   ROWCOUNT Cardinality Cardinality    FRP
----- ------------- ---------- ---------- ----------- ----------- ------
    8 EMP_DIM          6243035    6243035      240117      215414    3.5
    9 EMP_LOC_DIM       329699     329699      296337      329699  100.0
   15 EMPLR_LOC_DIM       8874       8874        8874        8872  100.0
   19 EMP_DIM          6243035    6243035      240117      236469    3.8
   21 EMP_LOC_DIM       329699     329699      251761      212993   64.6

5 rows selected.
```

Notice all the information provided. In particular, there are four types of row counts along with the Actual FRP based on filtered cardinality. Also shown is plan ID from the query plan for each table reference we are interested in, so that if necessary all this information can be related back to the query plan.

These are the metrics of the generated FRP.

- NUM_ROWS = DBA_TABLES.NUM_ROWS which is what the database thinks is the number of rows in the table.
- ROW COUNT = the actual number of rows in the table obtained using a COUNT QUERY.
- Plan Cardinality = the optimizer's guess at filtered cardinality for this plan step. This is taken directly from the PLAN_TABLE table for the explained query.
- Filtered Cardinality = true cardinality obtained using a FILTER QUERY. This is the actual number of rows returned after filtering.
- Actual FRP = round (Filtered Cardinality / ROWCOUNT*100, 1) and tells us the percentage of rows that remain after filtering. THIS IS DIFFERENT FROM WHAT WE LEARNED. Again it is the percentage of rows that remain after filtering. So instead of seeking 100 as best, we seek 0 as best. Or said another way, this is the percentage of rows that feed into the query from this plan step. This is a new column and is one of the destinations of the FRP spreadsheet.

You can see some serious sophistication here. This FRP spreadsheet is powerful because it combines data from three different places and relates these metrics to each other. It has data taken from database metadata objects, data taken from the table data directly, and data taken from the plan table for the explained plan of our problem query. You will see all this again in Chapter 7: Row Counts and Run Times. That chapter will use this example to work through a real life tuning episode to show how the FRP spreadsheet is used and how it fits in relation to other tuning artifacts. This code generator and the reporting SQL to create the FRP spreadsheet generated are wicked time savers and make it much easier to use FRP in your analysis of a problem SQL statement.

As an example of the initial way to use this information, evaluate the first line of the spreadsheet.

ID	TABLE_NAME	NUM_ROWS	ROWCOUNT	Plan Cardinality	Filtered Cardinality	Actual FRP
8	EMP_DIM	6243035	6243035	240117	215414	3.5

Consider these points and questions.

- NUM_ROWS exactly matches ROWCOUNT. What does this suggest to you?
- Plan Cardinality is very close to Filtered Cardinality. What does this suggest to you?
- Actual FRP is 3.5%. What does this suggest to you?
- And across all lines of the FRP Spreadsheet, what looks like a good DRIVING TABLE?

These should suggest the following.

- Statistics are likely up-to-date.
- The CBO accurately estimated the workload feeding into the query for this plan step.
- A full table scan looks good for EMP_DIM which suggests HASH JOIN when joining to it from this plan step. This is based on the 2% RULE (a guideline) which you have not learned yet.

- Given that this is the lowest FRP of all tables in the query, EMP_DIM might make a good DRIVING TABLE for this query too since it removes the highest percentage of rows for any table in the query (only 3.5% remain after filtering).

Examining the relevant portion of the query plan, we see that things match our expectations. A FULL TABLE SCAN has indeed been used to access the table EMP_DIM at ID=8, and results of this plan step feed into a HASH JOIN at ID=4.

```
|*  4 |     HASH JOIN                      |                    |   296K|
|*  5 |      VIEW                          |                    |   240K|
|*  6 |       WINDOW SORT PUSHED RANK      |                    |   240K|
|*  7 |        FILTER                      |                    |       |
|*  8 |         TABLE ACCESS STORAGE FULL  | EMP_DIM            |   240K|
|*  9 |      TABLE ACCESS STORAGE FULL     | EMP_LOC_DIM        |   296K|
```

A reasonable conclusion from this is that the CBO has correctly dealt with WORKLOAD ESTIMATE / ACCESS METHOD / JOIN METHOD for ID=8.

Things the CBO has done correctly for the query plan above.

- Accurately estimated the initial workload for the plan step ID=8.
- Used a reasonable access method to get rows from the table at plan step ID=8.
- Used a reasonable join method to join results from ID=8 to another row source in the plan.

The same analysis can be done for all rows of the FRP spreadsheet. Do this now for yourself please so that you can gain actual experience in the method of FRP.

This space intentionally left blank so you can think.

If you worked through the rest of the rows in FRP, you would see that the same is true for all plan steps shown by the FRP spreadsheet. They have all correctly dealt with WORKLOAD ESTIMATE / ACCESS METHOD / JOIN METHOD for their respective table. A query plan which has good workload estimates and which accesses data and joins data using reasonable choices is likely a very good plan.

Of course this still does not tell us if the CBO has made reasonable choices for Driving Table and Join Order. To learn this we need to follow up with a join tree of some kind. Here is one possible join tree using the query plan data. Because the actual query is hidden behind at least one view, we cannot construct a query diagram for the underlying query without looking at the view text. This is something we should do but in the meantime here is a join tree created using only information provided by the query plan. In actuality, over time we will prefer using the data in explain plan rather than going to the view text.

```
PLAN_STEP    TABLE JOINS                        ROWS
---------    -------------------------------    ----
19           EMP_DIM                            240K
17              EMP_LOC_DIM                     240K
11                 (EMPLYR_LOC_DIM)             152M
 3                    (EMP_DIM/EMP_LOC_DIM)     198M
```

ROWS is the cardinality estimate of the CBO for the plan step. Please note that given the nature of the query plan we have chosen to combine some of the steps to more readily see the overall join sequence. The

increasing values in the ROWS column shows us why the CBO may have selected this specific join order. It keeps row counts smallest going forward.

But as we know, these are estimates. We want to know if these join cardinality estimates are accurate so that we can evaluate the goodness of fit of the CBO choices for Driving Table and Join Order. Driving table here is EMP_DIM at line 19 and Join Order is as seen in the sequence of rows. Refer back to the query plan previously shown to see these lines. To validate these we need to build RECONSTRUCTION QUERIES (also presented in this chapter). Sadly, I do not yet have a code generator for this. I leave that to you as an exercise should you care to have a go at it.

Scripts used in the Chapter

There are about two dozen scripts that go with this book. They are all intended to reduce time needed to tune and to provide unique insights about your query, its query plan, and its runtime environment. There is in fact a script for almost every artifact described in this book as part of the FRP method. You will see them as you progress through the book, and especially in the LAB. Here are the scripts used in this chapter.

SHOWOWNER

SHOWPLAN11G

SHOWPLAN11GSHORT

SHOWPLANFRPSPREADSHEETCODE11G (formerly genfrpspreadsheetcode411g)

Chapter Summary

It was only forty seven pages, but this chapter has introduced you to the idea that the fastest query plan is almost always the query plan that removes the largest percentage of rows as early as possible in query execution. It has shown you how this is determined by using a CARDINALITY BASED ANALYSIS called FILTER ROWS PERCENTAGE. Additionally, you have been given a script that does most of the work needed to acquire and present the necessary information for this analysis. You are expected to use this script in your future tuning travels.

Consider for a moment what this chapter has taught you; do these terms mean anything to you? They should. If you are hazy or drawing a blank from any of these terms, maybe you should go back to the relevant portion of this chapter and review it.

- Join Sentence
- Query Diagram
- Driving Table
- Join Order
- Preferred Join Order
- Initial Join Order
- Cardinality Feedback Join Order
- Count Query
- Filter Query

- Reconstruction Query
- Initial Query Workload
- FILTERED ROWS PERCENTAGE
- Magic Script that does the work

Please visit the message board on www.orafaq.com and search for the book title Oracle SQL Performance Tuning and Optimization in order to download a .rar file with the scripts. They are free even if you don't buy the book. Or, you can contact me directly using my email, km133688@sbcglobal.net and I will reply to you with the latest set of scripts.

Enjoy. Kevin Meade