I POCKET GUIDES

Oracle Performance Tuning Essentials

Practical Steps for Diagnosing and Improving SQL

Sanjay Mishra

Apress®

Apress Pocket Guides

Apress Pocket Guides present concise summaries of cutting-edge developments and working practices throughout the tech industry. Shorter in length, books in this series aims to deliver quick-to-read guides that are easy to absorb, perfect for the time-poor professional.

This series covers the full spectrum of topics relevant to the modern industry, from security, AI, machine learning, cloud computing, web development, product design, to programming techniques and business topics too.

Typical topics might include:

- A concise guide to a particular topic, method, function or framework
- Professional best practices and industry trends
- A snapshot of a hot or emerging topic
- Industry case studies
- Concise presentations of core concepts suited for students and those interested in entering the tech industry
- Short reference guides outlining 'need-to-know' concepts and practices.

Sanjay Mishra

Oracle Performance Tuning Essentials

Practical Steps for Diagnosing and Improving SQL

Apress®

Sanjay Mishra
Fuquay Varina, NC, USA

ISSN 3004-927X ISSN 3004-9288 (electronic)
Apress Pocket Guides
ISBN 979-8-8688-3155-3 ISBN 979-8-8688-3156-0 (eBook)
https://doi.org/10.1007/979-8-8688-3156-0

© Sanjay Mishra 2026

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Unless otherwise expressly agreed by the Publisher in writing, the Publisher does not authorize use or reproduction of all or any part of this work in any manner for the purpose of training, fine-tuning, evaluating, or developing artificial intelligence models, technologies, or systems. In accordance with Article 4(3) of the EU Directive 2019/790 on copyright and related rights in the Digital Single Market, the Publisher expressly reserves all rights with respect to this work under the text and data mining exception of the directive.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.
The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

If disposing of this product, please recycle the paper.

To my daughter Prisha, who is super excited for her dad writes books and suggests me to write more. :)
To my son Krishiv, whose smile makes every hard day easier.
To my wife Richa, my constant through it all.
And to my father, Taranath Mishra, who fills me with positivity.

Preface

Performance tuning in Oracle databases often appears daunting, wrapped in layers of complexity and technical jargon. But through years of experience in the field, I've learned that most performance problems have simple, structured solutions—if approached with clarity and the right mindset.

This book is built on that principle. It offers a result-oriented approach to diagnosing and resolving performance issues, with practical techniques that have consistently worked in real-world environments. Whether you're a developer writing PL/SQL code or a DBA managing mission-critical systems, the strategies here are designed to be actionable and effective.

One of the most important tools I share is the *Performance Tuning Checklist*. This isn't just a theoretical framework—it's a must-have for any organization serious about writing efficient PL/SQL code. I've explained how to apply this checklist using simple examples, so readers can confidently integrate it into their daily workflow.

You will also find chapters dedicated to:

- Understanding Oracle Performance Problems
- The Simple Diagnostic Approach
- Advanced SQL Identification and Analysis
- Locking and Concurrency Management

My goal is to simplify the tuning process and empower you to make Oracle run smoother, faster, and smarter. I hope this book becomes a trusted companion in your journey toward better performance and deeper understanding.

Fuquay Varina, NC, USA Sanjay Mishra

Contents

1 The Simple Truth About Oracle Performance 1
The 80/20 Rule in Action 1
What's Usually Going Wrong? 3
Indexing Issues 3
Query Design 3
Data and Statistics 4
Schema and Design 4
Infrastructure 4
Application Layer 5
How These Problems Show Up in Real Life 5
Small Fixes, Big Impact 6
Summary 6

2 The Simple Diagnostic Approach 9
When Performance Goes Wrong: Where Do You Actually Start? 9
Step 1: Identify the Problem SQL 11
Using AWR to Find the Problem 12
Using ASH to Go Deeper 12
Watch for Duration Creep 13
The Alternate Strategy: Often the Most Practical First Step 14
Step 2: Analyze the Execution Plan 16
Step 3: Check the Obvious Issues 16
Step 4: Implement the Fix 17
Summary 17

3 Understanding EXPLAIN PLAN: Your SQL's Roadmap to Performance 19
What Is EXPLAIN PLAN? 19
Why Is EXPLAIN PLAN So Important? 20
A Simple Example 20
How to Generate EXPLAIN PLAN 22
In SQL Developer 22

In Allround Automations' PL/SQL Developer 22
In TOAD 23
What to Look for in the Plan 23
Summary 25

4 EXPLAIN PLAN: Deep Dive 27
Core Components of the Plan 27
Access Paths: How the Data Is Found 28
Join Methods: How Tables Are Connected 30
Practical Examples 31
Example A: The Problematic Plan (High Cost) 31
Example B: The Efficient Plan (Low Cost) 32
Example C: The Cost of Sorting 33
Summary 33

5 Proactive Optimization 35
Write SQL That Gives the Optimizer Good Choices 35
Avoid `SELECT *` 35
Use Explicit ANSI JOIN Syntax 37
Replace Subqueries with JOINs or CTEs 37
Use Bind Variables 38
Respect the Index Contract 39
Do Not Apply Functions to Indexed Columns 39
Avoid Implicit Type Conversion 40
Avoid Leading Wildcards 41
Write Predicates for Correctness and Index Use (Not Evaluation Order) 41
Aggregation 42
Use Index-Only Aggregation 42
Pre-compute with Materialized Views 42
Use `ROLLUP` for Multilevel Aggregation 43
Set Operations 43
Prefer `UNION ALL` over `UNION` 43
Maintain Current Statistics 44
Summary 45

6 Advanced SQL Discovery: Hunting for Performance Bottlenecks 47
The Shared Pool: Oracle's Central Statement Repository 47
`V$SQL`: The Statement Register 48
Identifying High-Impact Queries 48
`V$SQL_PLAN`: Examining Plans in the Cursor Cache 50
Retrieving Cached Plans with `DBMS_XPLAN.DISPLAY_CURSOR` 51
Locating a `SQL_ID` for a Statement 51
Detecting Plan Changes with `PLAN_HASH_VALUE` 52
`V$ACTIVE_SESSION_HISTORY`: Real-Time Wait Analysis 52
Common Wait Events and Their Meaning 53

ASH Queries for Incident Analysis 53
Optimizer Statistics Management 55
Gathering Table and Index Statistics 55
Histograms for Skewed Column Distributions 55
Locking Statistics 56
Restoring Statistics After a Plan Regression 56
Correlating V$SQL, ASH, and Statistics 57
Summary 59

7 Comprehensive Index Strategy 61
Types of Indexes 61
Strategic Index Creation 62
Index Usage Verification with EXPLAIN PLAN 63
Identifying and Removing Redundant Indexes 64
When NOT to Use Indexes 65
Advanced Indexing Techniques 66
Function-Based Indexes 66
Covering Indexes 66
Invisible Indexes 67
Index Skip Scans 67
Reverse Key Indexes 68
Index Maintenance 68
Summary 69

8 Advanced Locking and Concurrency Management 71
Oracle's Multiversion Read Consistency 71
Oracle's Lock Types 72
Comprehensive Lock Detection and Resolution 73
Prolonged Blocking: Check V$SESSION_LONGOPS Before You Kill 73
Deadlock Detection and Prevention 74
Oracle Isolation Level Optimization 75
Long Transaction Optimization via Batch Processing 76
Summary 77

9 Execution Plan Stabilization 79
SQL Plan Baselines 79
Enabling Automatic Plan Capture 80
Loading a Known-Good Plan Manually 80
Monitoring and Managing Plan Baselines 81
Detecting Unauthorized Plan Changes 82
Evolving Unaccepted Plans 83
Dropping Stale Baselines 83
SPM Across Database Upgrades 84
Summary 85

10 Query Frequency and Volume Optimization 87
Eliminate Query Loops in PL/SQL 87
`FORALL` for Bulk DML 89
Advanced Caching Strategies 89
`DETERMINISTIC` Functions 90
Materialized Views for Expensive Aggregations 90
Result Set Limiting and Pagination 91
Efficient Pagination (Oracle 12c and Later) 91
Legacy Pagination (Pre-12c) 92
Identifying High-Frequency SQL 93
Summary 93

11 Database Configuration Optimization 95
Memory Configuration Tuning 95
System Global Area (SGA) 96
Program Global Area (PGA) 97
Buffer Cache Sizing with Advisory Views 98
Shared Pool Sizing 98
Connection Pool Optimization 99
Redo Log and I/O Configuration 100
Summary 100

12 Application Layer Optimization 103
Parameterized Queries for Security and Performance 103
ORM Query Optimization 105
Avoiding the N+1 Problem 105
Use Projections to Reduce Payload 106
Push Filters Down, Not Up 107
Application-Level Pagination 109
Caching and Read Scaling 110
Cache-Aside Pattern 110
Read and Write Splitting 111
Summary 111

13 Batch Processing and Parallel Execution 113
Prefer Set-Based SQL Over Row-by-Row Work 113
Replace Looping Updates with a Single Statement 113
PL/SQL Bulk Processing: `BULK COLLECT` and `FORALL` 114
Bulk Collect in Batches 115
Use `FORALL` for DML 115
Commit Strategy and Undo/Redo Pressure 116
Parallelism: Use It Like a Power Tool 116
Parallel Query and Parallel DML 117
Chunked Parallelism with `DBMS_PARALLEL_EXECUTE` 117
Direct-Path Loads and Minimal Logging 118
Partition-Friendly Batch Design 118

Operational Guardrails: Make Batch Jobs Boring 119
Summary 119

14 Observability: Measuring, Monitoring, and Proving Improvements . 121
Start With a Measurement Contract 121
Live Diagnosis: What Are Sessions Waiting For? 122
Active Sessions and Wait Events 122
Who Is Blocking Whom? 123
SQL-Level Evidence: From `SQL_ID` to Plan and Monitoring 123
Confirm Plan Identity and Per-Execution Cost 123
SQL Monitoring for Long-Running Statements 124
AWR and ASH: Baselines, Regressions, and "What Changed?" 125
AWR Snapshots and Top SQL 125
ASH: Sampled Session History 126
Tracing and Targeted Instrumentation 126
Tag Work with `MODULE`, `ACTION`, and Client IDs 126
Enable a Targeted SQL Trace 127
Regression Proof: Before/After Checklists 127
Summary 127

15 Instance Resource Tuning: CPU, Memory, and I/O 129
CPU: When *Everything* Looks Like a Performance Problem 129
Find CPU-Heavy SQL 129
Reduce Parse and Library Cache Pressure 130
Memory: SGA, PGA, and the Cost of Spilling to Disk 130
SGA: Buffer Cache and Shared Pool 130
PGA: Sort/Hash Work Areas 131
TEMP Usage: Detect the Top Offenders 132
I/O: Separate Random Reads, Scans, and Logging 132
Identify the Dominant I/O Wait Classes 133
Redo and Commit Latency 133
Managing Mixed Workloads with Resource Manager 134
A Minimal Instance Tuning Checklist 134
Summary 135

16 High Availability and Performance: RAC, Data Guard, and Resilience 137
Define the Availability Target (RTO/RPO) and Performance Target 137
RAC: Scaling Reads and Surviving Node Failure 138
The Global Cache: Why "Hot" Blocks Hurt More in RAC 138
Identify Global Cache Waits 138
RAC-Friendly Design Patterns 139
Data Guard: Protecting Data Without Surprising Latency 139
SYNC vs. ASYNC Transport 140
Observe Transport and Apply Lag 140
Read Scaling: Offload Queries Safely 140

Route by Service and Workload Class 141
Failover Without a Thundering Herd 141
Resilience Patterns That Improve Performance 142
Summary 142

17 Performance Tuning Playbook: From Alert to Verified Fix 145
The Tuning Loop 145
Step 1: Triage the Incident (Minutes) 146
Ask Three Questions 146
Immediate Queries 146
Step 2: Isolate the Top Cost (Hours) 147
Top SQL by Total and Per-Execution Cost 147
Step 3: Choose the Right Fix Type 148
If It Is I/O Heavy 149
If It Is CPU-Heavy 149
If It Is Concurrency-Heavy 149
If It Is a Plan Regression 150
Step 4: Verify and Prove the Improvement 150
Step 5: Make the Fix Stick 150
Regression Prevention Checklist 150
Common Failure Modes (and How to Avoid Them) 151
Summary 151

Conclusion 153

References 155

Index 157

About the Author



Sanjay Mishra is a principal software engineer at Fidelity Investments, based in North Carolina, USA, where he leads a full-stack engineering squad building AI-powered applications. He brings over 14 years of experience in Oracle database systems and full-stack engineering, with prior roles as technical lead at IBM and Tata Consultancy Services.

A fellow of the British Computer Society (BCS) and a senior member of IEEE, Sanjay is an independent researcher whose work has been published in Q1 journals and conferences, spanning natural language-to-SQL (NL2SQL) systems, retrieval-augmented generation (RAG) architectures, LLM safety and guardrails, enterprise AI governance, and database performance engineering. He is the author of *The SQL Universe* and speaks frequently at academic and industry venues, including Weber State University, Columbia University, and conferences across the United States. He is a mentor and created his application https://easynlsql.netlify.app/, a free platform offering SQL exam preparation resources.

Sanjay holds an MBA in Finance from the University of Massachusetts Amherst and has a huge interest in personal finance.

About the Technical Reviewer



Krishnakumar Kunka Mohanram is an experienced cloud and database architect with more than 20 years of expertise in enterprise databases, cloud infrastructure, data platforms, high availability, disaster recovery, and AI-ready architectures. He specializes in Oracle Database, Oracle Cloud Infrastructure, Autonomous Database, Exadata, database modernization, complex migrations, and multicloud solutions. Krishnakumar is an Oracle ACE pro, IEEE senior member, and Sigma Xi full member. He is a coauthor of several technical books and an experienced technical reviewer, blogger, conference speaker, and industry awards judge. He has presented at Oracle CloudWorld and numerous international technology conferences and has received the Oracle MOS Community Gold Trophy for his sustained technical contributions to the Oracle community.

Acknowledgments

This book would not exist without the support of many.
To my colleagues and collaborators—thank you for the debates, the late nights, and the shared passion for getting things right.
To the team at Apress—thank you for your guidance and patience throughout this journey.
And to my family—Richa, Prisha, and Krishiv—thank you for giving me the time, the space, and the love to see this through.

Special thanks to my peers and mentors who challenged me to think deeper and tune smarter. Your insights helped shape this book.

Chapter 1
The Simple Truth About Oracle Performance



If you're reading this book, you're probably running Oracle in your organization (or planning to), and something isn't performing the way it should.

Maybe your application's UI feels sluggish and users are complaining. Maybe a critical business report takes minutes when it should take seconds. Maybe you built a PL/SQL-based ETL pipeline that worked fine in development but crawls in production.

This book is built on practical experience, not theory. Every concept, technique, and recommendation you'll find here has been used in real production environments, on real workloads, with real constraints. Whether you're a developer trying to speed up one slow query, a DBA managing a multi-terabyte warehouse, or an architect designing a new system, you'll get a repeatable approach to diagnosing and fixing performance issues.

You're in the right place. By the end of this book, you'll be able to improve the performance of an Oracle system and, just as importantly, understand why it improved so you can do it again next time.

In this opening chapter, we set the stage for everything that follows. You will see why most Oracle performance problems trace back to a small amount of SQL, how an SQL statement executes inside the database, and the most common root causes behind the symptoms you hear in production. You will also see why small, focused fixes usually deliver the biggest wins, and how the rest of the book is organized to help you move from diagnosis to verified improvement.

The 80/20 Rule in Action

As a database engineer who has worked with Oracle databases for years, I've noticed the same thing again and again: most performance problems come down to SQL. And when Oracle slows down, your application eventually slows down too. Yes,

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_1

you also need to review the overall architecture and make sure the right inputs are being passed from your application, reports, ETL pipelines, and so on (we'll cover that in a later chapter). But first and foremost, let's fix the basics, and since this is an Oracle performance tuning guide, let's fix the SQL.

> **Note**
> While this book focuses on Oracle, most of these SQL tuning concepts apply to other transactional (OLTP) database systems as well.

To fix SQL effectively, it helps to understand how a statement executes inside Oracle in the first place; that is where the most common failure points hide. First, remember that Oracle already has an optimizer. Ideally, if we write good, efficient SQL (and the database has the right supporting objects and statistics), the optimizer should choose a fast execution plan and run it.

More precisely, the optimizer chooses the plan with the lowest *estimated cost*; if statistics are inaccurate, data distribution shifts, or bind values change, that "lowest-cost" plan may not be the fastest plan in practice.

A simple way to think about it is a map: to reach the same destination, you can take multiple routes then pick the best one. Oracle's optimizer does the same thing: it considers different ways to execute the query and picks the best plan based on its cost estimates. As shown in Fig. 1.1, the SQL lifecycle has a few distinct stages, and each stage has its own failure modes.

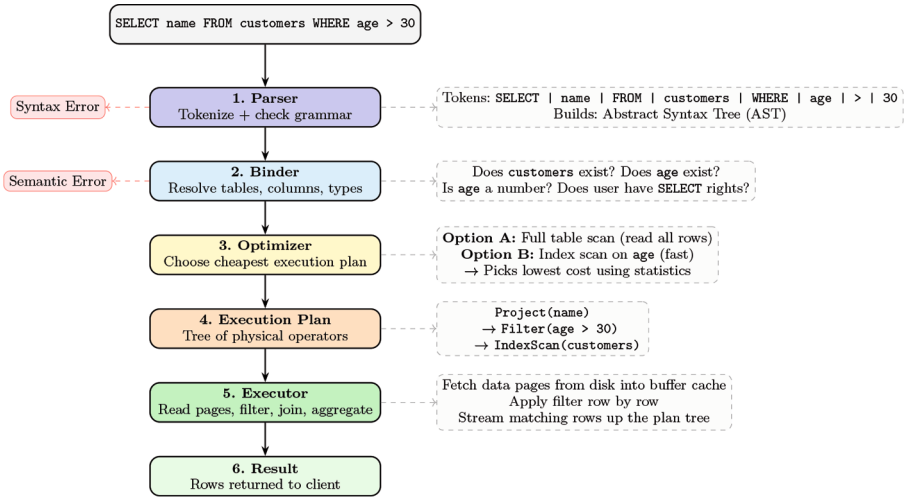


Fig. 1.1 High-level lifecycle of an SQL statement: parse, bind, optimize, plan, execute, and return results

In Fig. 1.1, the label "Syntax check" describes what happens during parsing: the parser checks the statement's grammar, and a syntax error is simply the outcome when that check fails. Follow Fig. 1.1 from top to bottom: the statement is parsed and validated, the optimizer weighs the candidate plans (a full scan vs. an index scan, for example) and picks the cheapest one, and the executor then carries that plan out and streams the rows back.

When statistics are inaccurate or data distribution changes, the "lowest-cost" plan may be expensive at runtime. The stage that matters most for this book is step 3, the optimizer: when performance goes wrong, it is very often because the optimizer chose an expensive route, usually based on bad inputs such as stale statistics or SQL that hides its intent.

What's Usually Going Wrong?

When Oracle slows down, it's usually because of a small set of common problems. The categories below cover the vast majority of real-world incidents I have diagnosed over the years. Don't worry about memorizing them; each one gets a full treatment later in the book. For now, just get a feel for how varied (and how ordinary) the root causes tend to be.

Indexing Issues

Indexes are the single most common lever in Oracle tuning and also the most commonly misused one. The recurring patterns look like this:

- No index at all: The database has no choice but to read every single row.
- You have an index but wrapped the column in a function (e.g., `UPPER(col)`): The index becomes useless.
- You did math on the column (e.g., `col + 1 > 30`): Same problem, the index is ignored.
- You used `LIKE '%text%'` with a wildcard at the start: The index can't help here.
- Too many indexes on one table: Every write now has to update all of them.

In every one of these cases, the fix is cheap once you see it; the trick is recognizing that the index (or its misuse) is the culprit rather than the database itself.

Query Design

Even with perfect indexes, the shape of the SQL itself can force Oracle to do far more work than the question requires:

- `SELECT *`: You're pulling columns you'll never even look at.
- Querying inside a loop (N+1): One query per row instead of one join for all.
- Correlated subquery: The inner query is secretly running thousands of times.
- Missing join condition (e.g., forgetting the `ON` in a join): Every row is matched with every other row.
- Adding `DISTINCT` or `ORDER BY` on a huge result set just to clean up a messy query.
- No `WHERE` clause, no row limiting: You asked for everything and the database delivered.

What these have in common is that the query asks for more than the application needs, and Oracle faithfully delivers it.

Data and Statistics

The optimizer is only as good as the information it is given. When its picture of the data is wrong, its plan will be too:

- Statistics haven't been refreshed: the optimizer is guessing based on old numbers.
- Comparing a number column to a string: Oracle quietly converts every row before comparing.
- Data is heavily skewed, but statistics say it's uniform: the optimizer picks the wrong plan.

Schema and Design

Some problems are baked in before a single query runs, in the way the tables themselves are laid out:

- **Giant table with no partitioning**: Every query scans more data than it should.
- **Too many joins**: The schema is over-normalized.
- **Too much duplicated data**: The schema is under-normalized.

Infrastructure

Then, there is the machinery underneath: memory, disk, network, and concurrency. Well-written SQL can still crawl when the platform is squeezed:

- Buffer cache too small: Going to disk on every read.
- Pulling millions of rows over the network when the application uses ten.
- Queries sitting in a queue, waiting for a locked row to be released.
- No parallelism configured for a query that scans 100M rows.

Application Layer

Finally, some of the worst database pain is inflicted from outside the database by how the application talks to it:

- Opening a new database connection for every single request.
- Running the same query hundreds of times with no caching.
- No pagination: The whole table lands in memory before the user sees page one.

These are common issues and are fixable, and the good news is that you do not need a complicated process to get them resolved. With a simple diagnostic approach, you can

- Identify the SQL that's really hurting you
- Read and interpret the execution plan
- Confirm whether indexes and statistics are doing their job
- Apply focused fixes that resolve the issue

This is the core rhythm of the book: locate the cost, understand why Oracle is paying it, then make the smallest change that you can verify.

How These Problems Show Up in Real Life

The categories above describe root causes. In practice, though, nobody calls you and says, "Our statistics are stale." What you actually hear (and see) is one of a handful of familiar symptoms. When Oracle performance drops, it's usually one (or more) of these troublemakers:

Slow SQL queries: You might have faced this from users: "The report is taking forever." "The page isn't loading." A single slow query running thousands of times a minute can quietly eat up CPU and I/O, and suddenly every query on the system is suffering, not just the bad one. What looks like a minor inconvenience at 9 a.m. can turn into a full outage by noon.

Inefficient execution plans: The code hasn't changed. The data looks fine. Nobody touched anything. But Oracle decided to take a completely different path to get the same result, and that path is ten times more expensive. You won't see it by reading the SQL. You have to look at what Oracle is actually doing under the hood, and that's exactly why these problems can linger for weeks before anyone figures out what's wrong.

Missing indexes: On a small table, a missing index is barely noticeable. On a table with half a billion rows, it means Oracle is reading and throwing away almost everything just to find the few rows you actually need. And when multiple users hit that same query at once, all those full scans compete for the same disk I/O, which means everyone waits.

Resource contention: This is when a performance problem stops being one person's problem and becomes everyone's problem. Sessions pile up, all waiting for the same CPU, the same memory, the same locked row. A query that normally takes a second now takes 30, not because anything broke, but because there's a traffic jam. The more users you have, the worse it gets.

Plan instability: This is the one that keeps DBAs up at night. Everything was fine yesterday. Nothing changed. But this morning, the overnight batch that usually finishes in two hours is still running at 8 a.m. when the business day starts. The plan switched for reasons the optimizer thought were good, and now SLAs are blown and someone is getting a phone call. A system that was working perfectly just stopped.

Every one of these symptoms traces back to the root causes we just cataloged, and every one of them yields to the same evidence-based approach you will learn in the next chapter.

Small Fixes, Big Impact

Here's the best part: you don't always need a complicated fix to get big results. A well-placed index, a small query rewrite, or a fresh round of statistics can take a system from sluggish to snappy in a hurry.

Why Simple Solutions Work
A performance problem can look complex, but the root cause is often simple. If you stick to the fundamentals, find the SQL that's doing the damage, fix the execution plan, and keep statistics current, you can resolve most Oracle issues faster than you'd expect.

Summary

This chapter set the foundation. Oracle performance issues are rarely random. Most of the time, a small number of SQL statements drives most of the load, and there are only a handful of repeatable reasons those statements become slow. In the chapters ahead, we'll go through the core ideas step by step: you'll learn to catch problems early, apply techniques that work in the real world, and lean on the Performance Tuning Checklist so issues get handled before they turn into outages.

Quick Checklist

Use the checklist below as a safe wrap-up for Chap. 1. It summarizes what you should verify before you start chasing specific queries or changing anything in production:

- Find the few statements that account for most DB time (the 80/20 rule).
- Do not guess from the SQL text alone; confirm the execution plan.
- Check the usual suspects first: missing or misused indexes, stale statistics, contention, and plan instability.
- Measure before and after so improvements are real, not accidental.

If you follow this sequence, the next chapters become easier because you will have a consistent way to move from symptom to evidence.

Next, Chap. 2 (*The Simple Diagnostic Approach*) shows where to start when someone says, "The system is slow," and how to narrow the search to the SQL that is actually hurting you. Let's jump in and make Oracle performance simpler, one query at a time.

Chapter 2
The Simple Diagnostic Approach



Chapter 1 showed you *why* Oracle slows down. This chapter shows you what to do about it, a simple, repeatable diagnostic workflow that takes you from "the system is slow" to the exact SQL statement responsible, backed by evidence rather than guesswork. You will walk through the core steps in order: identify the problem SQL, analyze its execution plan, check the obvious issues, and implement a focused fix. You will also see an alternate, practical path for environments where you do not have DBA-level access, so you can still make progress without waiting on special permissions.

When Performance Goes Wrong: Where Do You Actually Start?

The first thing most people do when something feels slow is open a database monitoring tool. That is usually the wrong first move.

Before you touch Oracle at all, go to your application logs.

A well-written application log will tell you exactly what it experienced: which operation failed, how long it waited, whether it hit a time-out or an error, and roughly when it happened. If your logs are doing their job, they will not just say, "Something was slow." They will say, "This specific API call took 47 seconds at 2:13 a.m." or "This batch job started failing with ORA-01555 after step 3." That is not a vague complaint. That is a lead.

Not every application has the provision (or discipline) to capture detailed database-related errors and timings. In those cases, the Oracle database alert log is also an important input for analysis, especially for ORA- errors, background process issues, and time-correlated warnings.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_2

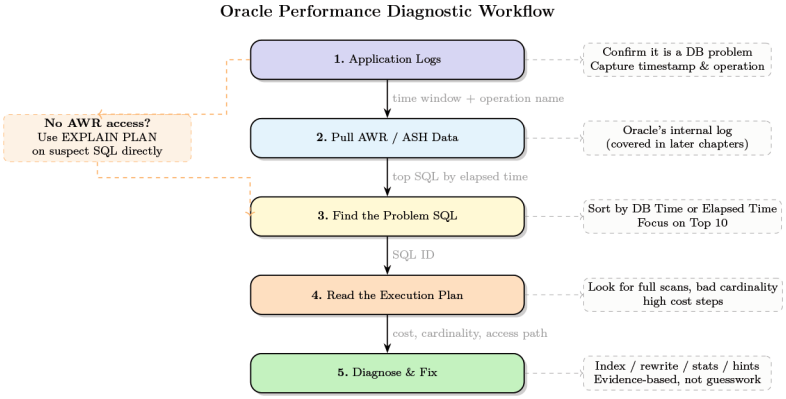


Fig. 2.1 A simple, evidence-based workflow for diagnosing Oracle performance issues. Start with application logs, then use Oracle diagnostics to narrow down to the exact SQL and execution plan

The log is your starting point because it answers the question you need to answer first: is this actually a database problem? Slow applications have many causes. Network latency, application bugs, memory pressure on the app server, and third-party API time-outs are not Oracle's fault and will not show up in a database report. Chasing a database bottleneck when the real problem is in the application layer wastes hours.

Once the logs point clearly at the database (slow query responses, connection time-outs, ORA-errors), you can move into Oracle's diagnostics with confidence. Figure 2.1 lays out the entire diagnostic workflow we'll follow in this chapter, from the first look at the application logs all the way to an evidence-based fix.

Read Fig. 2.1 from top to bottom: each step narrows the search, and the labels on the arrows show what each step hands to the next, including a time window, a ranked list of SQL, an SQL ID, and finally a plan to act on. Notice the orange branch on the left; if you do not have AWR access, you can skip straight from the logs to running `EXPLAIN PLAN` on the suspect SQL. The key takeaway is that every step is driven by evidence produced in the step before it.

Log quality matters. Vague logs produce vague investigations. Logs that capture operation names, durations, and timestamps turn a half-day debugging session into a 30-minute one. Once you have that time window and operation, you are ready to identify the SQL behind the slowdown. In plain words, the workflow boils down to five moves:

1. Start with application logs to confirm it is a database problem, and capture the time window and operation name.
2. Use that time window to pull AWR or ASH data for exactly that period.
3. Find the SQL that was running during that window.
4. Read the execution plan for that SQL.
5. Diagnose and fix with evidence, not guesses.

Note
AWR (Automatic Workload Repository) and ASH (Active Session History) are part of Oracle's Diagnostic Pack and require the appropriate licensing. If you do not have that license (or do not have access to the views/reports), use the alternate workflow later in this chapter based on application evidence, `V$` views you can access, and targeted `EXPLAIN PLAN` / `DBMS_XPLAN` analysis.

The rest of this chapter works through these steps one at a time, with just enough detail to help you apply them the next time you are on call.

Step 1: Identify the Problem SQL

Once you've confirmed the issue is in the database and you have a time window from your application logs, the next job is finding the exact SQL that's causing the pain.

If you have AWR access, start with the Top 10 SQL statements by DB time or elapsed time. These are your prime suspects, the statements consuming the most database resources during the problem window. Look for queries with high execution counts combined with long elapsed times. A query that runs a million times a day at 50 ms each is often a bigger problem than one that runs once and takes 10 seconds. The sheer volume matters as much as the individual duration. A single bad query that every user triggers on every page load is silently eating your database alive, and it will not look dramatic until the load doubles.

One thing worth keeping in mind, and this is a mindset shift that changes how you approach tuning, is to prioritize by where time is actually being spent, not just by which query text looks suspicious. Is the time in CPU, in I/O waits, or in lock waits? The answer points you in completely different directions.

High CPU usually means the query is doing too much logical work. This can be a bad join order, a missing index, or processing more rows than it needs to. High I/O waits mean data is not in the buffer cache and Oracle is going to disk repeatedly. This is often a missing index, too large a scan, or a cache that is simply undersized for the workload. Lock waits mean sessions are blocked behind each other, for example, a row locked by one transaction that others are queuing behind. Treating all three the same way wastes time.

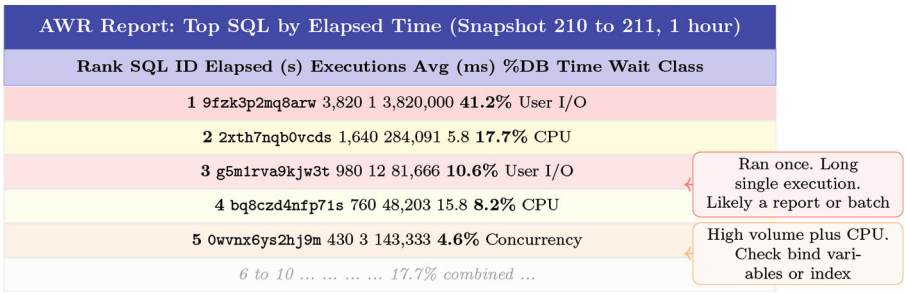


Fig. 2.2 Simplified AWR Top SQL output. The first SQL dominates total DB time inside the snapshot window

Using AWR to Find the Problem

AWR takes a snapshot of database activity at regular intervals (often one hour, though it is configurable). Those snapshots are kept only as long as the AWR retention settings allow; on many systems, the default retention is about eight days, but it can be shorter or longer depending on configuration (and storage). When you have a problem window from your application logs, identify the two snapshots that bracket that window and generate a report for that period. The AWR report shows you the top SQL by elapsed time, CPU, I/O, buffer gets, and executions, all scoped to your window. You are not looking at the whole history of the database. You are looking at what was happening precisely when things went wrong.

The most useful part of the report is the "Top SQL" section. It ranks statements by their contribution to total DB time during the snapshot window. The SQL at the top of that list is where you start. See Fig. 2.2 for a simplified view of what an AWR top SQL output looks like.

Two things are worth noticing in Fig. 2.2. First, the SQL in rank 1 consumed 41% of all database time in the window with a *single* execution; that pattern usually points to a report or batch job with a bad plan. Second, rank 2 ran 284,000 times at under 6 ms each; individually harmless, but its sheer volume made it the second-largest consumer. This is exactly why you rank by total DB time rather than by how slow one execution looks.

Using ASH to Go Deeper

Where AWR gives you aggregated statistics over a time period, ASH gives you a second-by-second sample of what every active session was doing. If a problem lasted five minutes, ASH recorded what each session was waiting on during every one of those seconds. This makes it invaluable for short spikes that AWR can average away.

Remember that ASH is part of the Oracle Diagnostic Pack, so its availability depends on licensing and on what your environment permits you to query.

The key view is `V$ACTIVE_SESSION_HISTORY`. Each row is one session sample, captured every second, with the SQL ID being executed, the wait event, and the wait class. Aggregate those samples over your problem window and you get a precise picture of where time was going. Listing 2.1 shows a query you can run directly against `V$ACTIVE_SESSION_HISTORY` to rank the SQL and wait events that dominated your window.

Listing 2.1 Ranking SQL and wait events by ASH sample count for a problem window

```
SELECT sql_id,
       event,
       wait_class,
       COUNT(*) AS sample_count,
       ROUND(COUNT(*) * 100 / SUM(COUNT(*)) OVER (), 1) AS pct_db_time
FROM v$active_session_history
WHERE sample_time BETWEEN :start_time AND :end_time
  AND session_type = 'FOREGROUND'
GROUP BY sql_id, event, wait_class
ORDER BY sample_count DESC
FETCH FIRST 15 ROWS ONLY;
```

The output of Listing 2.1 reads like a leaderboard: each row tells you how many one-second samples a given SQL ID and wait event combination accounted for, and `pct_db_time` converts that into a share of everything the database was doing. An SQL ID that appears in 800 out of 900 samples during a five-minute window is responsible for nearly all of what Oracle was doing in that period. That is your problem. Figure 2.3 shows how those same samples look when you plot them over time.

In Fig. 2.3, each bar is one moment in time, and its height is the number of active sessions; the colors show what those sessions were waiting on. The picture makes the diagnosis almost self-evident: a single SQL ID dominates 87% of the samples, and the dominant color (User I/O) tells you the time went to disk reads, not CPU.

At this point, you should have two things: a short list of SQL IDs that dominated the window and evidence of where the time went (CPU, I/O, or locks). With that, you can either keep drilling into the same incident or zoom out and look for slowdowns that build up over time.

Watch for Duration Creep

Also watch for duration creep in off-hours jobs. ETL pipelines, reports, and index rebuilds may have been designed to finish quietly overnight, but over time, they can spill into business hours as data grows. The job that used to finish at 4 a.m. now

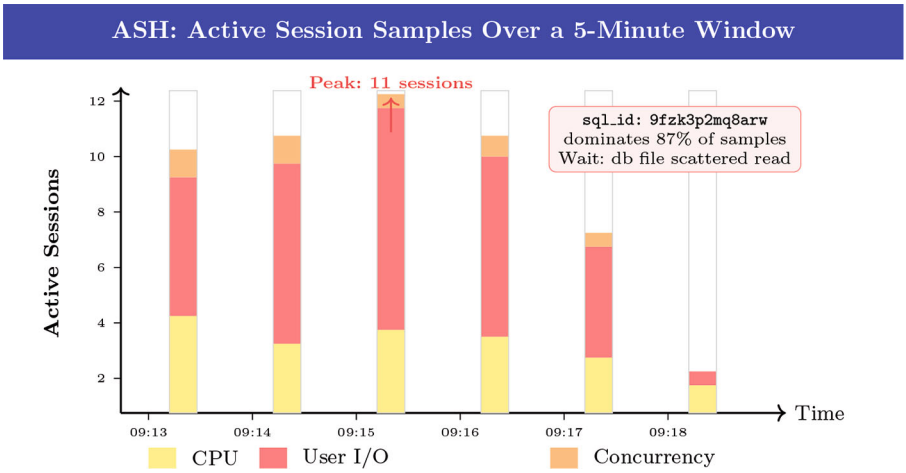


Fig. 2.3 ASH session samples over a five-minute spike. The dominant wait class tells you where time was going during the window

finishes at 8 a.m. Then 9 a.m. Then it is still running when the first users log in and a batch workload and an OLTP workload are competing for the same resources.

AWR makes this visible over time. If you pull AWR reports for the same overnight window across several weeks and watch the elapsed time of the same SQL creep upward week by week, you can catch the problem before it becomes a crisis. ASH will show you exactly which batch SQL is still active when the morning load begins. The overlap is often the smoking gun.

The Alternate Strategy: Often the Most Practical First Step

If you do not have AWR or ASH access (e.g., because the Oracle Diagnostic Pack license is not available or you do not have the required privileges), you can still make progress. The goal is the same: narrow down from "something is slow" to the few SQL statements most likely responsible. Figure 2.4 shows the four-step manual process.

The flow in Fig. 2.4 deliberately mirrors the main workflow: you still narrow from a suspect module to specific statements, and you still rank by evidence (here, the plan cost) before acting. The warning box in step 2 matters most in practice: a modest-cost statement sitting inside a loop can outweigh an expensive one that runs once.

When AWR access is restricted, and in many organizations it is, unless you are a DBA, this manual approach becomes not just an alternative but often the fastest and most actionable path to a diagnosis.

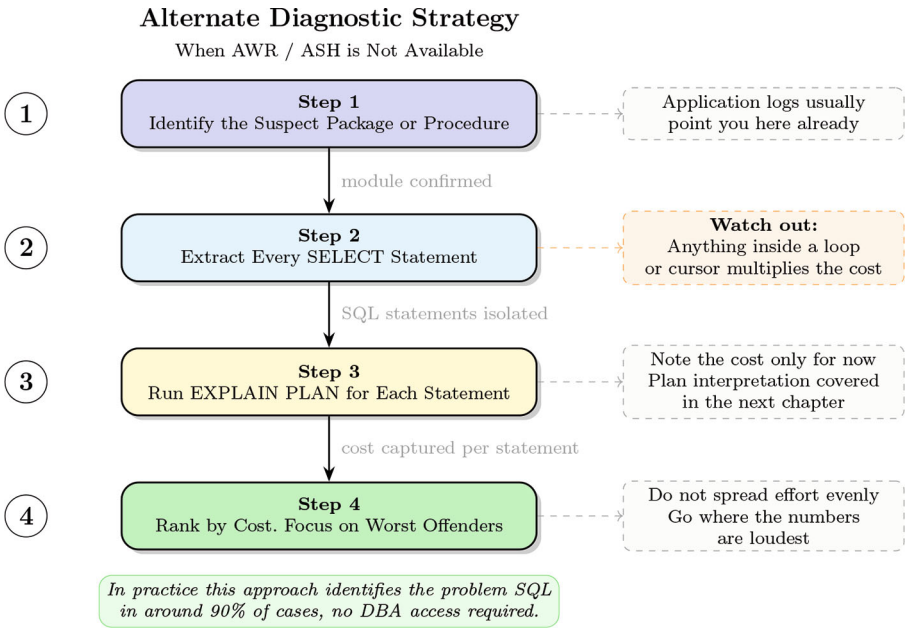


Fig. 2.4 Alternate diagnostic strategy when AWR or ASH is not available

This approach works well in practice because it puts control back in your hands. You don't need to raise a ticket, wait for approval, or explain why you need production access. You work with what you have: the code and a database connection.

Working through the code and running explain plans on the SQL inside the suspect module identifies the problem in around 90% of cases. It works consistently across development, UAT, and production environments, and it scales to any team regardless of how tightly the DBA team controls access.

1. Start by identifying the suspect package or procedure, the module that is known or suspected to be causing the slowdown. Your application logs will usually point you here already.
2. Extract every `SELECT` statement from that module. Pay particular attention to anything inside a loop or a cursor.
3. Run **EXPLAIN PLAN** for each statement and note the cost. Do not worry about interpreting the plan in detail yet; we will cover that thoroughly in the next chapter.
4. Rank the statements by cost and concentrate on the worst offenders. Do not spread your effort evenly. Go where the numbers are loudest.

This is not a workaround for when the real tools are not available. It is a disciplined, repeatable process that works in any environment and gives you a depth of understanding of your own code that no monitoring dashboard can.

Once you have a short list of candidate SQL statements (or a confirmed SQL ID), the next step is to understand how Oracle plans to execute that SQL.

Step 2: Analyze the Execution Plan

With a short list of suspect SQL in hand, the next step is to see *how* Oracle executes each statement. When you run an **EXPLAIN PLAN**, Oracle hands you a lot of numbers. Most of them you can ignore at first. Three of them matter most.

Cost: Oracle's way of saying, "How hard is this step going to be?" Think of cost as a difficulty score. The higher the number, the more work Oracle expects to do. It is not expressed in milliseconds; it is a relative unit. It is still extremely useful for finding the most expensive step in a complex query.

Cardinality: Oracle's estimate of how many rows will come out of a step. This is critically important because the optimizer builds its plan around these estimates. If Oracle thinks a step will return 10 rows and it actually returns 10,000, every decision downstream was built on a false assumption.

Bytes: How much data Oracle expects to move through a step. Larger numbers mean more memory pressure and more I/O.

When you see all three of those numbers elevated on the same step, stop looking. That is your trouble spot. Start tuning there.

Once you understand the plan and where the work is happening, take a quick pass through the basics. These checks are fast, and they eliminate the most common causes before you spend time on deeper rewrites.

Step 3: Check the Obvious Issues

Before reaching for anything sophisticated, rule out the basics. These three questions catch a surprisingly large percentage of real-world performance problems, and it takes minutes to check them:

- Are table and index statistics up to date?
- Are indexes missing on columns that are frequently used in `WHERE` clauses or joins?
- Is the SQL using bind variables, or is it hard-coding literal values?

After you answer these, you should be able to say *why* Oracle is doing more work than you expected, not just that it is slow. If something obvious is off, fix it first and retest. If the basics look good, you are ready to implement a focused change based on the evidence you collected from AWR, ASH, and the execution plan.

Step 4: Implement the Fix

Most performance problems, once properly diagnosed, have straightforward solutions. The diagnosis is the hard part. By the time you've worked through the steps above, the fix is usually obvious and falls into one of these categories:

- Add a missing index to support the filtering or join condition that is forcing the full scan.
- Update statistics so the optimizer has an accurate and current picture of the data.
- Rewrite the SQL to express the intent more clearly and give the optimizer a better chance of making a good decision.
- Use hints to steer Oracle toward a specific access path when you have good reason to believe you know better in that case.

The goal is not to apply every technique, it is to apply the smallest change that removes the measured bottleneck and then prove it with a re-test.

One important caution on indexes: they are not free. Every index you add has a maintenance cost. Oracle has to update that index on every `INSERT`, `UPDATE`, and `DELETE` against the table. Before you push a new index to production, test it against a realistic workload that includes writes, not just the query you are trying to speed up.

An index that cuts a critical `SELECT` from 30 seconds to 1 second can still be a net loss if it pushes your nightly batch job from 4 hours to 7 hours and it starts overlapping with business hours. Always measure the full picture before committing. A fix that helps one thing and quietly breaks another is not a fix at all.

Summary

This chapter gave you a practical starting point for performance incidents. Instead of guessing, you narrow the scope quickly, collect evidence, and then apply changes you can validate.

Quick Checklist
Use the checklist below as a safe wrap-up for Chap. 2. It keeps the workflow in the right order, logs first, evidence second, plan third, then a focused fix:

- Start with application logs: operation name, timestamp, and duration.
- Use AWR or ASH to find the SQL that dominated the window.
- Read the execution plan to locate the expensive step.
- Fix the obvious first (statistics, indexing, bind variables) then retest.

If you keep this sequence consistent, you will spend less time guessing and more time making changes you can defend with evidence.

Next, Chap. 3 (*Understanding EXPLAIN PLAN*) focuses on reading execution plans confidently and spotting the operations that drive database time.

Chapter 3
Understanding EXPLAIN PLAN: Your SQL's Roadmap to Performance



This chapter will not *fix* a slow query by itself, but it will help you *see* what Oracle is planning to do. The goal is simple, learn to read an execution plan well enough to spot where the time is likely being spent, even if you are not a DBA.

You will start with a small before-and-after example (a missing index), then learn how to generate plans in common tools, and finally learn the handful of numbers and operation names that tell you where the work really is. By the end of the chapter, you should be able to look at a plan and decide what to check next, instead of guessing.

What Is EXPLAIN PLAN?

Think of an SQL query like running errands across town. You can take different roads, visit places in a different order, and choose routes that avoid traffic. The fastest trip usually comes from a good route, not from driving faster.

Oracle has the same problem to solve for your SQL. Before it runs a statement, it needs to decide

- Which tables (or indexes) to access
- In what order to access them
- Which methods to use (full scans, index scans, join types)
- How much data to move between steps

Those decisions are the difference between a one-second query and a one-minute query, even when the SQL text looks the same.

EXPLAIN PLAN is you asking Oracle, "Show me the route you plan to take." The plan is Oracle's preview of the steps it *estimates* it will perform to return your results.

It is important to understand what EXPLAIN PLAN is (and is not). The plan you see is the optimizer's estimated plan at parse time, and it can differ from the *actual*

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_3

plan used at runtime due to adaptive optimization, bind variable values, and runtime statistics. When you need to see the plan that actually ran, execute the SQL and then use `DBMS_XPLAN.DISPLAY_CURSOR` (typically with `ALLSTATS LAST`) to view the cursor's real execution plan and row counts.

Why Is EXPLAIN PLAN So Important?

If you learn only one diagnostic tool from this book, make it this one. Four properties make EXPLAIN PLAN the workhorse of everyday tuning:

- It shows you **the strategy Oracle chose**, not just the SQL text.
- It helps you **catch expensive choices** before they become production incidents.
- It is often the **first place to look** when a query slows down.
- You can use it without AWR/ASH access in many environments.

That last point deserves emphasis: unlike AWR and ASH, EXPLAIN PLAN requires no special license or DBA privileges, which is exactly why the alternate diagnostic strategy from Chap. 2 leans on it so heavily.

A Simple Example

The best way to see what a plan tells you is to watch one change. We'll take a simple query, look at the plan Oracle produces without an index, add the index, and compare. Suppose you run the query in Listing 3.1.

Listing 3.1 Simple customer query without an index

```
SELECT customer_name
FROM   customers
WHERE  city = 'New York';
```

Oracle can either scan the `customers` table row by row (a **full table scan**) or use an index on `city` to jump straight to the matching rows. If the table has one million rows and there is no index, a full scan is usually what Oracle has to do. Table 3.1 shows what that plan looks like.

Table 3.1 Execution plan before index creation

ID	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		50,000	2 MB	1200
1	TABLE ACCESS FULL	CUSTOMERS	50,000	2 MB	1200

Read Table 3.1 from the bottom up: the `TABLE ACCESS FULL` operation on line 1 means Oracle touches every row in `CUSTOMERS`, and the cost of 1200 reflects that. Oracle expects to sift through the entire table just to keep the 50,000 New York rows.

Now give the optimizer a better route. Listing 3.2 creates an index on the `city` column.

Listing 3.2 Creating an index on the city column

```
CREATE INDEX idx_customers_city ON customers(city);
```

Run the same query again and the plan changes shape, as Table 3.2 shows.

Two things changed in Table 3.2. The access path is now `INDEX RANGE SCAN` followed by `TABLE ACCESS BY INDEX ROWID`: Oracle uses the index to find exactly the matching rows and then fetches only those. And the cost dropped from 1200 to 50. Remember that cost is a relative estimate of work, not a direct measure of runtime, but a big cost drop like this is usually a strong signal that the optimizer found a much cheaper route. Figure 3.1 puts the two plans side by side so you can see the difference at a glance.

The comparison in Fig. 3.1 is the essence of plan reading: same query, same result, radically different work. Whenever you tune, this before-and-after

Table 3.2 Execution plan after index creation

ID	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		50,000	2 MB	50
1	TABLE ACCESS BY INDEX ROWID	CUSTOMERS	50,000	2 MB	50
2	INDEX RANGE SCAN	IDX_CUSTOMERS_CITY	50,000		10

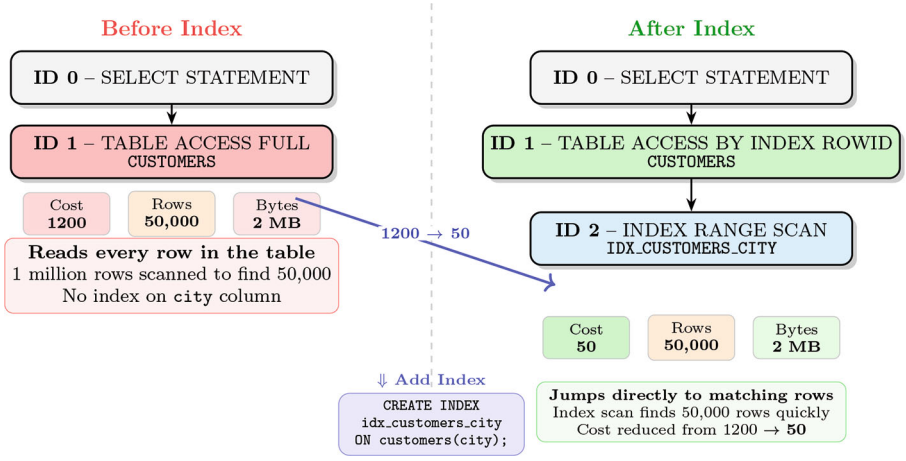


Fig. 3.1 Execution plan before and after index creation (conceptual view)

comparison of the plan (not just the runtime) is your proof that the fix worked for the right reason.

How to Generate EXPLAIN PLAN

You don't need any special setup to produce a plan; every mainstream Oracle client can do it in a couple of clicks. The steps below cover the three tools you are most likely to have on your desk: SQL Developer, Allround Automations' PL/SQL Developer, and TOAD. Pick whichever one you already use.

In SQL Developer

1. Open SQL Developer and connect to your database.
2. Paste your SQL query into a worksheet.
3. Right-click and choose **Explain Plan**.
4. Review the steps, cost, cardinality (rows), and access paths.

> **Tip**
> You can also press **F10** to run EXPLAIN PLAN directly.

SQL Developer presents the plan as an expandable tree, which makes it easy to spot the most expensive step: look for the operation with the highest cost and work outward from there.

In Allround Automations' PL/SQL Developer

1. Open Allround Automations' PL/SQL Developer and connect to your schema.
2. Paste your SQL in the command window.
3. Right-click and choose **Explain Plan** (or press **Ctrl+E**).
4. Review the plan in the tab/tree view.

> **Tip**
> You can save the plan or copy it into documentation.

The tree view in PL/SQL Developer shows the same information as SQL Developer's grid, just organized differently; the numbers to focus on (cost, cardinality, bytes) are identical.

In TOAD

1. Open TOAD and connect to your database.
2. Paste your SQL in the Editor window.
3. Click the **Explain Plan** button on the toolbar.
4. Review the graphical plan, including cost and access methods.

> **Tip**
> TOAD can also highlight expensive operations with color-coded indicators.

Whichever tool you use, the plan itself is the same; only the presentation differs. What matters next is knowing which parts of that plan to read first.

What to Look for in the Plan

A plan can contain dozens of columns, but three numbers carry most of the signal. Table 3.3 summarizes what each one means and why you should care.

The pattern to watch for in Table 3.3: when cost, cardinality, and bytes are all elevated on the *same* step, that step is your trouble spot. Beyond the numbers, a handful of operation names come up constantly, and it pays to recognize them on sight:

Table 3.3 Key metrics in Oracle execution plans

Metric	What it means	Why it matters
Cost	Oracle's relative estimate of total work (not time)	Useful for comparing options in a plan; lower is usually preferred, but it is not a guarantee of the fastest runtime (especially with stale or skewed statistics)
Cardinality	Number of rows Oracle expects to return	Bad estimates often lead to bad join methods and access paths
Bytes	Amount of data Oracle expects to process	Large values often mean heavier I/O and memory pressure

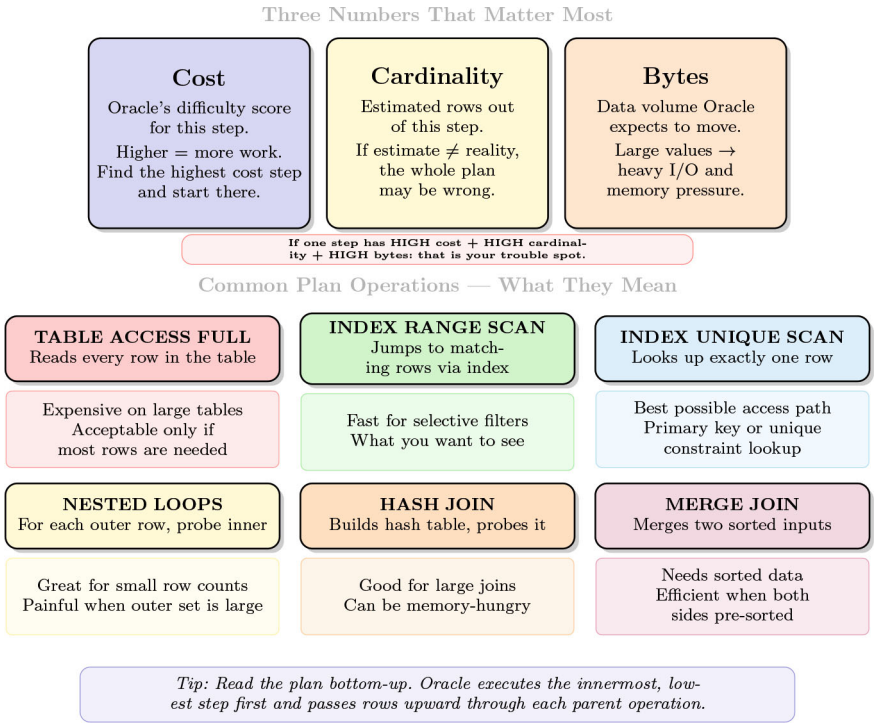


Fig. 3.2 Reading an Oracle execution plan: key metrics and common operations (conceptual guide)

- **TABLE ACCESS FULL**: Oracle scans the whole table (often expensive on large tables).
- **INDEX RANGE SCAN**: Oracle uses an index to find a range of keys (often faster).
- **NESTED LOOPS**: Great for small row counts; painful when the driving row set is large.
- **HASH JOIN**: Usually good for larger joins, but it can be memory-hungry.

None of these operations is inherently bad; a full table scan of a 200-row lookup table is perfectly fine. The question is always whether the operation fits the data volume. Figure 3.2 pulls the whole chapter together into a single visual guide to reading a plan.

Keep Fig. 3.2 handy as a reference while you practice: it maps each metric and operation to the question it answers, so you always know what to check next.

Summary

This chapter introduced **EXPLAIN PLAN** as the fastest way to see what Oracle is planning to do. You do not need to memorize every operation. You just need to recognize where the work is.

Quick Checklist
Use the checklist below as a safe wrap-up for Chap. 3. It reinforces the minimum you should verify every time you look at a plan, so you do not jump to rewrites before you understand where Oracle is paying the cost:

- Identify the access path: full table scan, index range scan, or unique scan.
- Check the key estimates: cost, rows (cardinality), and bytes.
- Look for expensive patterns: big sorts, Cartesian joins, and nested loops on large row sets.
- Pick one likely cause and test it (index, statistics, rewrite) then recheck the plan.

If you keep this sequence consistent, plan reading becomes a habit, and tuning becomes a series of small changes that you can verify.

Next, Chap. 4 (*EXPLAIN PLAN: Deep Dive*) goes further into plan structure and decision points so you can spot expensive access paths and join methods quickly and tune with confidence.

Chapter 4
EXPLAIN PLAN: Deep Dive



An execution plan is Oracle's "play-by-play" for how it intends to run your SQL. The **Optimizer** chooses an approach (access paths, join methods, sorts, and so on) based on its best estimate of the work required.

This chapter is a deeper, more practical guide to reading **EXPLAIN PLAN** quickly. You will learn to spot the expensive step, form a hypothesis, and decide what to test next.

The chapter builds the skill in three passes. First, Fig. 4.1 shows the reading habit that makes plans understandable, read them bottom-up. Next, Table 4.1 names the fields you will keep returning to (operation, object, cost, rows, predicates) and what each one tells you. Finally, the last two sections focus on the two decisions every plan encodes, how Oracle finds rows and how it joins row sources, using Figs. 4.2 and 4.3 as the mental models.

Core Components of the Plan

Execution plans are displayed as a hierarchy of operations. A simple rule that keeps you sane: **read the plan bottom-up**. Oracle starts at the most indented step, produces rows, and passes them up to the parent operations until it reaches the final `SELECT STATEMENT` (ID 0). Figure 4.1 illustrates this reading order on a typical plan.

As Fig. 4.1 shows, the deepest (most indented) operation runs first, and every parent consumes the rows its children produce. Once you internalize that order, the plan stops looking like a wall of text and starts reading like a story. Each line of that story is built from the same handful of components, which Table 4.1 breaks down.

Of the components in Table 4.1, rows (cardinality) and predicates are often the highest-signal clues (they explain *why* a plan shape was chosen). Cost is still useful as a compass—it is the optimizer's way of ranking alternatives—but it is a model, not a stopwatch. A lower-cost plan is usually preferred, yet real performance should

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_4

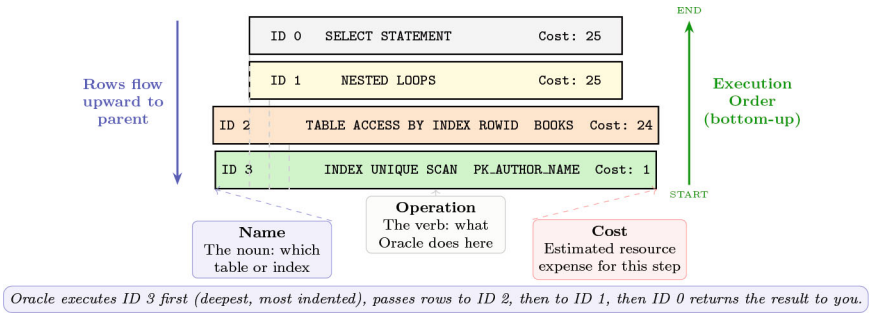


Fig. 4.1 How to read an execution plan bottom-up

Table 4.1 Core components of an Oracle execution plan

Component	Description	Why it matters
Operation	The action the database takes (scan, look up, join, sort)	Identifies *how* rows are produced and where work can explode
Name	The table or index being acted upon	Tells you *what* object is doing the work (and where an index is, or is not, used)
Cost	The optimizer's relative estimate of work, used to compare alternative plans	Lower is usually preferred, but cost is not elapsed time; validate performance with actual execution statistics when it matters
Rows (cardinality)	Estimated number of rows each step will produce/process	Bad estimates often lead to wrong join methods and access paths
Predicates	Filters (WHERE clauses) and join conditions applied at each step	Shows *where* filtering happens and whether predicates can use indexes

always be validated with actual execution statistics, and a full table scan can be the optimal choice when the query needs a large percentage of a table (high data volume, low selectivity). With the anatomy covered, let's look at the two big decisions every plan encodes: how the data is found and how tables are joined.

Access Paths: How the Data Is Found

Access paths answer a basic question: *how will Oracle find the rows?* On small tables, many paths work fine. On large tables, the difference between an index lookup and a full scan can be the difference between a snappy application and a user staring at a spinner. Figure 4.2 ranks the paths you will meet most often, from the most expensive to the cheapest.

Keep the ordering in Fig. 4.2 in mind as a tuning compass: the goal is not "index good, full scan bad"—the goal is to match the access path to data volume and selectivity. When a predicate is selective, an index-driven plan often wins; when a

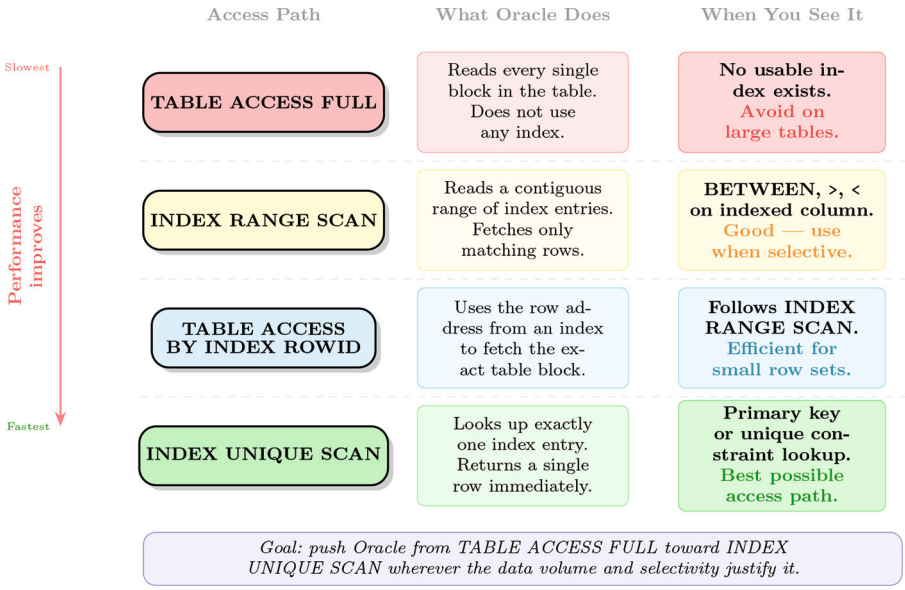


Fig. 4.2 Oracle access paths from worst to best

Table 4.2 Oracle access paths

Access path	Description	Performance	Tuning goal
TABLE ACCESS FULL	Reads most/all blocks of the table and applies filters as it goes	Can be optimal when many rows are needed; expensive when only a small subset is needed	Prefer when selectivity is low; avoid when a selective index exists
INDEX UNIQUE SCAN	Probes a unique index to find a single key (often followed by a table fetch by ROWID if needed)	Extremely fast for single-row lookups.	Ideal for primary-key / unique-key access
INDEX RANGE SCAN	Scans a range of index entries that match a predicate, then typically fetches the matching table rows via **TABLE ACCESS BY INDEX ROWID**	Fast for selective filters; can degrade if the range returns a large portion of the table	Best for selective predicates (e.g., ranges) and well-designed composite indexes

query needs a large percentage of the rows, a full table scan can be the most efficient route. Table 4.2 summarizes the common paths and the situations they fit.

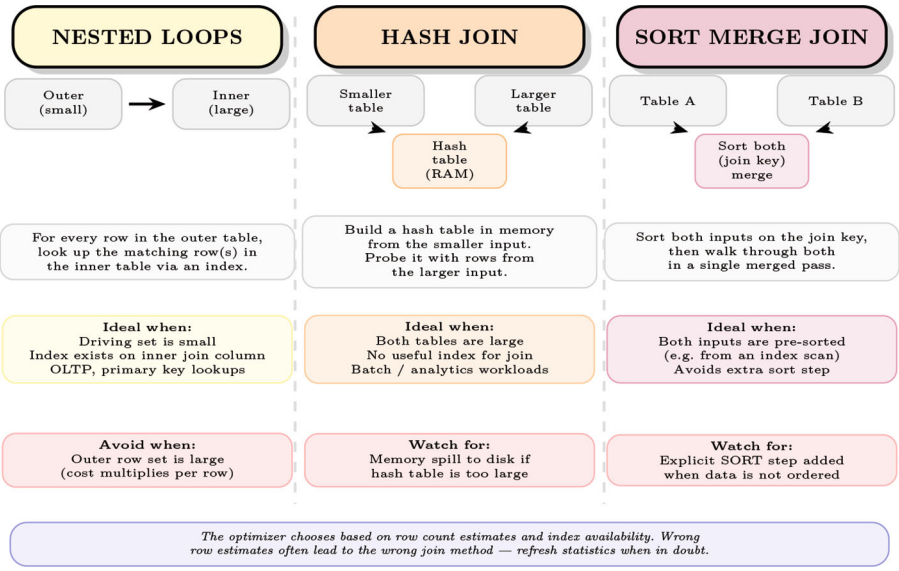


Fig. 4.3 Oracle join methods: mechanics and when to use each

The optimizer chooses among these by estimating how much data will be touched. That is why cardinality and predicates matter so much: they drive whether Oracle expects an index path to be cheaper than scanning the table.

Join Methods: How Tables Are Connected

Join methods explain *how* Oracle combines row sources. The same SQL can produce very different performance depending on whether Oracle uses **NESTED LOOPS**, a **HASH JOIN**, or a **SORT MERGE JOIN**. The "best" choice depends mainly on row counts and whether useful indexes exist. Figure 4.3 shows how each method actually moves data.

The mechanics in Fig. 4.3 explain the rule of thumb you will use constantly: nested loops shine when the driving row set is small and an index exists on the inner table, while hash joins win when both inputs are large. Table 4.3 condenses this into a quick reference.

When a plan surprises you, Table 4.3 is the first thing to check: a nested loop driving millions of rows, or a hash join used where a selective index exists, usually means the optimizer's row estimates are off.

Table 4.3 Oracle join methods

Join method	Ideal scenario	Mechanics
NESTED LOOPS	Small result set joined to a large table	For every outer row, look up the matching inner row via index
HASH JOIN	Two large, unsorted tables	Smaller table loaded into a hash table; larger table scanned and matched
SORT MERGE JOIN	Both inputs must be sorted by join key	A costly SORT step is applied to both tables before merging

Practical Examples

The fastest way to build intuition is to look at a few common patterns. Before you read the details, keep two questions in mind so you always know what you are trying to extract from the plan:

- **Where the work is**, scans, big joins, sorts.
- **Why Oracle chose it**, row estimates and available indexes.

If you can answer those two questions, you can usually propose a small change that is worth testing.

Example A: The Problematic Plan (High Cost)

This one is a classic: wrapping an indexed column in a function can prevent Oracle from using the index, which often pushes the optimizer toward a full table scan. Listing 4.1 shows the query; Table 4.4 shows the plan it produces.

Listing 4.1 Applying a function on an indexed column disables the index

```
-- Assume an index exists on last_login_date
SELECT *
FROM   users
WHERE  TO_CHAR(last_login_date, 'YYYY') = '2024';
```

Analysis and Fix
Applying `TO_CHAR` to `last_login_date` forces Oracle to evaluate the function for every row, which typically prevents an index on the raw date column from being used.

Rewrite the predicate as a date range so Oracle can do an **INDEX RANGE SCAN**, as shown in Listing 4.2.

Table 4.4 Problematic execution plan with full table scan

ID	Operation	Name	Cost	Predicate
1	TABLE ACCESS FULL	USERS	25000	FILTER(TO_CHAR(LAST_LOGIN_DATE,'YYYY')='2024')

Listing 4.2 Rewritten query that allows index range scan

```
SELECT *
FROM   users
WHERE  last_login_date >= DATE '2024-01-01'
AND    last_login_date <  DATE '2025-01-01';
```

Example B: The Efficient Plan (Low Cost)

Now for the kind of plan you want to see. The query in Listing 4.3 joins two tables on a very selective predicate, and Table 4.5 shows the plan Oracle chooses.

Listing 4.3 Efficient query using indexed lookup and NESTED LOOPS

```
SELECT b.title, a.name
FROM   books   b
JOIN   authors a ON b.author_id = a.author_id
WHERE  a.name = 'Sanjay';
```

Table 4.5 Efficient execution plan using indexes

ID	Operation	Name	Cost
0	SELECT STATEMENT		25
1	NESTED LOOPS		25
2	INDEX UNIQUE SCAN	PK_AUTHOR_NAME	1
3	TABLE ACCESS BY INDEX ROWID	BOOKS	24

Why This Plan Is Efficient
Step 2 is the key: **INDEX UNIQUE SCAN** on `PK_AUTHOR_NAME` means Oracle can locate the single matching author row for `a.name = 'Sanjay'` with a handful of index probes instead of scanning the authors table. That produces a very small driving row set (typically one row).

(continued)

With that one author row in hand, Oracle uses **NESTED LOOPS** (step 1) to join to `BOOKS`: for each outer row (the author), it looks up the matching book rows using the join key. The **TABLE ACCESS BY INDEX ROWID** on `BOOKS` (step 3) reflects the final table fetch for those matching book rows. Because the outer row set is tiny and the lookups are targeted, this plan minimizes the total data touched.

Example C: The Cost of Sorting

Sorting is easy to overlook because it hides in an innocent-looking `ORDER BY`. Listing 4.4 asks for the entire inventory in date order, and Table 4.6 shows what that request costs.

Listing 4.4 ORDER BY clause introducing a SORT operation

```
SELECT *
FROM   inventory
ORDER BY stock_date DESC;
```

Table 4.6 Execution plan showing expensive SORT ORDER BY

ID	Operation	Name	Cost
1	SORT ORDER BY		500
2	TABLE ACCESS FULL	INVENTORY	150

In Table 4.6, the `SORT ORDER BY` step alone costs more than three times the scan that feeds it; the sort, not the read, is the expensive part of this query.

Tip
If you need results in `stock_date` order frequently, add an index on `stock_date` so Oracle can read the rows in order and avoid an explicit **SORT ORDER BY**.

Summary

This chapter took **EXPLAIN PLAN** beyond the basics and focused on the parts that usually matter most in real tuning: access paths, join methods, and sorting.

Quick Checklist

Use the checklist below as a safe wrap-up for Chap. 4. It reinforces the sequence that keeps tuning grounded in the plan, first find the expensive step, then confirm access path, join method, and sorts:

- Read plans bottom-up and identify the first expensive operation.
- Check whether predicates allow index usage or accidentally disable it.
- Match join method to row counts, nested loops for small, hash join for large.
- Watch for sorts, and add an index when order is required frequently.

If you keep this sequence consistent, you will spend less time debating theory and more time making changes you can verify.

Next, Chap. 5 (*Proactive Optimization*) shifts from diagnosis to prevention: writing SQL and maintaining metadata so slow queries are less likely to reach production in the first place.

Chapter 5
Proactive Optimization



Reactive tuning, waiting for a complaint and then diagnosing the cause, is necessary but insufficient. The strategies in this chapter are applied before problems emerge, at the point of writing SQL, designing predicates, structuring aggregations, and maintaining the metadata the optimizer depends on. Organized into five layers, this framework gives developers a systematic approach to building Oracle applications that perform well under real production conditions and continue to perform well as data volumes grow.

The chapter progresses layer by layer. It starts with SQL writing habits that remove accidental complexity, then moves through predicate design and aggregation choices, and ends with statistics practices that keep the optimizer's model aligned with reality. As you read, use Fig. 5.1 as the map, it summarizes the five layers in one place and shows how they connect.

Write SQL That Gives the Optimizer Good Choices

The optimizer can only work with the SQL you hand it. The habits in this first layer cost nothing at development time, yet each one removes a common way of forcing Oracle into an expensive plan.

Avoid `SELECT *`

`SELECT *` retrieves every column from the table regardless of what the calling application actually uses. The cost is threefold: increased logical I/O to read wider rows, increased network transmission, and elimination of covering index access.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_5

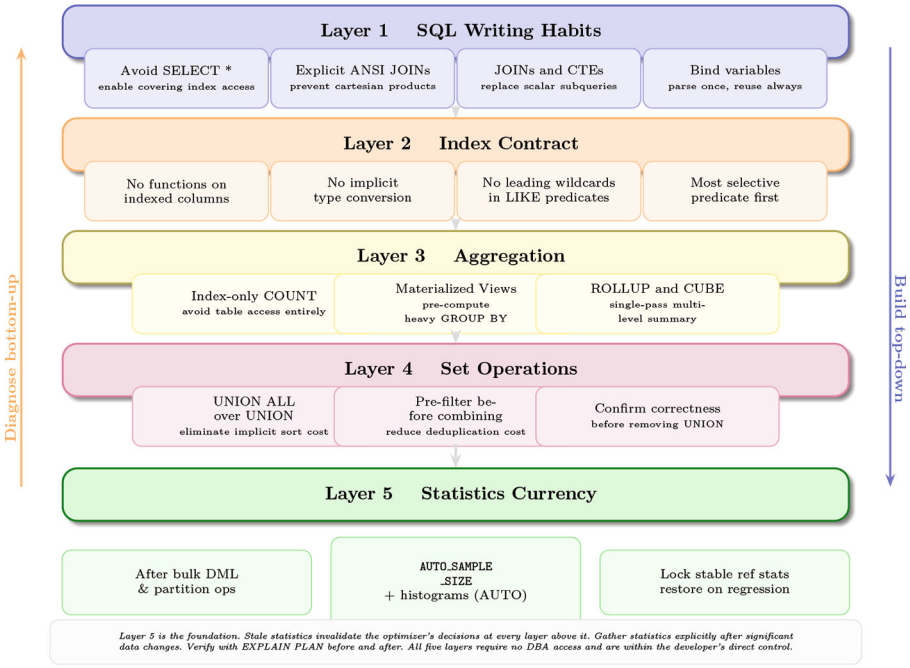


Fig. 5.1 Proactive Optimization Framework (five developer-controlled layers)

A **covering index** is one that contains all columns referenced by a query, allowing Oracle to satisfy the query entirely from the index without accessing the table. This is one of the most efficient access paths available. `SELECT *` makes it impossible, because no index contains every column.

Listing 5.1 shows the wasteful form; Listing 5.2 shows the disciplined alternative.

Listing 5.1 Inefficient, full row retrieval

```
SELECT * FROM employees;
```

Listing 5.2 Efficient, projection to required columns only

```
SELECT employee_id, first_name, last_name, status
FROM employees;
```

If an index exists on `(employee_id, last_name, status)`, Listing 5.2 can be satisfied entirely from the index. Listing 5.1 cannot. Additionally, specifying columns explicitly makes the query resilient to schema changes and communicates data dependencies clearly to anyone maintaining the code.

Use Explicit ANSI JOIN Syntax

Implicit comma-style joins mix join conditions with filter conditions in the `WHERE` clause. A missing or misplaced join condition produces a Cartesian product without any syntax error.

Listing 5.3 Implicit join, Cartesian product risk

```
SELECT *
FROM orders, customers
WHERE orders.customer_id = customers.customer_id;
```

Compare that with the explicit form, where the join condition has a dedicated home.

Listing 5.4 Explicit ANSI JOIN, join condition structurally separated

```
SELECT o.order_id, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_status = 'OPEN';
```

With explicit JOIN syntax, the `ON` clause contains join conditions and the `WHERE` clause contains filter conditions. A missing `ON` clause is a parse error, not a silent logic error. For outer joins, use standard `LEFT JOIN`, `RIGHT JOIN`, and `FULL OUTER JOIN` syntax rather than Oracle's proprietary `(+)` notation.

Replace Subqueries with JOINs or CTEs

Correlated Subqueries and `IN` Subqueries

The optimizer can transform many `IN` subqueries into joins internally, but the transformation depends on cardinality estimates and optimizer settings. Writing the join explicitly removes that dependency.

Listing 5.5 `IN` subquery

```
SELECT employee_id, first_name
FROM employees
WHERE department_id IN (
    SELECT department_id
    FROM departments
    WHERE location_id = 1700
);
```

The equivalent join expresses the same logic in a form the optimizer can cost directly.

Listing 5.6 Equivalent JOIN (often easier for the optimizer to cost)

```
SELECT e.employee_id, e.first_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id
WHERE d.location_id = 1700;
```

Scalar Subqueries in the SELECT List

A scalar subquery in the SELECT list executes once per row in the outer result set. This is a structural property of the query.

Listing 5.7 Scalar subquery, executes N times for N outer rows

```
SELECT e.employee_id,
       (SELECT department_name
        FROM departments d
        WHERE d.department_id = e.department_id) AS dept_name
FROM employees e;
```

Rewriting the lookup as a join (here via a CTE) turns N executions into one set operation.

Listing 5.8 Join/CTE rewrite, avoids per-row scalar subquery execution

```
WITH dept_info AS (
    SELECT department_id, department_name
    FROM departments
)
SELECT e.employee_id, d.department_name
FROM employees e
JOIN dept_info d ON e.department_id = d.department_id;
```

In Oracle, a common table expression (CTE) may be inlined rather than materialized, depending on version and cost. The key improvement here is structural: the lookup becomes a join operation that the optimizer can cost and execute as a set, instead of running a scalar subquery once per output row.

Use Bind Variables

Oracle identifies SQL statements by hashing their text. Statements with different literal values produce different hashes and are treated as distinct SQL, each requiring a full parse and optimization cycle.

Listing 5.9 Literal values, each statement is unique

```
SELECT * FROM orders WHERE customer_id = 1042;
SELECT * FROM orders WHERE customer_id = 1891;
SELECT * FROM orders WHERE customer_id = 2204;
```

With a bind variable, all three calls share one statement, one parse, and one plan:

Listing 5.10 Bind variable, parsed and optimized once

```
SELECT * FROM orders WHERE customer_id = :cust_id;
```

> **Note**
> Oracle's bind variable peeking samples the first bind value to inform plan selection. Adaptive cursor sharing can maintain multiple plans for the same statement when different bind values warrant different access paths due to data skew.

Respect the Index Contract

An index is a sorted data structure built on a specific expression, typically a raw column value. Oracle can perform an index lookup only when the predicate presents that same expression, unmodified, on the left side of a comparison operator. Any transformation applied to the indexed column, whether a function call, an arithmetic operation, or an implicit type conversion, renders the existing index unusable for that predicate.

Do Not Apply Functions to Indexed Columns

The most common contract violation is wrapping the column in a function. The two queries below return the same rows, but only the second one can use an index on `order_date`.

Listing 5.11 Function on indexed column, index not usable

```
-- Index exists on order_date (DATE)
SELECT order_id, order_amount
FROM orders
WHERE TO_CHAR(order_date, 'YYYY') = '2023';
```

The fix is to move the transformation to the constant side and express the intent as a range.

Listing 5.12 Range predicate, index fully usable

```
SELECT order_id, order_amount
FROM orders
WHERE order_date >= DATE '2023-01-01'
  AND order_date <  DATE '2024-01-01';
```

When the query text cannot be modified, a function-based index built on the same expression provides an alternative.

Listing 5.13 Function-based index as a last resort

```
CREATE INDEX idx_orders_year ON orders (TO_CHAR(order_date, 'YYYY'));
```

> **Note**
> A function-based index introduces storage overhead and increases write cost on the base table. Use it when the query cannot be changed. When the query can be changed, fix the query.

Avoid Implicit Type Conversion

When a column of one data type is compared against a value of a different type, Oracle implicitly converts one side to match the other.

Listing 5.14 Type mismatch, implicit conversion on every row

```
-- customer_id is NUMBER(10)
SELECT * FROM customers WHERE customer_id = '1042';
```

The conversion lands on the column side, which disables the index. Matching the literal's type to the column costs nothing and keeps the index in play.

Listing 5.15 Matching types, index usable

```
SELECT * FROM customers WHERE customer_id = 1042;
```

Implicit conversion is especially dangerous because the query still returns correct results; nothing warns you that every execution is silently scanning the table.

Avoid Leading Wildcards

A trailing wildcard provides a starting point: `LIKE 'Laptop%'` starts at the first index entry beginning with `Laptop` and scans forward. A leading wildcard provides no starting point, so the index cannot be used.

Listing 5.16 Leading wildcard, full table scan

```
SELECT product_id, product_name
FROM products
WHERE product_name LIKE '%Laptop';
```

Move the wildcard to the end and the index becomes a usable starting point.

Listing 5.17 Trailing wildcard, INDEX RANGE SCAN available

```
SELECT product_id, product_name
FROM products
WHERE product_name LIKE 'Laptop%';
```

When mid-string search is a genuine requirement, Oracle Text indexes are the correct solution.

Write Predicates for Correctness and Index Use (Not Evaluation Order)

SQL is nonprocedural: the optimizer is free to rearrange and combine predicates as it chooses, using statistics to estimate selectivity and cost. In other words, the order you write predicates in the `WHERE` clause does not guarantee the order they are evaluated.

Predicate *form* matters much more than predicate *order*. Focus on writing predicates that preserve index usability (no functions on indexed columns, matching data types, sargable ranges), and keep statistics current so the optimizer can estimate selectivity correctly. If you order predicates from most selective to least selective, treat it as a readability convention for humans, not as a performance technique.

Listing 5.18 Multiple predicates (order does not force evaluation order)

```
SELECT transaction_id, transaction_amount
FROM transactions
WHERE transaction_amount > 50000        -- highly selective predicate
  AND transaction_date > SYSDATE - 30;  -- additional filter
```

Aggregation

Aggregation queries (counts, sums, and grouped summaries) tend to touch far more rows than they return, which makes them prime candidates for proactive design. Three techniques cover most situations: index-only aggregation, materialized views, and `ROLLUP`.

Use Index-Only Aggregation

For `COUNT` queries with a selective `WHERE` clause, an index on the filter column allows Oracle to count matching index entries without accessing the table.

Listing 5.19 COUNT accelerated by index

```
CREATE INDEX idx_orders_status ON orders(status);

SELECT COUNT(*)
FROM orders
WHERE status = 'SHIPPED';
```

For queries that also retrieve non-indexed columns, a composite index that includes both the filter column and the projected columns enables a covering index scan.

Listing 5.20 Composite covering index for aggregation

```
CREATE INDEX idx_orders_status_amount ON orders(status, order_amount);

SELECT status, SUM(order_amount)
FROM orders
WHERE status IN ('SHIPPED', 'PENDING')
GROUP BY status;
```

Pre-compute with Materialized Views

A materialized view stores a pre-computed aggregation as a physical table.

Listing 5.21 Materialized view with fast refresh

```
CREATE MATERIALIZED VIEW LOG ON orders
WITH ROWID, SEQUENCE (order_date, order_amount)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW daily_sales_mv
```

```
BUILD IMMEDIATE
REFRESH FAST ON COMMIT
AS
SELECT TRUNC(order_date)   AS sales_day,
       COUNT(*)             AS total_orders,
       SUM(order_amount)    AS total_revenue
FROM orders
GROUP BY TRUNC(order_date);
```

Oracle's **Query Rewrite** feature can transparently redirect queries against the base table to the materialized view when the summary satisfies the query.

Use `ROLLUP` for Multilevel Aggregation

When a report needs subtotals at several levels (by region, by department within region, and a grand total), running one `GROUP BY` per level scans the data repeatedly. `ROLLUP` produces every level in a single pass.

Listing 5.22 ROLLUP producing all grouping levels in one pass

```
SELECT region_id, department_id, job_id,
       COUNT(*) AS headcount,
       SUM(salary) AS total_salary
FROM employees
GROUP BY ROLLUP(region_id, department_id, job_id);
```

One statement, one scan, and every aggregation level the report needs; the alternative would be three separate queries stitched together with `UNION ALL`.

Set Operations

Set operations look innocent in SQL text, but some of them carry a hidden sort of the entire combined result. Choosing the right operator upfront avoids paying that cost.

Prefer `UNION ALL` over `UNION`

`UNION` deduplicates its combined output by sorting the full result set and discarding duplicate rows. `UNION ALL` concatenates without deduplication.

Listing 5.23 UNION, SORT UNIQUE on full combined result

```
SELECT customer_id, order_date, order_amount FROM current_orders
UNION
SELECT customer_id, order_date, order_amount FROM archived_orders;
```

If current and archived orders can never overlap, the deduplication buys nothing. `UNION ALL` skips it entirely.

Listing 5.24 UNION ALL, no sort, immediate concatenation

```
SELECT customer_id, order_date, order_amount FROM current_orders
UNION ALL
SELECT customer_id, order_date, order_amount FROM archived_orders;
```

Caution
Replacing `UNION` with `UNION ALL` is a semantic change. Confirm that duplicates cannot exist between the two result sets or that their presence is acceptable.

Maintain Current Statistics

Oracle's Cost-Based Optimizer makes plan decisions based on object statistics: table row counts, column value distributions, index clustering factor, and partition statistics. When those numbers are stale, every technique in the previous four layers can be undermined by a plan built on wrong assumptions.

The `DBMS_STATS` package gives you direct control, but it requires appropriate privileges. In many production environments, statistics gathering is restricted to DBAs or automated maintenance jobs, so treat the examples below as guidance and coordinate with your DBA team if you cannot run them directly.

To refresh a single table after a significant data change, use the call shown here.

Listing 5.25 Gather statistics for a specific table

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(
    ownname          => 'schema_name',
    tabname          => 'orders',
    estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE,
    method_opt       => 'FOR ALL COLUMNS SIZE AUTO',
    cascade          => TRUE
);
```

When many objects changed at once (after a release or a large data load), gathering at the schema level is more practical.

Listing 5.26 Gather statistics for an entire schema

```
EXEC DBMS_STATS.GATHER_SCHEMA_STATS(
    ownname          => 'schema_name',
    estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE,
    method_opt       => 'FOR ALL COLUMNS SIZE AUTO',
    cascade          => TRUE
);
```

Two less obvious capabilities round out the toolkit. For small reference tables whose plans are already good, locking the statistics prevents the nightly automated job from disturbing them.

Listing 5.27 Locking statistics to prevent automated refresh

```
-- Lock statistics on a reference table that changes rarely
EXEC DBMS_STATS.LOCK_TABLE_STATS('schema_name', 'country_codes');
```

And if a statistics refresh triggers a plan regression, you do not have to live with it. Oracle keeps a history you can roll back to.

Listing 5.28 Restoring previous statistics after a regression

```
-- Restore statistics to a prior snapshot to reverse a plan regression
EXEC DBMS_STATS.RESTORE_TABLE_STATS(
    ownname          => 'schema_name',
    tabname          => 'orders',
    as_of_timestamp  => SYSTIMESTAMP - INTERVAL '1' DAY
);
```

Figure 5.1 assembles all five layers into a single reference view of the framework this chapter has built.

As the figure's side arrows show, the framework reads in two directions depending on what you're doing: top-down when you are building new SQL (start with Layer 1's writing habits), and bottom-up when you are diagnosing something already slow (start with Layer 5's statistics, since stale statistics undermine every layer above them).

Summary

The five layers of the Proactive Optimization Framework address performance at every level within the developer's direct control:

- **Layer 1** eliminates waste in the SQL itself: unnecessary column retrieval, inefficient join syntax, repeated subquery execution, and redundant parse overhead from literal values.
- **Layer 2** ensures that predicates are written in a form that allows existing indexes to be used: columns kept bare, types matched, wildcards placed correctly, and predicates expressed in sargable forms the optimizer can cost accurately.

- **Layer 3** directs aggregation work to the most efficient available mechanism: index-only counts, pre-computed materialized view summaries, and single-pass `ROLLUP` operations.
- **Layer 4** prevents hidden sort costs from set operations by using `UNION ALL` where correctness permits and by filtering source sets aggressively when deduplication is unavoidable.
- **Layer 5** ensures that the optimizer has accurate data to work with, through timely statistics gathering and selective histogram collection.

Quick Checklist

Use the checklist below as a safe wrap-up for Chap. 5. It keeps the framework actionable by turning the five layers into a short sequence you can apply during development and code review:

- Project only required columns and use bind variables to keep parsing stable.
- Keep predicates *sargable*, no functions on indexed columns, no implicit type conversion, and careful wildcard placement.
- Use set-based rewrites (JOINs/CTEs) and choose set operations intentionally (`UNION ALL` vs. `UNION`).
- Gather statistics after meaningful data change, and verify critical plans with **EXPLAIN PLAN**.

If you keep these checks consistent, many performance problems never make it to production, and the ones that do are easier to diagnose.

Next, Chap. 6 (*Advanced SQL Discovery: Hunting for Performance Bottlenecks*) moves from prevention to visibility: using Oracle's runtime views to find the SQL statements that actually consume the most resources and to see the plans they are using in production.

Chapter 6
Advanced SQL Discovery: Hunting for Performance Bottlenecks



Oracle keeps score on every SQL statement it runs: how often it executed, how much CPU it burned, how many blocks it touched, and which plan it used. In this chapter, you will use `V$SQL`, `V$SQL_PLAN`, and `V$ACTIVE_SESSION_HISTORY`, plus `DBMS_STATS`, to answer the questions that matter in production: *Which SQL is expensive? Which plan is it really using? What was it waiting on when things slowed down? Are bad estimates pushing the optimizer into a poor plan?*

The chapter follows the same evidence chain you will use during a real incident. First, you will pull a short list of high-impact statements from `V$SQL`. Next, you will confirm the *runtime* plan with `DBMS_XPLAN.DISPLAY_CURSOR` so you are tuning what Oracle actually executed. Then, you will use ASH to explain *why* the statement was slow (I/O, CPU, or concurrency). Finally, you will validate the optimizer inputs—statistics and histograms—so the fix addresses the root cause rather than the symptoms.

The Shared Pool: Oracle's Central Statement Repository

When Oracle parses and runs a statement, it caches the cursor, its execution plan, and cumulative runtime statistics in the **Shared Pool** (inside the SGA). Those details are exposed through the dynamic performance views, so you can query them with plain SQL.

The big win over **EXPLAIN PLAN** is that the Shared Pool shows *what happened*, not just *what Oracle thought it would do*. **EXPLAIN PLAN** is a forecast. The cursor cache is the postgame report: the plan that was used, how many times it ran, and where the time and I/O went.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_6

V$SQL: The Statement Register

V$SQL is the front door. It has one row per *child cursor* for each statement currently in the Shared Pool. A single SQL text can split into multiple child cursors when something about the execution environment changes (schemas, NLS settings, bind sensitivity, optimizer settings, and so on). Use SQL_ID to identify the statement and CHILD_NUMBER to keep the variants straight. Table 6.1 lists the columns you will use constantly and why each one matters.

Of the columns in Table 6.1, BUFFER_GETS deserves special attention. It is easy to miss a problem query when you look only at DISK_READS: if the data is already cached, physical reads stay low while the statement still churns through thousands (or millions) of logical reads. When BUFFER_GETS per execution is high, the query is doing too much work to return its rows, even if it never touches disk.

Identifying High-Impact Queries

When you hunt for offenders in V$SQL, focus on *per-execution* cost. Totals mostly tell you what ran the most, not what hurts the most each time it runs.

Listing 6.1 Queries with the highest average disk I/O per execution

```
SELECT sql_id,
       executions,
       disk_reads,
       buffer_gets,
       elapsed_time,
       ROUND(disk_reads    / NULLIF(executions, 0))    AS
    avg_disk_reads,
       ROUND(buffer_gets   / NULLIF(executions, 0))    AS
    avg_buffer_gets,
       ROUND(elapsed_time  / NULLIF(executions, 0))    AS
    avg_elapsed_us,
       SUBSTR(sql_text, 1, 80)                         AS sql_preview
FROM   v$sql
WHERE  executions > 100
ORDER BY avg_disk_reads DESC
FETCH FIRST 20 ROWS ONLY;
```

Disk reads only tell part of the story. The next query ranks by logical I/O instead, which catches the statements that hammer the buffer cache even when nothing goes to disk.

Table 6.1 Key columns in `V$SQL` and their tuning relevance

Column	Data type	Description	Tuning relevance
SQL_ID	VARCHAR2(13)	Unique identifier for the SQL statement	Used to join to `V$SQL_PLAN`, `V$ACTIVE_SESSION_HISTORY`, and `DBMS_XPLAN`
CHILD_NUMBER	NUMBER	Child cursor number under the parent SQL_ID	Identifies cursor sharing misses and bind-sensitive plans
EXECUTIONS	NUMBER	Total number of executions since cursor load	Distinguishes one-off queries from high-frequency workload drivers
CPU_TIME	NUMBER	Cumulative CPU time in microseconds	Identifies CPU-intensive queries; divide by EXECUTIONS for per-run cost
ELAPSED_TIME	NUMBER	Cumulative wall-clock time in microseconds	Primary measure of user-visible impact
DISK_READS	NUMBER	Total disk blocks read across all executions	High values indicate missing indexes, poor clustering, or full table scans
BUFFER_GETS	NUMBER	Total logical reads from the buffer cache	High logical reads indicate inefficient access paths even when disk reads are low
ROWS_PROCESSED	NUMBER	Cumulative number of rows processed/returned across all executions	With EXECUTIONS, gives average rows per execution; useful to spot high-row statements and validate whether the cursor returns far more rows than expected
PARSE_CALLS	NUMBER	Number of parse requests for this cursor	High parse calls relative to executions indicate cursor reuse failures
LAST_ACTIVE_TIME	DATE	Most recent execution timestamp	Confirms whether the statement is part of the current workload
PLAN_HASH_VALUE	NUMBER	Hash of the execution plan	Detects plan changes without reading the full plan text
SQL_TEXT	VARCHAR2(1000)	First 1000 characters of the SQL statement	Entry point for identifying the statement; use SQL_FULLTEXT for the complete text

Listing 6.2 Queries with the highest average logical I/O per execution

```
SELECT sql_id,
       executions,
       buffer_gets,
       rows_processed,
       ROUND(buffer_gets    / NULLIF(executions,    0)) AS
    avg_buf_gets,
       ROUND(rows_processed / NULLIF(executions,    0)) AS avg_rows,
       ROUND(buffer_gets    / NULLIF(rows_processed, 0)) AS
    buf_per_row,
       SUBSTR(sql_text, 1, 80)                          AS sql_preview
FROM   v$sql
WHERE  executions > 100
ORDER BY avg_buf_gets DESC
FETCH FIRST 20 ROWS ONLY;
```

A third pattern worth hunting for is parse overhead. A statement that is parsed nearly as often as it executes is usually built with literals instead of bind variables, or is failing to reuse cursors, and this query flushes those candidates out.

Listing 6.3 Queries with excessive parse-to-execution ratio (literal SQL candidates)

```
SELECT sql_id,
       parse_calls,
       executions,
       ROUND(parse_calls / NULLIF(executions, 0), 2) AS parse_ratio,
       SUBSTR(sql_text, 1, 80)                       AS sql_preview
FROM   v$sql
WHERE  executions > 100
  AND  (parse_calls / NULLIF(executions, 0)) > 0.5
ORDER BY parse_calls DESC
FETCH FIRST 20 ROWS ONLY;
```

Interpreting `parse_ratio` is straightforward: it is `PARSE_CALLS / EXECUTIONS`. A value near `0.0` means the statement was parsed once and executed many times (good reuse). A value near `1.0` means it is being reparsed on nearly every execution, which is a strong signal of literal SQL (no bind variables), hard-parse churn, or cursor sharing failures. Values above `1.0` can occur when a cursor is reparsed multiple times between executions.

`V$SQL_PLAN`: Examining Plans in the Cursor Cache

Once you have a `SQL_ID`, the next step is to see the plan Oracle cached for it. `V$SQL_PLAN` holds the plan steps for cursors currently in the Shared Pool. Each row is one operation in the plan for a specific `SQL_ID`/`CHILD_NUMBER` combination.

Retrieving Cached Plans with DBMS_XPLAN.DISPLAY_CURSOR

You could query V$SQL_PLAN directly, but DBMS_XPLAN.DISPLAY_CURSOR formats the same information far more readably. Given a SQL_ID, one call returns the full cached plan.

Listing 6.4 Retrieving the execution plan for a known SQL_ID

```
SELECT *
FROM TABLE(
    DBMS_XPLAN.DISPLAY_CURSOR(
        sql_id          => 'your_sql_id_here',
        cursor_child_no => 0,
        format          => 'ALLSTATS LAST'
    )
);
```

The format parameter controls how much detail comes back; Table 6.2 summarizes the options you will actually use.

Of the options in Table 6.2, ALLSTATS LAST is the one to reach for first: it shows estimated rows (E-Rows) next to actual rows (A-Rows), and a large gap between the two is the classic signature of a statistics problem.

Locating a SQL_ID for a Statement

If you know the SQL text but not its SQL_ID, a pattern search against V$SQL gets you there.

Table 6.2 DBMS_XPLAN.DISPLAY_CURSOR format options

Format value	Content displayed	Use case
BASIC	Operation, object name, rows	Quick identification of access paths
TYPICAL	BASIC plus cost, bytes, predicate information	Standard plan review
ALL	TYPICAL plus column projections, parallel details	Complete plan analysis
ALLSTATS	Runtime statistics from all executions	Cumulative actuals across all runs
ALLSTATS LAST	Runtime statistics from the most recent execution	Actuals for the last single run
IOSTATS LAST	I/O statistics from the last execution	Buffer gets and disk reads per step

Listing 6.5 Locating the SQL_ID for a statement using V$SQL

```
SELECT sql_id,
       child_number,
       executions,
       plan_hash_value,
       SUBSTR(sql_text, 1, 100) AS sql_preview
FROM   v$sql
WHERE  sql_text LIKE '%orders%'
  AND  sql_text NOT LIKE '%v$sql%'
ORDER BY last_active_time DESC;
```

Detecting Plan Changes with PLAN_HASH_VALUE

Every distinct plan hashes to a distinct PLAN_HASH_VALUE, which makes plan changes trivially detectable: if the same SQL_ID shows two hash values with very different average elapsed times, you are looking at a plan regression. This query lays that history out.

Listing 6.6 Querying plan history for a specific SQL_ID

```
SELECT sql_id,
       plan_hash_value,
       executions,
       elapsed_time,
       ROUND(elapsed_time / NULLIF(executions, 0)) AS avg_elapsed_us,
       last_active_time
FROM   v$sql
WHERE  sql_id = 'your_sql_id_here'
ORDER BY last_active_time DESC;
```

V$ACTIVE_SESSION_HISTORY: Real-Time Wait Analysis

V$ACTIVE_SESSION_HISTORY (ASH) is Oracle's black box recorder. Once per second, it samples every active session and records what it was doing (or waiting for). If you need the same information after the in-memory buffer has wrapped, DBA_HIST_ACTIVE_SESS_HISTORY is the AWR-persisted copy. Table 6.3 covers the ASH columns that carry the diagnostic signal.

The columns in Table 6.3 combine naturally: SQL_ID tells you *what* was running, EVENT tells you *why* it was slow, and BLOCKING_SESSION tells you *who* was in the way.

Table 6.3 Key columns in V$ACTIVE_SESSION_HISTORY

Column	Description	Tuning relevance
SAMPLE_TIME	Timestamp of the sample	Isolates the exact time window of a performance event
SESSION_ID	SID of the sampled session	Links to V$SESSION for session details
SQL_ID	SQL being executed at sample time	Identifies which query the wait is associated with
EVENT	Wait event name at sample time	What the session was waiting for
WAIT_CLASS	Category of the wait event	High-level grouping: User I/O, Concurrency, System I/O, CPU
BLOCKING_SESSION	SID of the blocking session	Identifies the holder in lock contention scenarios
MODULE	Application module name	Identifies which application component generated the workload
SQL_PLAN_HASH_VALUE	Plan hash at sample time	Confirms which plan was in use during the incident

Common Wait Events and Their Meaning

The EVENT column is only useful if you can read it. Thousands of wait events exist, but in practice, a handful account for nearly all the samples you will see

`db file sequential read`: Single-block read, typically from index access
`db file scattered read`: Multi-block read, typically from a full table scan
`latch: cache buffers chains`: Buffer cache latch contention on hot blocks
`enq: TX - row lock contention`: Waiting on row locks held by another transaction
`log file sync`: Commit latency; often indicates commits are too frequent

Learn to recognize these five and you can classify most incidents at a glance: the first two point to I/O and access-path problems, the middle two to contention, and the last to commit design.

ASH Queries for Incident Analysis

When an incident is in progress (or just ended), three queries cover most of the investigation. The first ranks the SQL that dominated the window.

Listing 6.7 Top SQL statements by wait time in a specific time window

```
SELECT sql_id,
       event,
       COUNT(*)                                  AS sample_count,
       ROUND(COUNT(*) / SUM(COUNT(*)) OVER ()
             * 100, 1)                           AS pct_of_window
FROM   v$active_session_history
WHERE  sample_time >= SYSDATE - INTERVAL '30' MINUTE
  AND  session_type = 'FOREGROUND'
GROUP BY sql_id, event
ORDER BY sample_count DESC
FETCH FIRST 20 ROWS ONLY;
```

If the dominant events are lock-related, the next question is who is blocking whom. ASH records the blocker's session ID with every sample, so the chains fall straight out.

Listing 6.8 Identifying blocking session chains from ASH

```
SELECT ash.sql_id,
       ash.event,
       ash.session_id          AS waiting_sid,
       ash.blocking_session    AS blocking_sid,
       COUNT(*)                AS wait_samples,
       ROUND(COUNT(*) * 1, 0) AS estimated_wait_seconds
FROM   v$active_session_history ash
WHERE  ash.sample_time >= SYSDATE - INTERVAL '1' HOUR
  AND  ash.blocking_session IS NOT NULL
GROUP BY ash.sql_id, ash.event,
         ash.session_id, ash.blocking_session
ORDER BY wait_samples DESC;
```

For incidents that happened hours or days ago, the in-memory ASH buffer has usually wrapped; switch to the AWR-persisted history instead.

Listing 6.9 Historical incident analysis using DBA_HIST_ACTIVE_SESS_HISTORY

```
SELECT TO_CHAR(sample_time, 'YYYY-MM-DD HH24:MI') AS sample_minute,
       sql_id,
       event,
       COUNT(*)                                    AS samples
FROM   dba_hist_active_sess_history
WHERE  sample_time >= SYSDATE - 7
  AND  session_type  = 'FOREGROUND'
GROUP BY TO_CHAR(sample_time, 'YYYY-MM-DD HH24:MI'),
         sql_id, event
ORDER BY sample_minute, samples DESC;
```

Optimizer Statistics Management

The Cost-Based Optimizer makes every plan choice from object statistics. When those stats are stale, missing, or simply wrong, row estimates drift away from reality—and that is how you end up with plans that look "reasonable" on paper but perform terribly under load.

Gathering Table and Index Statistics

The workhorse call refreshes a table and, via `cascade`, all of its indexes in one operation.

Listing 6.10 Gathering statistics for a single table and its indexes

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(
    ownname          => 'SCHEMA_NAME',
    tabname          => 'TABLE_NAME',
    cascade          => TRUE,
    estimate_percent => DBMS_STATS.AUTO_SAMPLE_SIZE,
    method_opt       => 'FOR ALL COLUMNS SIZE AUTO'
);
```

Histograms for Skewed Column Distributions

Basic statistics assume values are evenly distributed. When a column is heavily skewed (say, 95% of orders are `COMPLETE` and 1% are `PENDING`), that assumption produces wildly wrong estimates, and a histogram is the cure. For a low-cardinality column, request a frequency histogram explicitly.

Listing 6.11 Forcing a frequency histogram on a specific column

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(
    ownname    => 'SCHEMA_NAME',
    tabname    => 'ORDERS',
    method_opt => 'FOR COLUMNS SIZE 254 ORDER_STATUS'
);
```

For a high-cardinality column with skew (amounts, timestamps), let Oracle choose the bucket strategy with `SIZE AUTO`.

Listing 6.12 Forcing a height-balanced histogram on a high-cardinality column

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(
    ownname    => 'SCHEMA_NAME',
```

```
    tabname    => 'ORDERS',
    method_opt => 'FOR COLUMNS SIZE AUTO ORDER_AMOUNT'
);
```

Before adding histograms, check what already exists; `DBA_TAB_COL_STATISTICS` shows the histogram type and bucket count per column.

Listing 6.13 Checking existing histogram information

```
SELECT column_name,
       histogram,
       num_buckets,
       num_distinct,
       num_nulls,
       last_analyzed
FROM   dba_tab_col_statistics
WHERE  owner      = 'SCHEMA_NAME'
  AND  table_name = 'ORDERS'
ORDER BY column_name;
```

Locking Statistics

For small reference tables whose data (and good plans) rarely change, locking the statistics protects them from the automated maintenance job.

Listing 6.14 Locking statistics on a stable reference table

```
EXEC DBMS_STATS.LOCK_TABLE_STATS('SCHEMA_NAME', 'COUNTRY_CODES');
```

A quick check against `DBA_TAB_STATISTICS` confirms which tables are protected.

Listing 6.15 Verifying lock status

```
SELECT table_name,
       stattype_locked
FROM   dba_tab_statistics
WHERE  owner = 'SCHEMA_NAME'
  AND  table_name IN ('COUNTRY_CODES', 'CURRENCY_CODES',
    'REGION_CODES');
```

Restoring Statistics After a Plan Regression

If a statistics refresh sends a critical plan sideways, you can put the old statistics back while you investigate; Oracle retains history for 31 days by default.

Listing 6.16 Restoring table statistics to the state from 24 hours ago

```
EXEC DBMS_STATS.RESTORE_TABLE_STATS(
    ownname         => 'SCHEMA_NAME',
    tabname         => 'ORDERS',
    as_of_timestamp => SYSTIMESTAMP - INTERVAL '1' DAY
);
```

To pick the right restore point, review when statistics were actually gathered.

Listing 6.17 Viewing the history of statistics gathers for a table

```
SELECT start_time,
       end_time,
       savtime,
       operation,
       target
FROM   dba_optstat_operations
WHERE  target LIKE 'SCHEMA_NAME.ORDERS%'
ORDER BY start_time DESC
FETCH FIRST 10 ROWS ONLY;
```

Correlating V$SQL, ASH, and Statistics

Each tool in this chapter answers one question; the real power comes from chaining them. The script below walks a single investigation end to end: find the expensive statement, see what it waits on, pull its runtime plan, and check the statistics feeding the optimizer.

Listing 6.18 End-to-end correlation: SQL_ID to wait events to plan

```
-- Step 1: Identify a high-cost statement
SELECT sql_id,
       ROUND(elapsed_time / NULLIF(executions, 0)) AS avg_elapsed_us,
       SUBSTR(sql_text, 1, 80)                     AS sql_preview
FROM   v$sql
WHERE  executions > 50
ORDER BY avg_elapsed_us DESC
FETCH FIRST 5 ROWS ONLY;

-- Step 2: Check wait events for that SQL_ID (substitute value from
    Step 1)
SELECT event,
       COUNT(*) AS samples
FROM   v$active_session_history
WHERE  sql_id    = 'target_sql_id'
  AND  sample_time >= SYSDATE - INTERVAL '1' HOUR
GROUP BY event
ORDER BY samples DESC;
```

```sql
-- Step 3: Retrieve runtime plan (substitute SQL_ID and child number)
SELECT *
FROM TABLE(
    DBMS_XPLAN.DISPLAY_CURSOR('target_sql_id', 0, 'ALLSTATS LAST')
);

-- Step 4: Check statistics on the affected table
SELECT column_name, histogram, num_distinct, last_analyzed
FROM   dba_tab_col_statistics
WHERE  owner      = 'SCHEMA_NAME'
  AND  table_name = 'AFFECTED_TABLE'
ORDER BY column_name;
```

Figure 6.1 shows the same investigation as a decision flow, including how the dominant wait class steers the middle of the analysis.

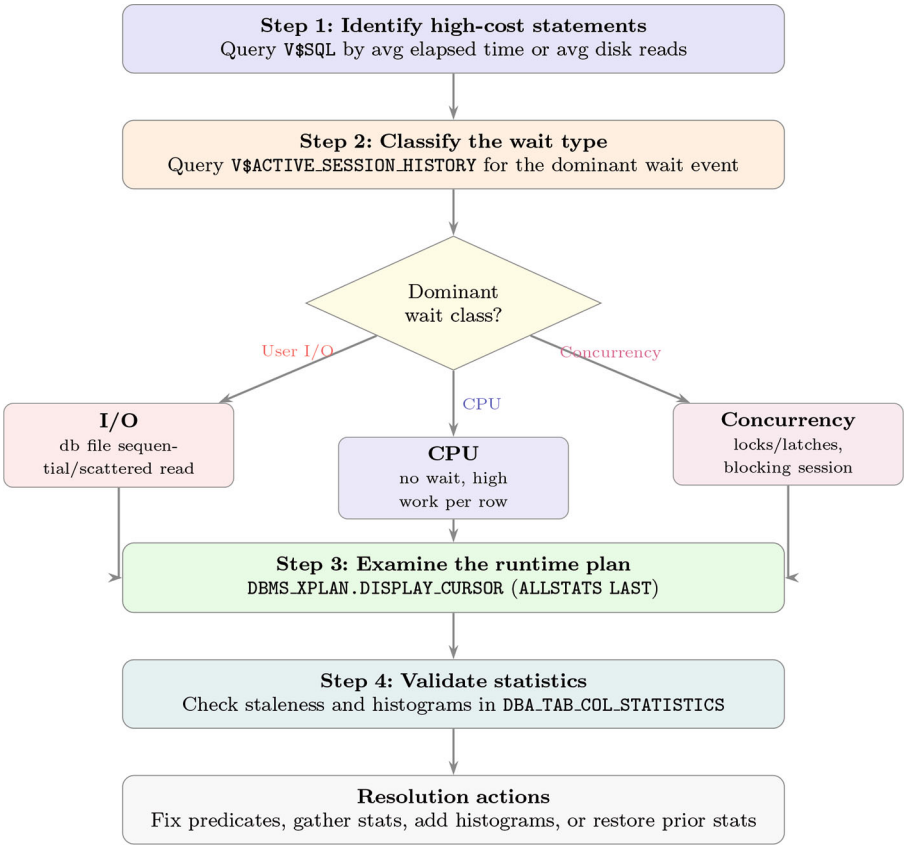


Fig. 6.1 Advanced SQL discovery workflow for correlating V$SQL, ASH, and statistics

The branch in the middle of Fig. 6.1 is the pivotal step: classifying the dominant wait as I/O, CPU, or concurrency decides where your tuning effort goes. Whichever branch you take, the flow always ends the same way, at the plan and the statistics behind it.

Summary

As shown in Fig. 6.1, effective SQL tuning is a short evidence loop: identify the expensive statement, confirm the *runtime* plan, explain the dominant wait, and then validate the statistics that shaped the optimizer's decision. In practice, you do that with four tools—and each one answers a different question:

- Start with `V$SQL` to build the hit list: which cursors are expensive, how often they run, and whether the plan is stable (via `PLAN_HASH_VALUE`).
- Next, use `DBMS_XPLAN.DISPLAY_CURSOR` to see the plan Oracle really used, plus actual rows and timing. When A-ROWS and E-ROWS are far apart, treat it as a statistics/estimates problem until proven otherwise.
- Then, consult `V$ACTIVE_SESSION_HISTORY` (ASH) to explain *why* users felt pain by showing what sessions were waiting for during the incident window.
- Finally, use `DBMS_STATS` to keep the optimizer honest by gathering, locking, and restoring statistics when plans regress.

Taken together, these tools keep you out of guesswork: they force you to connect symptoms (waits) to a specific cursor and plan, and then to the inputs that produced it.

Quick Checklist
Use the checklist below as a repeatable sequence. Each item is intentionally ordered so you collect identifiers first, then verify the plan, and only then change structure (SQL or indexes).

- In `V$SQL`, rank by *per-execution* cost first (avg elapsed, avg buffer gets, avg disk reads), then sanity-check totals.
- Capture `SQL_ID`, `CHILD_NUMBER`, and `PLAN_HASH_VALUE` so you know *which* cursor and *which* plan you are tuning.
- Pull the runtime plan with `DBMS_XPLAN.DISPLAY_CURSOR` (`ALLSTATS LAST`) and hunt for big A-ROWS vs. E-ROWS gaps.
- Use ASH to classify the bottleneck quickly: User I/O, CPU, or Concurrency (and the top event behind it).
- Fix the inputs before the structure: verify stats freshness and histograms, then retest before adding indexes or rewriting SQL.

If you follow that sequence, your fixes are both faster and safer: you can explain what changed, prove the impact, and roll back confidently if needed.

Next, Chap. 7 (*Comprehensive Index Strategy*) shows how to turn what you discovered here—especially high buffer gets, high disk reads, and I/O-dominant ASH profiles—into concrete index designs, validation queries, and an index maintenance mindset.

Chapter 7
Comprehensive Index Strategy



If SQL is the language of data retrieval, the *index* is the lookup key that makes retrieval fast. A database index is conceptually similar to the index in the back of a textbook: instead of reading every page to find a topic, you use the index to jump straight to the exact page number.

This chapter builds an index strategy the same way you should build indexes in production. It starts by surveying the common index types, as summarized in Table 7.1, so you can match structure to workload. Next, Fig. 7.1 grounds the discussion in the mechanics of a B-Tree, because design decisions make more sense when you can picture where the work happens. The middle of the chapter then walks through practical index patterns, each tied to a concrete query shape, and shows how to verify usage with `EXPLAIN PLAN` and `DBMS_XPLAN`. Finally, it covers the operational side, which indexes to remove, when to use function-based and covering indexes, and how to maintain and rebuild only when there is measurable benefit.

Types of Indexes

Oracle supports several index types, each optimized for different access patterns. Table 7.1 summarizes the most commonly used types.

Figure 7.1 illustrates the internal structure of a B-Tree index, showing how a root node fans out to branch and leaf nodes that store the actual row-IDs (ROWIDs).

The shape in Fig. 7.1 is what makes indexes fast: because the tree stays shallow even for millions of rows, Oracle reaches any key in just a few block reads instead of scanning the table.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_7

Table 7.1 Oracle index types and their best use cases

Index type	Purpose	Best used for
B-Tree (B*Tree)	Default balanced-tree structure	Primary keys, foreign keys, most `WHERE`-clause lookups
Bitmap	Stores a bitmap per key value	Low-cardinality columns: `gender`, `status`, `country`
Composite	Index on two or more columns	Queries filtering multiple columns, `GROUP BY`, `ORDER BY`
Function-Based	Indexes the result of a function	Queries where the indexed column is modified by a function

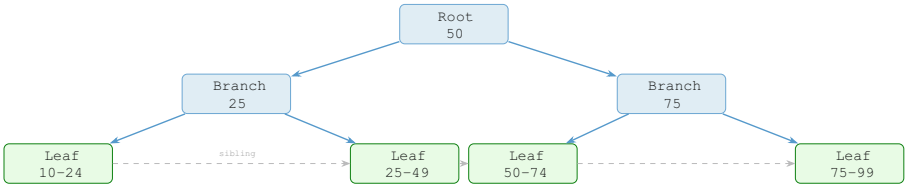


Fig. 7.1 B-Tree index structure: root → branch → leaf nodes with ROWID pointers. Leaf nodes are linked to support range scans efficiently

Strategic Index Creation

The examples in this section are meant to be copied and adapted, not memorized. Listing 7.1 starts with the simplest case, a single-column lookup index that supports an equality predicate.

Listing 7.1 Index supporting a `WHERE`-clause lookup

```
-- Query
SELECT * FROM orders WHERE customer_id = 101;

-- Supporting index
CREATE INDEX idx_orders_customer_id
ON orders(customer_id);
```

Joins deserve the same treatment as filters, because join keys are predicates too. Listing 7.2 shows the standard pattern, index the join column on both sides so Oracle can drive the join from either table when row counts change.

Listing 7.2 Indexes supporting a JOIN operation

```
SELECT e.employee_id, d.department_name
FROM   employees   e
JOIN   departments d ON e.department_id = d.department_id;

CREATE INDEX idx_employees_dept_id
ON employees(department_id);
```

```
CREATE INDEX idx_departments_dept_id
ON departments(department_id);
```

Sorting is the third common pattern. When a query filters and sorts, you can often eliminate the sort step entirely by using an index whose leading columns match the filters and whose trailing columns match the `ORDER BY`. Listing 7.3 shows the pattern.

Listing 7.3 Composite index supporting a filtered `ORDER BY`

```
-- Query
SELECT *
FROM   orders
WHERE  status = 'SHIPPED'
ORDER  BY order_date DESC;

-- Composite index (leading column = equality predicate)
CREATE INDEX idx_orders_status_date
ON orders(status, order_date DESC);
```

Tip
Always place equality-predicate columns *first* in a composite index. Range predicates and `ORDER BY` columns follow, maximizing the index's ability to satisfy the query without a separate sort step.

Index Usage Verification with EXPLAIN PLAN

Oracle's `EXPLAIN PLAN` reveals whether the optimizer uses an index or falls back to a full table scan. Listing 7.4 shows the basic workflow for generating the plan and printing it.

Listing 7.4 Generating and displaying an execution plan

```
EXPLAIN PLAN FOR
  SELECT * FROM orders WHERE customer_id = 101;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

Look for `INDEX RANGE SCAN` or `INDEX UNIQUE SCAN` in the plan output. If you see `TABLE ACCESS FULL`, the index is either absent, not selective enough, or suppressed by a function on the indexed column.

Figure 7.2 contrasts an index range scan with a full table scan to illustrate why the former is dramatically cheaper for selective queries.

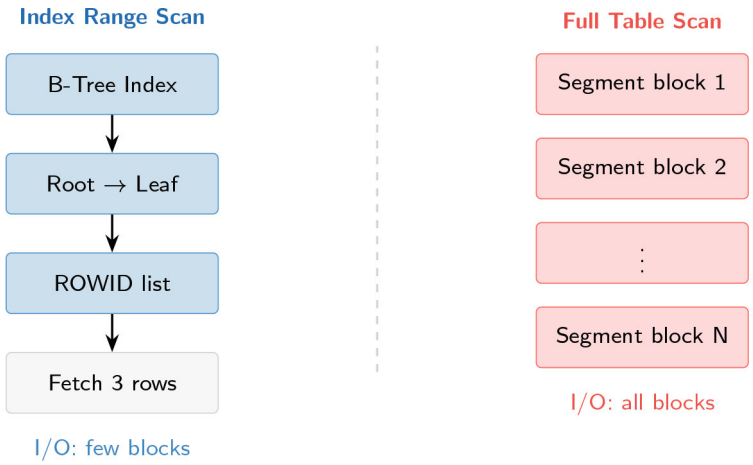


Fig. 7.2 Index range scan (left) vs. full table scan (right). The index navigates directly to matching rows; the full table scan reads every block regardless of selectivity

The contrast in Fig. 7.2 is the payoff of every technique in this chapter: the indexed path touches only the blocks that matter, while the full scan pays for every block in the table whether it qualifies or not.

Identifying and Removing Redundant Indexes

Every index incurs a write overhead on `INSERT`, `UPDATE`, and `DELETE` operations. Redundant indexes amplify this cost without adding read benefit. Before you drop anything, make the redundancy explicit in a concrete example. Listing 7.5 shows the classic case, a composite index that makes a single-column index unnecessary.

Listing 7.5 Example of redundant indexes

```
-- These two indexes are redundant:
CREATE INDEX idx_orders_customer_id
ON orders(customer_id);                          -- single-column

CREATE INDEX idx_orders_customer_date
ON orders(customer_id, order_date);              -- composite
-- The single-column index is redundant because the composite
-- index can serve queries on customer_id alone.
```

Suspecting an index is unused is not the same as knowing it. Before dropping anything, enable monitoring and let it observe a full business cycle. Listing 7.6 shows the minimal workflow.

Listing 7.6 Monitoring index usage to find unused indexes

```
-- Enable monitoring
ALTER INDEX idx_orders_customer_id MONITORING USAGE;

-- Check usage after an observation period
SELECT index_name, table_name, monitoring, used
FROM   v$object_usage
WHERE  table_name = 'ORDERS';
```

Note
`V$OBJECT_USAGE` resets each time monitoring is enabled. Allow at least one full business cycle (e.g., a month-end batch) before concluding that an index is truly unused.

When NOT to Use Indexes

Indexes are not always beneficial. Before you add one, sanity-check that you are not optimizing a case where a full scan is already the right answer, or where write overhead will dominate the read benefit:

- **Low-cardinality columns:** Columns with very few distinct values (e.g., gender, Y/N flags) yield poor selectivity. A bitmap index may be appropriate in read-heavy environments, but bitmap locking can cause contention under concurrent DML.
- **Small tables:** Tables with fewer than roughly 100 rows rarely benefit from an index; the optimizer may choose a full table scan because the entire table fits in a handful of blocks.
- **Frequently updated columns:** High DML frequency (inserts, updates, deletes) negates the read benefit because every change requires the index to be maintained.

If you are in one of these scenarios, step back and confirm the access path first, then add an index only when you can demonstrate a net reduction in blocks touched per execution.

Advanced Indexing Techniques

Function-Based Indexes

When a query applies a function to an indexed column, the standard index is bypassed. A function-based index resolves this by pre-computing and storing the function result.

Listing 7.7 shows a function-based index that matches the expression in the predicate, which allows Oracle to use an index access path even though the query applies `TO_CHAR`.

Listing 7.7 Function-based index for a formatted date predicate

```
CREATE INDEX idx_orders_date_fmt
ON orders(TO_CHAR(order_date, 'YYYY-MM-DD'));

-- Now this query can use the index:
SELECT * FROM orders
WHERE  TO_CHAR(order_date, 'YYYY-MM-DD') = '2025-01-15';
```

Covering Indexes

A *covering index* contains every column referenced by the query, allowing Oracle to satisfy the query entirely from the index without touching the table.

Listing 7.8 demonstrates a covering index, where all referenced columns live in the index so Oracle can avoid a separate table lookup.

Listing 7.8 Covering index to avoid table access

```
CREATE INDEX idx_orders_cover
ON orders(customer_id, order_date, status, order_amount);

-- All projected columns are in the index; no table access needed
SELECT order_date, status, order_amount
FROM   orders
WHERE  customer_id = 123;
```

Tip
Verify that the execution plan shows `INDEX FAST FULL SCAN` or `INDEX RANGE SCAN` with no subsequent `TABLE ACCESS BY ROWID` to confirm the covering-index optimization is active.

Invisible Indexes

An *invisible index* exists physically but is ignored by the optimizer for normal queries. This makes it a safe way to test whether a new index helps (or whether an existing index can be dropped) without changing application code.

Listing 7.9 Creating and testing an invisible index

```
-- Create the index, but keep it invisible to the optimizer
CREATE INDEX idx_orders_customer_id_test
ON orders(customer_id)
INVISIBLE;

-- Make it visible (goes live)
ALTER INDEX idx_orders_customer_id_test VISIBLE;

-- Or toggle back to invisible
ALTER INDEX idx_orders_customer_id_test INVISIBLE;
```

If you need to test it for a specific statement only, you can force usage with hints (or session settings) in a controlled test environment, then validate with the runtime plan and per-execution metrics.

Index Skip Scans

Oracle can sometimes use a composite index even when the leading column is not in the predicate. This is called an *index skip scan*. It works best when the leading column has low cardinality (few distinct values), allowing Oracle to "skip" through each leading-key value and probe the remaining key parts efficiently.

Listing 7.10 Composite index that may support an index skip scan

```
-- Composite index (leading column has few distinct values)
CREATE INDEX idx_orders_region_customer
ON orders(region_code, customer_id);

-- Predicate does not include REGION_CODE, but Oracle may still use
    the index
-- via INDEX SKIP SCAN when REGION_CODE has low cardinality
SELECT *
FROM   orders
WHERE  customer_id = 101;
```

Treat skip scans as an optimization opportunity, not a design goal: if this query is important and frequent, a dedicated index on `customer_id` (or a better composite key) is usually more predictable.

Reverse Key Indexes

A *reverse key index* reverses the bytes of the indexed value before storing it. This spreads monotonically increasing keys (such as sequence-driven IDs) across many leaf blocks instead of funneling inserts into the "rightmost" block. The main use case is reducing hot-block contention in high-insert workloads, especially on RAC.

Listing 7.11 Creating a reverse key index for a monotonically increasing key

```
-- Useful when ORDER_ID increases steadily and insert contention is
    observed
CREATE INDEX idx_orders_order_id_rev
ON orders(order_id)
REVERSE;
```

A reverse key index is usually a poor fit for range predicates on the reversed column (e.g., `order_id BETWEEN ...`) because the reversal destroys key order for range scans. Use it when the dominant access pattern is equality lookup and the problem is insert contention.

Index Maintenance

B-Tree indexes accumulate *deleted leaf rows* over time, leading to fragmentation and increased I/O. Before you rebuild, measure. Listing 7.12 shows a simple fragmentation query and an online rebuild command.

Listing 7.12 Checking fragmentation and rebuilding an index

```
SELECT index_name,
       del_lf_rows,
       lf_rows,
       ROUND((del_lf_rows / lf_rows) * 100, 2)
           AS fragmentation_pct
FROM   user_indexes
WHERE  table_name = 'ORDERS'
AND    lf_rows    > 0;

-- Rebuild online when fragmentation exceeds ~20%
ALTER INDEX idx_orders_customer_id REBUILD ONLINE;
```

Note
`REBUILD ONLINE` allows concurrent DML during the rebuild, making it safer for production use. An offline rebuild is faster but locks the table for the duration of the operation.

Summary

This chapter built an index strategy from the inside out: how a B-Tree is structured, how Oracle chooses an access path, and how to keep the index inventory healthy over time. As shown in Fig. 7.1, the reason indexes are fast is structural: the tree stays shallow while leaf blocks stay linked for efficient range scans. Figure 7.2 then shows the practical payoff, touch a few blocks via an index range scan instead of reading the entire table.

With that context in mind, the following takeaways capture the decisions you make repeatedly when you design and maintain indexes:

- Choose the index type to match the column's cardinality and the query pattern: B-Tree for most lookups, Bitmap for read-heavy low-cardinality workloads, and composite indexes for multicolumn predicates.
- Verify that Oracle is using the index you intended by checking the plan (and, when possible, the runtime plan) rather than relying on intuition.
- Treat indexes as part of your write path: redundant and unused indexes quietly tax every `INSERT`, `UPDATE`, and `DELETE`.
- Maintain what you create: rebuild only when it is justified by fragmentation and operational constraints.

Those principles are deliberately general; the details change from schema to schema, but the reasoning does not.

Quick Checklist

Use the checklist below as a repeatable build-and-verify loop. Read it in order: it starts with the SQL access pattern, moves through index design, and ends with proof and operational safety.

- Start from the SQL: list the `WHERE` predicates and `JOIN` keys, then build the index to match the *access pattern*, not the table definition.
- Put the most selective columns first in a composite index (and align the leading column with the most common predicate).
- If the query is *index-only*, add the missing columns to create a covering index and reduce table access.
- Confirm with the *runtime* plan (`DBMS_XPLAN`) and per-execution metrics (buffer gets/elapsed), not just `EXPLAIN PLAN`.
- Recheck write overhead: every extra index adds work to inserts/updates and can become a hidden throughput limiter.

If you follow that sequence, index work stays grounded in measurable outcomes: fewer blocks touched, fewer logical reads per execution, and fewer surprise regressions after deployment.

Next, Chap. 8 (*Locking and Concurrency Management*) shows how to diagnose and eliminate blocking sessions and deadlocks in production by connecting what users feel to the lock types, wait events, and transaction patterns causing contention.

Chapter 8
Advanced Locking and Concurrency Management



Query tuning is about making one statement run fast. *Concurrency management* is about making *many* statements run at the same time, with thousands of users reading and changing data simultaneously without corruption, deadlocks, or long wait chains.

When locking is designed well, the database stays responsive under load. When it isn't, performance collapses the moment concurrency ramps up.

This chapter moves from foundations to incident response. It starts with Oracle's multiversion read consistency, so you understand why reads usually do not block writes. Next, it introduces the two DML lock types you will see most often, as summarized in Table 8.1, and it shows how to identify a blocker, the wait chain, and the locked objects with targeted queries. It then uses Fig. 8.2 to explain how deadlocks form and how to prevent them with consistent lock ordering. The final sections cover practical control knobs, isolation levels, `NOWAIT` and `SKIP LOCKED`, and batching strategies that shorten lock holds in long-running maintenance work.

Oracle's Multiversion Read Consistency

Oracle uses *Multiversion Read Consistency* (MVCC) so readers and writers can coexist.

In plain terms, MVCC has two consequences that shape almost every concurrency problem you will diagnose:

- **Writers don't block readers.** A query reads a consistent snapshot of the data as of its start time, even if another session is in the middle of changing those rows.
- **Readers don't block writers.** A normal `SELECT` doesn't take row locks, so it doesn't get in the way of concurrent `INSERT`, `UPDATE`, or `DELETE`.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_8

Table 8.1 Oracle DML lock types

Lock Type	Description	Purpose	Problematic state
TX (Row Lock)	Acquired on `INSERT`, `UPDATE`, or `DELETE` of a single row	Only one transaction can modify a specific row at a time	*Row Lock Contention*: session waits for another to commit
TM (Table Lock)	Acquired whenever a Row Lock is acquired on any row	Prevents conflicting DDL (e.g., `DROP TABLE`)	Exclusive mode can block all subsequent DML

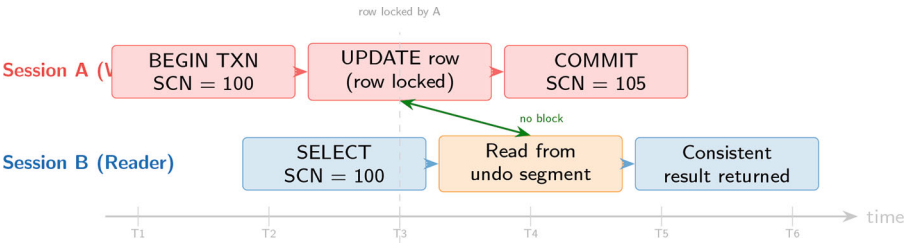


Fig. 8.1 Oracle MVCC: Session B's `SELECT` (initiated at SCN = 100) reads a consistent snapshot from undo while Session A holds a row lock. Neither session waits for the other

Those statements are the rule, and they explain why many "locking" complaints are really about writers competing with other writers.

Oracle pulls this off by using *undo*: if a row has changed since the query's SCN, Oracle reconstructs the older version on the fly. Figure 8.1 shows this mechanism in action, with a reader and a writer touching the same row at the same time.

The key point in Fig. 8.1 is that Session B never waits and never sees Session A's uncommitted change: it reads the pre-change version reconstructed from undo. Readers don't block writers, and writers don't block readers; only writers competing for the *same rows* ever queue.

Oracle's Lock Types

Most day-to-day DML locking boils down to two lock types (Table 8.1).

> **Note**
> Oracle does *not* escalate row locks into a big table lock. A single transaction can touch (and lock) huge numbers of rows without suddenly blocking all other DML on the table, one of the reasons Oracle scales well under OLTP load.

Comprehensive Lock Detection and Resolution

When one session is stuck waiting on another, Oracle tells you directly: the waiting session shows up in `V$SESSION` with `BLOCKING_SESSION` populated.

The next two queries show (1) who is blocked by whom and (2) which objects are locked. Listing 8.1 starts by extracting the wait chain from `V$SESSION`.

Listing 8.1 Finding blocking sessions and their blocked victims

```
SELECT s.sid                AS blocked_sid,
       s.serial#            AS blocked_serial,
       s.username           AS blocked_user,
       s.event              AS wait_event,
       s.seconds_in_wait,
       s.state,
       s.blocking_session   AS blocker_sid,
       bs.serial#           AS blocker_serial,
       bs.username          AS blocker_user,
       bs.program           AS blocker_program,
       bs.module            AS blocker_module
FROM   v$session  s
LEFT   JOIN v$session bs ON bs.sid = s.blocking_session
WHERE  s.blocking_session IS NOT NULL
ORDER  BY s.seconds_in_wait DESC;
```

Once you know who is blocking whom, the follow-up question is which objects are involved. Listing 8.2 names the locked objects and associates them with the holding session.

Listing 8.2 Identifying locked objects and who holds the lock

```
SELECT lo.session_id    AS sid,
       s.serial#,
       s.username,
       s.program,
       o.owner,
       o.object_name,
       o.object_type
FROM   v$locked_object lo
JOIN   dba_objects o ON o.object_id = lo.object_id
JOIN   v$session   s ON s.sid       = lo.session_id
ORDER  BY s.username, o.owner, o.object_name;
```

Prolonged Blocking: Check `V$SESSION_LONGOPS` Before You Kill

When a blocker has held locks for a long time, the most useful question is not just *who* it is, but *what it is doing*. `V$SESSION_LONGOPS` tracks long-running operations

(table scans, index builds, large sorts, RMAN, parallel operations) and can provide context before you decide to terminate a session.

Listing 8.3 Long-running operations for blocker sessions (progress and ETA)

```
SELECT l.sid,
       s.serial#,
       s.username,
       s.module,
       l.opname,
       l.target,
       l.sofar,
       l.totalwork,
       ROUND(l.sofar * 100 / NULLIF(l.totalwork, 0), 1) AS pct_done,
       l.elapsed_seconds,
       l.time_remaining
FROM   v$session_longops l
JOIN   v$session        s ON s.sid = l.sid
WHERE  l.totalwork > 0
  AND  l.sofar <> l.totalwork
  AND  s.sid IN (SELECT DISTINCT blocking_session
                 FROM v$session
                 WHERE blocking_session IS NOT NULL)
ORDER  BY l.elapsed_seconds DESC;
```

Tip
If you have the blocker's `SID` and `SERIAL#`, you can terminate it:

```
ALTER SYSTEM KILL SESSION 'sid,serial#' IMMEDIATE;
```

This is the *last resort*. Use it only when you're confident the session is stuck or abandoned (or is doing unnecessary work), and the cost of waiting is worse than the rollback that will follow.

Deadlock Detection and Prevention

A *deadlock* is a lock standoff: two (or more) sessions each hold something the other needs, so everyone waits forever.

Oracle detects the cycle, chooses a victim, and rolls back *only the statement* that hit the deadlock (not the whole transaction), raising `ORA-00060`. Figure 8.2 diagrams the classic two-session version of this trap.

Trace the arrows in Fig. 8.2 and you can see why no amount of waiting resolves the situation: each session holds exactly the resource the other needs.

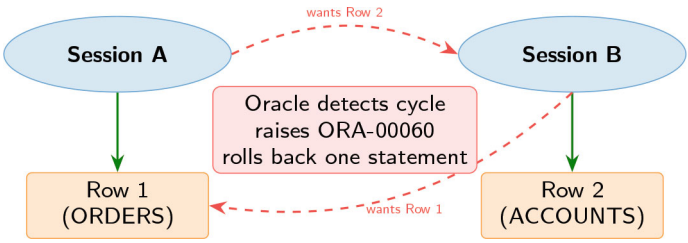


Fig. 8.2 Classic two-session deadlock: Session A holds Row 1 and waits for Row 2; Session B holds Row 2 and waits for Row 1. Oracle detects the cycle and automatically resolves it

The safest prevention is boring but effective: make every code path lock rows in the *same order*. Listing 8.4 shows a simple technique, force a deterministic ordering before `FOR UPDATE`.

Listing 8.4 Preventing deadlocks by acquiring locks in consistent order

```
-- Always acquire locks in ascending ID order
SELECT *
FROM   accounts
WHERE  id IN (:id1, :id2)
ORDER  BY id
FOR UPDATE;
```

Warning
Don't treat Oracle's deadlock handling as a feature you can lean on. Every `ORA-00060` means a statement rollback and a retry path in your application, which adds latency and hides a real design problem.

Oracle Isolation Level Optimization

Oracle gives you three isolation levels. The right choice depends on whether you care most about *freshness*, *repeatable results*, or *minimal overhead* for long reads.

Listing 8.5 shows the syntax for the three isolation levels, and the comments capture the intent for each.

Listing 8.5 Setting isolation levels for different workload types

```
-- Default: best for OLTP (statement-level read consistency)
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- For consistent reporting snapshots (transaction-level consistency)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- For long-running read-only reports (no undo consumption)
SET TRANSACTION READ ONLY;
```

Beyond isolation levels, Oracle gives you fine-grained control over how long a statement is willing to wait for a lock, which is invaluable in queue-style workloads. Listing 8.6 shows the two patterns, fail fast with `NOWAIT`, or keep throughput high with `SKIP LOCKED`.

Listing 8.6 Advanced lock control with `NOWAIT` and `SKIP LOCKED`

```
-- Fail immediately if the row is already locked (avoid indefinite
    wait)
SELECT *
FROM   accounts
WHERE  id = :id
FOR UPDATE NOWAIT;

-- Skip locked rows (ideal for parallel work-queue consumers)
SELECT *
FROM   job_queue
WHERE  processed = 'N'
FOR UPDATE SKIP LOCKED;
```

Tip
`SKIP LOCKED` (introduced in Oracle 11g Release 2) is the go-to pattern for multi-consumer work queues. Each worker simply grabs what it can; locked rows are skipped, so workers naturally stay out of each other's way.

Long Transaction Optimization via Batch Processing

Long transactions are where concurrency goes to die: they hold locks for a long time and keep generating undo.

The fix is almost always the same: do the work in smaller chunks and commit frequently. Listing 8.7 shows the basic batching pattern, update a limited number of rows, commit to release locks, then loop.

Listing 8.7 Batch processing pattern: commit every 1,000 rows

```
DECLARE
  v_rows INTEGER := 1;
BEGIN
  WHILE v_rows > 0 LOOP

    UPDATE orders o
    SET    o.status = 'PROCESSED'
    WHERE  o.status     <> 'PROCESSED'
    AND    o.order_date < DATE '2025-01-01'
    AND    ROWID IN (
             SELECT rid
             FROM  (
               SELECT ROWID rid
               FROM   orders
               WHERE  status     <> 'PROCESSED'
               AND    order_date < DATE '2025-01-01'
               AND    ROWNUM    <= 1000
             )
           );

    v_rows := SQL%ROWCOUNT;
    COMMIT;  % Release locks; free undo after each batch

  END LOOP;
END;
/
```

Note
The inner `ROWNUM <= 1000` keeps each update to a manageable bite-size batch. Tune that number based on your undo capacity and how long you can afford to hold locks. In production, it's common to see batch sizes anywhere from 500 to 10,000 rows.

Summary

This chapter turned Oracle locking from a mystery into a diagnosable system: what locks exist, why sessions block, and how to fix contention without making things worse. As shown in Fig. 8.1, Oracle's MVCC model is the reason reads usually do not block writes, but when sessions do block, it is because a specific transaction is holding a lock that another session needs. Figure 8.2 then shows the failure mode to avoid: two sessions each holding one resource while waiting on the other.

With that context, the following takeaways summarize the ideas you will reuse in every blocking incident:

- MVCC keeps readers and writers out of each other's way, but writers can still block writers when they touch the same rows.
- Use `V$SESSION` and `V$LOCKED_OBJECT` to identify the blocker, the wait chain, and the objects involved.
- Prevent deadlocks and chronic blocking by keeping transactions short and enforcing consistent lock ordering in application code.

If you can name the blocker, the lock type, and the SQL, you are no longer guessing, you are debugging.

Quick Checklist

Use the checklist below as a safe sequence. Each item deliberately narrows uncertainty before you take an action that is hard to undo (like killing a session).

- Identify the blocker and all waiters (who is waiting on whom?).
- Confirm the locked object and lock type (TX/TM) and the SQL causing it.
- Check transaction duration and undo pressure (long TXNs = long lock holds).
- Apply the least-risk fix first: commit/rollback; kill the session only if needed.
- Prevent recurrence: consistent lock ordering, short transactions, and batching long updates.

Follow that order and you will resolve incidents faster while reducing the chance of collateral damage.

Next, Chap. 9 (*Execution Plan Stabilization*) shows how to keep critical SQL from regressing when statistics change or the optimizer changes its mind, using SQL Plan Management (SPM) and plan baselines.

Chapter 9
Execution Plan Stabilization



A tuned SQL statement is only as good as the execution plan Oracle chooses *today*. When statistics change, index availability shifts, bind-value patterns evolve, or the optimizer's behavior changes after a database upgrade, a previously fast query can suddenly flip to a new plan and regress badly.

SQL Plan Management (SPM) addresses this by letting Oracle remember known-good plans, called *plan baselines*, and control when new plans are allowed to replace them. A baseline does not prevent the optimizer from finding a better plan; it simply insists that any candidate replacement plan must be verified and accepted before it can be used in production.

This chapter follows the lifecycle of a baseline in production. It starts by showing how to capture plans automatically or load a known-good plan manually, so you can protect a critical statement immediately. It then shows how to query baseline status and interpret the baseline flags, as summarized in Table 9.1, so you can see which plans are eligible for use. Finally, it ties SPM to operational workflows, monitoring for plan changes, evolving candidates safely, and using baselines as a guardrail during database upgrades.

SQL Plan Baselines

A plan baseline is a set of one or more accepted execution plans stored in the SQL Management Base (SMB) in the `SYSAUX` tablespace. Once a baseline exists for a statement, the optimizer will only use plans that appear in that baseline (unless no accepted plan exists, in which case it falls back to normal cost-based optimization).

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_9

Table 9.1 SPM plan baseline status flags

Flag	Description	Tuning relevance
ENABLED	If YES, the optimizer considers this baseline	Keep accepted baselines enabled; disabling prevents the optimizer from using the stored plan
ACCEPTED	If YES, the optimizer may use this baseline	Accept only known-good plans; new plans enter as unaccepted until verified
FIXED	If YES, only baseline plans are used; the optimizer will not consider any plan outside the baseline	Use sparingly, only for highly critical, fragile queries where even a verified improvement should be gated by a change-control process

Enabling Automatic Plan Capture

The simplest way to start building baselines is to enable automatic capture system-wide. Oracle then captures a new baseline whenever a SQL statement is executed for the second time, ensuring that repeating workload statements are protected.

Listing 9.1 shows the simplest on-ramp, enabling automatic capture so Oracle records repeating statements.

Listing 9.1 Enabling automatic plan capture system-wide

```
ALTER SYSTEM
  SET optimizer_capture_sql_plan_baselines = TRUE;
```

Warning
Automatic capture populates the SMB with every repeating SQL statement in the workload. On a busy OLTP system, this can grow the SYSAUX tablespace rapidly. Prefer targeted, manual loading (Listing 9.2) for critical queries; reserve automatic capture for initial baselining exercises or post-upgrade stabilization windows.

Loading a Known-Good Plan Manually

When you already know which SQL statement to protect, for example, after tuning it with SQL Tuning Advisor, you can load its current plan directly from the shared cursor cache using DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE.

Listing 9.2 shows the manual approach, load the current cursor plan into a baseline when you already know the plan you want to protect.

Listing 9.2 Loading a known-good plan from the cursor cache

```
DECLARE
  l_plans_loaded PLS_INTEGER;
BEGIN
  l_plans_loaded :=
    DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE(
      sql_id => '&sql_id'
    );
  DBMS_OUTPUT.PUT_LINE(
    'Plans loaded: ' || l_plans_loaded
  );
END;
/
```

Tip
Find the `sql_id` of any statement from `V$SQL` by searching on a fragment of the SQL text:
`SELECT sql_id, sql_text FROM v$sql WHERE sql_text LIKE '%your_table%';`
The returned `sql_id` is a stable 13-character hash that persists across executions.

Monitoring and Managing Plan Baselines

All plan baselines are accessible through the `DBA_SQL_PLAN_BASELINES` view (or `USER_SQL_PLAN_BASELINES` for the current schema owner).

Before you run the next queries, it helps to understand the two identifiers you will see everywhere:

- `SQL_HANDLE`: a unique identifier for the *SQL statement* in the baseline repository. Think of it as the "folder" that groups all baseline plans for one statement.
- `PLAN_NAME`: a unique identifier for a *specific plan variant* within that `SQL_HANDLE`. A single `SQL_HANDLE` can have multiple `PLAN_NAME` entries (for example, after plan evolution).

In practice you use `SQL_HANDLE` when you want to manage a statement as a whole (list its plans, drop the baseline), and `PLAN_NAME` when you want to enable/disable, accept, fix, or evolve one particular plan.

The three most important status flags are explained in Table 9.1.

Listing 9.3 shows the simplest way to inspect what SPM is tracking: query the baseline view and pull the flags you will use to control plan eligibility.

Listing 9.3 Querying the DBA_SQL_PLAN_BASELINES view

```
SELECT sql_handle,
       plan_name,
       enabled,
       accepted,
       fixed,
       sql_text
FROM   dba_sql_plan_baselines
WHERE  sql_text LIKE '%your_critical_sql%';
```

Note
A newly loaded plan starts as ACCEPTED = YES only when loaded from the cursor cache (LOAD_PLANS_FROM_CURSOR_CACHE). Plans captured automatically start as ACCEPTED = NO and must be evolved (verified) before the optimizer can use them.

Detecting Unauthorized Plan Changes

Even with SPM active, it is good practice to monitor V$SQL for unexpected plan hash value changes. A sudden appearance of a new plan_hash_value for the same sql_id, especially accompanied by a spike in average elapsed time, is a reliable signal that the optimizer has chosen a different plan.

Listing 9.4 shows a simple query that makes plan hash changes visible for a single sql_id.

Listing 9.4 Monitoring for unauthorized plan hash changes

```
SELECT sql_id,
       plan_hash_value,
       executions,
       elapsed_time / executions / 1000  AS avg_ms,
       first_load_time,
       last_active_time
FROM   v$sql
WHERE  sql_id = '&your_sql_id'
ORDER  BY plan_hash_value;
```

If multiple plan_hash_value rows appear for the same sql_id, compare the avg_ms values. A large increase in the newer plan's average elapsed time confirms a regression that warrants loading the old plan into a baseline and marking it fixed until the root cause is resolved.

Evolving Unaccepted Plans

When the optimizer finds a new plan that it believes is better than the accepted baseline, it stores the candidate as an unaccepted plan. You can verify and promote it using `DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE`, which test-executes the candidate and accepts it if it performs at least as well as the current baseline.

Listing 9.5 shows how to evolve an unaccepted candidate plan, so a new plan only becomes usable after it has been verified.

Listing 9.5 Evolving an unaccepted candidate plan into the baseline

```
DECLARE
  l_report CLOB;
BEGIN
  l_report :=
    DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE(
      sql_handle  => '&sql_handle',
      plan_name   => '&plan_name',  -- leave NULL to evolve all
      verify      => 'YES',         -- test-execute the candidate
      commit      => 'YES'          -- accept if performance improves
    );
  DBMS_OUTPUT.PUT_LINE(l_report);
END;
/
```

> **Tip**
> The `EVOLVE_SQL_PLAN_BASELINE` report shows the before-and-after performance comparison. Run with `commit => 'NO'` first to preview the outcome and only then rerun with `commit => 'YES'` in production.

Dropping Stale Baselines

Over time, the SMB accumulates baselines for retired queries. Stale baselines waste `SYSAUX` space and slow down plan lookup. Drop them with `DBMS_SPM.DROP_SQL_PLAN_BASELINE`.

Listing 9.6 shows a minimal drop workflow for removing stale baselines once a statement is retired.

Listing 9.6 Dropping a specific plan baseline

```
DECLARE
  l_plans_dropped PLS_INTEGER;
BEGIN
  l_plans_dropped :=
    DBMS_SPM.DROP_SQL_PLAN_BASELINE(
      sql_handle => '&sql_handle',
      plan_name  => '&plan_name'   -- NULL drops all plans for the
    handle
    );
  DBMS_OUTPUT.PUT_LINE(
    'Plans dropped: ' || l_plans_dropped
  );
END;
/
```

Note
Oracle also provides `DBMS_SPM.CONFIGURE` to set a retention period for unaccepted plans (default: 53 weeks) and a space budget for the SMB (default: 10% of `SYSAUX`). Review these settings after enabling automatic capture on a high-volume system.

SPM Across Database Upgrades

Database upgrades are a common trigger for plan regressions, because the optimizer's cost model and statistics can change between versions. A practical upgrade workflow using SPM is

1. **Before upgrade:** Enable automatic capture (Listing 9.1) for at least one full business cycle to build a baseline set for the critical workload.
2. **After upgrade:** Keep `optimizer_use_sql_plan_baselines = TRUE` (the default). The optimizer will use the pre-upgrade baselines while it evaluates new plans.
3. **Evolve:** Run `EVOLVE_SQL_PLAN_BASELINE` in batch to promote new plans that genuinely improve on the baselines; reject those that regress.
4. **Clean up:** Once the workload is stable, drop obsolete baselines and disable automatic capture to stop unbounded SMB growth.

Tip
Oracle's *SQL Performance Analyzer* (SPA), part of the Real Application Testing option, can automate the evolve-and-compare steps by replaying a captured workload against the upgraded database and producing a regression report. SPA and SPM complement each other: SPA identifies regressed statements, and SPM pins the known-good pre-upgrade plans for them.

Summary

This chapter showed how to keep performance gains from evaporating when the optimizer changes its mind. The goal of SQL Plan Management is not to freeze the database in time, it is to make plan change a controlled process, with evidence before adoption.

With that framing, the key takeaways below summarize the levers you will use most often:

- Plan baselines store known-good execution plans in the SQL Management Base and prevent the optimizer from silently switching to an unverified alternative.
- For critical queries, load baselines manually from the cursor cache (`LOAD_PLANS_FROM_CURSOR_CACHE`), or enable automatic capture during a baselining window.
- Use the `ENABLED`, `ACCEPTED`, and `FIXED` flags to control which plans the optimizer may use, and verify candidates before accepting them.
- Monitor `V$SQL` for new `plan_hash_value` entries to detect regressions early, then use `EVOLVE_SQL_PLAN_BASELINE` to promote genuinely better plans in a controlled way.
- During database upgrades, pre-build baselines before the upgrade, then evolve them post-upgrade to keep the workload stable with minimal manual intervention.

Taken together, these steps keep you out of emergency tuning. You can keep the known-good plan in place while you test changes, then accept improvements deliberately.

Quick Checklist
Use the checklist below as a repeatable sequence. It starts with selecting the right statements, then moves through capture, verification, and ongoing hygiene.

- Pick the SQL to protect (critical path, high cost, or frequent regressions).
- Capture or load a known-good plan into a baseline.
- Keep it `ENABLED` and `ACCEPTED`; use `FIXED` only when necessary.

- Watch for new plans (new `plan_hash_value`), evolve and accept only after testing.
- Periodically prune obsolete baselines to control SMB growth.

If you follow that order, plan stability becomes routine operational work rather than a surprise outage.

Next, Chap. 10 (*Query Frequency and Volume Optimization*) shows how to reduce how often SQL runs and how much data it processes, so you can lower load even when individual statements are already well tuned.

Chapter 10
Query Frequency and Volume Optimization



Even perfectly tuned SQL can overwhelm a system if executed thousands of times per second or if it returns unnecessarily large volumes of data. This chapter targets two orthogonal issues, how often SQL runs and how much work each run forces the database to do.

The first problem is **query frequency**, too many repeated SQL executions caused by chatty access patterns like N+1 loops. The second is **result volume**, too much data moved, sorted, materialized, and processed when a query returns more rows or columns than the application needs.

A quick terminology note: when the loop runs in a client application, each execution can be a true network round-trip. When the loop runs inside the database (for example, inside PL/SQL), the cost is usually *PL/SQL-to-SQL engine context switching* and repeated parsing/execution within the same session, not network latency.

The chapter follows that same two-part structure. It starts with loop elimination in PL/SQL, using concrete before-and-after examples. Next, it covers caching strategies and precomputation, including deterministic functions and materialized views. It then closes with pragmatic volume controls, pagination and ranking hot statements, so you can pick the highest leverage targets first.

Eliminate Query Loops in PL/SQL

The *N+1 query problem* is the most common cause of excessive repeated SQL execution. It appears when a loop executes one SQL statement per iteration instead of fetching the required rows in a single set operation. Listing 10.1 shows the anti-pattern, one query per loop iteration, which turns a set problem into thousands of repeated SQL executions.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_10

Listing 10.1 Less efficient: N+1 query loop pattern to avoid

```
DECLARE
  CURSOR user_cursor IS
    SELECT user_id FROM user_ids;
  user_data users%ROWTYPE;
BEGIN
  FOR rec IN user_cursor LOOP
    SELECT *
    INTO   user_data
    FROM   users
    WHERE  id = rec.user_id;
    -- 1 cursor query + up to N individual SQL executions
    -- (inside PL/SQL this is repeated SQL execution/context
     switching, not network I/O)
  END LOOP;
END;
/
```

For a `user_ids` set of 10,000 rows, this pattern runs one cursor query followed by up to 10,000 single-row `SELECT` executions. In a client application this often means 10,001 network round-trips; inside PL/SQL it is 10,001 SQL executions in one session, dominated by context switching and parse/ execution overhead.

Listing 10.2 shows the set-based alternative, fetch the required rows in one call, then process them in memory.

Listing 10.2 More efficient: `BULK COLLECT` reduces row-by-row SQL execution

```
DECLARE
  TYPE user_table_type IS
    TABLE OF users%ROWTYPE
    INDEX BY PLS_INTEGER;
  user_data user_table_type;
BEGIN
  SELECT *
  BULK COLLECT INTO user_data
  FROM   users
  WHERE  id IN (SELECT user_id FROM user_ids);

  FOR i IN 1 .. user_data.COUNT LOOP
    -- Process user_data(i)
    NULL;
  END LOOP;
END;
/
```

Note
`BULK COLLECT` without a `LIMIT` clause fetches the entire result set into the PGA in a single call. For very large results, use a loop with `FETCH ...BULK COLLECT INTO ...LIMIT :n` so you still avoid per-row round-trips without exhausting memory.

FORALL for Bulk DML

The same anti-pattern appears on the write side when code issues one `INSERT` or `UPDATE` per iteration. Replace it with `FORALL` to batch the DML into a single server call.

On the write side, Listing 10.3 shows the equivalent fix, replace per-row DML with `FORALL` so the database does the work in one server call.

Listing 10.3 `FORALL` for bulk DML: single round-trip for many rows

```
DECLARE
  TYPE id_list   IS TABLE OF orders.id%TYPE INDEX BY PLS_INTEGER;
  TYPE stat_list IS TABLE OF orders.status%TYPE INDEX BY PLS_INTEGER;
  l_ids    id_list;
  l_status stat_list;
BEGIN
  -- Populate collections (e.g., from a prior BULK COLLECT)
  -- ...

  FORALL i IN 1 .. l_ids.COUNT
    UPDATE orders
    SET    status = l_status(i)
    WHERE  id     = l_ids(i);

  COMMIT;
END;
/
```

Advanced Caching Strategies

When identical computations repeat across many queries, caching the result removes redundant work entirely, either inside the database engine or via precomputed structures.

DETERMINISTIC Functions

A DETERMINISTIC function tells Oracle that, for a given input, the function always returns the same output. Oracle may then cache the return value *within a single SQL execution* and reuse it for rows with identical arguments.

Listing 10.4 shows a simple DETERMINISTIC function, the intent is to let Oracle reuse results inside a single execution when many rows share the same inputs.

Listing 10.4 DETERMINISTIC function: allows Oracle to cache results within a query

```
CREATE OR REPLACE FUNCTION get_tax_rate(income NUMBER)
  RETURN NUMBER DETERMINISTIC
IS
BEGIN
  IF    income < 50000  THEN RETURN 0.10;
  ELSIF income < 100000 THEN RETURN 0.20;
  ELSE                       RETURN 0.30;
  END IF;
END;
/
```

Note
A DETERMINISTIC function must not rely on SYSDATE, query tables without receiving their state as arguments, perform DML, or depend on changing package variables. If it does, Oracle can return stale cached results silently.

Materialized Views for Expensive Aggregations

A materialized view (MV) stores the result of an expensive query (typically an aggregation or multi-table join) so later queries can read the prebuilt result instead of recomputing it.

Listing 10.5 shows a materialized view that precomputes an expensive aggregation, so later queries read a small summary instead of re-aggregating the base tables.

Listing 10.5 Materialized view pre-calculating per-user order summaries

```
CREATE MATERIALIZED VIEW user_order_summary
  BUILD IMMEDIATE
  REFRESH FAST ON COMMIT
AS
SELECT u.id                AS user_id,
       COUNT(o.id)         AS order_count,
       SUM(o.total_amount) AS total_spent,
```

```
       MAX(o.order_date)   AS last_order_date
FROM   users  u
JOIN   orders o ON u.id = o.user_id
GROUP  BY u.id;
```

Not every workload can afford `REFRESH FAST ON COMMIT`. When staleness of a few minutes or hours is acceptable, refresh the view on your own schedule instead.

When staleness is acceptable, Listing 10.6 shows the two refresh modes you will use most often, complete refresh and fast refresh.

Listing 10.6 Manual refresh options for a materialized view

```
-- Complete refresh: re-execute the full defining query
EXEC DBMS_MVIEW.REFRESH('user_order_summary', 'C');

-- Fast refresh: apply only changes since last refresh
EXEC DBMS_MVIEW.REFRESH('user_order_summary', 'F');
```

Tip
`REFRESH FAST ON COMMIT` requires a materialized view log on each base table (`CREATE MATERIALIZED VIEW LOG ON ...`). With `REFRESH FAST`, if a fast refresh cannot be performed (for example, the required log is missing), Oracle raises an error rather than silently falling back.

A fallback to complete refresh happens only when you use `REFRESH FORCE` (or a refresh method that allows Oracle to choose). Confirm what happened by checking `DBA_MVIEWS.LAST_REFRESH_TYPE` after refresh.

Result Set Limiting and Pagination

Returning more rows than the application can display or process wastes network bandwidth and memory. Pagination restricts each query to the window of rows the caller actually needs.

Efficient Pagination (Oracle 12c and Later)

Oracle 12c introduced the SQL standard `OFFSET ...FETCH` syntax.

Listing 10.7 shows the modern pagination syntax, which is both readable and optimizer-friendly when paired with a stable ordering.

Listing 10.7 `OFFSET/FETCH` pagination: Oracle 12c and later

```
SELECT *
FROM   users
ORDER  BY created_at
OFFSET :offset ROWS FETCH NEXT :limit ROWS ONLY;
```

Tip
Always include `ORDER BY` when paginating. Without deterministic ordering, repeated calls can return duplicates or skip rows as data changes.

Legacy Pagination (Pre-12c)

On Oracle 11g and earlier, implement pagination with a double-nested subquery and `ROWNUM`.

For older releases, Listing 10.8 shows the classic `ROWNUM` pattern, with the stopkey predicate positioned to limit work.

Listing 10.8 `ROWNUM`-based pagination for pre-12c databases

```
SELECT *
FROM  (
  SELECT u.*, ROWNUM rnum
  FROM  (
    SELECT * FROM users ORDER BY created_at
  ) u
  WHERE  ROWNUM <= :end_row
)
WHERE  rnum >= :start_row;
```

Note
Apply the `ROWNUM <= :end_row` predicate in the middle query so Oracle can stop after reaching the upper bound (`COUNT STOPKEY`). Putting the predicate in the innermost query can force Oracle to sort and scan far more data than necessary.

Identifying High-Frequency SQL

Before tuning frequency, find the statements executed most often. `V$SQLSTATS` provides aggregated execution counts without the heavier overhead of `V$SQL`. Listing 10.9 shows a simple ranking query that surfaces "chatty" statements quickly.

Listing 10.9 Finding the top ten most frequently executed statements

```
SELECT sql_id,
       executions,
       rows_processed,
       rows_processed / NULLIF(executions, 0) AS rows_per_exec,
       elapsed_time   / NULLIF(executions, 0) / 1000 AS avg_ms,
       SUBSTR(sql_text, 1, 80) AS sql_excerpt
FROM   v$sqlstats
ORDER  BY executions DESC
FETCH  FIRST 10 ROWS ONLY;
```

A statement with `rows_per_exec` near 1 and very high `executions` is a prime candidate for eliminating N+1 lookups (`BULK COLLECT`/set-based SQL) or introducing caching.

Summary

This chapter focused on reducing workload pressure by cutting round-trips and unnecessary data movement. The core idea is simple, the fastest query is the one you never run, and the cheapest row is the one you never fetch.

With that framing, the takeaways below summarize the patterns you can apply across code bases and workloads:

- Eliminate N+1 patterns with `BULK COLLECT` and `FORALL`.
- Control result volume with pagination (`OFFSET ...FETCH` or legacy `ROWNUM`).
- Find the best targets first by ranking statements in `V$SQLSTATS`.

If you combine frequency reduction with volume reduction, the gains multiply, fewer calls means fewer parses and less contention, and fewer rows means less I/O, sorting, and memory pressure per call.

Quick Checklist

Use the checklist below as a repeatable sequence. It is ordered to help you pick targets first, then reduce round-trips, then reduce rows, and finally confirm the improvement with the same workload.

- Rank by impact: highest executions, highest elapsed time, and highest rows processed.
- Remove chatty patterns (N+1, per-row lookups); prefer set-based SQL or bulk binds.
- Reduce data moved: select only needed columns/rows; add pagination where appropriate.
- Cache stable reference data (app cache or result cache) instead of re-querying.
- Remeasure after each change (same workload, compare deltas).

If the numbers do not move, treat it as a signal to change the hypothesis, not as a reason to keep iterating on the same statement.

Next, Chap. 11 (*Database Configuration Optimization*) shows how to tune at the instance level, memory sizing, key initialization parameters, and safe iterative change, after you have removed the obvious SQL and application inefficiencies.

Chapter 11
Database Configuration Optimization



Suboptimal instance configuration limits Oracle's ability to cache data, reuse compiled code, and execute complex queries efficiently. No amount of index tuning can compensate for a buffer cache that cannot hold the working set, a shared pool that constantly reloads cursors, or a PGA that forces sorts and hash joins to spill to temp.

This chapter focuses on the configuration decisions that most directly affect throughput and stability: SGA and PGA sizing, advisory views that guide changes, and connection pool limits that prevent session sprawl.

The chapter is organized as a practical tuning loop. It starts with memory, how to size the SGA and PGA so the database can cache the working set and keep joins and sorts in memory. Next, it shows how to use advisory views to test sizing changes safely before you touch parameters. It then covers the shared pool and session limits, because a perfectly sized cache still performs poorly if the instance is forced to hard parse constantly or to support more sessions than the host can handle. Finally, it closes with redo and I/O checks, so you can catch configuration limits that only show up under write-heavy load.

Memory Configuration Tuning

Oracle memory divides into two main regions. The SGA is shared across all sessions and the PGA is private to each server process:

- **System Global Area (SGA)**: Shared across all sessions
- **Program Global Area (PGA)**: Private to each server process

Getting the balance right is the foundation of instance tuning. If the SGA cannot hold the working set or the PGA is too small for work areas, the symptoms will look like SQL problems, but the root cause is configuration.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_11

Table 11.1 Key SGA components and the parameters that size them

Component	Parameter	Role
Buffer cache	`db_cache_size`	Caches data blocks and reduces physical I/O
Shared pool	`shared_pool_size`	Stores parsed SQL, PL/SQL, and dictionary cache entries
Large pool	`large_pool_size`	Used for PQ, RMAN, and shared server UGA allocations
Streams pool	`streams_pool_size`	Used by Streams, GoldenGate capture, and LogMiner

System Global Area (SGA)

The SGA contains multiple pools. The two most performance-sensitive for most workloads are the buffer cache and the shared pool. Table 11.1 lists the main components and the parameters that size them.

Of the pools in Table 11.1, start with the buffer cache and the shared pool; the others matter only for specific features. Listing 11.1 shows a basic review query and a few common sizing changes.

Listing 11.1 Reviewing and adjusting SGA configuration

```
-- Review current SGA component sizes
SELECT name, value
FROM   v$sga;

-- Dynamic SGA components (useful when ASMM/AMM is enabled)
SELECT component, current_size, min_size, max_size
FROM   v$sga_dynamic_components
ORDER  BY component;

-- Example: set buffer cache size explicitly
ALTER SYSTEM
  SET db_cache_size = 2G SCOPE = BOTH;

-- Example: set shared pool size explicitly
ALTER SYSTEM
  SET shared_pool_size = 1G SCOPE = BOTH;

-- Example: Automatic Memory Management (AMM, available since Oracle
    Database 11g)
ALTER SYSTEM
  SET memory_target = 4G SCOPE = SPFILE;
ALTER SYSTEM
  SET memory_max_target = 6G SCOPE = SPFILE;
```

Note
`SCOPE = BOTH` applies the change to the running instance and persists it in the SPFILE. Use `SCOPE = SPFILE` for parameters that require a restart so you do not assume a change is active when it is not.

Warning
On Linux systems configured with HugePages, AMM (`memory_target`) is usually not a good fit. In those environments, disable AMM and use ASMM (`sga_target`) plus explicit PGA sizing.

Program Global Area (PGA)

When PGA work areas are too small, Oracle spills sorts and hash joins to temp, which is typically far slower than staying in memory.

Listing 11.2 shows a minimal PGA configuration and a quick check against `V$PGASTAT` to confirm whether work areas are spilling.

Listing 11.2 Configuring and monitoring PGA usage

```
ALTER SYSTEM
  SET pga_aggregate_target = 1G SCOPE = BOTH;

SELECT name, value
FROM   v$pgastat
WHERE  name IN (
  'aggregate PGA target parameter',
  'total PGA inuse',
  'total PGA allocated'
);
```

Tip
If you are on Oracle 12c or later, set `pga_aggregate_limit` to a reasonable ceiling so runaway queries do not exhaust OS memory.

Buffer Cache Sizing with Advisory Views

The buffer cache advisory estimates the effect of different cache sizes without actually resizing the cache. Listing 11.3 shows a simple query you can run, then use the read factor to decide whether a larger cache would meaningfully reduce physical I/O.

Listing 11.3 Using `V$DB_CACHE_ADVICE` to guide cache sizing

```
SELECT size_for_estimate,
       buffers_for_estimate,
       estd_physical_read_factor,
       estd_physical_reads
FROM   v$db_cache_advice
WHERE  name = 'DEFAULT'
ORDER  BY size_for_estimate;
```

> **Note**
> If `V$DB_CACHE_ADVICE` shows only the current size, confirm `db_cache_advice` is enabled and `statistics_level` is `TYPICAL` or `ALL`.

Shared Pool Sizing

The shared pool includes the library cache. Chronic hard parses and frequent reloads indicate that the shared pool is undersized or that the application is not using bind variables.

Listing 11.4 shows a focused library cache query. It helps you decide whether the shared pool is undersized, or whether the application is forcing hard parses by not using bind variables.

Listing 11.4 Diagnosing shared pool pressure

```
SELECT namespace,
       gets,
       gethits,
       ROUND(gethitratio * 100, 2) AS hit_pct,
       reloads,
       invalidations
FROM   v$librarycache
WHERE  namespace IN ('SQL AREA', 'TABLE/PROCEDURE', 'BODY')
ORDER  BY namespace;
```

Tip
Treat `reloads/gets` above 1% in the `SQL AREA` namespace as an action signal. Resize in small steps and remeasure.

Connection Pool Optimization

Every dedicated session consumes memory and contributes to concurrency pressure inside the database. Connection pooling is required for most applications, but an oversized pool simply moves the bottleneck.

Listing 11.5 shows a simple HikariCP configuration. Use it as a template, then tune the maximum pool size to match the database session budget and the application's real concurrency.

Listing 11.5 HikariCP connection pool configuration for Oracle

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:oracle:thin:@//host:port/service");
config.setUsername("your_user");
config.setPassword("your_password");

config.setMaximumPoolSize(20);
config.setMinimumIdle(5);

config.setIdleTimeout(30_000);
config.setConnectionTimeout(30_000);
config.setLeakDetectionThreshold(60_000);

HikariDataSource ds = new HikariDataSource(config);
```

Tip
Set session level NLS state in the pool (e.g., via an init SQL hook) so all pooled connections behave consistently and the application does not need per request `ALTER SESSION` calls.

Connection pools only work if the database is configured to accept the sessions they hold. Make sure the instance limits leave headroom above the sum of all pool maximums. Listing 11.6 shows the two parameters you most often have to align with the pool.

Listing 11.6 Configuring database session and process limits

```
ALTER SYSTEM SET sessions  = 500 SCOPE = SPFILE;
ALTER SYSTEM SET processes = 300 SCOPE = SPFILE;
```

Redo Log and I/O Configuration

Write heavy workloads are often limited by redo log I/O or frequent log switches.

Listing 11.7 shows two quick checks, current redo log sizing and recent switch frequency, so you can spot write pressure that looks like application latency.

Listing 11.7 Checking redo log size and switch frequency

```
SELECT group#, members, bytes / 1024 / 1024 AS size_mb, status
FROM   v$log
ORDER  BY group#;

SELECT TO_CHAR(first_time, 'YYYY-MM-DD HH24') AS hour,
       COUNT(*) AS switches
FROM   v$log_history
WHERE  first_time >= SYSDATE - 1
GROUP  BY TO_CHAR(first_time, 'YYYY-MM-DD HH24')
ORDER  BY hour;
```

Warning
Do not drop redo log groups with `STATUS = CURRENT` or `ACTIVE`. Wait until the group is `INACTIVE`.

Summary

This chapter covered the instance level settings that underpin every SQL tuning technique. The theme is consistency, when the instance is sized and configured well, tuning improvements persist under load rather than disappearing as concurrency rises.

With that framing, the key takeaways below summarize the decisions that matter most often:

- Size SGA and PGA with measurement and advisory views, not guesses.
- Use `V$DB_CACHE_ADVICE` and `V$LIBRARYCACHE` to decide what to change.
- Keep connection pools and database session limits aligned.

Taken together, these practices keep you from chasing symptoms. They help you distinguish a SQL problem from a memory, parsing, or session pressure problem.

Quick Checklist

Use the checklist below as a safe change process. Each item is ordered to force a baseline first, a small change next, and verification last.

- Establish a baseline before changing parameters. Use AWR only when the Oracle Diagnostic Pack is appropriately licensed; otherwise use Statspack (and OS metrics) as the no-additional-license alternative.
- Check memory advisories (buffer cache, shared pool, PGA), and change one knob at a time.
- Validate session and process limits against your connection pool peak usage.
- Verify redo and undo health, log switch rate, checkpoint behavior, and contention.
- Remeasure after the change, roll back if DB time or waits worsen.

If the change does not move the metrics you expected, treat it as a signal to revisit the hypothesis rather than to keep turning the same knob.

Next, Chap. 12 (*Application Layer Optimization*) shows how application patterns, bind variables, ORM behavior, pagination, and caching, can dominate performance even when the database configuration is solid.

Chapter 12
Application Layer Optimization



This chapter connects the database and the application.

Earlier chapters focused on what Oracle does internally, such as indexes, execution plans, locking, and memory. Here, we focus on what the application can do to help Oracle.

The core idea is straightforward: *do not fetch data and then filter it in memory*. Push filters, sorting, and row limits into SQL so Oracle can use indexes and stop early.

This chapter covers application patterns that reduce CPU, round-trips, and result size. It starts with bind variables and parameterized queries, because plan reuse and safety depend on keeping SQL text stable. Next, it looks at ORM behavior, fetch strategies, and projections, so the application stops generating N+1 queries and over-fetching columns. It then focuses on pushing filtering, ordering, and row limiting into SQL, followed by server-side pagination, including keyset pagination for deep pages. Finally, it covers caching and read scaling, so repeated reads do not become repeated database work.

As you read, treat the examples as a checklist for code review. Each listing shows a common anti-pattern and a safer alternative, and each figure is meant to be read as a data flow, where the work happens and where the bytes move.

Parameterized Queries for Security and Performance

Dynamic SQL that concatenates user input is simultaneously a security risk and a performance hazard. Each distinct SQL string becomes a distinct cursor. Oracle must hard parse it, allocate a child cursor in the library cache, and displace an existing reusable plan.

Listing 12.1 shows the anti-pattern, concatenating user input into SQL creates both an injection risk and a hard parse storm.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_12

Listing 12.1 Vulnerable dynamic SQL: injection risk and hard parse storm

```
// AVOID: injection risk plus a hard parse for each distinct value
String query = "SELECT * FROM users "
             + "WHERE username = '" + userInput + "'";
```

Warning
This pattern is dangerous for two reasons. First, it is vulnerable to SQL injection because user input becomes part of the SQL text. Second, it prevents plan reuse: every different input value produces a different SQL string, so Oracle must hard-parse and store a separate cursor each time. Under load, these near-duplicate cursors fill the shared pool and evict reusable statements.

The prepared statement form solves both problems at once; the placeholder keeps the SQL text constant, so Oracle parses it once and reuses the plan.

Listing 12.2 shows the correct pattern, keep the SQL text constant and pass values as parameters so Oracle can reuse the cursor.

Listing 12.2 Prepared statement: prevents injection and enables plan reuse

```
PreparedStatement stmt = conn.prepareStatement(
    "SELECT * FROM users WHERE username = ?"
);
stmt.setString(1, userInput);
ResultSet rs = stmt.executeQuery();
```

In application SQL frameworks, named parameters do the same job with better readability.

In frameworks that support named parameters, Listing 12.3 shows the same idea with better readability, values are data, not part of the SQL text.

Listing 12.3 Named bind variables: recommended form for application SQL

```
SELECT *
FROM   orders
WHERE  customer_id = :customer_id
AND    order_date  >= :start_date;
```

Tip
Treat `CURSOR_SHARING = FORCE` as a temporary compatibility setting only. Fix the application to bind explicitly as soon as possible.

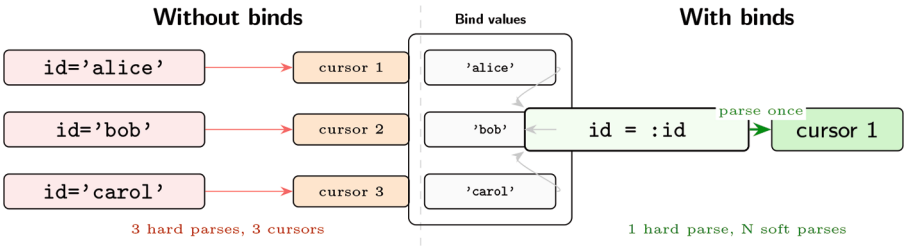


Fig. 12.1 Each distinct literal produces a separate cursor and hard parse (left). A bind variable parses once; all callers reuse the same plan (right)

Figure 12.1 visualizes the difference between literal SQL and bound SQL from the shared pool's point of view.

As Fig. 12.1 shows, the bound form funnels every execution through a single cursor, while the literal form manufactures a new cursor per value; the shared pool pays for that difference on every request.

ORM Query Optimization

Object relational mapping frameworks reduce boilerplate, but their defaults are optimized for correctness rather than throughput. Two problems dominate: N+1 queries caused by lazy loading and over fetching caused by loading full entities when only a few columns are needed.

Avoiding the N+1 Problem

Listing 12.4 shows the common trigger for N+1 behavior, lazy loading of a collection that is later iterated.

Listing 12.4 Less efficient: lazy loading causes N+1 queries in a loop

```
@Entity
public class User {

    @OneToMany(fetch = FetchType.LAZY)
    private List<Order> orders;

}
```

Fetching the association eagerly in the query itself collapses the N+1 round-trips into a single join.

To collapse those round-trips, Listing 12.5 shows the JOIN FETCH pattern, load the parent and its related rows in one query.

Listing 12.5 More efficient: JOIN FETCH loads related rows in one query

```
@Query("SELECT u FROM User u "
     + "JOIN FETCH u.orders "
     + "WHERE u.id = :id")
User getUserWithOrders(@Param("id") Long id);
```

Note
Be careful when combining collection fetch joins with pagination. Many ORM implementations cannot page correctly when the join creates duplicate parent rows, so they fetch a large result set and page in memory.

Use Projections to Reduce Payload

When you only need a subset of columns, Listing 12.6 shows a projection query that returns only the required fields.

Listing 12.6 Projection query: return only required fields

```
@Query("SELECT u.id, u.name, u.email "
     + "FROM User u WHERE u.active = true")
List<UserSummary> findActiveUserSummaries();
```

Spring Data can generate the mapping for you when the projection is declared as an interface.

Listing 12.7 shows the interface-based version, where the framework generates the mapping automatically.

Listing 12.7 Interface based projection: Spring Data generates the mapping

```
public interface UserSummary {
    Long   getId();
    String getName();
    String getEmail();
}
```

Push Filters Down, Not Up

A common and expensive bug in back-end code is *unbounded fetching*. The application loads many rows and then filters, sorts, or limits in memory. The result can look correct in development, but it becomes slow and memory hungry in production.

Use one rule: if you can express it in SQL, do it in SQL. That includes `WHERE` predicates, `ORDER BY`, and row limiting. Before looking at code, it helps to name the two shapes you will see in practice:

- **Wrong**: Fetch everything then `filter()` and `limit()` in code.
- **Right**: Pass API parameters into SQL as bind variables.

Once you know which shape you are looking at, the fix is usually straightforward. The goal is to move the limiting step as close to the data as possible.

The wrong version looks harmless in code review, which is exactly what makes it dangerous. Listing 12.8 shows the anti-pattern, a full fetch followed by in-memory filtering and limiting.

Listing 12.8 Anti pattern: filter and limit applied after a full table fetch

```
List<Order> all = orderRepository.findAll();

List<Order> pending = all.stream()
    .filter(o -> o.getCustomerId() == customerId)
    .filter(o -> o.getStatus().equals("PENDING"))
    .limit(limit)
    .toList();
```

Push the same parameters into the query, and the database returns only the rows the caller asked for. Listing 12.9 shows the same logic expressed at the data layer.

Listing 12.9 Correct pattern: parameters shape the query at the data layer

```
@Query("""
    SELECT o FROM Order o
    WHERE  o.customerId = :customerId
      AND  o.status     = :status
    ORDER  BY o.createdAt DESC
""")
Page<Order> findByCustomerAndStatus(
    @Param("customerId") long customerId,
    @Param("status") String status,
    Pageable pageable
);
```

The raw SQL equivalent uses `OFFSET` and `FETCH` directly. Listing 12.10 shows the same filters and limits expressed in SQL with bind variables.

Listing 12.10 Bounded SQL query: filtering and row limiting in the database

```
SELECT *
FROM   orders
WHERE  customer_id = :customer_id
AND    status      = :status
ORDER  BY created_at DESC
FETCH  FIRST :limit ROWS ONLY;
```

Tip
In code review, flag application side slicing that immediately follows a data call. It is often a sign that limiting is happening in the wrong layer.

Note
At the API layer, enforce bounded defaults and a server side maximum page size. An endpoint that accepts an unbounded `limit` parameter is a performance bug with a friendly URL.

Figure 12.2 contrasts the two data flows: filtering in the application vs. pushing the filter down into SQL.

The left path in Fig. 12.2 moves the entire table across the network just to throw most of it away; the right path lets Oracle do the filtering where the data lives, which is the whole point of a database.

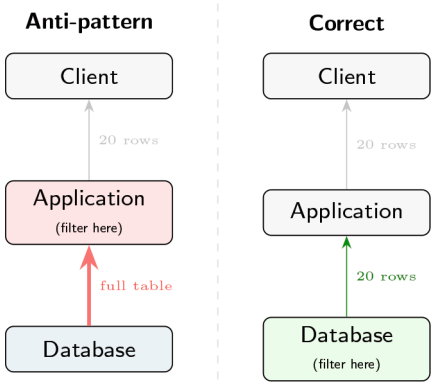


Fig. 12.2 Filtering in the application layer (left) transmits the full table before discarding rows. Filtering in SQL (right) transfers only qualifying rows

Application-Level Pagination

Apply pagination in SQL so Oracle can stop as soon as it has enough rows. Offset-based pagination is easy to implement, but it gets slower as the offset grows. Listing 12.11 shows the basic pattern.

Listing 12.11 Offset-based pagination using OFFSET and FETCH

```
public Page<User> getUsers(int page, int pageSize) {
    int offset = (page - 1) * pageSize;

    PreparedStatement stmt = conn.prepareStatement(
        "SELECT * FROM users ORDER BY created_at "
      + "OFFSET ? ROWS FETCH NEXT ? ROWS ONLY"
    );
    stmt.setInt(1, offset);
    stmt.setInt(2, pageSize);

    return executeAndBuildPage(stmt, page, pageSize);
}
```

> **Warning**
> With offsets, Oracle still has to scan and discard the earlier rows. A large page number can turn a fast query into a slow one.

Keyset pagination sidesteps that cost entirely by remembering where the last page ended and seeking straight to it. Listing 12.12 shows the idea, the first query is the first page, the second continues from the last seen key.

Listing 12.12 Keyset pagination: scales to deep paging by seeking on the last key

```
SELECT *
FROM   users
ORDER  BY created_at, id
FETCH  NEXT :page_size ROWS ONLY;

SELECT *
FROM   users
WHERE  created_at > :last_created_at
   OR (created_at = :last_created_at AND id > :last_id)
ORDER  BY created_at, id
FETCH  NEXT :page_size ROWS ONLY;
```

Caching and Read Scaling

Caching avoids repeated database reads. Use it for data that is read often and changes rarely, where brief staleness is acceptable.

Cache-Aside Pattern

Listing 12.13 shows the cache-aside pattern, check the cache first, hit returns immediately, miss loads from the database and repopulates the cache.

Listing 12.13 Cache-aside pattern using Redis with a TTL

```
public User getUserById(Long userId) {
    String cacheKey = "user:" + userId;

    String cached = redisTemplate.opsForValue().get(cacheKey);
    if (cached != null) {
        return JSON.parse(cached, User.class);
    }

    User user = userRepository.findById(userId).orElseThrow();

    redisTemplate.opsForValue().set(
        cacheKey,
        JSON.stringify(user),
        Duration.ofMinutes(30)
    );

    return user;
}
```

Figure 12.3 traces both paths through this code: the cheap cache hit and the miss that repopulates the cache.

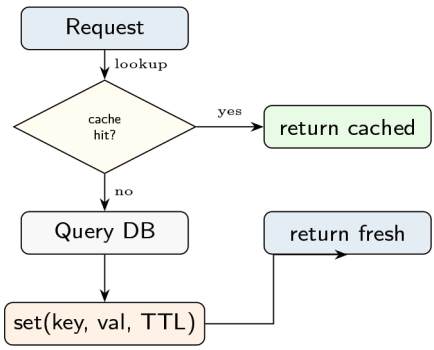


Fig. 12.3 Cache-aside flow: a hit returns immediately (right); a miss queries the database, populates the cache with a TTL, then returns the fresh value

Note in Fig. 12.3 that the database is touched only on a miss; with a sensible TTL, the vast majority of reads never reach Oracle at all.

> **Note**
> On updates, invalidate or refresh the cache after the database commit, never before.

Read and Write Splitting

Listing 12.14 shows one way to route reads and writes, which can scale read-heavy workloads when replica consistency rules are clear.

Listing 12.14 Read and write routing with AbstractRoutingDataSource

```
public class RoutingDataSource
        extends AbstractRoutingDataSource {

    @Override
    protected Object determineCurrentLookupKey() {
        return TransactionSynchronizationManager
                    .isCurrentTransactionReadOnly()
                ? "REPLICA"
                : "PRIMARY";
    }
}
```

> **Note**
> Replica lag can break read your writes expectations. Consider routing a user to the primary for a short window after a write.

Summary

This chapter moved the tuning lens up a layer, to the application code that generates the SQL in the first place. The theme is leverage, small changes in how the application issues SQL can remove entire classes of waste, repeated parsing, repeated round-trips, and repeated full result transfers.

With that framing, the main levers below summarize the patterns you will reuse most often:

- Use bind variables to prevent injection and enable plan reuse.
- Eliminate ORM-generated N+1 queries and reduce payload with projections.
- Push filters, ordering, and row limits into SQL rather than filtering in memory.
- Paginate on the server, and use keyset pagination for deep pages.
- Cache stable data and use replicas for read-heavy traffic when consistency permits.

Taken together, these practices keep the database doing database work, filtering, sorting, joining, and stopping early, instead of turning Oracle into a row store that the application then re-processes.

Quick Checklist

Use the checklist below as a code and architecture review sequence. It is ordered to make volume visible first, then fix correctness and reuse, then introduce caching only after the data contract is clear.

- Turn on SQL tracing or profiling in the app tier to see real query volume and latency.
- Ensure binds are used for hot statements, no literal concatenation.
- Audit ORM output, look for N+1, missing predicates, and over-fetching columns.
- Add server-side limits (pagination) and prefer keyset pagination for large offsets.
- Add caching only after correctness rules are clear (TTL, invalidation, consistency).

If you cannot explain the correctness story, when values change and when stale reads are acceptable, do not scale reads with caching or replicas yet.

Next, Chap. 13 (*End to End Tuning Case Studies*) ties these techniques together with real-world scenarios, showing how to move from symptoms to evidence then to fixes that hold up under production load.

Chapter 13
Batch Processing and Parallel Execution



Application tuning reduces round-trips and improves plan quality, but many systems still spend a large fraction of their time in *batch* work: nightly loads, report refreshes, backfills, and reprocessing jobs. Batch workloads are where seemingly small choices (array size, commit frequency, logging mode, parallelism) can change runtimes by orders of magnitude.

This chapter focuses on three outcomes that matter for batch workloads. First, it pushes you toward set-based data movement, so the database does work in large, optimized operations instead of row-by-row loops. Next, it shows how to use parallel execution deliberately and safely, so you shorten wall-clock time without starving OLTP sessions. Finally, it treats predictability as a feature, restartability, measurement, and isolation from peak traffic are what make batch work operationally boring.

Prefer Set-Based SQL Over Row-by-Row Work

Row-by-row processing multiplies parse/execute overhead, increases context switching between PL/SQL and SQL, and usually generates excessive logical I/O. Start every batch tuning effort by asking: can this be expressed as a single `INSERT ...SELECT`, `MERGE`, or `UPDATE`?

Replace Looping Updates with a Single Statement

The most common batch anti-pattern is a cursor loop that issues one DML statement per row. Listing 13.1 shows the pattern, it works, but it pays PL/SQL context switch and execute overhead for every row.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_13

Listing 13.1 Avoid: row-by-row update in a loop

```
DECLARE
  CURSOR c IS
    SELECT id, new_status
    FROM   stage_orders;
BEGIN
  FOR r IN c LOOP
    UPDATE orders
    SET    status = r.new_status
    WHERE  id = r.id;
  END LOOP;
  COMMIT;
END;
/
```

One statement replaces the entire loop, and Oracle optimizes it as a single set operation. Listing 13.2 shows a set-based rewrite.

Listing 13.2 Preferred: set-based update

```
UPDATE orders o
SET    o.status = (
         SELECT s.new_status
         FROM   stage_orders s
         WHERE  s.id = o.id
       )
WHERE  EXISTS (
         SELECT 1
         FROM   stage_orders s
         WHERE  s.id = o.id
       );

COMMIT;
```

If the set-based rewrite changes concurrency characteristics (e.g., it locks more rows at once), schedule the job in smaller chunks (see the section "Chunked Parallelism with `DBMS_PARALLEL_EXECUTE`" later in this chapter).

PL/SQL Bulk Processing: `BULK COLLECT` and `FORALL`

When the work truly must be procedural (complex validations, calls to external APIs, or conditional logic), use bulk binds to reduce SQL/PLSQL context switching.

Bulk Collect in Batches

BULK COLLECT fetches many rows per round-trip instead of one; the LIMIT clause keeps memory bounded no matter how large the table grows. Listing 13.3 shows the basic pattern.

Listing 13.3 Bulk-collect in batches to control memory and reduce round-trips

```
DECLARE
  TYPE t_id_tab IS TABLE OF orders.id%TYPE;
  l_ids t_id_tab;
  l_limit PLS_INTEGER := 1000;

  CURSOR c IS
    SELECT id
    FROM   orders
    WHERE  status = 'PENDING'
    ORDER  BY id;
BEGIN
  OPEN c;
  LOOP
    FETCH c BULK COLLECT INTO l_ids LIMIT l_limit;
    EXIT WHEN l_ids.COUNT = 0;

    -- Process l_ids here (typically with FORALL)
    NULL;
  END LOOP;
  CLOSE c;
END;
/
```

Use FORALL for DML

On the write side, FORALL is the mirror image of BULK COLLECT: it sends an entire array of values to the SQL engine in one context switch. Listing 13.4 shows the idea.

Listing 13.4 FORALL turns N DML statements into a single-array operation

```
DECLARE
  TYPE t_id_tab IS TABLE OF orders.id%TYPE;
  l_ids t_id_tab := t_id_tab(101, 102, 103);
BEGIN
  FORALL i IN 1 .. l_ids.COUNT
    UPDATE orders
    SET    status = 'DONE'
```

```
    WHERE  id = l_ids(i);

  COMMIT;
END;
/
```

Bulk processing reduces CPU and latch pressure in the shared pool by lowering execute calls. It also makes jobs easier to tune because the SQL becomes visible as a smaller number of high-impact statements.

Commit Strategy and Undo/Redo Pressure

The right commit frequency is a balance. Before you change it, be explicit about the trade-off you are making:

- Large transactions reduce commit overhead and can improve throughput.
- Smaller transactions reduce undo retention requirements, shrink the blast radius of failures, and shorten lock durations.

Those trade-offs show up as user-visible symptoms, long locks, long rollback, and undo pressure, so commit strategy is not just a batch performance knob.

As a practical baseline, start with the workload type and adjust from there:

- For OLTP-style corrections, commit in small, predictable batches.
- For pure loads into new segments (staging tables), commit less often, but ensure the job can restart.

Measure the impact with wait events (`log file sync`, `log file parallel write`) and redo and undo volume.

Parallelism: Use It Like a Power Tool

Parallel execution can shorten wall-clock time, but it can also create new bottlenecks that did not exist before:

- Saturate I/O
- Increase contention on hot objects
- Starve OLTP sessions of CPU
- Amplify skew (one slow slave can gate the whole step)

If you cannot explain which resource you are trading for time, parallelism is a gamble.

Treat parallelism as a *scheduling* decision, not a default optimizer hint.

Parallel Query and Parallel DML

Parallel DML is off by default at the session level; enabling it is a prerequisite for any parallel insert, update, or delete. Listing 13.5 shows the one-line switch.

Listing 13.5 Enabling parallel DML for a session

```
ALTER SESSION ENABLE PARALLEL DML;
```

With the session prepared, a hinted insert-select spreads the work across parallel slaves. Listing 13.6 shows the pattern.

Listing 13.6 Example: parallel insert-select into a staging table

```
INSERT /*+ APPEND PARALLEL(s, 8) */
INTO   stage_sales s
SELECT /*+ PARALLEL(f, 8) */
       f.*
FROM   fact_sales f
WHERE  f.sales_dt >= DATE '2026-01-01';

COMMIT;
```

Notes:

- `APPEND` requests direct-path insert (more in the section "Direct-Path Loads and Minimal Logging" later in this chapter).
- Use degrees that match the system's real capacity and the job's I/O pattern. "8" is an example, not a recommendation.

Use these notes as guardrails. They are the difference between a faster job and a faster outage.

Chunked Parallelism with DBMS_PARALLEL_EXECUTE

For large updates and deletes, chunking is often safer than a single massive transaction. It improves restartability and reduces rollback risk. Listing 13.7 shows how to chunk work by ROWID with `DBMS_PARALLEL_EXECUTE`.

Listing 13.7 Chunking a large update by ROWID with DBMS_PARALLEL_EXECUTE

```
DECLARE
  l_task VARCHAR2(30) := 'ORDERS_FIX_01';
BEGIN
  DBMS_PARALLEL_EXECUTE.create_task(task_name => l_task);

  DBMS_PARALLEL_EXECUTE.create_chunks_by_rowid(
    task_name   => l_task,
```

```
    table_owner => USER,
    table_name  => 'ORDERS',
    by_row      => TRUE,
    chunk_size  => 50000
  );

  DBMS_PARALLEL_EXECUTE.run_task(
    task_name      => l_task,
    sql_stmt       => q'[
      UPDATE orders
      SET    status = 'CLOSED'
      WHERE  status = 'DONE'
      AND    rowid BETWEEN :start_id AND :end_id
    ]',
    language_flag  => DBMS_SQL.NATIVE,
    parallel_level => 8
  );

  DBMS_PARALLEL_EXECUTE.drop_task(task_name => l_task);
END;
/
```

Use chunk status views (`USER_PARALLEL_EXECUTE_CHUNKS`) to identify and rerun failed chunks rather than restarting the entire job.

Direct-Path Loads and Minimal Logging

For large loads into new or transient segments, direct-path insert can reduce buffer cache churn and improve throughput. Before you use it, be clear about what changes operationally:

- `INSERT /*+ APPEND */` writes above the high-water mark.
- The load may generate different redo characteristics than conventional inserts.

Those differences are why direct-path can be faster, and also why it must be used intentionally.

Be conservative with reduced-logging approaches in systems that require strict recoverability. Treat them as a controlled, documented operational decision.

Partition-Friendly Batch Design

If tables are partitioned, batch operations should align with the partition key. Common patterns are simple, and they all aim at the same goal, move work from row processing to metadata changes:

- Load into a staging table and use *partition exchange*.
- Drop or truncate old partitions instead of deleting rows.
- Gather statistics at the partition level after large changes.

Even without deep partitioning knowledge, one rule holds: if you can convert a row-level operation into a metadata-level operation, you usually win.

Operational Guardrails: Make Batch Jobs Boring

Batch jobs fail in production for mundane reasons: unexpected data volumes, blocking locks, plan regressions, or resource contention. To keep failures boring and recoverable, build guardrails that make progress measurable and reruns safe:

- **Idempotency**: Rerunning the job should not corrupt results.
- **Checkpoints**: Record progress by key ranges or chunk IDs.
- **Resource caps**: Schedule parallelism by time windows or consumer groups.
- **Telemetry**: Log counts, elapsed time, and key wait events.

If you add these guardrails early, you can tune more aggressively later because rollback and restart are planned.

Summary

Batch workloads reward set-based thinking: the fewer round-trips between PL/SQL and SQL, the faster the job. The deeper point is operational, when batch is set-based, measurable, and restartable, you can push it hard without fear.

With that framing, the essentials from this chapter are

- Rewrite batch work into set-based SQL first; bulk processing is the fallback when you need procedural logic.
- Use `BULK COLLECT` and `FORALL` to reduce context switching.
- Treat commit frequency as a tuning lever tied to undo and redo, locks, and restart strategy.
- Prefer chunked parallelism (`DBMS_PARALLEL_EXECUTE`) for large corrective DML.
- Use direct-path and reduced-logging techniques only with clear recovery requirements and operational controls.

Taken together, these ideas keep batch work fast, and they keep it safe.

Quick Checklist

Use the checklist below as a tuning sequence. It starts by eliminating row by row work, then adds bulk binds, then increases parallelism only after restartability and measurement are in place.

- Convert row-by-row logic to set-based SQL where possible.
- If PL/SQL is required, bulk bind (`BULK COLLECT` and `FORALL`) and batch commits.
- Pick a restart strategy (idempotent logic and checkpoints) before increasing parallelism.
- Control resource usage (parallel degree, consumer groups, time windows) to protect OLTP.
- Capture telemetry (rows, elapsed, waits) so regressions are obvious.

If you cannot prove that the batch job got faster and stayed correct, you did not finish tuning.

Next, Chap. 14 (“Observability: Measuring, Monitoring, and Proving Improvements”) shows how to collect evidence, build baselines, and validate that your fixes improved user experience without regressing other workloads.

Chapter 14
Observability: Measuring, Monitoring, and Proving Improvements



Tuning without measurement is guesswork. In production, the hard part is often not *finding* a change, but proving (to yourself and everyone else) that it improved user experience, did not break other workloads, and will remain stable next week.

This chapter builds a practical observability toolkit for Oracle performance. It starts by defining a measurement contract, what you are optimizing, where, and what must not regress. Next, it shows how to diagnose live pain with `V$SESSION`, wait events, and blocking chains. It then moves to SQL-level evidence, confirm plan identity and per-execution cost, and use SQL Monitor for long-running executions. After that, it covers time-series tools, AWR and ASH, for baselines, regressions, and the "what changed" question. Finally, it shows how to instrument application work with module and action tags, and how to use targeted tracing when the built-in views are not enough.

Start With a Measurement Contract

Before changing anything, define three things. Treat them as a contract, they make it clear what success looks like and what failure looks like:

- **Goal metric**: Response time, throughput, CPU, I/O, or queue time
- **Scope**: Which endpoint, job, or report, and which time window
- **Safety**: What must *not* regress (e.g., OLTP latency during a batch load)

If you cannot state these, you will end up "optimizing" something that does not matter.

Once the contract is written down, the rest of the chapter becomes a recipe, pick the tool that matches the question and collect evidence that supports a specific action.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_14

Live Diagnosis: What Are Sessions Waiting For?

When users report slowdowns *right now*, start with live session state.

Active Sessions and Wait Events

When someone reports a slowdown in progress, the first query to run shows who is active right now and what each session is waiting on. Listing 14.1 is a compact starting point for live diagnosis.

Listing 14.1 Live view of active sessions: what they are waiting on right now

```
SELECT s.sid,
       s.serial#,
       s.username,
       s.status,
       s.state,
       s.event,
       s.wait_class,
       s.seconds_in_wait,
       s.sql_id,
       s.module,
       s.action
FROM   v$session s
WHERE  s.type   = 'USER'
AND    s.status = 'ACTIVE'
ORDER  BY s.seconds_in_wait DESC;
```

Interpretation hints (based on documented `V$SESSION.STATE` values such as `WAITING` and the `WAIT_CLASS` classification), these are the signposts that help you choose the next tool:

- **CPU pressure**: sessions with `WAIT_CLASS='CPU'` are running on CPU (not waiting on an event). Do not expect `STATE` to show "ON CPU"; `STATE` reports wait-state, not CPU.
- **Concurrency**: lock/latch related waits (for example, `WAIT_CLASS='Concurrency'` or `WAIT_CLASS='Application'`) usually point to hot objects, blocking chains, or serialization.
- **I/O**: `WAIT_CLASS='User I/O'` (and the specific event) guides whether the workload is index-driven (single-block reads) or scan-driven (multiblock reads).

Do not over-interpret a single sample. Use the hints to form a hypothesis, then validate it with SQL-level and time-series evidence.

Who Is Blocking Whom?

If the waits are lock-related, the next question is who is holding things up. Listing 14.2 pairs each waiter with its blocker and the SQL involved.

Listing 14.2 Identify blockers and the SQL they are running

```
SELECT w.sid         AS waiting_sid,
       w.sql_id      AS waiting_sql_id,
       w.event       AS waiting_event,
       b.sid         AS blocking_sid,
       b.sql_id      AS blocking_sql_id,
       b.module      AS blocking_module,
       b.action      AS blocking_action
FROM   v$session w
JOIN   v$session b
       ON b.sid = w.blocking_session
WHERE  w.blocking_session IS NOT NULL;
```

Once you know the blocking session, you can take a few concrete follow-up steps:

- Inspect its current SQL and execution plan.
- Check whether it is stuck on I/O.
- Decide whether to wait, fix the root cause, or, rarely, terminate it.

The key is sequencing, identify the blocker first, then decide on the least-risk intervention.

SQL-Level Evidence: From SQL_ID to Plan and Monitoring

Confirm Plan Identity and Per-Execution Cost

Before and after any fix, capture the plan hash and per-execution cost so you can prove the change helped for the right reason. Listing 14.3 shows a compact query you can reuse during tuning.

Listing 14.3 Per-execution cost and plan hash for a single statement

```
SELECT sql_id,
       child_number,
       plan_hash_value,
       executions,
       ROUND(elapsed_time/NULLIF(executions,0)/1e6, 4) AS
    avg_elapsed_s,
       ROUND(cpu_time/NULLIF(executions,0)/1e6,     4) AS avg_cpu_s,
       ROUND(buffer_gets/NULLIF(executions,0))         AS avg_buf_gets,
```

```
       ROUND(disk_reads/NULLIF(executions,0))          AS
    avg_disk_reads
FROM   v$sql
WHERE  sql_id = :sql_id
ORDER  BY child_number;
```

A plan change is often visible first as a new `PLAN_HASH_VALUE` for the same `SQL_ID`. If performance changed, validate whether the plan changed *and* whether execution statistics changed.

SQL Monitoring for Long-Running Statements

For statements that run long enough, Oracle can generate SQL Monitor reports. This is especially useful for parallel query and large join pipelines.

> **Licensing**
> Real-Time SQL Monitoring (SQL Monitor) is part of the Oracle Tuning Pack. In many environments you can query `V$SQL_MONITOR`, but generating and using full SQL Monitor reports requires the appropriate licensing.

Listing 14.4 shows a quick way to find executions that are currently being monitored, so you can jump straight to a SQL Monitor report.

Listing 14.4 Find currently monitored SQL

```
SELECT sql_id,
       status,
       sql_exec_id,
       sql_exec_start,
       elapsed_time/1e6 AS elapsed_s,
       px_servers_requested,
       px_servers_allocated
FROM   v$sql_monitor
WHERE  status = 'EXECUTING'
ORDER  BY sql_exec_start;
```

An SQL Monitor report answers questions that plans alone cannot: where time is spent *inside* the execution (I/O vs. CPU vs. waits) and whether parallel slaves are balanced.

AWR and ASH: Baselines, Regressions, and "What Changed?"

Live views explain the present; AWR and ASH explain the past.

Licensing
AWR (Automatic Workload Repository) and ASH (Active Session History) are part of Oracle's Diagnostic Pack and require the appropriate licensing. If Diagnostic Pack is not available in your environment, use alternatives such as Statspack, OS metrics, targeted sampling queries against `V$` views, and application instrumentation to build before/after evidence.

AWR Snapshots and Top SQL

AWR is designed for trend and regression analysis. Use it to compare "before" vs. "after" periods and to answer: did we shift CPU, I/O, parses, or concurrency?

Listing 14.5 shows a time-windowed AWR query that ranks SQL by elapsed time, which is a good starting point for before and after comparisons.

Listing 14.5 Top SQL by elapsed time in a time window (AWR)

```
SELECT *
FROM (
  SELECT s.sql_id,
         SUM(s.elapsed_time_delta)/1e6 AS elapsed_s,
         SUM(s.cpu_time_delta)/1e6     AS cpu_s,
         SUM(s.buffer_gets_delta)      AS buffer_gets,
         SUM(s.disk_reads_delta)       AS disk_reads
  FROM   dba_hist_sqlstat s
  JOIN   dba_hist_snapshot sn
         ON sn.snap_id = s.snap_id
        AND sn.dbid    = s.dbid
        AND sn.instance_number = s.instance_number
  WHERE  sn.begin_interval_time >= :begin_ts
  AND    sn.end_interval_time   <= :end_ts
  GROUP  BY s.sql_id
  ORDER  BY SUM(s.elapsed_time_delta) DESC
)
WHERE  ROWNUM <= 20;
```

ASH: Sampled Session History

ASH is invaluable when a problem is intermittent. Instead of relying on a single "snapshot" query, ASH can show what sessions spent time doing over an interval.

Listing 14.6 shows a simple ASH query that turns an intermittent complaint into a ranked list of waits for a specific SQL_ID.

Listing 14.6 ASH: top waits for a SQL_ID over an interval

```
SELECT event,
       wait_class,
       COUNT(*) AS samples
FROM   v$active_session_history
WHERE  sample_time BETWEEN :begin_ts AND :end_ts
AND    sql_id = :sql_id
GROUP  BY event, wait_class
ORDER  BY samples DESC;
```

When you can link a user-visible slowdown to a spike in ASH samples for a specific wait class, you have evidence strong enough to guide engineering work.

Tracing and Targeted Instrumentation

AWR/ASH are broad. When you need microlevel detail for a single session or a single module, trace.

Tag Work with `MODULE`, `ACTION`, and Client IDs

Good instrumentation turns "the database is slow" into "checkout latency is slow."

Listing 14.7 shows how to set `MODULE` and `ACTION` tags, which makes AWR, ASH, and `V$SESSION` data searchable by feature.

Listing 14.7 Set module/action so DBAs can attribute workload

```
BEGIN
  DBMS_APPLICATION_INFO.set_module(
    module_name => 'checkout',
    action_name => 'place_order'
  );
END;
/
```

Once set, these fields appear in `V$SESSION`, ASH, and AWR, allowing you to filter by application feature.

Enable a Targeted SQL Trace

When the views are not enough, a targeted 10046 trace records every wait and bind for one session, without the overhead of tracing the whole instance. Listing 14.8 shows the minimal workflow.

Listing 14.8 Enable and disable trace for a specific session

```
EXEC DBMS_MONITOR.session_trace_enable(session_id => :sid, serial_num
    => :serial, waits => TRUE, binds => TRUE);
-- reproduce the issue
EXEC DBMS_MONITOR.session_trace_disable(session_id => :sid, serial_num
    => :serial);
```

Trace files can be formatted with `tkprof`. Use tracing sparingly in busy systems, and prefer short, targeted captures.

Regression Proof: Before/After Checklists

A tuning change is complete only when you can show specific evidence. Use the list below as a definition of done:

- The plan, or the reason you accepted a plan change
- The per-execution cost change
- The time-series behavior (AWR and ASH) over representative load
- The operational risk (locks, undo, redo, and parallel impact)

If you cannot produce this evidence, the change may still be correct, but it is not yet defensible.

Do not stop at "it feels faster." Stop when you can prove what changed and why it will stay improved.

Summary

You cannot tune what you cannot see. This chapter assembled the observability toolkit that makes every other chapter's techniques measurable. The goal is to move from opinions to evidence, so tuning discussions are about data, not anecdotes.

With that framing, the takeaways below summarize which tool to use for which question:

- Use live views (`V$SESSION`) for immediate symptoms, and identify blockers early.
- Use `V$SQL` to validate per-execution cost and detect plan changes via `PLAN_HASH_VALUE`.

- Use SQL Monitor for long-running statements to understand where time is spent inside the execution.
- Use AWR for trends and ASH for intermittent issues and wait-class breakdowns.
- Instrument application features with module and action so performance work maps to user-visible behavior.

Taken together, these tools let you prove improvements, and just as importantly, they let you prove when a change did not help.

Quick Checklist

Use the checklist below as an investigation sequence. It starts with live symptoms, then confirms the plan and per-execution cost, then picks the right historical tool for the time horizon.

- Start with the symptom: top waits, top SQL, and whether sessions are blocked.
- Confirm the plan and per-exec cost (do not tune from total elapsed alone).
- For long SQL, use SQL Monitor to find the expensive operations and skew.
- For intermittent issues, use ASH; for long-term trends, use AWR.
- Tag the workload (module, action, client ID) so findings map to real features.

If you follow that order, you will spend less time arguing about causes and more time fixing the right thing.

Next, Chap. 15 (“Instance Resource Tuning: CPU, Memory, and I/O”) shows how to use this measurement toolkit to identify the true limiting resource then tune the instance and host so well-tuned SQL can actually run fast under load.

Chapter 15
Instance Resource Tuning: CPU, Memory, and I/O



SQL tuning and application changes deliver the biggest wins, but they still run inside an Oracle instance with finite CPU, memory, and storage bandwidth. When systems become CPU-saturated, memory-constrained, or I/O-bound, even "good" SQL slows down.

This chapter provides a practical, DBA-friendly workflow for instance-level resource tuning. It starts with classification, confirm whether the limiting factor is CPU, memory, I/O, or concurrency, because symptoms often look the same at the application layer. It then walks through the three major resource domains, CPU, memory, and I/O, with queries you can run immediately to find the top offenders and the dominant wait profile. Finally, it shows how to keep mixed workloads predictable using services and Resource Manager, so batch throughput does not quietly steal interactive latency.

CPU: When *Everything* Looks Like a Performance Problem

CPU pressure often masks itself as many different symptoms: higher latency, queueing, "random" timeouts, and increased wait times across unrelated modules.

Find CPU-Heavy SQL

CPU pressure almost always traces back to a handful of statements. Listing 15.1 ranks the cursor cache by total CPU, and it is often enough to identify the dominant workload drivers.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_15

Listing 15.1 Top SQL by total CPU time (current cursor cache)

```
SELECT sql_id,
       executions,
       cpu_time/1e6 AS cpu_s,
       elapsed_time/1e6 AS elapsed_s,
       ROUND(cpu_time/NULLIF(executions,0)/1e6,4) AS avg_cpu_s,
       ROUND(buffer_gets/NULLIF(executions,0))    AS avg_buf_gets,
       SUBSTR(sql_text,1,80)                      AS sql_preview
FROM   v$sql
WHERE  executions > 0
ORDER  BY cpu_time DESC
FETCH FIRST 20 ROWS ONLY;
```

If CPU is saturated, the fastest way to improve throughput is usually to reduce logical I/O (`BUFFER_GETS`) and parse overhead, not to change a single initialization parameter.

Reduce Parse and Library Cache Pressure

Hard parses consume CPU and can create shared pool contention. Before you tune memory or add CPU, eliminate avoidable parse work:

- Use bind variables (Chap. 12).
- Ensure the application reuses sessions and prepared statements.
- Avoid generating "unique" SQL text for every request.

If parse pressure is high, instance sizing changes will not fix the root cause, stable SQL text and cursor reuse will.

Memory: SGA, PGA, and the Cost of Spilling to Disk

Oracle performance depends heavily on avoiding unnecessary physical I/O and on keeping work areas (sorts, hash joins) in memory.

SGA: Buffer Cache and Shared Pool

The buffer cache reduces disk reads; the shared pool reduces parse cost. Instead of tuning blindly, observe outcomes. Listing 15.2 shows the key dynamic SGA components so you can see where memory is allocated.

Listing 15.2 Key SGA components

```
SELECT component,
       ROUND(current_size/1024/1024) AS mb
FROM   v$sga_dynamic_components
ORDER  BY current_size DESC;
```

High CPU from SQL is one story; the other common pattern is CPU burned on logical reads that should have been avoided. Listing 15.3 helps you decide whether physical reads are a meaningful part of the story.

Listing 15.3 Are we doing physical reads because caching is insufficient?

```
SELECT name,
       value
FROM   v$sysstat
WHERE  name IN (
  'physical reads',
  'db block gets',
  'consistent gets'
);
```

Heuristic guidance, use these patterns to decide whether you should tune SQL or memory first:

- High physical reads *with* high buffer gets often means poor access paths (SQL issue).
- High physical reads *with* modest buffer gets can indicate a working set larger than memory (cache pressure).

Treat this as a starting hypothesis, then validate it with the top SQL and the wait profile.

PGA: Sort/Hash Work Areas

When work areas spill to TEMP, response time can degrade sharply. Listing 15.4 shows a quick view of PGA pressure and whether the instance is routinely spilling.

Listing 15.4 PGA work area efficiency (spills vs. optimal)

```
SELECT name,
       value
FROM   v$pgastat
WHERE  name IN (
  'cache hit percentage',
  'total PGA allocated',
  'total PGA inuse',
  'over allocation count'
);
```

If `over allocation count` grows and users report slow sorts or hash joins, consider PGA pressure, but also check whether queries are forcing large sorts (missing indexes, overly broad result sets).

TEMP Usage: Detect the Top Offenders

TEMP exhaustion tends to strike suddenly, usually from one oversized sort or hash join. Listing 15.5 names the sessions consuming TEMP right now.

Listing 15.5 Sessions currently consuming the most TEMP

```
SELECT s.sid,
       s.serial#,
       s.username,
       s.sql_id,
       ROUND(u.blocks * t.block_size / 1024 / 1024) AS temp_mb,
       s.module,
       s.action
FROM   v$sort_usage u
JOIN   v$session s ON s.saddr = u.session_addr
JOIN   dba_tablespaces t ON t.tablespace_name = u.tablespace
ORDER  BY temp_mb DESC;
```

TEMP spikes are often caused by a small number of predictable query shapes:

- Large `ORDER BY` or `GROUP BY` without selective predicates
- Hash joins on very large rowsets
- Analytics over broad time ranges
- Insufficient PGA for the concurrent workload

If you can identify which shape applies, you can often fix the root cause in SQL before you resize TEMP.

I/O: Separate Random Reads, Scans, and Logging

Not all I/O is the same. Before you change storage or chase an I/O wait, classify which subsystem is actually under pressure:

- Random single-block reads (index-driven OLTP)
- Large multiblock reads (scans, reporting)
- Redo log writes (commit latency)
- TEMP spills

Each subsystem has different fixes, and mixing them leads to wasted effort.

Identify the Dominant I/O Wait Classes

Before touching storage, confirm that I/O actually dominates the wait profile. Listing 15.6 ranks system-wide wait events by time waited, and it quickly tells you whether you should be looking at I/O at all.

Listing 15.6 Top wait events by time waited (system-wide)

```
SELECT event,
       wait_class,
       time_waited/100 AS time_s
FROM   v$system_event
WHERE  wait_class <> 'Idle'
ORDER  BY time_waited DESC
FETCH FIRST 15 ROWS ONLY;
```

Note
`V$SYSTEM_EVENT` values are cumulative since instance startup. For incident diagnosis, compare deltas between two points in time (take two snapshots and subtract), or use recent-window views such as ASH (if licensed) to avoid chasing an event that was dominant hours ago but is quiet now.

Common interpretations, use these wait names as a translation layer from symptoms to root cause:

- `db file sequential read`: Random single-block reads (often index lookups)
- `db file scattered read`: Multiblock reads (full scans)
- `log file sync`: Commit latency (application commits, redo I/O)
- `direct path read/write temp`: TEMP spills

Once you can name the dominant event, you can choose the next diagnostic query instead of guessing.

Redo and Commit Latency

Commit performance is a frequent bottleneck for write-heavy OLTP systems. Listing 15.7 shows a high-level view of redo volume and commit behavior.

Listing 15.7 Redo generation rate and commit behavior (high-level)

```
SELECT name,
       value
FROM   v$sysstat
WHERE  name IN (
  'redo size',
  'user commits',
  'user rollbacks'
);
```

If `log file sync` dominates, look for a small set of common causes:

- Too many commits (chatty application)
- Undersized or slow redo storage
- Log switches that happen too frequently

Each cause has a different fix, reduce commit frequency, fix redo storage, or adjust redo log sizing, so it is worth confirming which one applies.

Managing Mixed Workloads with Resource Manager

When batch jobs and OLTP share a database, the right goal is usually not "maximum throughput" but *predictable* latency for the business-critical work.

Oracle Database Resource Manager can cap CPU usage for lower-priority consumer groups and prevent batch parallelism from starving interactive work.

Listing 15.8 shows a simple ASH query that groups activity by service name, which is a practical starting point for workload isolation and Resource Manager design.

Listing 15.8 Identify workload by service name (useful for Resource Manager strategy)

```
SELECT service_name,
       COUNT(*) AS active_sessions
FROM   v$active_session_history
WHERE  sample_time >= SYSTIMESTAMP - INTERVAL '5' MINUTE
GROUP  BY service_name
ORDER  BY active_sessions DESC;
```

Even if you do not implement Resource Manager immediately, separating traffic by *service* is a foundational observability and control technique.

A Minimal Instance Tuning Checklist

Use this checklist before making parameter changes. It forces you to name the bottleneck and the evidence first, so you do not confuse workload problems with configuration problems:

- Is the bottleneck CPU, memory, I/O, or concurrency (Chap. 14)?
- Do top SQL statements have high per-execution cost?
- Are we spilling to TEMP (sort and hash work areas)?
- Is commit latency dominated by redo I/O (`log file sync`)?
- Are batch jobs isolated (services, consumer groups, schedule windows)?

If you cannot answer these, changing SGA and PGA sizes is likely to be placebo.

Once you can answer them, you can make smaller, safer changes, and you can verify their effect under comparable load.

Summary

Instance resources set the ceiling for everything SQL-level tuning can achieve. The theme is constraint, once you know which resource is limiting, you can stop guessing and start making changes that move a measurable metric.

With that framing, the core lessons are:

- Confirm the real bottleneck first; symptoms are often misleading.
- CPU pressure is usually solved by reducing logical I/O and parse cost.
- Memory tuning is about avoiding physical reads and TEMP spills, not chasing a single "perfect" cache hit ratio.
- I/O problems split into random reads, scans, redo writes, and TEMP.
- Use services (and optionally Resource Manager) to keep mixed workloads predictable.

Taken together, these lessons prevent the most common failure mode in instance tuning, changing parameters without changing outcomes.

Quick Checklist

Use the checklist below as a safe tuning sequence. It starts with classification, then targets the top SQL, then changes one knob at a time so you can attribute cause and effect.

- Classify the bottleneck (CPU vs. I/O vs. memory vs. contention) from wait data.
- List the top SQL by DB time and validate per-execution cost and plans.
- Check for TEMP spills and redo bottlenecks (`log file sync`).
- Make one change at a time (parameter, index, plan, workload isolation).
- Remeasure under comparable load; keep a rollback plan.

If you cannot explain why a change helped, or why it did not, treat it as a signal to revisit the measurement, not as a reason to keep turning knobs.

Next, Chap. 16 ("High Availability and Performance: RAC, Data Guard, and Resilience") shows how availability architecture changes performance behavior, and how to design for resilience without accidental regressions.

Chapter 16
High Availability and Performance: RAC, Data Guard, and Resilience



High availability features are often adopted for uptime, but they have a performance footprint. RAC changes how caching and locking behave. Data Guard changes commit and shipping behavior. Failover strategies influence connection pool design and workload routing.

This chapter previews the trade-offs you need to make explicit (RTO/RPO vs. latency), then walks through what typically slows down RAC and Data Guard in practice, how to measure it, and the design patterns that keep resilience from turning into surprise regression. The goal is simple: engineer for availability *and* keep performance predictable under both steady-state load and failover conditions.

Define the Availability Target (RTO/RPO) and Performance Target

Before choosing technology, write down what "good" looks like for both availability and performance:

- **RTO (Recovery Time Objective)**: How long you can be down
- **RPO (Recovery Point Objective)**: How much data you can lose
- **Latency SLO**: Acceptable commit/response latency under normal load

These targets become the lens for every choice that follows (RAC design, Data Guard transport mode, and failover behavior). Many availability decisions are really trade-offs between RPO and latency, so make the trade explicit and measure it.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_16

RAC: Scaling Reads and Surviving Node Failure

RAC can increase availability and scale out some workloads, but it introduces a new class of performance constraints: global cache coordination.

The Global Cache: Why "Hot" Blocks Hurt More in RAC

In a single instance, a hot block causes contention locally. In RAC, that same hot block can also cause cross-instance traffic.

Typical causes of global cache overhead include a few repeat offenders that turn local contention into cross-instance block traffic:

- Sequences without caching
- Indexes with monotonically increasing keys (right-hand growth)
- Single-row lookups that become a write hotspot
- Frequent updates to the same small set of rows

If you can reduce or localize these hotspots, you usually reduce *both* cluster wait time and tail latency.

Identify Global Cache Waits

In an RAC environment, the first diagnostic question changes: before asking which SQL is slow, ask how much time the cluster spends shipping blocks between instances. As shown in Listing 16.1, start by ranking the `gc` waits across the cluster.

Listing 16.1 System-wide cluster wait events (RAC)

```
SELECT inst_id,
       event,
       wait_class,
       time_waited/100 AS time_s
FROM   gv$system_event
WHERE  wait_class <> 'Idle'
AND    event LIKE 'gc%'
ORDER  BY time_waited DESC;
```

This view is a fast "are we paying for RAC coordination?" check: if `gc` events dominate the top of the list, you are likely dealing with cross-instance block pinging rather than a purely local SQL tuning problem.

To see whether the interconnect traffic is balanced or one instance is doing all the shipping, break the statistics down per instance. Listing 16.2 shows one simple way to compare key `gc` counters by instance:

Listing 16.2 Global cache statistics by instance

```
SELECT inst_id,
       name,
       value
FROM   gv$sysstat
WHERE  name IN (
  'gc current blocks received',
  'gc cr blocks received',
  'gc current block receive time',
  'gc cr block receive time'
)
ORDER  BY inst_id, name;
```

Use this breakdown to spot imbalance: if one instance shows much higher receive counts or receive time, it is often where the shared-state hotspot is landing. If you see heavy `gc` wait time overall, look for a small number of objects or SQL statements driving block transfers.

RAC-Friendly Design Patterns

Most global cache pain is designed in, not configured in. The point of the patterns below is to reduce shared-state contention so each instance can do more work locally:

- **Avoid hot blocks**: Use sequence caching, reverse key indexes (selectively), and spread inserts when appropriate.
- **Partition-aware access**: Route work by partition key to reduce cross-instance block movement.
- **Service placement**: Pin services to preferred instances when you need locality.

Applied consistently, these choices tend to reduce interconnect traffic and make RAC scaling more predictable.

RAC performance is often less about “more nodes” and more about “less shared state”.

Data Guard: Protecting Data Without Surprising Latency

Data Guard provides standby databases for HA/DR. The main performance question is commit behavior.

SYNC vs. ASYNC Transport

At a high level, the transport choice determines whether the primary "pays" for durability in commit latency:

- **SYNC** modes can add commit latency because the primary waits for acknowledgment.
- **ASYNC** modes reduce commit latency but allow some data loss in a disaster (larger effective RPO).

The key is to make the trade-off explicit: pick the mode that meets business RPO, then validate the actual latency impact with measured network/storage behavior.

Observe Transport and Apply Lag

For Data Guard, the numbers that matter are transport lag and apply lag; both are exposed directly in the standby views:

Listing 16.3 Data Guard lag and status (primary/standby views)

```
SELECT name,
       value,
       unit,
       time_computed,
       datum_time
FROM   v$dataguard_stats
WHERE  name IN ('transport lag', 'apply lag', 'apply finish time');
```

A simple rule of thumb is that stable lag during peak load suggests the standby has capacity headroom, while rising lag is a signal to investigate apply throughput, redo generation rate, or the network path.

Transport/apply lag is an operational signal: rising lag during peak load may mean the standby cannot keep up, or the network path is constrained.

Read Scaling: Offload Queries Safely

A standby can be used for read-only workloads (reporting, extracts), but only if you make the consistency model explicit and enforce it in routing.

Licensing
Querying a physical standby while Redo Apply remains active uses the Active Data Guard real-time query feature and requires the appropriate Oracle Active Data Guard license. Without Active Data Guard, you can query the standby only when it is opened read-only with apply stopped.

- Data may be slightly stale (apply lag).
- Some reports require read consistency relative to primary commits.
- Connection routing must be explicit and testable.

The takeaway is to treat standby reads as a product feature with documented expectations, not a "free" performance win.

Route by Service and Workload Class

The cleanest pattern is to separate responsibilities so routing stays predictable and testable:

- OLTP reads/writes on the primary service
- Analytics/reporting on a standby service

When you do this, the performance model stays simple, and you avoid accidental "report storms" on the primary.

Failover Without a Thundering Herd

Failover often creates a second problem: every application instance reconnects at once, rewarms caches, and re-parses SQL.

Mitigations work best when you think of failover as a temporary overload event that you can shape and throttle:

- Use connection pools with sane retry/backoff.
- Ensure bind usage and cursor reuse so shared pool warm-up is fast.
- Keep critical objects in the buffer cache with stable access patterns.

Together, these reduce reconnect storms and shorten the "back to steady state" window. Treat failover as a *load test scenario*, not a theoretical capability.

Resilience Patterns That Improve Performance

Some HA patterns also improve performance stability. The common thread is that they prevent uncontrolled queuing and protect the database from overload:

- **Workload shedding**: Degrade noncritical features when the database is under pressure.
- **Queueing and async processing**: Smooth spikes and protect OLTP.
- **Circuit breakers**: Fail fast instead of piling up waiting sessions.

Used together, these patterns reduce tail latency and help prevent cascading failures during partial outages or demand spikes.

Summary

High availability features change the performance rules. The goal is not just "stay up," but to stay up *predictably*: architecture choices that improve resilience can also change latency, contention, and operational behavior.

With that framing, the core lessons from this chapter are:

- Availability choices are latency choices; define RTO/RPO and performance SLOs first so trade-offs are explicit.
- In RAC, the biggest surprises come from shared-state hotspots; measure `gc` waits and design to reduce cross-instance block movement.
- In Data Guard, transport mode determines how much commit latency you are willing to buy to reduce RPO; monitor transport/apply lag as an ongoing operating metric.
- If you offload reads, do it deliberately with services and documented consistency expectations.
- Treat failover as a performance event: plan for reconnect storms, cache warm-up, and post-failover throughput.

Quick Checklist

Use this checklist as a safe sequence for designing and validating HA without accidental regressions:

- Define RTO/RPO and a latency SLO; confirm the business trade-off you are making (durability vs. commit latency).
- For RAC: Look for shared-state hotspots ("hot" blocks) and quantify `gc` waits; reduce pinging with caching/partition-aware access.
- For Data Guard: Choose SYNC/ASYNC consciously; track commit impact and transport/apply lag during peak load.
- For read scaling: Route by service; document staleness/consistency rules and test that routing is enforced.

- For failover: Test under load; validate pool backoff, retry behavior, cursor reuse, and time-to-recovery for performance.

Next, Chap. 17 (“Performance Tuning Playbook: From Alert to Verified Fix”) turns these ideas into an end-to-end workflow you can run from first symptom to measured, regression-resistant improvement.

Chapter 17
Performance Tuning Playbook: From Alert to Verified Fix



The final chapter ties the book together into a repeatable workflow. Oracle performance tuning is not a bag of tricks; it is a discipline: observe, hypothesize, test, measure, and operationalize.

This chapter lays out a practical playbook you can run in an incident (minutes), through deeper diagnosis (hours), and into verified, regression-resistant fixes (days). It is designed to work under real constraints: partial information, production risk, competing stakeholders, and the pressure to "just make it faster."

Version, Privileges, and Licensing
Some views, syntax, and tooling referenced in this playbook vary by Oracle Database version and by privileges. In addition, features such as AWR/ASH and SQL Monitor have licensing implications (Diagnostic Pack / Tuning Pack). Verify your database version, required privileges, and applicable licensing before using any example in a production environment.

The Tuning Loop

Use the same loop at every scale (single query or whole system). The point is not to be clever, it is to be repeatable under pressure:

1. **Reproduce or capture evidence** (Chap. 14).
2. **Localize the cost**: which SQL, which waits, which resource.
3. **Choose the smallest safe change**.
4. **Verify improvement** with before/after measurements.
5. **Prevent regression**: baselines, tests, and dashboards.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0_17

If you skip steps (especially verification), you can spend days "tuning" while never changing user-visible outcomes.

Step 1: Triage the Incident (Minutes)

When performance is actively bad, focus on stabilizing the system.

Ask Three Questions

Before running a single query, spend 30 seconds framing the incident. These three questions decide the direction of everything that follows:

- Is the issue **global** (many sessions) or **localized**?
- Is the bottleneck **CPU**, **I/O**, or **concurrency**?
- Did something **change** (deploy, stats, data volume, plan, batch job)?

Answering them upfront prevents a common failure mode: jumping straight to "top SQL" when the system is actually blocked, overloaded, or reacting to a recent change.

Immediate Queries

The first two queries you run in any incident are always the same. Start with what active sessions are waiting on at this moment. As shown in Listing 17.1, you want a quick, current snapshot of the wait profile.

Listing 17.1 What are active sessions waiting on right now? (single-instance view)

```
SELECT s.sid,
       s.serial#,
       s.username,
       s.state,
       s.event,
       s.wait_class,
       s.sql_id,
       s.module,
       s.action
FROM   v$session s
WHERE  s.type='USER'
AND    s.status='ACTIVE'
ORDER  BY s.wait_class, s.event;
```

Tip
Listing 17.1 uses `V$SESSION`, which reflects a single instance. In RAC, use `GV$SESSION` and include `INST_ID` so you can see which instance the session is on.

This is the fastest way to confirm whether you are dealing with a broad resource issue (CPU/I/O) or a concurrency issue that will show up as specific waits and blocking patterns.

Then check immediately for blocking, because a lock chain changes the entire investigation. As shown in Listing 17.2, identify who is waiting and who is blocking:

Listing 17.2 Are we blocked?

```
SELECT w.sid AS waiting_sid,
       b.sid AS blocking_sid,
       w.event,
       w.sql_id AS waiting_sql_id,
       b.sql_id AS blocking_sql_id
FROM   v$session w
JOIN   v$session b
       ON b.sid = w.blocking_session
WHERE  w.blocking_session IS NOT NULL;
```

Treat this as a fork in the road: if you find blockers, resolve the lock chain first; if you do not, you can focus on the dominant wait class and resource.

If the system is unstable, reduce pressure first (pause batch jobs, reduce parallelism, shed noncritical traffic) before attempting deeper tuning.

Step 2: Isolate the Top Cost (Hours)

Once stable, identify what is driving time.

Top SQL by Total and Per-Execution Cost

With the wait picture in hand, rank the SQL. As shown in Listing 17.3, total elapsed time finds the statements dominating the system.

Listing 17.3 Top SQL by total elapsed time (cursor cache)

```
SELECT sql_id,
       executions,
       elapsed_time/1e6 AS elapsed_s,
       cpu_time/1e6     AS cpu_s,
       buffer_gets,
       disk_reads,
       plan_hash_value,
       SUBSTR(sql_text,1,80) AS sql_preview
FROM   v$sql
WHERE  executions > 0
ORDER  BY elapsed_time DESC
FETCH FIRST 20 ROWS ONLY;
```

Use Listing 17.3 to identify which statements dominate DB time at the system level. Once you have a short candidate list, confirm whether the pain is caused by *frequency* (many executions) or *cost per call*.

Per-execution cost tells a different story; it surfaces the queries that hurt *every single time* they run. As shown in Listing 17.4, this view is often where you find high-leverage rewrites and indexing opportunities.

Listing 17.4 Top SQL by per-execution elapsed time (find painful queries)

```
SELECT sql_id,
       executions,
       ROUND(elapsed_time/NULLIF(executions,0)/1e6,4) AS avg_elapsed_s,
       ROUND(buffer_gets/NULLIF(executions,0))        AS avg_buf_gets,
       ROUND(disk_reads/NULLIF(executions,0))         AS
    avg_disk_reads,
       plan_hash_value,
       SUBSTR(sql_text,1,80)                          AS sql_preview
FROM   v$sql
WHERE  executions >= 10
ORDER  BY avg_elapsed_s DESC
FETCH FIRST 20 ROWS ONLY;
```

Use this ranking to find the statements where a single plan change, rewrite, or index can improve every request. Taken together, totals point to workload drivers, while per-execution cost points to the statements most likely to benefit from targeted tuning.

Step 3: Choose the Right Fix Type

Map symptoms to the smallest safe fix.

If It Is I/O Heavy

Typical fixes for I/O-heavy work focus on reading less data, reading it more selectively, or making reads cheaper:

- Add/adjust indexes (Chap. 7).
- Rewrite predicates to be sargable (Chaps. 3 and 4).
- Reduce result volume (Chap. 10).
- Fix skew with better stats/histograms (Chap. 5).

Tie the fix back to the evidence: random reads, scans, and TEMP spills usually point to different changes.

If It Is CPU-Heavy

Typical fixes for CPU-heavy work focus on doing less work per call (fewer rows, fewer operations) and removing waste (parsing and unnecessary computation):

- Reduce logical I/O (buffer gets).
- Reduce parse (binds, session reuse).
- Simplify expensive expressions.
- Avoid unnecessary sorts and scalar subqueries.

If CPU is saturated, a small reduction in buffer gets or parse overhead often shows up immediately in throughput and tail latency.

If It Is Concurrency-Heavy

Typical fixes for concurrency-heavy systems focus on removing serialization: shorter transactions, fewer hot rows/blocks, and less shared state:

- Remove hot rows/blocks (Chap. 8).
- Change transaction scope/commit behavior (Chap. 13).
- Avoid serializing patterns (single-row counters, un-cached sequences).
- Schedule parallel/batch work away from OLTP peaks.

The practical test is whether you reduce waiting sessions and improve tail latency, not whether you improve averages.

If It Is a Plan Regression

Typical fixes for plan regressions focus on proving what changed and then restoring stability safely:

- Compare plan hash values and row-source stats.
- Gather stats (carefully).
- Stabilize with SPM/plan baselines (Chap. 9).

The goal is to make performance predictable again; optimize only after you have re-established a stable baseline.

Step 4: Verify and Prove the Improvement

A fix is not real until it is measured. This step is where you turn a plausible idea into a verified improvement:

- Capture before/after `V$SQL` per-exec metrics.
- Validate the execution plan and cardinality assumptions.
- Confirm system-level impact (CPU, I/O, waits) did not regress other work.

If you cannot show the before/after delta, treat the change as unproven and keep looking.

Step 5: Make the Fix Stick

Operationalize the improvement.

Regression Prevention Checklist

The difference between a one-time win and sustained performance is what you do *after* the fix lands:

- Add an automated test (query plan check, response-time budget, or workload replay).
- Instrument the feature (module/action) so it is observable (Chap. 14).
- Document the change and the rollback plan.
- If needed, lock in stability (plan baseline) and monitor for drift.

This is how you prevent “we fixed it last quarter” from becoming the start of the next incident.

Common Failure Modes (and How to Avoid Them)

Experienced tuners fail in predictable ways too. These traps consume the most time for the least benefit, and they usually show up when you move too fast or skip verification:

- **Tuning the wrong thing**: Optimize a query that is not on the critical path.
- **Chasing ratios**: Cache hit ratio, parse ratio, etc., without user-visible improvements.
- **Fixing one workload while breaking another**: No safety metric.
- **No restart strategy**: Batch jobs that fail require full reruns.
- **No ownership**: Fixes are not monitored and silently regress.

When you feel stuck, come back to the loop: recheck the evidence, re-localize cost, and pick a smaller change you can prove.

Summary

This closing chapter turns the material from the previous chapters into a repeatable operating procedure. The key idea is that tuning is an engineering loop: you form a hypothesis from evidence, change one thing, and then prove the outcome.

With that framing, the core lessons are:

- Use a disciplined loop: evidence → localization → smallest safe fix → verification → regression prevention.
- Separate incident triage (stabilize) from optimization (improve) so you do not take risky changes under pressure.
- Prove improvements with per-execution SQL metrics and time-series evidence. Use AWR/ASH only when the Oracle Diagnostic Pack is appropriately licensed; SQL Monitor is part of the Oracle Tuning Pack (and has Diagnostic Pack licensing implications as well). If those packs are not licensed, use alternatives such as Statspack, OS metrics, and targeted sampling queries against `V$` views.
- Operationalize fixes so they survive deployments, data growth, and upgrades (tests, baselines, dashboards, and ownership).

Quick Checklist

Use this as a safe sequence when you are on the hook to make things faster *and* keep production safe:

- Stabilize first: Stop the bleeding (blockers, runaway SQL, resource storms) and reduce load if needed.
- Localize with evidence: Identify the limiting resource and the top SQL on the critical path.

- Choose the smallest safe change (plan, index, code path, parameter) and retest under comparable conditions.
- Verify with before/after measurements (per-exec SQL metrics and time-series), and confirm you did not regress other workloads.
- Prevent regression: baseline what matters, add monitoring, document the change and rollback plan, and assign ownership.

If you cannot explain why a change helped (or why it did not), treat it as a signal to revisit the measurement, not as a reason to keep turning knobs.

Conclusion

Oracle performance work is rarely about a single magic change. More often, it is about removing friction, one constraint at a time: a bad plan caused by stale statistics, an index that does not match the real access pattern, a hot block that turns into a concurrency bottleneck, or an application that quietly issues thousands of unnecessary queries.

The practical takeaway from this book is a mindset and a workflow. Start with measurement. Read the plan. Form a hypothesis. Change one thing. Measure again. Then lock in the win with stability controls and a monitoring cadence so the same incident does not return six months later.

Optimization is a habit. Systems change, and your process has to keep up.

As you apply these techniques, aim for improvements that are not only faster but also safer: predictable plans, bounded workloads, clear baselines, and simple rollback paths. When performance is treated as a normal engineering practice instead of emergency response, the database becomes easier to run, easier to change, and easier to trust.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0

References

- **Oracle Database Performance Tuning Guide.** Oracle Corporation. https://docs.oracle.com/en/database/oracle/oracle-database/21/tgdba/index.html. Comprehensive official documentation covering tuning methodology, SQL tuning, memory management, and diagnostic tools.
- **Troubleshooting Oracle Performance.** Christian Antognini, Apress. A deep dive into execution plans, wait events, and real-world tuning strategies.
- **Optimizing Oracle Performance.** Cary Millsap and Jeff Holt, O'Reilly Media. Introduces the Method R approach to performance diagnosis and optimization.
- **Oracle SQL Tuning Pocket Reference.** Mark Gurry, O'Reilly Media. Quick reference for tuning SQL statements and understanding optimizer behavior.
- **Oracle DBMS_PARALLEL_EXECUTE Documentation.** Oracle Corporation. https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/DBMS_PARALLEL_EXECUTE.html.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0

Index

A
ACTION tag, 126
Active session
  immediate queries, 146
  and wait events, 122
Active Session History (ASH), 11–13, 16, 17, 125, 126, 133, 134
Aggregation, 42
  index-only aggregation, 42
  materialized view, 42–43
  multilevel aggregation, 43
  pre-computed aggregation, 42–43
  ROLLUP, 43
Allround Automations' PL/SQL Developer, 22–23
ANSI JOIN, 37
Application-level pagination, 109
Application logs, 9
Automatic Memory Management (AMM), 96, 97
Automatic Workload Repository (AWR), 11, 12, 16, 17, 125, 126

B
Batch processing, 76–77
  long transaction optimization, 76–77
  and parallel execution, 113–120
Bind variables, 38–39, 98, 104, 105, 107, 112, 130
Bitmap index, 62
BMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE, 80
Bounded SQL query, 108
B-Tree index, 61, 62, 68, 69
BULK COLLECT, 88, 89, 93, 114–116

C
Cache-aside pattern, 110–111
Cardinality, 16, 23, 27, 28, 150
Common table expression (CTE), 37–38
Composite index, 62–64, 67, 69
Concurrency management, 71, 76–77
Connection pools
  database session and process limits, 99, 100
  HikariCP configuration, 99
  with retry/backoff, 141
Cost-Based Optimizer (CBO), 44, 55
Covering index, 36, 42, 66
CPU pressure
  reduce parse and library cache pressure, 130
  SQL by total CPU time, 130
  symptoms, 129

D
Data Guard
  ASYNC modes, 140
  commit behavior, 139
  observe transport and apply lag, 140
  SYNC modes, 140
DBMS_SPM.CONFIGURE, 84
DBMS_SPM.DROP_SQL_PLAN_BASELINE, 83
DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE, 83

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026

S. Mishra, *Oracle Performance Tuning Essentials*, Apress Pocket Guides,
https://doi.org/10.1007/979-8-8688-3156-0

DBMS_STATS, 44, 47, 59
DBMS_XPLAN, 49, 61, 69
DBMS_XPLAN.DISPLAY_CURSOR, 51, 59
Deadlock, 71, 74–75, 78
DETERMINISTIC function, 90
Diagnostic Pack, 11, 13, 101, 125, 145
Diagnostic workflow, 10
  application logs, 9–10
  checklist, 17
  execution plan, 16
  fix implementation, 17
  log quality, 10
  moves, 10
  obvious issues, 16
  SQL, problem identification
    alternate strategy, 14–16
    ASH, 12–13
    AWR, 11, 12
    CPU, 11
    duration creep, 13–14
Direct-path insert, 118
Dynamic SQL, 103, 104

E
80/20 Rule, 1–2
Execution
  parallel, 113, 116
  per-execution cost, 123–124, 147–148
  SQL, 87, 88, 90
Execution plan stabilization
  checklist, 85–86
  database upgrades, 84–85
  plan baselines (*see* Plan baselines)
  unaccepted plans, 83
  unauthorized plan changes, 82
EXPLAIN PLAN, 19
  access paths, 28–29
  checklists, 25, 34
  common operations, 24
  components, 27–28
  creation
    Allround Automations' PL/SQL Developer, 22–23
    SQL Developer, 22
    TOAD, 23
  customer query, without index, 20
  execution plan, before/after index creation, 20–21
  importance, 20
  index
    creation, 21
    usage verification, 63–64
  join methods, 30–31
  key metrics, 23, 24
  optimizer's plan, 19–20, 27
  patterns
    cost of sorting, 33
    efficient plan, 32–33
    problematic plan, 31–32

F, G
Function-based index, 62, 66

H
HASH JOIN, 24, 30, 31
High availability features
  Data Guard, 139–140
  failover as load test scenario, 141
  RAC, 138–139
  read scaling, 140–141
  resilience patterns, 142
  RTO/RPO, 137
High-frequency SQL, 93
HikariCP connection pool configuration, 99
Histograms, 55–56, 149
History (ASH), 125, 126, 128, 134

I
Index, 39
  checklist, 69
  creation, 62–63
  database index, 61
  EXPLAIN PLAN, 63–64
  functions, indexed column, 39–40
  implicit type conversion, 40
  index-driven OLTP, 132
  issues, 3
  maintenance, 68, 69
  predicates, 41
  redundant indexes, 64–65, 69
  scenarios, 65
  techniques
  covering index, 66
    function-based index, 66
    index skip scan, 67
    invisible index, 67
    reverse key index, 68
  types, 61–62, 69
  wildcards, 41
INDEX RANGE SCAN, 21, 24, 29, 31, 32, 41, 63, 64
Invisible index, 67
I/O
  identify dominant I/O wait classes, 133

redo volume and commit behavior, 133–134
subsystem, 132
V$SYSTEM_EVENT values, 133
wait names, 133

J
Join methods, 30–31

K
Keyset pagination, 103, 109, 112

L
Latch contention, 53
Latency SLO, 137
Live diagnosis
active sessions and wait events, 122
blockers and SQL, 123
concurrency, 122
CPU pressure, 122
I/O, 122
Locking management
checklist, 78
deadlock, 74–75, 78
detection, 73–74, 78
isolation levels, 75–76
lock types, 72
prolonged blocking, 73–74
Loop elimination
FORALL for bulk DML, 89
N+1 query problem, 87, 88
in PL/SQL, 87–89
set-based alternative, 88

M
Management
concurrency management, 71, 76–77
locking management (*see* Locking management)
optimizer statistics management (*see* Optimizer statistics management)
Materialized views (MV), 42–43, 90–91
Memory
PGA, 131–132
SGA, 130–131
TEMP exhaustion, 132
Memory configuration tuning
PGA, 95, 97
SGA, 95–97
Modern pagination syntax, 91
MODULE tag, 126
Multiversion concurrency, 71
Multiversion Read Consistency (MVCC), 71, 72, 77, 78

N
NESTED LOOPS, 24, 30–33
N+1 query problem, 87, 88, 105, 112

O
Object relational mapping (ORM) query optimization
interface based projection, 106
JOIN FETCH pattern, 106
N+1 queries, 105
projection query, 106
Offset-based pagination, 109
OFFSET/FETCH pagination, 92
Operational guardrails, 119
Optimization, 153
connection pooling, 99–100
ORM query optimization, 105–106
Optimizer statistics management
cost-based optimizer, 55
histograms, 55–56
statistics
locking, 56
restoring, 56–57
table and index, 55
Oracle
central statement repository (*see* Shared pool)
execution plan, 23, 24, 28
full table scan, 20, 28, 32, 41, 63, 64
performance issues, 10
performance tuning (*see* Performance tuning)
query rewrite, 43
SQL, 19
Oracle Database Resource Manager, 134
Oracle performance, 1
checklist, 7
diagnostic workflow (*see* Diagnostic workflow)
80/20 Rule, 1–2
fixes, problems, 6
issues, 3, 6
application layer, 5
data, 4
design, 4
indexing issues, 3
infrastructure, 4

schema, 4
statistics, 4
lifecycle, 2–3
troublemakers, 5–6

P
Pagination
application-level, 109
efficient pagination (Oracle 12c and later), 91–92
keyset pagination, 103, 109, 112
legacy pagination (pre-12c), 92
OFFSET/FETCH pagination, 92
ROWNUM-based pagination, 92
Parallelism
chunked parallelism, 117–118
DBMS_PARALLEL_EXECUTE, 117
parallel DML, 117
parallel query, 117
as scheduling decision, 116
Parameterized queries, 103–105
Partition-friendly batch design, 118–119
Performance tuning
common failure modes, 151
concurrency-heavy systems, 149
CPU-heavy work, 149
immediate queries, 146–147
I/O-heavy work, 149
plan regressions, 150
regression prevention checklist, 150
three questions, 146
top SQL by total and per-execution cost, 147–148
tuning loop, 145–146
verify and prove improvement, 150
Plan
PLAN_HASH_VALUE, 124
regressions, 119, 150
SQL-level evidence, 123–124
Plan baselines
automatic plan capture, 80
definition, 79
dropping, 83–84
known-good plan, 80–81, 85
management, 81–82
monitoring, 81–82
status flags, 80, 85
PLAN_HASH_VALUE, 52, 59
PLAN_NAME, 81
PL/SQL bulk processing
BULK COLLECT, 115
use FORALL for DML, 115–116
PL/SQL-to-SQL engine context switching, 87
Proactive optimization
aggregation, 42
index-only aggregation, 42
materialized view, 42–43
multilevel aggregation, 43
pre-computed aggregation, 42–43
ROLLUP, 43
checklist, 46
framework, 35–36, 45–46
index, 39
functions, indexed column, 39–40
implicit type conversion, 40
predicates, 41
wildcards, 41
set operations, 43–44
SQL
ANSI JOIN, 37
bind variables, 38–39
SELECT *, 35–36
subqueries, JOINs/CTEs, 37–38
statistics maintaince, 44–45
UNION ALL and UNION, 43–44
Program Global Area (PGA)
configure and monitor PGA usage, 97
and SGA sizing, 95
sort/hash work areas, 131–132
work area efficiency, 131

Q
Query frequency, 87

R
RAC design
avoid hot blocks, 139
global cache, 138
global cache statistics by instance, 139
identify global cache waits, 138
partition-aware access, 139
performance, 139
service placement, 139
system-wide cluster wait events, 138
Reactive tuning, 35
Read and write routing, 111
Read scaling, 110, 140–141
Recovery Point Objective (RPO), 137
Recovery Time Objective (RTO), 137
Redo log I/O configuration, 100
Redundant indexes, 64–65
REFRESH FAST ON COMMIT, 91
Result volume, 87
Reverse key index, 68
Row-by-row processing, 113–114

Row Lock, 72
ROWNUM-based pagination, 92

S
SELECT *, 35–36, 88, 92, 104, 108, 109, 125
Set-based SQL, 113
Set operations, 43–44
Shared pool, 47
  ASH queries, 53
  buffer cache, SGA, 130–131
  bulk processing, 116
  cached plans, 51
  DBMS_XPLAN.DISPLAY_CURSOR, 51
  diagnose shared pool pressure, 98
  high-impact queries, 48, 50
  library cache, 98
  literal and bound SQL, 105
  plan changes, 52
  PLAN_HASH_VALUE, 52
  real-time wait analysis, 52–53
SGA components, 96
SQL_ID, 51–52
  V$ACTIVE_SESSION_HISTORY (ASH), 52–53
  V$SQL, 48, 49
  V$SQL_PLAN, 50
  Wait Events, 53
SORT MERGE JOIN, 30, 31
SQL
  ANSI JOIN, 37
  bind variables, 38–39
  lifecycle, 1–2
  optimizer, 41
  plan baselines (*see* Plan baselines)
  problem identification, 11–12
  ASH, 12–13
  AWR, 11, 12
  CPU, 11
  diagnostic strategy, 14–16
  duration creep, 13–14
  queries, 5
SELECT *, 35–36
SQL_ID, 51–52
  subqueries, JOINs/CTEs, 37–38
SQL Developer, 22
SQL discovery
  checklist, 59
  end-to-end correlation, 57–58
  optimizer statistics management (*see* Optimizer statistics management)
  shared pool (*see* Shared pool)
  tools, 59
  workflow, 58–59
SQL_HANDLE, 81
SQL_ID, 51–52
SQL-level evidence
  confirm plan identity and per-execution cost, 123–124
  PLAN_HASH_VALU, 124
SQL monitoring for long-running statements, 124
SQL Management Base (SMB), 79, 83, 85
SQL Monitor, 121, 124, 128, 145
SQL Performance Analyzer (SPA), 85
SQL Plan Management (SPM), 78, 84, 85
Statistics
  and data, 4
  end-to-end correlation, 57–58
  global cache, 138–139
  index statistics, 55
  locking, 56
  maintenance, 44–45
  restoring, 56–57
  table statistics, 55
System Global Area (SGA), 95
  buffer cache and shared pool, 130–131
  components and parameters, 96
  multiple pools, 96
  physical reads, 131
  review and adjust SGA configuration, 96

T
Targeted SQL trace, 127
TEMP exhaustion, 132
TOAD, 23

U
Unbounded fetching, 107
Undo/redo pressure, 116

V
V$ACTIVE_SESSION_HISTORY, 52–53, 59
V$DB_CACHE_ADVICE, 98
V$SESSION_LONGOPS, 73–74
V$SQL, 48, 49, 59, 93, 124, 127, 130, 148, 150
V$SQL_PLAN, 50

W, X, Y, Z
Wait events, 122, 133, 138