BANDLEI

A BANDLEI FIELD GUIDE

# Rein In Your Database

## A practical tuning guide for PostgreSQL, MySQL, SQL Server and Oracle

From the first look at a slow system to the monitoring that keeps it fast: methods, queries, version notes and checklists for DBAs, developers, performance engineers and data architects.

INSIDE THIS GUIDE


01 How to start
02 Set up your toolkit
03 Build a data profile
04 Choose the right platform
05 Decide what to look for
06 The statement tuning procedure
07 Joins and WHERE clauses
08 Indexing
09 Procedure code
10 Materialized views and partitioning
11 Settings
12 Baselining
13 Ongoing observability
14 How ReinDB fits in



Version 1.3 · November 2026
bandlei.com · reindb.com


ReinDB

# Contents


**About this guide** ..... 4
- Who it's for ..... 4
- What it covers ..... 4
- Conventions ..... 4
- Trademarks ..... 5

**1 How to start** ..... 6
- 1.1 Start with a symptom and a target ..... 6
- 1.2 The tuning loop ..... 6
- 1.3 Ground rules ..... 7
- 1.4 Ask "what changed?" ..... 7
- 1.5 Your first hour on an unfamiliar database ..... 8

**2 Set up your toolkit** ..... 10
- 2.1 What a complete toolkit covers ..... 10
- 2.2 What the engine gives you for free ..... 10
- 2.3 Know what you're licensed to use ..... 11
- 2.4 Turn on the instrumentation you'll need ..... 13
- 2.5 Monitoring accounts with least privilege ..... 14
- 2.6 Cloud and managed databases: what changes ..... 15
- 2.7 Beyond the built-ins: choosing a monitoring tool ..... 18

**3 Build a data profile** ..... 19
- 3.1 What goes in the profile ..... 19
- 3.2 Sizes and activity ..... 19
- 3.3 Column shape: cardinality, skew and nulls ..... 21
- 3.4 Growth and temperature ..... 21
- 3.5 The workload profile ..... 22
- 3.6 A one-page profile template ..... 23

**4 Choose the right platform for the data** ..... 24
- 4.1 The familiar-platform trap ..... 24
- 4.2 Does it all need to be relational? ..... 25
- 4.3 Tier inside the engine first ..... 28
- 4.4 Hybrid data tiers: when part of the data leaves ..... 28
- 4.5 Keeping a hybrid tier cohesive ..... 29
- 4.6 Before you move a table out ..... 31

**5 Decide what to look for** ..... 32
- 5.1 Slow queries or overall execution time? Use three lenses ..... 32
- 5.2 Four proven methods ..... 32
- 5.3 What comes before statement-level tuning ..... 33
- 5.4 Collecting the evidence: counters, deltas and samples ..... 34
- 5.5 Where does the time go? ..... 35
- 5.6 From symptom to strategy ..... 38
- 5.7 The same waits under different names ..... 39

**6 The statement tuning procedure** ..... 40
- 6.1 Ten steps ..... 40

6.2 Getting the actual plan ..... 41
6.3 Reading the plan ..... 43
6.4 Cardinality: the root of most bad plans ..... 44
6.5 Plan stability tools: a stopgap, not a cure ..... 46

**7 Tuning joins and WHERE clauses ..... 48**

7.1 Keep predicates sargable ..... 48
7.2 Data type mismatches and implicit conversions ..... 49
7.3 Patterns that hide rows from indexes ..... 49
7.4 Joins: get the shape right and the method follows ..... 50
7.5 Ask for less ..... 51
7.6 Bind variables and parameterization ..... 53
7.7 Batch large data changes ..... 53

**8 Indexing: when, where, which type, and what goes wrong ..... 55**

8.1 When an index helps, and when it doesn't ..... 55
8.2 Where: designing composite indexes ..... 55
8.3 Index types and where each fits ..... 56
8.4 Finding index candidates ..... 58
8.5 Finding unused and duplicate indexes ..... 59
8.6 Build and maintain indexes without hurting production ..... 60
8.7 Common indexing pitfalls ..... 61

**9 Optimizing procedure code ..... 62**

9.1 The patterns that cost the most ..... 62
9.2 Set-based rewrites ..... 62
9.3 Dynamic SQL, done safely ..... 64
9.4 Functions called from SQL ..... 65
9.5 Parameter sensitivity in procedures ..... 65
9.6 Errors, logging and transactions ..... 66
9.7 Instrument your code so time can be attributed ..... 67
9.8 Profile procedure code ..... 67

**10 Materialized views and partitioning ..... 69**

10.1 Materialized views: precomputed answers ..... 69
10.2 Partitioning: one table, many pieces ..... 71
10.3 Using them together ..... 75
10.4 Which tool for which problem? ..... 75

**11 Settings: match the workload, then keep it there ..... 76**

11.1 Principles ..... 76
11.2 The settings that matter, by category ..... 76
11.3 Starting points by engine ..... 77
11.4 Scoped overrides for the outliers ..... 81
11.5 Keeping it there ..... 81

**12 Baselining: our approach and why it matters ..... 84**

12.1 Why baselines matter ..... 84
12.2 Kinds of baseline ..... 84
12.3 What to baseline ..... 84
12.4 Our approach in ReinDB ..... 85
12.5 Build a simple seasonal baseline yourself ..... 85
12.6 Baselines around changes ..... 87
12.7 Baseline pitfalls ..... 87

**13 Ongoing observability ..... 88**
13.1 What to track ..... 88
13.2 The shape of a monitoring environment ..... 89
13.3 Exporting to Grafana ..... 90
13.4 Designing dashboards people actually use ..... 91
13.5 Alerts: what to set and how to manage them ..... 92
13.6 Keeping it optimal: a review rhythm ..... 93

**14 How ReinDB fits in ..... 95**
14.1 ReinDB at each step of this guide ..... 95
14.2 A quick tour ..... 96
14.3 Getting started ..... 97
14.4 A week with ReinDB ..... 98
14.5 Built for strict security boundaries ..... 98
14.6 Plans ..... 99

**Appendix A. Checklists ..... 100**
A.1 First hour on an unfamiliar database ..... 100
A.2 Statement tuning ..... 100
A.3 Index change ..... 100
A.4 Settings change ..... 101
A.5 Monitoring starter set ..... 101
A.6 Running on a managed service ..... 101

**Appendix B. Version notes ..... 102**
B.1 Oracle 11g against 19c ..... 102
B.2 SQL Server ..... 103
B.3 PostgreSQL ..... 103
B.4 MySQL ..... 104

**Appendix C. Glossary ..... 105**

**Appendix D. Sources and further reading ..... 106**

# About this guide

This is a field guide to making relational databases fast and keeping them that way. It follows the path a working DBA takes: from the first look at a slow system, through the evidence, the fixes and the settings, to the monitoring that stops the same problem from coming back.

## Who it's for

DBAs, performance engineers, data architects, and developers who own database code. It assumes you can read SQL and an execution plan. It does not assume you know all four engines.

## What it covers

PostgreSQL, MySQL, Microsoft SQL Server and Oracle Database, self-managed or as managed cloud services. The main text is engine-neutral: it explains each method in terms that apply to any relational database. Engine differences sit in labeled notes and comparison tables, so you can read straight through, or jump to the notes for the engine you run.

| Part | Chapters | What you get |
|---|---|---|
| Strategy | 1 to 5 | How to start, the toolkit and what changes on managed cloud services, the data profile, platform choices, and what to look at first |
| Hands-on tuning | 6 to 11 | The statement tuning procedure, joins and predicates, indexing, procedure code, materialized views and partitioning, settings |
| Staying fast | 12 to 13 | Baselines and ongoing observability, including Grafana and alerting |
| ReinDB | 14 | How ReinDB fits each step, and how to put it to work |
| Reference | Appendices | Checklists, version notes, glossary and sources |

## Conventions

- **Versions** are noted in parentheses where a feature depends on them, for example (PostgreSQL 13+) or (Oracle 12c+). Appendix B collects the differences that matter most, including Oracle 11g against 19c.
- **Examples** use a small, familiar schema: customers, orders and order items. They are illustrations, not drop-in scripts. Test every change outside production first, with production-like data.
- **Oracle licensing.** The Oracle path in this guide uses V$ views and free tools only. Features that need the Diagnostics Pack or the Tuning Pack are marked wherever they come up. Confirm against Oracle's licensing documentation for your release and edition.
- **Engine notes.** Slate boxes labeled ORACLE, SQL SERVER, POSTGRESQL or MYSQL hold the syntax, views, settings and version details for one engine; ENGINE NOTES boxes cover several at once. Tables with a slate header compare the engines side by side.

- **Cloud notes.** Teal boxes cover managed services such as Amazon RDS and Aurora, Azure SQL, Google Cloud SQL and Oracle's own cloud services. Section 2.6 sums up what changes in the cloud.
- **Other callouts.** Green boxes show where ReinDB helps. Amber boxes are warnings and common traps. Grey boxes are side notes.

> WATCH OUT **Before you change anything**
>
> Every technique here can hurt as well as help. Make one change at a time, keep a rollback script, and verify the result against a baseline taken at the same time of day.

## Trademarks

Oracle and MySQL are registered trademarks of Oracle and/or its affiliates. Microsoft and SQL Server are trademarks of the Microsoft group of companies. PostgreSQL is a registered trademark of the PostgreSQL Community Association. Other names may be trademarks of their respective owners. ReinDB is a trademark of Bandlei, Inc.

© 2026 Bandlei, Inc., Evergreen, Colorado.

# 1 How to start

Most tuning goes wrong before anyone touches a query. Someone picks a suspect, applies a favorite fix, and declares victory because the next run was faster (often only because the cache was warm by then). Good tuning is calmer than that. It is a loop you can repeat, explain and defend.

## 1.1 Start with a symptom and a target

Write the problem down in the words of the people who feel it, then turn it into a number.

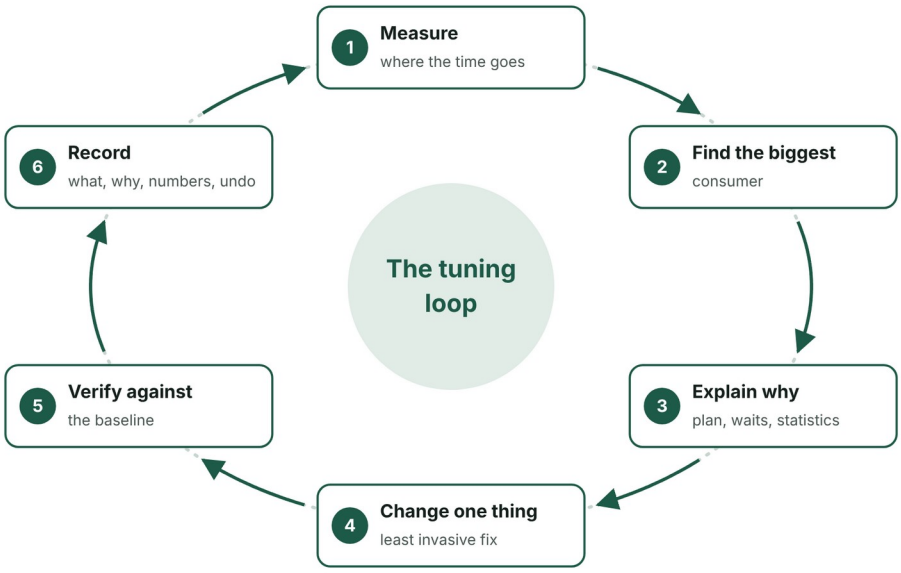
| Symptom as reported | Measurable target |
|---|---|
| "Order search is slow in the morning." | 95th percentile of the search statement under 300 ms between 08:00 and 11:00 |
| "The nightly load keeps missing its window." | Load finishes by 05:30 on 95% of nights |
| "The server has been maxed out since Tuesday." | Peak CPU under 70%, and load back to its usual level for each hour |
| "Reports time out at month end." | The month-end report set finishes in under 20 minutes with no timeouts |

A target tells you when to stop. Without one, tuning never ends, and every extra change carries risk without a clear payoff. Note the scope as well: one statement, one feature, one database, or the whole estate. The scope decides which data you need and who has to be involved.

## 1.2 The tuning loop

1. **Measure.** Collect how long things take and where the time goes, over a period that includes the problem.
2. **Find the biggest consumer.** Rank by the time it costs, not by how bad it looks.
3. **Explain why.** Read the plan, the waits and the statistics until you can say in one sentence why it is slow.
4. **Change one thing.** Choose the least invasive fix that addresses the cause.
5. **Verify against the baseline.** Compare at the same time of day, under the same kind of workload, with what was normal before.
6. **Record.** Write down what you changed, why, the before and after numbers, and how to undo it.

Then go around again with the next biggest consumer, until you reach the target.




*The tuning loop: measure, find the biggest consumer, explain why, change one thing, verify against the baseline, record.*

## 1.3 Ground rules

- **Tune for time, not ratios.** A buffer cache hit ratio of 99.9% says nothing about whether users are waiting. A statement doing ten million logical reads per execution has a wonderful hit ratio and is still the problem. Time, and the work behind it, is what users and budgets feel.
- **Fix the biggest consumer first.** If a statement accounts for 2% of database time, making it infinitely fast saves at most 2%.
- **Change one thing at a time.** With two changes you can't tell which one helped, or which one hurt.
- **Compare like with like.** Same hour, same kind of day, same data volume, warm cache against warm cache, representative parameter values.
- **Make every change reversible.** Write the rollback before you deploy.
- **Test with production-like data.** Plans depend on data volume and distribution. A query that is fast on 10,000 test rows can choose a completely different plan on 100 million.
- **Measure work, not just elapsed time, when you test.** Logical reads per execution, rows examined per row returned and temp usage are stable from run to run. Elapsed time swings with cache state and concurrency.
- **Know when to stop.** When the target is met, stop. The last 10% of improvement often carries most of the risk.

## 1.4 Ask "what changed?"

Databases rarely get slower for no reason. Before you measure anything, ask what changed in the days before the problem started.

- **Code:** a deployment, a new report, a new ORM version, a feature flag switched on.
- **Data:** growth that crossed a threshold, a bulk load, a purge that stopped running, a new customer with ten times the usual data.
- **Statistics:** gathered at a bad moment, not gathered after a big load, or locked long ago.
- **Configuration:** a parameter change, a new compatibility level, a bigger connection pool.
- **Software:** an upgrade or patch that changed optimizer behavior.
- **Infrastructure:** a new instance class, storage type or volume size; burst credits used up; a noisy neighbor; a failover to different hardware.
- **Plans:** a new execution plan for an existing statement, often after a statistics refresh or a restart.
- **Schedules:** batch jobs, backups or maintenance moved into busy hours.
- **Usage:** a marketing campaign, month end, a new integration calling one API in a tight loop.

Most sudden regressions trace back to one of four causes: a plan change, data growth, contention, or the infrastructure underneath. Keep those four in mind while you measure.

## 1.5 Your first hour on an unfamiliar database

When you inherit a database, or get paged for one you have never seen, work through this list before you change anything.

| Check | Why it matters | Where to look |
|---|---|---|
| Version, patch level, edition, platform | Features, known bugs and licensing limits | Version functions and views; cloud console |
| Time since the last restart | Cumulative counters and caches only cover this period | The engine's instance status views |
| Size and growth | Capacity risk and maintenance cost | Data dictionary sizes; your size history |
| Top waits and the time model | The class of problem: CPU, I/O, locks or commits | Chapter 5 |
| Top statements by total time | Where the database actually spends its time | The engine's statement statistics (Chapter 5) |
| Blocking and long transactions right now | Acute contention, often the real cause of "slowness" | Live session and lock views |
| Storage headroom: data, log, WAL or redo, temp, archive destination | A full disk is an outage, not a slowdown | File and tablespace usage; OS and cloud metrics |
| Last good backup and replica lag | Don't tune a database you can't restore | Backup catalog; replication views |
| Statistics freshness and maintenance jobs | Stale statistics and skipped maintenance cause many bad plans | Statistics views; job history |
| Recent changes | The fastest path to the cause | Deploy log, change tickets, parameter history |
| Settings far from defaults | Yesterday's fix can be today's problem | Non-default settings queries in Chapter 11 |

ENGINE NOTES **Where the first-hour facts live**

- **PostgreSQL:** `version()` and `pg_postmaster_start_time()`, `pg_stat_statements` for statements, `pg_stat_activity` and `pg_locks` for live activity, `pg_database_size()` and `pg_total_relation_size()` for sizes.
- **MySQL:** `VERSION()` and the `Uptime` status variable, `performance_schema` statement digests, `performance_schema.threads` and `sys.innodb_lock_waits` for live activity, `information_schema.TABLES` for sizes.
- **SQL Server:** `sys.dm_os_sys_info` (`sqlserver_start_time`), Query Store or `sys.dm_exec_query_stats` for statements, `sys.dm_exec_requests` for live activity, `sys.database_files` for sizes.
- **Oracle:** `V$INSTANCE` (version, `STARTUP_TIME`), `V$SQLSTATS` for statements, `V$SESSION` and `V$LOCK` for live activity, `DBA_SEGMENTS` for sizes.

REINDB **ReinDB shortcut**

The Now page covers most of this list on one screen: open flags, load against what is usual for this hour, the most drifted statement, the longest lock wait, and storage used with the days left until full. The health check report adds version and patch level, waits, the busiest tables, stored code findings, settings and a fix script.

# 2 Set up your toolkit

You can't tune what you can't see, and you can't prove a fix without history. Set the toolkit up before the next incident, not during it.

## 2.1 What a complete toolkit covers

| Capability | What it gives you | Typical sources |
|---|---|---|
| History | Low-overhead measurements of statements, waits and resources, kept for weeks | The engine's statement statistics and history stores, snapshot repositories, a monitoring product |
| Live view | Who is running what right now, who blocks whom, what sessions wait on | Session, lock and transaction views |
| Plan inspection | Estimated and actual plans with real row counts | The engine's explain facilities and actual (runtime) plans |
| Platform metrics | CPU, memory, storage latency and throughput, network, cloud limits | OS tools, node_exporter, Amazon CloudWatch, Azure Monitor, Google Cloud Monitoring |
| Change record | Deployments, parameter changes and schema changes, with timestamps | CI/CD logs, change tickets, dashboard annotations |
| Alerting and reporting | Notifications for real deviations, and reports for regular review | Alert manager, monitoring product, scheduled reports |

No single tool covers all six well. Aim for full coverage with as little overlap as possible, and make sure every source uses the same clock (UTC is easiest) so you can line events up during an incident.

## 2.2 What the engine gives you for free

Every major engine ships with serious instrumentation, and it comes in the same five families everywhere:

- **Statement statistics:** cumulative calls, time, rows and I/O for each normalized statement, since the last restart or reset.
- **Live activity:** the sessions, what each one is running and waiting on, and who blocks whom.
- **Plans:** the estimated plan, and the actual plan with real row counts and timings.
- **Waits and time accounting:** where the time goes, by wait event or class, for the whole instance and for each session.
- **Logs and traces:** slow statements, lock waits, deadlocks and temp spills, captured one execution at a time.

Learn these first: commercial tools read the same sources. The table maps the families to each engine.

| Engine | Statement history | Live activity | Plans | Waits and extras |
|---|---|---|---|---|
| PostgreSQL | `pg_stat_statements` extension | `pg_stat_activity`, `pg_locks`, `pg_blocking_pids()` | `EXPLAIN (ANALYZE, BUFFERS)`, `auto_explain` | `wait_event` in `pg_stat_activity` (point in time); `pg_wait_sampling` extension for history; `pg_stat_io` (16+); table and index statistics views |
| MySQL | `performance_schema` statement digests; `sys` schema views | `performance_schema.threads`, `SHOW PROCESSLIST`, `sys.innodb_lock_waits` | `EXPLAIN`, `EXPLAIN ANALYZE` (8.0.18+), optimizer trace | `performance_schema` wait events (some instruments are off by default); slow query log; `pt-query-digest` from Percona Toolkit |
| SQL Server | Query Store (2016+); `sys.dm_exec_query_stats` | `sys.dm_exec_requests`, `sys.dm_exec_sessions`, `sys.dm_os_waiting_tasks` | Actual execution plans; plans in Query Store | `sys.dm_os_wait_stats`; per-query waits in Query Store (2017+); Extended Events (the default `system_health` session records deadlocks) |
| Oracle, no packs | `V$SQL`, `V$SQLSTATS` | `V$SESSION`, `V$LOCK`, `V$TRANSACTION` | `DBMS_XPLAN.DISPLAY_CURSOR`, `V$SQL_PLAN_STATISTICS_ALL` | `V$SYSTEM_EVENT`, `V$SESSION_EVENT`, `V$SYS_TIME_MODEL`; SQL trace (event 10046) with TKPROF; Statspack for history |

## 2.3 Know what you're licensed to use

Some performance features cost extra, depend on the edition, or exist only on certain platforms. On some engines, querying a licensed view counts as use of the feature, even from a script or a monitoring tool. Before you build the toolkit, list what your editions and contracts actually include, and make everything else hard to use by accident.

> ORACLE **Diagnostics Pack and Tuning Pack**
>
> Some of Oracle's best-known performance features are licensed separately. The **Diagnostics Pack** covers AWR (the `DBA_HIST_*` views), Active Session History (`V$ACTIVE_SESSION_HISTORY`) and ADDM. The **Tuning Pack**, which requires the Diagnostics Pack, covers the SQL Tuning Advisor, the SQL Access Advisor, SQL profiles and Real-Time SQL Monitoring (`V$SQL_MONITOR`). Both packs require Enterprise Edition, and querying the views counts as use, even from a script.

If you are not licensed, set `CONTROL_MANAGEMENT_PACK_ACCESS = NONE` (11g and later) to mark both packs as unavailable. It doesn't stop anyone from querying the views, so keep pack views out of scripts and grants as well, and use the free path described throughout this guide: V$ views, `DBMS_XPLAN.DISPLAY_CURSOR`, SQL trace with TKPROF, and Statspack for history. Always confirm against Oracle's Database Licensing Information for your release.

ORACLE **Statspack: free history on every edition**

Statspack has shipped with Oracle for decades and still does, on every edition. It is not as rich as AWR, but it gives you hourly snapshots and a readable report at no extra cost:

```
-- As SYSDBA, once: creates the PERFSTAT schema (prompts for a password and tablespaces)
@?/rdbms/admin/spcreate.sql

-- Take a snapshot (schedule this hourly with DBMS_SCHEDULER).
-- Level 7 adds segment statistics; the default level 5 covers SQL statements.
EXEC PERFSTAT.STATSPACK.SNAP(i_snap_level => 7);

-- Report between two snapshots
@?/rdbms/admin/spreport.sql
```

SQL SERVER **Editions**

Since SQL Server 2016 SP1, most performance features are in every edition, including Query Store, partitioning, data compression and columnstore. Some stay in Enterprise Edition only: online index operations, automatic plan correction, and the batch-mode adaptive joins and memory grant feedback. Standard Edition is also capped on cores and memory. SQL Server 2025 raises that cap to 32 cores and 256 GB and brings Resource Governor to Standard Edition. Check "Editions and supported features" for your version.

MYSQL **Community and Enterprise**

Community Edition includes everything this guide relies on: `performance_schema`, the `sys` schema, `EXPLAIN ANALYZE`, the optimizer trace and the slow query log. The thread pool plugin, MySQL Enterprise Monitor and the audit and firewall plugins need Enterprise Edition. Percona Server and MariaDB ship their own thread pool.

POSTGRESQL **No editions, but check the extensions**

PostgreSQL has no paid editions. What varies are the extensions. `pg_stat_statements` ships with PostgreSQL's contrib modules, while others, such as `pg_wait_sampling` and `pg_hint_plan`, are installed separately, and managed services offer only their own approved list.

## 2.4 Turn on the instrumentation you'll need

Several of the most useful sources are off by default, or need a restart to enable. Turn them on during a quiet change window, not in the middle of an incident. Whatever the engine, you want the same things switched on: statement statistics sized for your workload, I/O timing, logs of slow statements, lock waits and temp spills, a history of statements and plans that outlives the cache, and application names on every connection.

| Engine | Setting | Why | Cost and notes |
|---|---|---|---|
| PostgreSQL | `shared_preload_libraries = 'pg_stat_statements'`, then `CREATE EXTENSION pg_stat_statements` | Statement history | Needs a restart; small overhead |
| PostgreSQL | `track_io_timing = on` | I/O time per statement and per table | Check timer overhead with `pg_test_timing`; usually low on modern hardware |
| PostgreSQL | `log_lock_waits = on`, `log_temp_files = 0`, `log_min_duration_statement` or `auto_explain.log_min_duration` | Evidence of lock waits, temp spills and slow statements | Log volume; start with a threshold such as 1 s |
| PostgreSQL | `track_functions = 'pl'`, `pg_stat_statements.track = 'all'` | Visibility into functions and the statements inside them | Small overhead |
| MySQL | `performance_schema = ON` (the default) with statement digest consumers enabled (the default) | Statement history | Fixed memory; for very varied workloads check `performance_schema_digests_size` |
| MySQL | `slow_query_log = ON`, `long_query_time = 1`, `log_slow_extra = ON` (8.0.14+) | Evidence for individual executions | Log volume |
| SQL Server | Query Store `ON` with `OPERATION_MODE = READ_WRITE` and `QUERY_CAPTURE_MODE = AUTO` | Statement and plan history that survives restarts, and plan forcing | On by default for new databases in SQL Server 2022; size `MAX_STORAGE_SIZE_MB` so it never flips to read-only |
| SQL Server | `blocked process threshold (s)` of 10 to 20, plus an Extended Events session for `blocked_process_report` | Evidence of blocking after the fact | Deadlocks are already in `system_health` |
| Oracle | `STATISTICS_LEVEL = TYPICAL` (the default) | Time model, advisories, timed statistics | Use `ALL` only per session while diagnosing |
| Oracle | Hourly Statspack snapshots, if you don't license the Diagnostics Pack | History | Purge old snapshots regularly |
| All | Module and application names set by the application | Lets every tool attribute time to a feature or service | Free; see Chapter 9 |

CLOUD **Changing these on a managed service**

On Amazon RDS and Aurora you change them in the DB parameter group (or the cluster parameter group), on Azure in the server parameters, and on Google Cloud SQL as database flags. Static parameters need a reboot, the provider fixes some values, and a few are already on: Query Store is on by default in Azure SQL Database, for example.

## 2.5 Monitoring accounts with least privilege

Monitoring needs to read performance views, not business data. Give it exactly that: a dedicated login, the engine's built-in monitoring role or a short list of view grants, no write access, and no access to application tables. Keep its password in a secrets store and rotate it like any other service credential.

POSTGRESQL **The pg_monitor role (10+)**

```
CREATE ROLE monitor LOGIN PASSWORD 'use-a-long-passphrase';
GRANT pg_monitor TO monitor;
```

`pg_monitor` reads every statistics view, including other sessions' statement text in `pg_stat_activity` and `pg_stat_statements`.

MYSQL **Process, replication and performance_schema (8.0)**

```
CREATE USER 'monitor'@'10.0.%' IDENTIFIED BY 'use-a-long-passphrase';
GRANT PROCESS, REPLICATION CLIENT ON *.* TO 'monitor'@'10.0.%';
GRANT SELECT ON performance_schema.* TO 'monitor'@'10.0.%';
```

The `sys` schema views also need `SELECT` and `EXECUTE` on `sys`.

SQL SERVER **Server state and database state**

```
CREATE LOGIN monitor WITH PASSWORD = 'use-a-long-passphrase';
GRANT VIEW SERVER STATE TO monitor;
-- In each database you monitor (Query Store views need database-level permission):
CREATE USER monitor FOR LOGIN monitor;
GRANT VIEW DATABASE STATE TO monitor;
```

SQL Server 2022 adds narrower permissions (`VIEW SERVER PERFORMANCE STATE`, `VIEW DATABASE PERFORMANCE STATE`); check each view's documented permission before switching.

ORACLE **Grant the V$ views you need**

```
-- Grants go on the V_$ views; the V$ names are public synonyms for them
CREATE USER monitor IDENTIFIED BY "use-a-long-passphrase";
GRANT CREATE SESSION TO monitor;
GRANT SELECT ON v_$session TO monitor;
GRANT SELECT ON v_$sqlstats TO monitor;
GRANT SELECT ON v_$system_event TO monitor;
GRANT SELECT ON v_$sys_time_model TO monitor;
```

`SELECT_CATALOG_ROLE` is the broad alternative. Avoid `SELECT ANY TABLE` and the DBA role for monitoring logins.

In a multitenant database (12c+), a local user in a PDB sees only that PDB. A common user in the root sees every container once you set `CONTAINER_DATA = ALL` for it.

CLOUD **Monitoring grants on managed services**

- **Amazon RDS for Oracle:** there is no SYSDBA, so grant V$ access through the provided procedure, for example `EXEC rdsadmin.rdsadmin_util.grant_sys_object(p_obj_name => 'V_$SESSION', p_grantee => 'MONITOR', p_privilege => 'SELECT');`
- **Azure SQL Database:** server-level permissions come only through server roles. Add the login to `##MS_ServerStateReader##`, or grant `VIEW DATABASE STATE` in each database.
- **Amazon RDS for SQL Server:** the master user holds `VIEW SERVER STATE` with the grant option, so it can grant it to a monitoring login.
- **Managed PostgreSQL and MySQL:** the same grants work, run by the service's admin user. On PostgreSQL that is a member of the provider's admin role, such as `rds_superuser`, `cloudsqlsuperuser` or `azure_pg_admin`.

REINDB **ReinDB and least privilege**

ReinDB ships a grant script for each engine with the smallest set of grants it needs. It reads, never writes. On Oracle it uses V$ views only and never queries pack-licensed views unless you tell it the database is licensed for the Diagnostics Pack, which is off by default.

## 2.6 Cloud and managed databases: what changes

Managed services run the same engines, so the method in this guide holds. What changes is your reach. You trade host access and some control for automation, and you inherit the provider's limits and defaults. Expect these differences:

- **No host access.** There is no operating system shell and no full superuser (SYSDBA, sysadmin or a PostgreSQL superuser). Logs and trace files come through the provider's console, API or helper procedures.

- **Settings through the provider.** Parameters change in parameter groups, flags or server parameters. The defaults are the provider's, some values are fixed, and static ones need a reboot (Chapter 11).
- **Features from a menu.** Only approved extensions, plugins and options are available, and some only through an option group or a higher tier.
- **Platform limits.** Storage throughput and IOPS are provisioned or tied to the instance size, burst credits can run out, and the instance class caps network and log throughput. Rule these out before you tune a statement (Chapter 5).
- **Provider monitoring.** Amazon CloudWatch Database Insights, Azure's Query Performance Insight and Cloud SQL Query Insights read the same engine statistics and keep them longer. Know what each tier keeps, for how long, and what it costs.
- **License included or brought.** The instance price either includes the license or you bring your own, and that decides which editions and paid features you may use.
- **Provider maintenance.** Minor version upgrades and maintenance windows can change plans. Annotate them like any other change (Chapter 12).
- **Different storage underneath.** Some services replace the engine's storage layer (Amazon Aurora and Google AlloyDB, for example), so advice about checkpoints, flushing and log sizing may not apply.

CLOUD **Oracle in the cloud: the service decides what you may use**

Oracle runs in every major cloud, but what you're licensed for and what you can reach depend on the service more than for any other engine.

| Service | Packs and licensing | Access |
|---|---|---|
| Amazon RDS for Oracle | License Included covers Standard Edition 2 only, with no packs. Enterprise Edition needs Bring Your Own License; the packs are available only if you own them, and a Multi-AZ standby must be licensed too | No SYSDBA or host access. V$ grants, trace files and AWR reports (packs required) go through `rdsadmin` procedures; Statspack is installed through an option group |
| Amazon RDS Custom for Oracle | Bring Your Own License | Host and SYSDBA access, with AWS still running backups and automation |
| OCI Base Database Service | The license-included Enterprise Edition tiers (Enterprise, High Performance, Extreme Performance) include the Diagnostics and Tuning Packs, and the two higher tiers add Partitioning and Advanced Compression; Standard Edition has no packs | SYSDBA and host access on your DB system |
| Exadata Database Service, including Oracle Database@Azure, @Google Cloud and @AWS | Licensed as Exadata, so the Exadata-only features below are allowed | SYSDBA and VM access |
| Autonomous Database | Diagnostic and tuning features are part of the service (Performance Hub, SQL monitoring) | No SYSDBA or host access; Oracle manages most settings |

Four 19c features are licensed only on Exadata: automatic indexing, SQL Quarantine, real-time statistics and high-frequency automatic statistics collection. You can use them on Exadata Database Service, Oracle Database@Azure, Google Cloud and AWS, and Autonomous Database, but not on Base Database Service, RDS or your own non-Exadata servers.

On Autonomous Database, automatic indexing is available but off until you turn it on with `DBMS_AUTO_INDEX.CONFIGURE`. The service name you connect through (HIGH, MEDIUM or LOW, plus TP and TPURGENT for transaction processing) sets priority and parallelism, so check the service before you tune the statement.

On RDS for Oracle, CloudWatch Database Insights also works on Standard Edition 2, so it doesn't depend on the packs. Its Advanced mode adds execution plans and longer retention.

CLOUD **SQL Server in the cloud**

- **Azure SQL Database** is a single database, not an instance. Use the database-scoped views: `sys.dm_db_resource_stats` shows resource use against the service tier's limits, and `sys.dm_db_wait_stats` replaces `sys.dm_os_wait_stats`. Query Store and automatic plan correction (`FORCE_LAST_GOOD_PLAN`) are on by default.
- **Tier limits show up as waits.** Reaching the tier's log rate limit, for example, appears as `LOG_RATE_GOVERNOR` waits. Check the limits before you tune the statements.
- **Azure SQL Managed Instance** behaves like a full instance, with SQL Server Agent and cross-database queries, but still without host access.
- **Amazon RDS for SQL Server** has no sysadmin login. The master user has a fixed set of permissions, server settings go through parameter groups, and RDS Custom for SQL Server adds host access when you need it.

CLOUD **PostgreSQL and MySQL in the cloud**

- **No true superuser.** The provider's admin role can do most monitoring setup, but not everything a superuser can.
- **Extensions from a list.** `pg_stat_statements` is offered everywhere; others, such as `pg_hint_plan` and `pg_wait_sampling`, depend on the service.
- **Aurora and AlloyDB replace the storage layer.** Checkpoint, WAL and InnoDB flushing advice for self-managed servers may not apply, and their waits differ. Aurora PostgreSQL's `IO:XactSync`, for example, is a commit waiting on Aurora storage.
- **MySQL's `performance_schema` is a startup setting.** Check that it is on before you need it, because turning it on means a reboot.

> **REINDB Managed platforms**
>
> ReinDB detects when a database runs on Amazon RDS or Aurora (including RDS for Oracle), Google Cloud SQL or Azure, and its fix scripts then name the parameter-group or provider setting to change instead of using `ALTER SYSTEM`. It needs only the monitoring grants in section 2.5, so it works without host access.

## 2.7 Beyond the built-ins: choosing a monitoring tool

Built-in views are deep but local: one database at a time, short memory, and no idea what is normal. A monitoring tool earns its place by adding history, comparison and reach. Judge candidates on these points:

- **Coverage.** All your engines and platforms, including managed services such as RDS and Aurora.
- **Footprint.** Read-only access with minimal grants, low overhead, and nothing installed inside the database if your security team objects to agents.
- **History.** At least two to four weeks kept, so baselines are meaningful.
- **Comparison.** It knows what normal looks like for each hour, instead of relying only on static thresholds.
- **Depth.** Statement-level drill-down, plan history and wait breakdown, not just server charts.
- **Licensing awareness.** It stays within the features you're licensed for, and says clearly when it would use a paid one, such as Oracle's Diagnostics and Tuning Packs.
- **Integration.** Exports to your observability stack (Prometheus and Grafana, for example) and alerting that reaches your on-call process.
- **Reviewable output.** Findings explained in plain terms, and fixes delivered as scripts your team can test, rather than changes made silently in production.
- **Security and deployment.** Credential handling, an audit trail, redaction of literals, and a deployment model that fits (hosted, in your cloud account, or on premises).

| Category | Examples | Strengths and limits |
|---|---|---|
| Built into the engine | Query Store, `pg_stat_statements`, `performance_schema`, V$ views and Statspack | Free and deep; one database at a time; little history or comparison unless you persist it |
| Vendor consoles | Oracle Enterprise Manager (performance pages need the packs), SQL Server Management Studio reports, cloud provider consoles | Close to the engine; usually tied to one vendor or one cloud |
| Open-source stacks | Prometheus with exporters and Grafana; Percona Monitoring and Management; pgBadger; pt-query-digest | Flexible and free to run; you build and maintain the dashboards, the alert rules and the baselines yourself |
| Commercial database monitoring | Cross-engine products from observability and database tool vendors | Broad coverage and polish; agents, pricing and lock-in vary |
| ReinDB | Drift monitoring for PostgreSQL, MySQL, SQL Server and Oracle | Learns normal per database and hour, flags drift with evidence, writes fix scripts; read-only; exports to Grafana |

# 3 Build a data profile

Before you decide what to tune, or where data should live, know what you have. A data profile is a short, factual description of each database: how big it is, how fast it grows, what is hot, and how it is used. It takes an afternoon, and it replaces weeks of guessing.

## 3.1 What goes in the profile

| Area | What to capture | Why it matters |
|---|---|---|
| Inventory | Engine, version, edition, platform, high availability and DR setup, total size, growth per month | Sets the constraints for every later decision |
| Objects | Largest tables and indexes (including LOB or TOAST storage), row counts, growth per table | Shows where space and maintenance time go |
| Activity | Reads and writes per table, full scans against index access, the hottest tables by time | Shows where the work goes |
| Temperature | How much data is touched each day, week and month; how old the rows are that queries actually read | Decides what can move to cheaper storage |
| Shape | Column cardinality and skew, null rates, wide columns, LOB and JSON use, data types of join keys | Explains misestimates and poor index choices |
| Workload | Transaction types (OLTP, reporting, batch, ETL), peak hours and days, concurrency, share of time taken by the top statements, read/write mix | Decides how to tune and what to protect |
| Constraints | Retention rules and legal holds, RPO and RTO, data classification (PII), residency, service levels | Limits what you're allowed to change or move |

## 3.2 Sizes and activity

For each database, capture the largest tables and indexes with their row counts, including large-object storage, and the activity on each table: rows read, inserted, updated and deleted, and full scans against index access. Activity counters are cumulative since the last restart or statistics reset, so compare two snapshots to get a rate. The engine notes give a starting query for each engine.

POSTGRESQL **Size and activity for the largest tables**

```
SELECT s.schemaname, s.relname,
       pg_size_pretty(pg_total_relation_size(s.relid)) AS total_size,
       pg_size_pretty(pg_indexes_size(s.relid))        AS index_size,
       s.n_live_tup, s.n_dead_tup,
       s.seq_scan, s.seq_tup_read, s.idx_scan,
       s.n_tup_ins, s.n_tup_upd, s.n_tup_del
FROM   pg_stat_user_tables s
ORDER  BY pg_total_relation_size(s.relid) DESC
LIMIT  25;
```

MYSQL **Sizes from the data dictionary, activity from the sys schema**

`TABLE_ROWS` is only an estimate for InnoDB.

```
SELECT table_schema, table_name, table_rows,
       ROUND(data_length  / 1024 / 1024) AS data_mb,
       ROUND(index_length / 1024 / 1024) AS index_mb
FROM   information_schema.tables
WHERE  table_schema NOT IN ('mysql', 'sys', 'performance_schema', 'information_schema')
ORDER  BY data_length + index_length DESC
LIMIT  25;

SELECT table_schema, table_name, rows_fetched, rows_inserted, rows_updated, rows_deleted
FROM   sys.schema_table_statistics
ORDER  BY rows_fetched DESC
LIMIT  25;
```

SQL SERVER **Sizes from partition statistics, activity from index usage**

Index usage statistics reset at every restart.

```
SELECT s.name AS schema_name, t.name AS table_name,
       SUM(CASE WHEN ps.index_id IN (0, 1) THEN ps.row_count END) AS row_count,
       SUM(ps.reserved_page_count) * 8 / 1024 AS reserved_mb
FROM   sys.dm_db_partition_stats ps
JOIN   sys.tables  t ON t.object_id = ps.object_id
JOIN   sys.schemas s ON s.schema_id = t.schema_id
GROUP  BY s.name, t.name
ORDER  BY reserved_mb DESC;
```

```
SELECT OBJECT_SCHEMA_NAME(u.object_id) AS schema_name,
       OBJECT_NAME(u.object_id)        AS table_name,
       SUM(u.user_seeks) AS seeks, SUM(u.user_scans) AS scans,
       SUM(u.user_lookups) AS lookups, SUM(u.user_updates) AS updates
FROM   sys.dm_db_index_usage_stats u
WHERE  u.database_id = DB_ID()
  AND  OBJECTPROPERTY(u.object_id, 'IsUserTable') = 1
GROUP  BY u.object_id
ORDER  BY SUM(u.user_seeks + u.user_scans + u.user_lookups) DESC;
```

ORACLE **Segment sizes and segment statistics**

The `ROWNUM` form works on 11g and later; on 12c and later you can use `FETCH FIRST 25 ROWS ONLY` instead.

```
SELECT * FROM (
  SELECT owner, segment_name, segment_type,
         ROUND(SUM(bytes) / 1024 / 1024) AS mb
  FROM   dba_segments
  WHERE  owner NOT IN (SELECT username FROM dba_users WHERE oracle_maintained = 'Y')  --
12c+
  GROUP  BY owner, segment_name, segment_type
  ORDER  BY mb DESC)
WHERE ROWNUM <= 25;
```

```
SELECT * FROM (
  SELECT owner, object_name, statistic_name, value
  FROM   v$segment_statistics
  WHERE  statistic_name IN ('logical reads', 'physical reads', 'row lock waits')
    AND  owner NOT IN ('SYS', 'SYSTEM')
  ORDER  BY value DESC)
WHERE ROWNUM <= 30;
```

On 11g, which has no `ORACLE_MAINTAINED` column, list the system schemas yourself. LOB segments appear with system-generated names (`SYS_LOB...`); join to `DBA_LOBS` to see which table they belong to. `DBA_TAB_MODIFICATIONS` shows inserts, updates and deletes since statistics were last gathered (run `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` first for current numbers).

## 3.3 Column shape: cardinality, skew and nulls

Skewed columns are behind many bad plans: the optimizer assumes an even spread unless statistics say otherwise. Check the columns your important statements filter and join on.

ENGINE NOTES **Where each engine keeps column statistics**

### PostgreSQL

```
SELECT attname, n_distinct, null_frac, most_common_vals, most_common_freqs
FROM   pg_stats
WHERE  schemaname = 'public' AND tablename = 'orders';
```

### MySQL (8.0+)

Only histograms you have created with `ANALYZE TABLE ... UPDATE HISTOGRAM` appear here:

```
SELECT schema_name, table_name, column_name, histogram->>'$."histogram-type"' AS
histogram_type
FROM   information_schema.column_statistics;
```

### SQL Server

The statistics header, density and histogram for one statistics object:

```
DBCC SHOW_STATISTICS ('dbo.orders', 'ix_orders_status');
```

### Oracle

```
SELECT column_name, num_distinct, num_nulls, histogram, last_analyzed
FROM   dba_tab_col_statistics
WHERE  owner = 'APP' AND table_name = 'ORDERS';
```

## 3.4 Growth and temperature

Growth needs history. If your monitoring doesn't keep size history, snapshot sizes once a day into a small table of your own; a year of daily rows per table is tiny.

> ORACLE **No free growth history without the Diagnostics Pack**
>
> Segment growth history (`DBA_HIST_SEG_STAT`) is part of AWR. Without the Diagnostics Pack, the daily snapshot of your own is the way.

Temperature tells you how much of the data is actually used. Where tables carry creation or event timestamps, two simple checks go a long way:

```
-- How rows are spread over time (standard SQL; see the note for SQL Server)
SELECT EXTRACT(YEAR FROM order_date)  AS yr,
       EXTRACT(MONTH FROM order_date) AS mon,
       COUNT(*)                       AS order_rows
FROM   orders
GROUP  BY EXTRACT(YEAR FROM order_date), EXTRACT(MONTH FROM order_date)
ORDER  BY yr, mon;
```

> SQL SERVER **No EXTRACT**
>
> Use `YEAR(order_date)` and `MONTH(order_date)` instead, or `DATETRUNC(month, order_date)` on SQL Server 2022 and later.

The second check is in the workload: look at the date predicates in your top statements. If nearly every query reads the last 90 days while the table holds seven years, you have found a tiering opportunity (Chapter 4) and probably a partitioning key (Chapter 10).

| Tier | Typical age | Access pattern | Where it usually belongs |
|---|---|---|---|
| Hot | Days to weeks | Frequent reads and updates; latency matters | Primary OLTP storage, fully indexed |
| Warm | Months | Reporting and occasional lookups | Older partitions in the same database, compressed; a replica or a warehouse |
| Cold | Years | Rare reads, compliance, analytics | Archive tables, or open file formats in object storage queried through external tables or a federated engine |

## 3.5 The workload profile

Data at rest is half the picture. The other half is how it is used.

- **Peaks.** Database time per hour across a typical week. Most systems have a clear daily shape and a weekly one.
- **Mix.** OLTP, reporting, batch and ETL, split by application or module where you can (see the instrumentation tips in Chapter 9).
- **Concentration.** The share of database time taken by the top 10 and top 50 statements. In most systems a small number of statements dominate, which is why statement-level tuning pays.
- **Concurrency.** Average active sessions against the number of CPU cores, at peak and on average.

- **Read/write mix and commit rate.** These drive log, WAL or redo volume, replication load and storage choices.
- **Overlap.** Batch windows, backups and statistics jobs, and how they overlap with online peaks.
- **Clients.** Which applications, tools and people connect, with connection counts and pool settings.

## 3.6 A one-page profile template

| Field | Example |
|---|---|
| Database, engine and version | orders-prod, PostgreSQL 16.4 on Amazon RDS |
| Size and growth | 1.9 TB, growing about 60 GB per month |
| Largest five tables, share of size | events 52%, order_items 14%, orders 8%, audit_log 7%, sessions 4% |
| Hottest five tables, share of reads | orders 31%, customers 18%, order_items 15%, inventory 9%, prices 6% |
| Workload mix | OLTP all day; reporting 07:00 to 09:00; ETL 01:00 to 03:00 |
| Peak window | Weekdays 09:00 to 11:00 local time |
| Top 10 statements, share of database time | 64% |
| Data older than 12 months | 71% of rows; read by two monthly reports |
| Retention and compliance | Orders 7 years; events 13 months; PII in customers and orders |
| RPO and RTO | 5 minutes and 1 hour |
| Known pain points | Morning search latency; the events table dominates backup time |

> REINDB **Let the profile build itself**
>
> ReinDB records database size every run and table sizes daily, projects the days until the disk fills, and keeps per-table activity (rows read and written, full scans) next to the top statements. After a few days of measurements, most of the one-page profile can be read straight off the Environment and Top SQL pages (and the Space page, on Oracle and PostgreSQL).

# 4 Choose the right platform for the data

Some performance problems are not tuning problems. They are placement problems: data kept on a platform that is wrong for the way it is used. No index will fix a four-billion-row clickstream table that is scanned for weekly aggregates on the same server that takes customer orders.

This chapter asks three questions. Is the data in the right storage? Does it all need to be relational? And if part of it moves elsewhere, how do you keep the whole data tier cohesive?

## 4.1 The familiar-platform trap

One of the most common sources of long-term performance debt has nothing to do with SQL. Teams extend the platform they already know with data that doesn't fit it.

It happens one reasonable step at a time. An events table lands in the OLTP database because the team knows that engine and the data is "just rows". Application logs go into the main database because the DBA team already backs it up. One relational database slowly becomes a queue, a search engine, a time-series store and a data lake, all at once. A key-value cache quietly turns into the system of record. Each step avoids new skills, new procurement and new runbooks, so each step looks cheap.

Together they build technical debt with interest:

- The platform is tuned for one workload and now fights four. Settings that help OLTP hurt the scans, and the other way around.
- Maintenance windows grow: statistics, vacuum, index rebuilds and backups are all sized by the biggest, least valuable table.
- Restores slow down, so recovery time objectives quietly stop being met.
- Licensing scales with data that uses none of the licensed features. On an engine licensed per core, archive data that is read twice a year can still drive the hardware, and with it the licence bill.
- Workarounds pile up: polling tables used as queues, `LIKE '%term%'` searches, triggers that maintain summary tables, generic entity-attribute-value tables, giant JSON blobs.
- Every later migration gets harder, because applications now depend on the wrong shape.

> WATCH OUT **Warning signs that a table is on the wrong platform**
>
> - One table, or a handful, dominates storage but carries little of the business logic.
> - It is append-only, or close to it, and rows are rarely updated after the first day.
> - Most queries against it are aggregate scans over time ranges, not lookups by key.
> - Its maintenance sets the length of the maintenance window or the backup.
> - Retention is "forever", because nobody owns deletion.
> - You keep adding indexes to make reports faster, and pay for every one of them on each insert.
> - The schema bends to fit the data: entity-attribute-value tables, untyped JSON, wide tables full of nulls.

How to avoid the trap:

1. **Make platform choice a design-review question,** with explicit criteria: access pattern, volume and growth, consistency needs, retention, and cost per terabyte including people.
2. **Prototype at realistic volume.** A design that works at one million rows can fall over at one billion.
3. **Decide the data lifecycle at creation:** who deletes it, when, and how. A purge job written on day one is cheap; one written in year five is a project.
4. **Set review triggers,** such as a table that grows tenfold or enters the top three by storage, and revisit its placement when one fires.
5. **Weigh familiarity, don't assume it.** Operational skill is a real asset and a valid criterion. It should be one input to the decision, not the whole decision.

## 4.2 Does it all need to be relational?

Relational engines are superb at what they were built for, and stretchable well beyond it. The question is where stretching stops paying.

| Data and workload | Relational fit | Alternatives when it outgrows the fit |
|---|---|---|
| Transactions with integrity rules and concurrent updates | Excellent | Keep it relational |
| Point lookups and short range scans | Excellent | Keep it relational |
| Reporting on recent data with joins | Good | Indexes, read replicas, materialized views |
| Batch processing and bulk loads | Excellent in sets, poor row by row | Set-based or bulk SQL first; an external pipeline only for very large transforms |
| High-volume, append-only events, logs and telemetry | Fair, then poor at scale | Partitioned and compressed tables first; then a columnar store, or Parquet files in object storage |
| Large history scanned for analytics | Poor in row stores | Columnar storage inside the engine, where offered; a columnar warehouse; Parquet or Iceberg tables with a query engine |
| Full-text search | Fair (built-in full-text features) | A dedicated search engine for heavy or relevance-ranked use |
| Documents with highly variable structure | Fair (JSON types) | A document store, if the relational part is small |
| Work queues | Fair (reads that skip locked rows work well at moderate rates) | A message broker for high throughput or fan-out |
| Files and large binary objects | Poor | Object storage, with a reference in the database |

ENGINE NOTES **Queue reads that skip locked rows**

`SELECT ... FOR UPDATE SKIP LOCKED` works in PostgreSQL (9.5+), MySQL (8.0+) and Oracle. SQL Server uses the `READPAST` table hint with `UPDLOCK` and `ROWLOCK` for the same pattern.

Batch work is the workload most often moved out too early. A nightly job that crawls is usually working one row at a time; rewritten as set-based or bulk SQL, it often fits its window again on the same platform. Every engine has a fast load path and a way to work on rows in sets. The engine notes show both, plus a batch read for a work queue (Chapter 9 shows longer rewrites).

POSTGRESQL **COPY and set-based batches**

PL/pgSQL has no bulk binds: every statement inside a loop runs once per row. Do the batch as one set-based statement, or pass the keys as an array and join to `unnest()` (or filter with `= ANY(...)`) instead of looping.

For loads, `COPY` beats any series of `INSERT` statements. Into a new table, load first and create the indexes afterwards. On managed services, use the client-side form (`COPY ... FROM STDIN`, or `\copy` in psql).

For a work queue, one statement claims a batch:

```
UPDATE work_queue
SET    status = 'TAKEN'
WHERE  id IN (SELECT id FROM work_queue
              WHERE  status = 'NEW'
              ORDER  BY id
              LIMIT  500
              FOR UPDATE SKIP LOCKED)
RETURNING id;
```

MYSQL **LOAD DATA and multi-row inserts**

Stored procedures have no bulk binds either, so do the batch as one set-based statement: `INSERT ... SELECT`, `INSERT ... ON DUPLICATE KEY UPDATE`, or a multi-table `UPDATE` or `DELETE` with a join.

For loads, MySQL's manual puts `LOAD DATA` at usually 20 times faster than `INSERT` statements; otherwise send multi-row `INSERT ... VALUES` lists, not one row per statement. `LOAD DATA LOCAL` reads the file from the client, the usual choice on managed services, and needs `local_infile` enabled on both the server and the client.

For a work queue (8.0+), lock a batch and update it in the same transaction:

```
START TRANSACTION;
SELECT id FROM work_queue
WHERE  status = 'NEW'
ORDER  BY id
LIMIT  500
FOR UPDATE SKIP LOCKED;            -- the application keeps the ids it gets back
UPDATE work_queue SET status = 'TAKEN'
WHERE  id IN (1001, 1002, 1003);   -- those ids
COMMIT;
```

SQL SERVER **Bulk import and table-valued parameters**

T-SQL has no bulk binds; cursors and `WHILE` loops work one row at a time. Do the batch as one set-based statement, and have the application send its rows as a table-valued parameter (2008+) instead of making one call per row.

For loads, use `BULK INSERT`, `bcp` or the client bulk-copy API (`SqlBulkCopy` in .NET). Under the simple or bulk-logged recovery model, a `TABLOCK` load into a heap or an empty table can be minimally logged; a load into a non-empty clustered index is fully logged whatever the recovery model.

For a work queue, one statement claims a batch:

```
WITH batch AS (
  SELECT TOP (500) id, status
  FROM   dbo.work_queue WITH (UPDLOCK, READPAST, ROWLOCK)
  WHERE  status = 'NEW'
  ORDER  BY id)
UPDATE batch
SET    status = 'TAKEN'
OUTPUT inserted.id;
```

ORACLE **Bulk SQL**

When PL/SQL has to loop, bulk SQL moves rows between PL/SQL and SQL in batches instead of one at a time. `BULK COLLECT` fetches many rows per call (add `LIMIT` to cap the memory each batch takes), and `FORALL` sends a whole collection of `INSERT`, `UPDATE` or `DELETE` statements to SQL in one call. With `SAVE EXCEPTIONS`, `FORALL` carries on past failing rows and reports them in `SQL%BULK_EXCEPTIONS` at the end. Bulk SQL works in every edition, 11g included.

For loads, use direct-path insert (`INSERT /*+ APPEND */ ... SELECT`), external tables or SQL*Loader rather than row-by-row inserts. A direct-path insert locks the table, so keep it to load windows.

For a work queue, pair bulk SQL with `SKIP LOCKED`. Oracle locks those rows as it fetches them, so each bulk fetch claims the next batch of rows no other session holds:

```
DECLARE
  CURSOR c IS
    SELECT id FROM work_queue
    WHERE  status = 'NEW'
    FOR UPDATE SKIP LOCKED;
  TYPE t_ids IS TABLE OF work_queue.id%TYPE;
  l_ids t_ids;
BEGIN
  OPEN c;
  FETCH c BULK COLLECT INTO l_ids LIMIT 500;   -- up to 500 free rows, locked as fetched
  FORALL i IN 1 .. l_ids.COUNT
    UPDATE work_queue SET status = 'TAKEN' WHERE id = l_ids(i);
  CLOSE c;
  COMMIT;
END;
/
```

Chapter 9 shows a longer bulk rewrite, and section 9.6 covers error handling for bulk binds.

## 4.3 Tier inside the engine first

The cheapest way to split hot and cold data is often to stay in the same engine. You keep one security model, one backup, one set of skills, and frequently that alone fixes the problem.

- **Partition by date** (Chapter 10) so old data can be compressed, moved or dropped a partition at a time instead of with huge `DELETE` statements.
- **Compress older partitions.** Most engines can compress data that rarely changes, by row, by page or by column. Check which kinds need a paid option.
- **Move cold partitions to cheaper storage** with tablespaces or filegroups on slower disks.
- **Use columnar storage for history,** where the engine offers it. On an archive table or on older partitions it can shrink the data several-fold and speed up scans dramatically.
- **Use archive tables or an archive database** with the same schema, and query across them with a view when needed.
- **Send reporting to a read replica** to protect the primary from long scans.

> ENGINE NOTES **Compression and columnar storage**
>
> - **PostgreSQL:** large values are compressed automatically (TOAST, with `lz4` available from 14), but core PostgreSQL has no general table or page compression and no columnar storage. Those need an extension, or a managed service that offers them.
> - **MySQL:** InnoDB compressed tables (`ROW_FORMAT=COMPRESSED`) or transparent page compression, which needs a file system that supports hole punching. The community server has no columnar storage.
> - **SQL Server:** row and page compression and clustered columnstore indexes, in all editions since 2016 SP1. A clustered columnstore index is the usual choice for archive tables and older partitions.
> - **Oracle:** basic table compression for direct-path loaded data comes with Enterprise Edition. Advanced Row Compression needs the Advanced Compression option, and Hybrid Columnar Compression needs Exadata or certain Oracle storage. The Database In-Memory column store is a separately licensed option, with a 16 GB base level that needs no extra license from 19.8.

## 4.4 Hybrid data tiers: when part of the data leaves

When data truly doesn't belong in the OLTP engine (large, append-mostly, scanned rather than looked up), move it to a tier built for that, and keep it queryable.

**Open file formats.** Parquet stores data by column, compressed, so analytic scans read only the columns they need. Table formats such as Apache Iceberg (and Delta Lake or Apache Hudi) add what plain files lack: atomic commits, snapshot isolation for readers, schema evolution, partition evolution and time travel.

**Foreign and external tables** let the database itself read data that lives elsewhere:

| Engine | Options |
|---|---|
| PostgreSQL | Foreign data wrappers: `postgres_fdw` and `file_fdw` ship with PostgreSQL; third-party wrappers and extensions read Parquet and object storage |

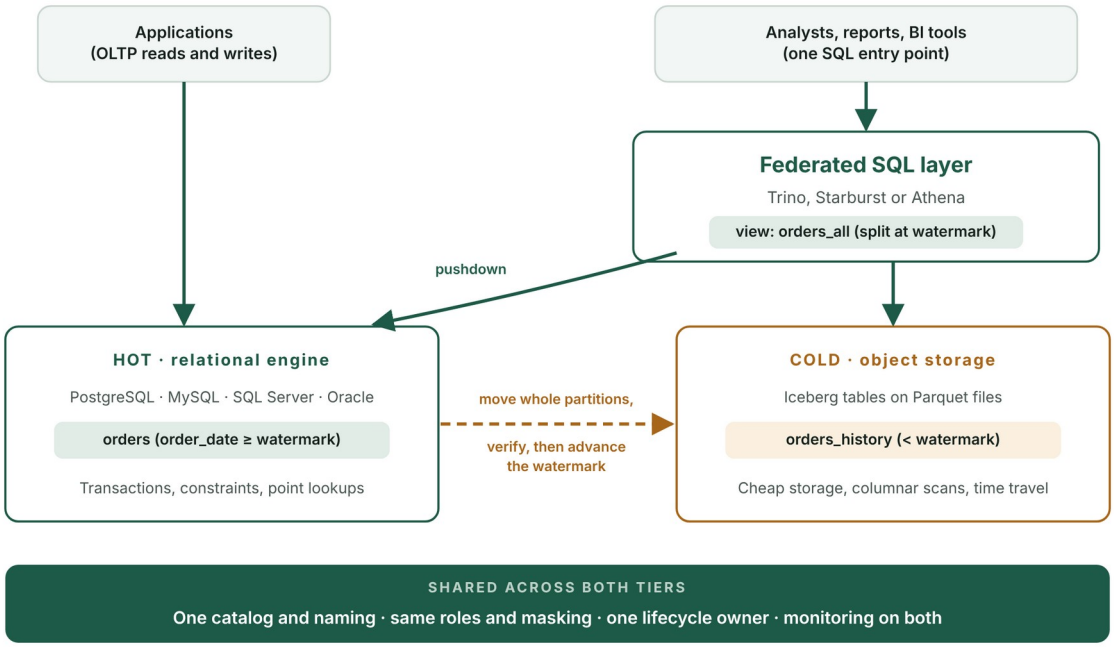
| Engine | Options |
|---|---|
| MySQL | No general foreign-table feature in the community server. The `FEDERATED` engine reaches only other MySQL servers and is rarely a good idea for performance. For files in object storage, use a federated engine (below) or a managed service with lakehouse features |
| SQL Server | PolyBase external tables. SQL Server 2022 reads Parquet, CSV and Delta files in S3-compatible object storage and Azure storage (`OPENROWSET`, `CREATE EXTERNAL TABLE`) and can write cold data out as Parquet with `CREATE EXTERNAL TABLE AS SELECT` |
| Oracle | External tables (`ORACLE_LOADER`, `ORACLE_DATAPUMP`); hybrid partitioned tables in 19c, with some partitions internal and some external (requires the Partitioning option); newer releases add an access driver for Parquet and other object-storage formats |

**Federated SQL engines** query many sources in one statement. Trino is the best-known open-source example, with connectors for PostgreSQL, MySQL, SQL Server, Oracle, Iceberg, Delta Lake, Hive and more. Starburst is its commercial distribution, and Amazon Athena's current engine is based on it. Trino turns "the hot data is in the OLTP database and the history is in Parquet" into one logical table.

> CLOUD **Managed services add their own routes to the lake**
>
> - **Amazon RDS and Aurora:** snapshots export to Amazon S3 as Parquet, the `aws_s3` extension imports and exports S3 files from PostgreSQL, and zero-ETL integrations replicate into Amazon Redshift.
> - **Microsoft:** Azure SQL Database and SQL Server 2025 can mirror into Microsoft Fabric.
> - **Google Cloud:** BigQuery can query Cloud SQL directly with federated queries, and AlloyDB adds a columnar engine.
> - **Oracle Autonomous Database:** external tables over object storage with `DBMS_CLOUD`.
> - **Only the provider's approved extensions run,** so check that a foreign data wrapper is offered before you design around it.

## 4.5 Keeping a hybrid tier cohesive

A hybrid tier is only an improvement if people can still find, trust and query the data. These practices keep it cohesive.




*Keeping a hybrid tier cohesive: hot data stays in the relational engine, cold data moves to Iceberg tables in object storage, and one federated SQL layer presents a single view, with the same catalog, security and monitoring across both.*

1. **One catalog of record.** Register lake tables in one catalog (a Hive metastore, AWS Glue or an Iceberg REST catalog) and keep names and keys consistent across tiers: `orders` in the OLTP database, `orders_history` in the lake, the same column names and types in both.
2. **One query entry point for analytics.** In Trino, each source is a catalog. A view can union the hot and cold parts behind one name, split at a watermark so no row is counted twice:

```
-- Trino: one logical orders table across two tiers.
-- 'pg' is a PostgreSQL catalog and 'lake' an Iceberg catalog; both names are yours to choose.
CREATE OR REPLACE VIEW lake.analytics.orders_all AS
SELECT order_id, customer_id, order_date, total_amount
FROM   pg.public.orders
WHERE  order_date >= DATE '2026-01-01'          -- the watermark: move it when you move data
UNION ALL
SELECT order_id, customer_id, order_date, total_amount
FROM   lake.archive.orders_history
WHERE  order_date <  DATE '2026-01-01';
```

1. **Move data with a contract.** Move whole partitions, verify row counts and checksums, move the watermark, then drop the source partition. Trino can do the copy itself (`INSERT INTO lake.archive.orders_history SELECT ... FROM pg.public.orders WHERE ...`), and some engines can write open formats directly (see the table in section 4.4). For continuous feeds, use change data capture (for example Debezium, AWS DMS, SQL Server CDC or Oracle GoldenGate, which is licensed separately).
2. **The same security everywhere.** Map roles across tiers, apply the same masking for personal data, and make sure archived copies are covered by the same retention and deletion rules as the source.
3. **Know what gets pushed down.** Trino pushes filters, and for many connectors also projections, aggregations, joins and `LIMIT`, down to the source database. What it can't push down, it pulls across the network. Filter early, and read `EXPLAIN` output in Trino to see what really runs where.

4. **Define freshness and consistency.** There are no transactions across sources. Agree how far behind the lake may be, and make the watermark boundary exclusive on one side and inclusive on the other.
5. **Observe the federated layer too.** Trino keeps query history (`system.runtime.queries`) and exposes worker metrics. A slow federated query is often a slow source query in disguise, so trace it back.
6. **Watch the cost model.** Object storage is cheap, scanning isn't always (some services charge per terabyte scanned). Partition and sort lake tables by the columns people filter on, and compact small files regularly.

> WATCH OUT **Common hybrid-tier mistakes**
>
> - Moving data out to "fix performance" when an index or partitioning would have fixed it in place.
> - Sending OLTP lookups through the federated engine. It is built for scans, not millisecond point reads.
> - Copies that drift: a lake table refreshed by a job nobody owns, with a schema that no longer matches.
> - The small-files problem: thousands of tiny files that make every scan slow.
> - Losing referential integrity and nobody noticing, because no constraint spans the tiers.
> - Ignoring data residency rules when the archive lands in a different region.
> - Underestimating operations: a federated engine is one more cluster to patch, secure, monitor and pay for.

## 4.6 Before you move a table out

Ask these questions, and write the answers down:

1. What share of database time, storage and maintenance time does the table account for today, and in a year?
2. Would partitioning, compression or archiving inside the engine solve the problem?
3. Who queries the data after it moves, how often, and how fast must the answer be?
4. What consistency do those readers need?
5. Who owns the lifecycle: the move, the verification and the deletion?
6. What will it cost to run, people included?

If the table is cheap to keep where it is, keep it there. Hybrid tiers pay off when data is large, cold and scanned; they cost you when data is small, hot or transactional.

> REINDB **Evidence for tiering decisions**
>
> ReinDB's busiest tables, table size history and storage projections show which tables cost the most in time, in space and in growth. Those are the numbers for question 1 above, and they make the case for (or against) moving data out.

# 5 Decide what to look for

"Find the slow queries" sounds like a plan, but it hides two questions: slow by what measure, and is a query the right place to start at all? This chapter answers both, then shows how to collect the evidence, with engine notes for the details.

## 5.1 Slow queries or overall execution time? Use three lenses

| Statement | Calls per hour | Average time | Total time per hour | Share of load |
|---|---|---|---|---|
| A: product lookup | 900,000 | 4 ms | 3,600 s | 81% |
| B: monthly finance report | 2 | 45 s | 90 s | 2% |
| C: checkout insert | 30,000 | 25 ms | 750 s | 17% |

Each statement is "the problem" through a different lens:

- **Throughput lens (total time = calls × average time).** Statement A costs the database the most. Making it 25% cheaper frees more capacity than making B instant. Rank by total time when the goal is lower load, lower cost or more headroom.
- **Experience lens (time per call, ideally the 95th or 99th percentile).** Statement B is what finance complains about, and C sits on the path to revenue. Rank by per-call latency of user-facing statements when the goal is a better experience.
- **Regression lens (now against its own normal).** A statement that used to take 4 ms and now takes 12 ms matters even if it isn't top of either list yet, because something changed: a plan, the data, or contention. Regressions are the earliest warning you get.

A good investigation uses all three. Start from the lens that matches the symptom, and check the other two before you finish.

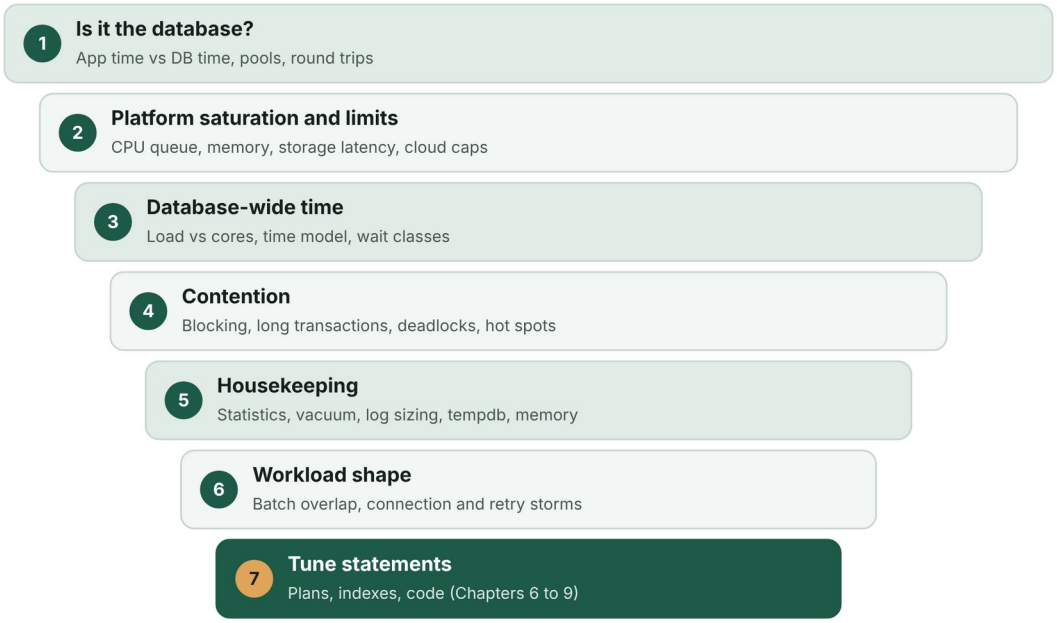
## 5.2 Four proven methods

- **Response-time analysis.** Measure where the time goes for the task that matters, and attack the biggest component. Database time is CPU time plus wait time. Cary Millsap's Method R is the classic statement of the idea.
- **Wait-based analysis.** Classify non-idle database time by wait class (CPU, I/O, locks, commit, concurrency, network) to find the dominant bottleneck.
- **USE method** (Brendan Gregg). For every resource, check **U**tilization, **S**aturation and **E**rrors. Ideal for the host and storage underneath the database.
- **RED method.** For every service, track **R**ate, **E**rrors and **D**uration. Ideal for the application in front of the database.

They complement each other: RED tells you a service is slow, response-time and wait analysis tell you whether the database is why, and USE tells you whether the hardware is the limit.

## 5.3 What comes before statement-level tuning

Statement tuning is powerful, but it is the last layer to look at, not the first. Work from the outside in.




*What comes before statement tuning: confirm it is the database, then check platform saturation, database-wide time, contention, housekeeping and workload shape, and only then tune individual statements.*

1. **Confirm the database is where the time goes.** Compare the application's request time with the database time per request. Check connection pool waits, network round trips (a chatty ORM making hundreds of small calls per page), and application pauses. If the database accounts for 50 ms of a 4-second request, tune elsewhere.
2. **Check the platform for saturation and limits.** CPU run queue longer than the number of cores; memory pressure or swapping; storage latency (SSD-class storage normally answers in low single-digit milliseconds or better, so sustained tens of milliseconds means trouble); cloud limits such as provisioned IOPS and throughput caps, burst credit balances on burstable volumes and instances, and per-instance network bandwidth.
3. **Look at database-wide time.** Compare average active sessions with the number of CPU cores. Read the time model and the top wait classes. This tells you what kind of problem you have.
4. **Check contention.** Blocking chains, lock waits, long transactions and sessions left idle inside an open transaction, deadlocks, hot rows and pages, latch and spinlock contention.
5. **Check housekeeping.** Stale statistics, cleanup that has fallen behind (and the bloat it leaves), skipped maintenance jobs, undersized log files, temporary space configuration, memory settings.

6. **Check the shape of the workload.** Batch jobs overlapping the online peak, connection storms after a deploy, retry storms from an unhealthy service, an ad hoc report running on the primary.
7. **Then tune statements.** Take the top statements by total time and the biggest regressions, and work through their plans, indexes and code (Chapters 6 to 9).

Why this order? A perfect statement fix does nothing while storage is being throttled or a forgotten session holds a lock. And a database-wide fix, such as fresh statistics or a corrected memory setting, can fix fifty statements at once.

> ENGINE NOTES **Housekeeping to check first**
>
> - **PostgreSQL:** autovacuum lag and bloat (`n_dead_tup` and `last_autovacuum` in `pg_stat_user_tables`), and transaction ID age (`age(datfrozenxid)`), which forces aggressive vacuums when it grows.
> - **MySQL:** purge lag, shown as the history list length in `SHOW ENGINE INNODB STATUS` or `trx_rseg_history_len` in `information_schema.INNODB_METRICS`. Long transactions hold it back.
> - **SQL Server:** tempdb data files (several, equally sized) and allocation contention; statistics updates and index maintenance jobs that failed or were skipped.
> - **Oracle:** redo log size (many log switches per hour, or `log file switch (checkpoint incomplete)` waits), undo retention, and stale statistics (`STALE_STATS = 'YES'` in `DBA_TAB_STATISTICS`).

> CLOUD **Where cloud limits show up**
>
> - **Amazon RDS:** compare `ReadIOPS` and `WriteIOPS` with the provisioned IOPS, and `ReadThroughput` and `WriteThroughput` with the volume's limit. `BurstBalance` (gp2 volumes), `CPUCreditBalance` (burstable instance classes) and `EBSIOBalance%` or `EBSByteBalance%` (smaller instance classes) show credits running out. Aurora has no provisioned volume, but the instance class still caps network and throughput.
> - **Azure SQL Database:** `sys.dm_db_resource_stats` reports CPU, data I/O and log writes as a percentage of the tier's limit, every 15 seconds for the last hour. Anything pinned near 100% is a limit, not a query problem.
> - **Google Cloud SQL:** disk IOPS and throughput grow with disk size and vCPU count, so a small disk can be the bottleneck on a large instance.

## 5.4 Collecting the evidence: counters, deltas and samples

Most performance views hold **cumulative counters**: totals since the instance started or since the last reset. A total on its own means little. Take snapshots on a fixed interval and subtract: hourly for trends and baselines, every one to five minutes during an incident.

Counters disappear or reset more often than people expect:

- Restarts reset most counters and caches.
- Someone resets the statistics by hand.
- Views built on the plan cache only show statements still in the cache.
- Fixed-size statement tables fill up, and then evict the least-used statements or lump new ones together.
- Persistent stores keep their data across restarts, which is one more reason to turn them on.

> ENGINE NOTES **Resets, eviction and persistence**
>
> - **PostgreSQL:** statistics survive a clean shutdown but not a crash. `pg_stat_reset()` and `pg_stat_statements_reset()` clear them by hand. `pg_stat_statements` keeps `pg_stat_statements.max` entries (5,000 by default) and evicts the least-executed.
> - **MySQL:** everything resets at restart, and `TRUNCATE` of a `performance_schema` summary table resets it. The digest summary table has a fixed size; when it is full, new statements are lumped into one catch-all row with a null digest.
> - **SQL Server:** `sys.dm_exec_query_stats` only shows statements still in the plan cache, and wait statistics reset at restart. Query Store persists across restarts.
> - **Oracle:** system-wide `V$` counters reset at instance startup. `V$SQL` only shows cursors still in the shared pool; `V$SQLSTATS` keeps their statistics a little longer, but it is still memory. Statspack snapshots persist.

For "what is happening right now", **sample** the active sessions every second or every few seconds: who is active, what each one is running and what it waits on. Added up, the samples give you a rough active-session history for free.

> ENGINE NOTES **Session views to sample**
>
> - **PostgreSQL:** `pg_stat_activity` where `state = 'active'`.
> - **MySQL:** `performance_schema.threads` with `performance_schema.events_waits_current`.
> - **SQL Server:** `sys.dm_exec_requests` with `sys.dm_os_waiting_tasks`.
> - **Oracle:** `V$SESSION` where `STATUS = 'ACTIVE'` and the session isn't waiting on an idle event. Sampling `V$SESSION` yourself is free; querying `V$ACTIVE_SESSION_HISTORY` requires the Diagnostics Pack.

Keep at least two weeks of history. Chapter 12 explains why that number matters.

## 5.5 Where does the time go?

Every engine can answer the same three questions, and it pays to ask them in this order. How does database time split between CPU and waiting? Which waits dominate? Which statements consume the most time? If the time is mostly CPU, look for statements with high logical reads per execution, and for heavy parsing or

compilation. If waits dominate, the top waits tell you where to look next (section 5.7). The engine notes show where each engine keeps the answers.

POSTGRESQL **Top statements and sampled waits**

Top statements from `pg_stat_statements`. The column names below are for PostgreSQL 13 and later; earlier versions use `total_time` and `mean_time`.

```
SELECT queryid, calls,
       round(total_exec_time)              AS total_ms,
       round(mean_exec_time::numeric, 2)   AS mean_ms,
       rows,
       shared_blks_hit, shared_blks_read, temp_blks_written,
       left(query, 80)                     AS query
FROM   pg_stat_statements
ORDER  BY total_exec_time DESC
LIMIT  20;
```

PostgreSQL has no cumulative wait history in core. Sample what active sessions are waiting on; repeat the query every few seconds and add up the results, or install the `pg_wait_sampling` extension.

```
SELECT coalesce(wait_event_type, 'CPU (or not instrumented)') AS wait_type,
       wait_event, count(*) AS sessions
FROM   pg_stat_activity
WHERE  state = 'active' AND pid <> pg_backend_pid()
GROUP  BY 1, 2
ORDER  BY 3 DESC;
```

MYSQL **Top statement digests**

Top statement digests from `performance_schema`. Timer columns are in picoseconds.

```
SELECT digest_text,
       count_star                          AS calls,
       ROUND(sum_timer_wait / 1e12, 1)     AS total_s,
       ROUND(avg_timer_wait / 1e9, 2)      AS avg_ms,
       sum_rows_examined, sum_rows_sent,
       sum_no_index_used, sum_created_tmp_disk_tables
FROM   performance_schema.events_statements_summary_by_digest
ORDER  BY sum_timer_wait DESC
LIMIT  20;
```

The ratio of rows examined to rows sent is one of the most useful numbers in MySQL tuning. A statement that examines 500,000 rows to return 20 is missing an index or filtering too late.

SQL SERVER **Wait statistics and top statements**

Wait statistics since the last restart, with common benign waits filtered out. The exclusion list below is shortened; community scripts maintain longer ones.

```
SELECT TOP (15)
       wait_type,
       waiting_tasks_count,
       wait_time_ms / 1000.0        AS wait_s,
       signal_wait_time_ms / 1000.0 AS signal_s
FROM   sys.dm_os_wait_stats
WHERE  wait_type NOT IN (
         N'BROKER_EVENTHANDLER', N'BROKER_RECEIVE_WAITFOR', N'BROKER_TASK_STOP',
         N'BROKER_TO_FLUSH', N'CHECKPOINT_QUEUE', N'DIRTY_PAGE_POLL',
         N'FT_IFTS_SCHEDULER_IDLE_WAIT', N'HADR_FILESTREAM_IOMGR_IOCOMPLETION',
         N'LAZYWRITER_SLEEP', N'LOGMGR_QUEUE', N'QDS_PERSIST_TASK_MAIN_LOOP_SLEEP',
         N'QDS_CLEANUP_STALE_QUERIES_TASK_MAIN_LOOP_SLEEP',
N'REQUEST_FOR_DEADLOCK_SEARCH',
         N'SOS_WORK_DISPATCHER', N'SP_SERVER_DIAGNOSTICS_SLEEP',
         N'SQLTRACE_INCREMENTAL_FLUSH_SLEEP', N'WAITFOR', N'XE_DISPATCHER_WAIT',
N'XE_TIMER_EVENT')
  AND  wait_type NOT LIKE N'SLEEP[_]%'
ORDER  BY wait_time_ms DESC;
```

A high share of signal wait time means tasks are waiting for CPU after their resource became available, a sign of CPU pressure.

Top statements over the last day from Query Store. Durations are stored in microseconds.

```
SELECT TOP (20)
       q.query_id,
       SUM(rs.count_executions)                                    AS executions,
       SUM(rs.avg_duration * rs.count_executions) / 1000000.0      AS total_duration_s,
       SUM(rs.avg_cpu_time * rs.count_executions) / 1000000.0      AS total_cpu_s,
       SUM(rs.avg_logical_io_reads * rs.count_executions)          AS logical_reads,
       MAX(qt.query_sql_text)                                      AS query_text
FROM   sys.query_store_runtime_stats rs
JOIN   sys.query_store_runtime_stats_interval i
       ON i.runtime_stats_interval_id = rs.runtime_stats_interval_id
JOIN   sys.query_store_plan p        ON p.plan_id = rs.plan_id
JOIN   sys.query_store_query q       ON q.query_id = p.query_id
JOIN   sys.query_store_query_text qt ON qt.query_text_id = q.query_text_id
WHERE  i.start_time >= DATEADD(day, -1, SYSDATETIMEOFFSET())
GROUP  BY q.query_id
ORDER  BY total_duration_s DESC;
```

Without Query Store, the plan cache gives a similar list for statements still cached:

```
SELECT TOP (20)
       qs.execution_count,
       qs.total_elapsed_time / 1000000.0                     AS total_elapsed_s,
       qs.total_worker_time  / 1000000.0                     AS total_cpu_s,
       qs.total_logical_reads,
       qs.total_elapsed_time / qs.execution_count / 1000.0   AS avg_ms,
       SUBSTRING(st.text,
                 (qs.statement_start_offset / 2) + 1,
                 ((CASE qs.statement_end_offset
                     WHEN -1 THEN DATALENGTH(st.text)
                     ELSE qs.statement_end_offset
                   END - qs.statement_start_offset) / 2) + 1) AS statement_text
FROM   sys.dm_exec_query_stats qs
CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) st
ORDER  BY qs.total_elapsed_time DESC;
```

ORACLE **Time model, waits and top statements, without the packs**

The time model gives the big picture: how much of database time is CPU, SQL execution, parsing and PL/SQL. Values are in microseconds since startup, so compare two snapshots.

```
SELECT stat_name, ROUND(value / 1e6) AS seconds
FROM   v$sys_time_model
WHERE  stat_name IN ('DB time', 'DB CPU', 'sql execute elapsed time', 'parse time
elapsed',
                     'hard parse elapsed time', 'PL/SQL execution elapsed time')
ORDER  BY value DESC;
```

Top non-idle waits:

```
SELECT * FROM (
  SELECT event, wait_class, total_waits,
         ROUND(time_waited_micro / 1e6)                          AS seconds,
         ROUND(time_waited_micro / NULLIF(total_waits, 0) / 1000, 2) AS avg_ms
  FROM   v$system_event
  WHERE  wait_class <> 'Idle'
  ORDER  BY time_waited_micro DESC)
WHERE ROWNUM <= 15;
```

Top statements by elapsed time:

```
SELECT * FROM (
  SELECT sql_id, plan_hash_value, executions,
         ROUND(elapsed_time / 1e6)                              AS elapsed_s,
         ROUND(cpu_time / 1e6)                                  AS cpu_s,
         buffer_gets, disk_reads,
         ROUND(elapsed_time / NULLIF(executions, 0) / 1000, 1)  AS avg_ms,
         ROUND(buffer_gets / NULLIF(executions, 0))             AS gets_per_exec,
         SUBSTR(sql_text, 1, 80)                                AS sql_text
  FROM   v$sqlstats
  ORDER  BY elapsed_time DESC)
WHERE ROWNUM <= 20;
```

If DB CPU is close to DB time, sessions are mostly on CPU: look for statements with high buffer gets per execution, and for hard parsing. If waits dominate, the top wait events tell you where to look next (see the table in section 5.7).

## 5.6 From symptom to strategy

| Symptom | Look first at | Typical fix area |
|---|---|---|
| Everything slow, CPU high | Top statements by CPU and logical reads; parse rates | Plans and indexes; bind variables; runaway parallelism |
| Everything slow, CPU low, I/O waits high | Storage latency; top statements by physical reads | Indexes; memory sizing; storage tier or limits |
| Spikes at the same times every day | Jobs, backups and statistics runs that overlap the peak | Scheduling; resource governance; batch design |
| One feature slow, the rest fine | That statement's plan history and statistics | Plan regression; statistics; parameter sensitivity |
| Intermittent timeouts | Blocking chains and lock waits; long transactions | Transaction scope; indexes on foreign keys; isolation level |

| Symptom | Look first at | Typical fix area |
|---|---|---|
| Slow commits | Log, WAL or redo write latency; commit rate | Faster log storage; fewer, larger commits |
| Gradual slowdown over weeks | Growth, bloat, statistics staleness | Maintenance; partitioning; archiving |
| Slow only on the replica | Replay lag; conflicts with long queries | Replica sizing; query timeouts; replica settings |

## 5.7 The same waits under different names

| What sessions wait for | PostgreSQL | MySQL | SQL Server | Oracle |
|---|---|---|---|---|
| Single-block data read | `IO: DataFileRead` | `wait/io/file/innodb/innodb_data_file` | `PAGEIOLATCH_SH` | `db file sequential read` |
| Multi-block or direct read | `IO: DataFileRead` during scans | `wait/io/file/innodb/innodb_data_file` | `PAGEIOLATCH_SH` during scans | `db file scattered read`, `direct path read` |
| Commit and log write | `IO: WALSync`, `IO: WALWrite` | `wait/io/file/innodb/innodb_log_file` | `WRITELOG` | `log file sync` |
| Row lock held by another transaction | `Lock: transactionid`, `Lock: tuple` | `Innodb_row_lock_time` and `sys.innodb_lock_waits` | `LCK_M_U`, `LCK_M_X` and other `LCK_M_*` | `enq: TX - row lock contention` |
| Memory for a query to start | (`work_mem` is allocated per operation) | (buffers allocated per session) | `RESOURCE_SEMAPHORE` | (PGA is allocated, not queued) |
| Parallel query coordination | `IPC: ExecuteGather` | (not applicable) | `CXPACKET`, `CXCONSUMER` | `PX Deq` events |
| Client not reading results fast enough | `Client: ClientWrite` | `wait/io/socket/sql/client_connection` | `ASYNC_NETWORK_IO` | `SQL*Net more data to client` |
| Hot shared memory structures | `LWLock: BufferContent` | `wait/synch/...` mutexes and rw-locks | `PAGELATCH_*` (often tempdb or last-page inserts) | `latch: cache buffers chains`, `buffer busy waits` |

> **REINDB** **Where ReinDB helps**
>
> ReinDB ranks drift flags by the extra database time each one costs, so the throughput and regression lenses come pre-sorted. A statement's page splits its time into CPU, I/O wait and lock wait (where the engine measures it), the Top SQL page lists the biggest consumers, the Environment page shows the wait profile over time, and Live now reads current sessions, blockers and waits with a 5-second limit.

# 6 The statement tuning procedure

Once the evidence points at specific statements, tune them with a fixed procedure. The procedure matters more than any single trick: it keeps you from fixing symptoms, and it leaves a record that the next person can follow.

## 6.1 Ten steps

1. **Pick the target and state the goal.** For example: "Statement 7f3k...: 2.1 s average between 09:00 and 11:00, 40,000 calls per hour. Goal: under 200 ms."
2. **Record its baseline.** Executions, average and worst time, logical reads per execution, rows per execution, the plan identifier, and the split between CPU and waits. Logical reads per execution is the most stable efficiency measure you have: it doesn't depend on cache state or concurrency.
3. **Get the full text and real parameter values.** Statement statistics normalize literals away, and plans depend on values, so find the values the slow executions actually used (see the engine note after these steps).
4. **Capture the actual plan with runtime row counts,** not just the estimated plan (section 6.2).
5. **Read the plan** (section 6.3). Find where estimated and actual rows diverge, where the time and buffers accumulate, and which access paths and join methods were chosen.
6. **Form one hypothesis** you can state in a sentence: "The optimizer expects 12 rows from orders but gets 480,000, because status is skewed and has no histogram, so it chose nested loops."
7. **Choose the least invasive fix** that addresses the cause, in roughly this order: rewrite the query (if you own the code); fix statistics; add or change an index; pin a plan as a temporary measure; change a setting; change the schema.
8. **Test with production-like data** and several representative parameter values, especially values from both ends of a skewed distribution. Compare logical reads and rows, not just elapsed time.
9. **Deploy with a rollback script,** at a quiet hour, and tell the people who own the application.
10. **Verify against the baseline** at the same hour on the following days, and record what changed, why, and the before and after numbers.

ENGINE NOTES **Where to find real parameter values**

- **PostgreSQL:** `auto_explain` or `log_min_duration_statement` output, which logs the parameters with the statement.
- **MySQL:** `performance_schema.events_statements_history` and `events_statements_history_long` keep recent statement text with its literals; so does the slow query log.
- **SQL Server:** the compiled parameter values stored in the plan XML (`ParameterCompiledValue`), in the plan cache and in Query Store.
- **Oracle:** `V$SQL_BIND_CAPTURE` (sampled, at most every 15 minutes per cursor by default), and the peeked binds that `DBMS_XPLAN` shows with `+PEEKED_BINDS`.

## 6.2 Getting the actual plan

An estimated plan shows what the optimizer intends. An actual plan shows what happened: the rows each step produced, how many times it ran, and where the time and buffers went. Capture the actual plan whenever you can, either by running the statement once with runtime statistics or from the engine's record of recent executions. Running a statement to get its actual plan really executes it, so for data changes use a copy of the data, or a transaction you roll back.

POSTGRESQL **EXPLAIN ANALYZE and auto_explain**

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT o.order_id, o.total_amount
FROM   orders o
WHERE  o.customer_id = 42 AND o.status = 'OPEN';

-- ANALYZE really runs the statement. For data changes, wrap it in a transaction:
BEGIN;
EXPLAIN (ANALYZE, BUFFERS) UPDATE orders SET status = 'CLOSED' WHERE order_id = 1001;
ROLLBACK;
```

For statements with parameters, `EXPLAIN (GENERIC_PLAN)` (PostgreSQL 16+) shows the generic plan without values. To catch slow plans in production, load `auto_explain`:

```
# postgresql.conf (add auto_explain to whatever is already listed)
shared_preload_libraries = 'pg_stat_statements,auto_explain'
auto_explain.log_min_duration = '500ms'
auto_explain.log_analyze = on      # actual row counts; adds overhead
auto_explain.log_timing = off      # keeps the overhead of log_analyze low
auto_explain.log_buffers = on
auto_explain.sample_rate = 0.1     # explain one statement in ten
```

MYSQL **EXPLAIN ANALYZE and the optimizer trace**

```
EXPLAIN FORMAT=TREE
SELECT o.order_id, o.total_amount FROM orders o WHERE o.customer_id = 42 AND o.status =
'OPEN';

-- 8.0.18+: runs the statement and reports actual rows and time per step
EXPLAIN ANALYZE
SELECT o.order_id, o.total_amount FROM orders o WHERE o.customer_id = 42 AND o.status =
'OPEN';

-- Why did the optimizer choose this plan?
SET optimizer_trace = 'enabled=on';
SELECT o.order_id FROM orders o WHERE o.customer_id = 42 AND o.status = 'OPEN';
SELECT trace FROM information_schema.optimizer_trace;
SET optimizer_trace = 'enabled=off';
```

SQL SERVER **Actual execution plans and Query Store**

```
SET STATISTICS IO, TIME ON;   -- reads per table and CPU/elapsed time in the Messages tab
SET STATISTICS XML ON;        -- returns the actual execution plan with the results
EXEC dbo.get_open_orders @customer_id = 42;
SET STATISTICS XML OFF;
```

In SQL Server Management Studio, "Include Actual Execution Plan" (Ctrl+M) does the same. Query Store keeps the plans each query used over time (`sys.query_store_plan`). For a query that is running right now, `sys.dm_exec_query_statistics_xml` shows its in-flight plan with live row counts (lightweight profiling, on by default since 2019); `sys.dm_exec_query_plan_stats` returns the last actual plan when the `LAST_QUERY_PLAN_STATS` database setting is on (2019+).

ORACLE **DBMS_XPLAN and SQL trace, without the packs**

```
-- In SQL*Plus or SQLcl, turn DBMS_OUTPUT off first, or DISPLAY_CURSOR shows the wrong
cursor
SET SERVEROUTPUT OFF

-- Run the statement once with runtime statistics for this execution
SELECT /*+ GATHER_PLAN_STATISTICS */ o.order_id, o.total_amount
FROM   orders o
WHERE  o.customer_id = :cust_id
AND    o.status = 'OPEN';

-- Then, in the same session: estimated and actual rows for each step
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, NULL, 'ALLSTATS LAST +PEEKED_BINDS'));

-- For a statement already in the cursor cache (no runtime row counts unless they were
collected)
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR('&sql_id', NULL, 'TYPICAL +PEEKED_BINDS'));
```

In the ALLSTATS output, `E-Rows` is the estimate **per start** of an operation, while `A-Rows` is the actual total across all starts. Compare `E-Rows × Starts` with `A-Rows`. On 12c and later, add `+ADAPTIVE` to the format to see adaptive plan decisions. All of this is free; Real-Time SQL Monitoring (`V$SQL_MONITOR` and its reports) needs the Tuning Pack. For a complete picture of one session's work, including waits per call, SQL trace with TKPROF is free and remains one of the most precise tools Oracle offers:

```
EXEC DBMS_MONITOR.SESSION_TRACE_ENABLE(session_id => 123, serial_num => 4567, -
                                       waits => TRUE, binds => TRUE);
-- ...let the workload run, then:
EXEC DBMS_MONITOR.SESSION_TRACE_DISABLE(session_id => 123, serial_num => 4567);
-- On the server: tkprof <tracefile>.trc report.txt sort=exeela,fchela
```

CLOUD **Trace files without a server shell**

On a managed service you can't run TKPROF or read logs on the server. Download trace and log files through the provider's console or API (on RDS for Oracle, `rdsadmin.rds_file_util.read_text_file` also reads them), and run TKPROF or your log analyzer on your own machine.

## 6.3 Reading the plan

Read a plan from the inside out, but **start with the numbers**: where do estimated and actual rows part ways, and where does the time go?

| What you see | What it usually means | Typical fix |
|---|---|---|
| Estimated rows far from actual (10× or more) | A bad cardinality estimate, and every decision after it is built on sand | Statistics, histograms, extended statistics, predicate rewrites (section 6.4) |
| Full scan of a large table that returns few rows | A missing or unusable index | Add an index, or make the predicate sargable (Chapter 7) |
| Nested loops with a large outer row count | An underestimate, or no index on the inner join column | Fix the estimate; index the inner join column; let the optimizer choose a hash join |
| Sort or hash spilling to temp or disk | Too many rows reaching the operation, or too little memory for it | Filter earlier; support the order with an index; raise memory for that session |
| Many lookups back to the table, one for each index entry | The index finds rows but doesn't cover the query | A covering index, or accept a scan when many rows qualify |
| A conversion function or cast wrapped around a column | A data type mismatch that disables an index | Match parameter and column types |
| A filter applied late, after many rows were read | A predicate the access path can't use | Rewrite the predicate; index the filtered column |
| One step executed thousands of times (high Starts or loops) | A correlated subquery or a function called per row | Rewrite as a join or a set-based aggregate |

| What you see | What it usually means | Typical fix |
|---|---|---|
| A parallel plan for a short OLTP query | A parallelism threshold that is too low, or an overestimate | Statistics; parallelism settings (Chapter 11) |

ENGINE NOTES **What these look like in each plan**

- **PostgreSQL:** late filters show as "Rows Removed by Filter", repeated steps as `loops`, spills as "Sort Method: external merge" or a hash with more than one batch, and conversions as casts in the filter or index condition.
- **MySQL:** late filters show as "Using where" after many rows examined; `EXPLAIN ANALYZE` reports rows and loops for each step. Temporary tables and sorts show as "Using temporary" and "Using filesort", and disk spills in the `Created_tmp_disk_tables` counter.
- **SQL Server:** lookups show as Key Lookup or RID Lookup operators, conversions as `CONVERT_IMPLICIT` with a plan warning, spills as sort or hash warnings that tempdb was used, and repeated steps in the number of executions.
- **Oracle:** lookups show as `TABLE ACCESS BY INDEX ROWID`, late filters in the filter predicates rather than the access predicates, conversions as `TO_NUMBER` or `INTERNAL_FUNCTION` around a column, repeated steps as `Starts`, and spills in the `Used-Tmp` column.

## 6.4 Cardinality: the root of most bad plans

The optimizer chooses join order, join method and access path from its row estimates. When the estimates are wrong, good plans happen only by luck.

| Cause | Remedy |
|---|---|
| Stale statistics | Fix the maintenance job; gather statistics after big loads and purges |
| Skewed column without a histogram | Build a histogram on the column, or a finer one |
| Correlated columns (city and postal code, status and region) | Statistics on the column combination |
| Expressions on columns | Statistics on the expression, or an index on it |
| Values beyond the histogram (ascending keys such as today's date) | Gather statistics more often on fast-growing tables |
| Parameter sensitivity (one plan can't serve all values) | Let the engine keep several plans, recompile infrequent calls, or split very different cases into separate code paths (Chapter 9) |
| Table variables and functions that return tables, with fixed row guesses | Temporary tables, which have statistics |

After a major upgrade, check whether the engine kept the optimizer statistics, and if it didn't, gather them before users arrive. On every engine, gather statistics after bulk loads and large purges rather than waiting for the automatic job.

POSTGRESQL **Statistics remedies**

- `ANALYZE`, which autovacuum also runs automatically. For skewed columns, raise the sample with `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS 1000`.
- `CREATE STATISTICS` for correlated columns (10+, with `mcv` from 12, and on expressions from 14). Expression indexes gather statistics on their expression.
- `plan_cache_mode` (12+) chooses between generic and custom plans for prepared statements.
- Before PostgreSQL 18, `pg_upgrade` doesn't carry planner statistics across, so run `vacuumdb --analyze-in-stages` right after an upgrade.

```
-- Multivariate statistics and a larger sample for one column
CREATE STATISTICS orders_status_region (dependencies, ndistinct, mcv) ON status, region
FROM orders;
ALTER TABLE orders ALTER COLUMN customer_id SET STATISTICS 1000;
ANALYZE orders;
```

MYSQL **Statistics remedies**

- `ANALYZE TABLE`. InnoDB's persistent statistics also recalculate automatically after about 10% of a table changes.
- Histograms (8.0+) with `ANALYZE TABLE ... UPDATE HISTOGRAM`. They are not refreshed automatically by default, so schedule them.
- There are no multi-column statistics; an index on both columns gives the optimizer combined information.

```
-- Histograms (8.0+; not refreshed automatically by default, so schedule them)
ANALYZE TABLE orders UPDATE HISTOGRAM ON status, region WITH 64 BUCKETS;
```

SQL SERVER **Statistics remedies**

- `UPDATE STATISTICS`, with `FULLSCAN` for skewed data. Statistics always carry a histogram (up to 200 steps), so increase the sample rather than asking for a histogram.
- Multi-column statistics hold densities for the combination but a histogram on the first column only. For expressions, create statistics on a computed column.
- The newer cardinality estimator (2014+) handles ascending keys better than the legacy one.
- For parameter sensitivity: Parameter Sensitive Plan optimization (2022), optional parameter plan optimization (2025), and `OPTION (RECOMPILE)` for infrequent calls.
- Table variables and multi-statement functions get better estimates from table variable deferred compilation (2019+) and interleaved execution (2017+); temporary tables have real statistics.

```
-- Full-scan statistics and a multi-column statistics object
UPDATE STATISTICS dbo.orders WITH FULLSCAN;
CREATE STATISTICS st_orders_status_region ON dbo.orders (status, region);
```

ORACLE **Statistics remedies**

- `DBMS_STATS`; the automatic job gathers stale statistics in the maintenance window. `METHOD_OPT => 'FOR ALL COLUMNS SIZE AUTO'` creates histograms for skewed columns that are used in predicates.
- Extended statistics (`DBMS_STATS.CREATE_EXTENDED_STATS`) cover column groups and expressions; a function-based index also helps with expressions.
- Adaptive cursor sharing (11g+) keeps several plans for statements that are sensitive to bind values.

```
-- Gather table statistics, then add a column group for correlated columns
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS(ownname => 'APP', tabname => 'ORDERS',
                                method_opt => 'FOR ALL COLUMNS SIZE AUTO', cascade =>
TRUE);
END;
/
SELECT DBMS_STATS.CREATE_EXTENDED_STATS('APP', 'ORDERS', '(STATUS, REGION)') FROM dual;
-- ...then gather statistics again so the column group gets values
```

## 6.5 Plan stability tools: a stopgap, not a cure

Sometimes you need a good plan today and the real fix takes a release cycle. Every engine offers a way to pin or steer a plan. Use it, write down why, and come back to it.

| Engine | Tools |
|---|---|
| PostgreSQL | No built-in plan forcing. Fix the root cause with statistics and indexes; `plan_cache_mode` (12+) for generic or custom plans of prepared statements; the third-party `pg_hint_plan` extension where hints are unavoidable |
| MySQL | Optimizer hints (`/*+ ... */`), index hints (`USE INDEX`, `FORCE INDEX`, `IGNORE INDEX`), `optimizer_switch`; histograms to correct estimates |
| SQL Server | Query Store plan forcing (2016+); Query Store hints (2022); automatic plan correction (2017+, Enterprise Edition); `USE HINT`, `OPTIMIZE FOR` and `OPTION (RECOMPILE)`; plan guides |
| Oracle | SQL plan baselines (SQL Plan Management, part of Enterprise Edition without a pack; Standard Edition 2 has restrictions); hints; adaptive cursor sharing. SQL profiles and the SQL Tuning Advisor need the Tuning Pack |

ENGINE NOTES **Pinning a known good plan**

### SQL Server

```
-- Force a plan that Query Store has seen for this query
EXEC sys.sp_query_store_force_plan @query_id = 1234, @plan_id = 5678;
```

**Oracle**

```
-- Load the known good plan from the cursor cache into a SQL plan baseline
DECLARE
  n PLS_INTEGER;
BEGIN
  n := DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE(sql_id          => '&sql_id',
                                             plan_hash_value => &good_plan_hash);
END;
/
```

WATCH OUT **Forced plans age badly**

A plan that is perfect for today's data can be terrible after a year of growth. Keep a list of every forced plan and hint with the reason and an owner, and review it every quarter and after every upgrade.

REINDB **Where ReinDB helps**

Click a statement id to see its time per execution by plan over 1 to 30 days, so plan changes are visible at a glance. The analysis reads the current plan and statistics for disk reads, temp spills, rows examined per row returned, full scans and N+1 patterns, and on Oracle it compares estimated with actual rows step by step and flags misestimates of 10× or more. The fix arrives as `fix_<id>.sql`, with every line commented out and an undo for each step. Pre-check lets you paste a query, procedure or DDL change before it ships; ReinDB only asks for the plan (it never runs the code) and returns Pass, Review or Risky, with the quietest hour to deploy.

# 7 Tuning joins and WHERE clauses

Most slow statements are slow for a handful of reasons that repeat across every engine: predicates that hide columns from indexes, type mismatches, joins that multiply rows, and queries that ask for far more than they need. This chapter covers the patterns and their rewrites.

## 7.1 Keep predicates sargable

A predicate is *sargable* (search-argument-able) when the engine can use an index to find the matching rows directly. The rule is simple: **leave the indexed column bare** and do the arithmetic on the other side.

```
-- Hides order_date from a normal index: a function wraps the column (each engine's version is
in the note below)
WHERE TRUNC(order_date) = DATE '2026-10-31'

-- Index-friendly: a half-open range on the bare column
WHERE order_date >= DATE '2026-10-31'
  AND order_date <  DATE '2026-11-01'
```

| Instead of | Write |
|---|---|
| `WHERE YEAR(order_date) = 2026` | `WHERE order_date >= '2026-01-01' AND order_date < '2027-01-01'` |
| `WHERE total_amount * 1.1 > 100` | `WHERE total_amount > 100 / 1.1` |
| `WHERE UPPER(email) = :email` | Store a normalized copy, or create a function-based or expression index on `UPPER(email)` |
| `WHERE COALESCE(discount, 0) = 0` | `WHERE (discount = 0 OR discount IS NULL)` |
| `WHERE SUBSTR(sku, 1, 3) = 'ABC'` | `WHERE sku LIKE 'ABC%'` |
| `WHERE order_id + 0 = :id` (an old trick to disable an index) | Remove it, and use a hint if you really must steer the plan |

ENGINE NOTES **The same trap, engine by engine**

- **PostgreSQL:** `order_date::date` or `date_trunc('day', order_date)`. An expression index on exactly the same expression is the alternative when you can't change the query.
- **MySQL:** `DATE(order_date)` or `YEAR(order_date)`. MySQL 8.0.13 and later can index an expression too, for example `CREATE INDEX ix_orders_day ON orders ((DATE(order_date)))`.
- **SQL Server:** `YEAR()`, `CONVERT()` or `DATEADD()` on the column. As a special case, `CAST(datetime_column AS date) = @d` can still seek; don't generalize from it. An indexed computed column is the alternative.
- **Oracle:** `TRUNC(order_date)` or `TO_CHAR(order_date, ...)`. A function-based index on the same expression is the alternative.

## 7.2 Data type mismatches and implicit conversions

When the two sides of a comparison have different types, the engine converts one of them. If it converts the **column**, it has to do it for every row, and the index on that column is useless.

The usual sources are application drivers that bind every string as Unicode or every number as a decimal, and join columns that were declared differently in different tables.

**Join keys should have identical types, lengths and collations on both sides.** Mixed integer sizes are usually handled well, but Unicode against non-Unicode strings, or fixed-length against variable-length strings, cause real trouble. Fix it in the schema, not in every query.

> ENGINE NOTES **How each engine handles a mismatch**
>
> - **PostgreSQL:** comparing an integer column with a numeric value (`WHERE id = 42.0`, or a driver that binds `numeric`) casts the column and skips its index. `text` and `varchar` mix freely.
> - **MySQL:** comparing a string column with a number (`WHERE phone = 5551234`) converts every row and scans the table. Joining columns with different character sets or collations can stop index use on the join.
> - **SQL Server:** comparing a `VARCHAR` column with an `NVARCHAR` parameter puts `CONVERT_IMPLICIT` on the column and can turn a seek into a scan, especially with SQL collations. Many drivers send strings as Unicode by default (for example the Microsoft JDBC driver's `sendStringParametersAsUnicode=true`). Match the parameter type to the column, or change the driver setting.
> - **Oracle:** comparing a `VARCHAR2` column with a number makes Oracle apply `TO_NUMBER` to the column. The index is ignored, and the query can fail with ORA-01722 the day a non-numeric value appears. Bind strings as strings. `CHAR` against `VARCHAR2` compares with nonpadded rules, so trailing blanks decide whether values match.

## 7.3 Patterns that hide rows from indexes

- **Leading wildcards.** `LIKE '%term'` and `LIKE '%term%'` can't use a normal B-tree index. Use a trigram index where the engine offers one, or full-text search. Full-text search matches words, not arbitrary substrings, so check that the semantics fit.
- **OR across different columns.** `WHERE customer_id = :c OR referrer_id = :c` may force a scan. Optimizers can sometimes combine two indexes, but not always. A `UNION ALL` of two indexed queries, written so it doesn't return duplicates, is the dependable rewrite.
- **NOT IN with a nullable subquery.** If the subquery returns a single NULL, `NOT IN` returns no rows at all. It also blocks some anti-join optimizations. Use `NOT EXISTS`.
- **Negative predicates.** `<>`, `NOT LIKE` and `NOT IN` rarely use indexes well, and they usually match most rows anyway.
- **Catch-all filters.** `WHERE (:status IS NULL OR status = :status) AND (:region IS NULL OR region = :region)` gives one plan for every combination, usually a bad one. Build the statement

dynamically with only the filters that were supplied, still using bind variables, or recompile on every call when calls are infrequent.

> ENGINE NOTES **Search, OR and catch-all filters**
>
> - **PostgreSQL:** `pg_trgm` trigram indexes with GIN support `LIKE '%term%'`; full-text search uses `tsvector`. Bitmap OR combines indexes.
> - **MySQL:** `FULLTEXT` indexes for word search; index merge combines indexes.
> - **SQL Server:** Full-Text Search; index union combines indexes. For catch-all filters, add `OPTION (RECOMPILE)` when calls are infrequent. SQL Server 2025 can also keep separate plans for this pattern (optional parameter plan optimization) at the newest compatibility level.
> - **Oracle:** Oracle Text for word search, included in every edition; OR expansion combines indexes.

## 7.4 Joins: get the shape right and the method follows

Optimizers pick between three join methods:

- **Nested loops:** for each row from the outer input, look up matches in the inner input. Excellent when the outer input is small and the inner join column is indexed.
- **Hash join:** build a hash table from the smaller input, probe it with the larger one. Excellent for large, unsorted inputs joined on equality. Needs memory, and spills to temp when it doesn't get enough.
- **Merge join:** walk two inputs already sorted on the join key. Excellent when indexes or earlier steps provide the order.

When the "wrong" method shows up, the cause is nearly always a bad row estimate (Chapter 6) or a missing index, not the optimizer's taste. Give it accurate statistics and the right indexes, and it chooses well.

**Index foreign keys.** An unindexed foreign key forces a full scan of the child table whenever a parent row is deleted or its key changes, because the engine must check for children. Some engines also lock the whole child table during those operations, which shows up as blocking far from the statement that caused it.

> ENGINE NOTES **Unindexed foreign keys**
>
> - **PostgreSQL** and **SQL Server** don't create an index for a foreign key.
> - **MySQL** InnoDB requires an index on foreign key columns and creates one if needed.
> - **Oracle** doesn't create one either, and an unindexed foreign key makes parent deletes and key updates lock the whole child table.

**PostgreSQL**

```
-- Single-column foreign keys with no index that starts with that column
SELECT c.conrelid::regclass AS table_name, c.conname, a.attname
FROM   pg_constraint c
JOIN   pg_attribute a ON a.attrelid = c.conrelid AND a.attnum = c.conkey[1]
WHERE  c.contype = 'f'
  AND  array_length(c.conkey, 1) = 1
  AND  NOT EXISTS (
         SELECT 1 FROM pg_index i
         WHERE  i.indrelid = c.conrelid
           AND  i.indkey[0] = c.conkey[1]);
```

**Oracle**

```
-- Foreign key columns with no index starting with the same column at the same position
-- (a quick check; multi-column keys need all their columns leading one index)
SELECT c.owner, c.table_name, c.constraint_name, cc.column_name, cc.position
FROM   dba_constraints c
JOIN   dba_cons_columns cc
       ON cc.owner = c.owner AND cc.constraint_name = c.constraint_name
WHERE  c.constraint_type = 'R'
  AND  c.owner = 'APP'
  AND  NOT EXISTS (
         SELECT 1 FROM dba_ind_columns ic
         WHERE  ic.table_owner     = cc.owner
           AND  ic.table_name      = cc.table_name
           AND  ic.column_name     = cc.column_name
           AND  ic.column_position = cc.position);
```

More join guidance:

- **Filter early.** Reduce rows before joining. Pre-aggregate in a derived table before joining a large detail table to a small dimension.
- **Mind the outer-join trap.** A condition on the outer table's columns in the `WHERE` clause quietly turns a `LEFT JOIN` into an inner join. If you mean "keep unmatched rows", put the condition in the `ON` clause.
- **Don't join and then DISTINCT.** If you only need to know that a match exists, use `EXISTS` (a semi-join). Joining and then removing duplicates does extra work and hides the intent.
- **Flatten deep view stacks.** Optimizers search a limited number of join orders, and views built on views on views can push a query past those limits.

> POSTGRESQL **Join search limits**
>
> `join_collapse_limit` and `from_collapse_limit` default to 8, and the planner switches to a genetic search at 12 tables (`geqo_threshold`).

## 7.5 Ask for less

- **Select only the columns you need.** It enables covering indexes, cuts I/O and network traffic, and avoids reading LOBs nobody looks at.
- **Paginate with keysets, not offsets.** `OFFSET 100000` reads and throws away 100,000 rows on every page. Remember the last key and seek past it instead:

```
-- Keyset pagination: seek past the last row of the previous page
SELECT order_id, order_date, total_amount
FROM   orders
WHERE  order_date < :last_date
   OR (order_date = :last_date AND order_id < :last_id)
ORDER  BY order_date DESC, order_id DESC
FETCH  FIRST 50 ROWS ONLY;   -- standard SQL; each engine's top-N syntax is in the note below
```

An index on `(order_date, order_id)` makes every page as fast as the first.

- **Use the engine's top-N syntax,** so the optimizer knows that only the first rows are needed and can stop early.

> ENGINE NOTES **Top-N and row-value syntax**
>
> - **PostgreSQL:** `LIMIT n` or `FETCH FIRST n ROWS ONLY`. A row-value comparison, `WHERE (order_date, order_id) < (:last_date, :last_id)`, is shorter and can use the index directly.
> - **MySQL:** `LIMIT n`. Row-value comparisons work too.
> - **SQL Server:** `TOP (n)`, or `OFFSET 0 ROWS FETCH NEXT n ROWS ONLY` after the `ORDER BY` (2012+). There are no row-value comparisons, so write the condition out as above.
> - **Oracle:** `FETCH FIRST n ROWS ONLY` (12c+). There is no row-value < comparison, so write the condition out. On 11g, filter on `ROWNUM` around an ordered subquery:
>
> ```
> SELECT * FROM (
>   SELECT order_id, order_date, total_amount
>   FROM   orders
>   WHERE  customer_id = :c
>   ORDER  BY order_date DESC)
> WHERE ROWNUM <= 10;
> ```

- **Replace per-row subqueries with set operations.** A scalar subquery in the select list can run once per row:

```
-- Before: one subquery execution per customer row
SELECT c.customer_id, c.name,
       (SELECT MAX(o.order_date) FROM orders o WHERE o.customer_id = c.customer_id) AS
last_order
FROM   customers c;

-- After: aggregate once, then join
SELECT c.customer_id, c.name, o.last_order
FROM   customers c
LEFT   JOIN (SELECT customer_id, MAX(order_date) AS last_order
             FROM   orders
             GROUP  BY customer_id) o
       ON o.customer_id = c.customer_id;
```

The rewrite wins when most customers are returned. When only a few are, the original with an index on `(customer_id, order_date)` can be faster, and many optimizers transform one into the other on their own. Check the plan rather than assuming.

- **Use window functions instead of self-joins** for running totals, rankings and "latest row per group" (every current release of the four engines supports them; MySQL from 8.0):

```
SELECT *
FROM  (SELECT o.*,
              ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC) AS rn
       FROM   orders o) x
WHERE  rn = 1;
```

## 7.6 Bind variables and parameterization

Literal values make every statement text unique, so the engine parses and optimizes each one from scratch and fills its plan cache with single-use plans. Use bind variables (parameters) in application code. They also protect against SQL injection.

> ENGINE NOTES **Literals and plan caching**
>
> - **PostgreSQL:** Prepared statements save planning time. After five executions, PostgreSQL may switch from custom plans to a generic plan; `plan_cache_mode` (12+) controls that choice.
> - **MySQL:** There is no shared plan cache, but prepared statements still save parsing and protect against SQL injection.
> - **SQL Server:** Ad hoc literals fill the plan cache with single-use plans. Turn on `optimize for ad hoc workloads`, parameterize in the application (`sp_executesql` with parameters), and consider forced parameterization per database.
> - **Oracle:** Literal values make every statement unique, so each one is hard parsed. That burns CPU and causes shared pool and library cache contention (waits such as `library cache: mutex X` and `cursor: pin S wait on X`). Use bind variables. `CURSOR_SHARING = FORCE` is an emergency measure for applications you can't change, not a design.

Bind variables plus skewed data equals parameter sensitivity: one plan for very different values. Chapter 9 covers the remedies.

## 7.7 Batch large data changes

One huge `UPDATE` or `DELETE` holds locks for its whole duration, generates large amounts of undo, redo, WAL or transaction log, and can stall replicas. Break it into batches with a commit after each (when the business rules allow), or better, design the table so old data leaves by dropping a partition (Chapter 10).

> ENGINE NOTES **Purging in batches**
>
> **PostgreSQL**
>
> ```
> -- One batch; repeat from a job or a procedure (procedures can COMMIT in 11+)
> DELETE FROM audit_log
> WHERE  id IN (SELECT id FROM audit_log
>               WHERE  logged_at < now() - interval '13 months'
>               LIMIT  5000);
> ```

**MySQL**

```
-- One batch; repeat until no rows are affected
DELETE FROM audit_log
WHERE  logged_at < NOW() - INTERVAL 13 MONTH
LIMIT  5000;
```

**SQL Server**

```
-- Purge in batches of 5,000 rows
DECLARE @rows int = 1;
WHILE @rows > 0
BEGIN
    DELETE TOP (5000) FROM dbo.audit_log
    WHERE logged_at < DATEADD(month, -13, SYSUTCDATETIME());
    SET @rows = @@ROWCOUNT;
END;
```

**Oracle**

```
-- Purge in batches of 5,000 rows, committing after each
BEGIN
  LOOP
    DELETE FROM audit_log
    WHERE  logged_at < ADD_MONTHS(SYSDATE, -13)
    AND    ROWNUM <= 5000;
    EXIT WHEN SQL%ROWCOUNT = 0;
    COMMIT;
  END LOOP;
  COMMIT;
END;
/
```

Make sure the column you filter on is indexed, or each batch rescans the table from the start.

# 8 Indexing: when, where, which type, and what goes wrong

Indexes are the most powerful tuning tool you have, and the easiest to overuse. Every index speeds up some reads and slows down every write to its table. Good indexing is a budget, not a reflex.

## 8.1 When an index helps, and when it doesn't

An index helps when:

- A query reads a **small fraction** of a large table through a selective predicate.
- It supports the **inner side of a nested loop** join (the join column of the table being looked up).
- It delivers rows **already in the order** a query needs for `ORDER BY`, `GROUP BY` or a merge join, avoiding a sort.
- It **covers** a hot query completely, so the table itself is never touched.
- It enforces **uniqueness**, or supports a **foreign key** (Chapter 7).

An index doesn't help, or costs more than it saves, when:

- The predicate matches a large share of the table. Past a break-even point, often in the low single-digit percent of rows, a full scan is cheaper than millions of individual lookups. Where exactly that point lies depends on how well the table's physical order matches the index, which statistics such as Oracle's clustering factor and PostgreSQL's correlation measure.
- The table is small enough to scan in a few pages.
- The table is write-heavy and the index serves a rare query.
- The query wraps the column in a function or converts its type (Chapter 7), unless you index the expression itself.

## 8.2 Where: designing composite indexes

Column order is the most important decision in a composite index. A dependable rule for a given query:

1. **Columns compared with equality** (`=`, `IN` with few values) first.
2. **Then one range column** (`<`, `>`, `BETWEEN`, `LIKE 'abc%'`). Columns after a range column can't narrow the search further, though they can still filter or cover.
3. **Then columns used for ordering**, when they follow the equality columns, so the sort disappears.
4. **Then columns needed only to cover** the query, or put them in an `INCLUDE` clause where the engine supports it.

```
-- The query
SELECT order_id, total_amount
FROM   orders
WHERE  customer_id = :c AND status = 'OPEN' AND order_date >= :since
ORDER  BY order_date;

-- An index that finds the rows, returns them in order, and covers the query
CREATE INDEX ix_orders_cust_status_date
    ON orders (customer_id, status, order_date) INCLUDE (total_amount);
```

- **The leftmost prefix rule.** An index on `(a, b, c)` serves predicates on `a`, on `a, b`, and on `a, b, c`, but not on `b` alone. Some optimizers can *skip scan* past a leading column with few distinct values; treat that as a bonus, not a design.
- **"Most selective column first" is a myth** as a universal rule. For a single query with several equality predicates, their order matters little. What matters is which column order lets the same index serve the most important queries.

> ENGINE NOTES **Composite indexes, engine by engine**
>
> - **PostgreSQL:** `INCLUDE` columns from 11, and skip scan from 18. Index-only scans depend on the visibility map, which `VACUUM` maintains; on tables that vacuum rarely, "index-only" scans still visit the table.
> - **MySQL:** no `INCLUDE` clause, so add covering columns as the last key columns. Skip scan from 8.0.13. InnoDB secondary indexes contain the primary key: an index on `(customer_id)` is effectively `(customer_id, order_id)`, so it already covers queries that need only those columns.
> - **SQL Server:** `INCLUDE` columns. There is no skip scan, so the leading column decides whether an index can seek.
> - **Oracle:** no `INCLUDE` clause, so add covering columns as the last key columns. Index skip scan can help when the leading column has few distinct values.

## 8.3 Index types and where each fits

| Type | Best for | Engines | Watch out for |
|---|---|---|---|
| B-tree | Equality, ranges, sorting: the default for almost everything | All | Nothing special; it's the baseline |
| Bitmap | Low-cardinality columns in read-mostly warehouses, combining many predicates | Oracle Enterprise Edition | Never on tables with concurrent DML: one bitmap entry covers many rows, so locks collide |
| Function-based or expression | Queries that filter on an expression such as `UPPER(email)` | Oracle; PostgreSQL; SQL Server via an indexed computed column; MySQL 8.0.13+ functional key parts | The query must use exactly the indexed expression |
| Partial or filtered | A small, hot subset of rows (`status = 'OPEN'`) | PostgreSQL partial indexes; SQL Server filtered indexes | The query predicate must match the filter; in SQL Server, parameterized predicates often can't use a filtered index |

| Type | Best for | Engines | Watch out for |
|---|---|---|---|
| Covering (`INCLUDE`) | Hot queries that would otherwise look up the table for a few columns | SQL Server; PostgreSQL 11+ | Wider indexes cost more on every write |
| Clustered or index-organized | Range access by key; primary key lookups | SQL Server clustered indexes; InnoDB (always clustered by primary key); Oracle index-organized tables | Random keys (UUIDv4) cause page splits and poor cache locality; ever-increasing keys can cause last-page insert contention at very high concurrency (SQL Server 2019 adds `OPTIMIZE_FOR_SEQUENTIAL_KEY`) |
| Hash | Equality only | PostgreSQL (crash-safe since 10); SQL Server memory-optimized tables | No ranges and no ordering |
| GIN | JSONB, arrays, full text, trigram searches for `LIKE '%x%'` | PostgreSQL | Slower writes and larger size |
| GiST and SP-GiST | Geometric data, ranges, nearest-neighbor searches, exclusion constraints | PostgreSQL | Choose the operator class carefully |
| BRIN | Very large tables whose physical order follows a column, such as time | PostgreSQL | Useless when the physical order doesn't correlate with the column |
| Columnstore | Analytics over many rows: aggregates, scans, batch mode | SQL Server (clustered and nonclustered columnstore); Oracle Database In-Memory (a separately licensed option; 19c's Base Level allows up to 16 GB without it) | Not for single-row lookups; trickle inserts land in row-based delta stores first |
| Full-text | Word and phrase search | Oracle Text; SQL Server Full-Text Search; MySQL `FULLTEXT`; PostgreSQL `tsvector` with GIN | Word semantics, language rules and update latency |
| Reverse key | Spreading sequential inserts across blocks on Oracle RAC | Oracle | Range scans on the key become impossible |

> ORACLE **Partial indexes the classic way**
>
> Oracle has no partial indexes for ordinary tables, but a classic trick gets the same effect. Entries whose key columns are all NULL are not stored in a B-tree index, so index an expression that is NULL for the rows you don't care about, and query with the same expression:
>
> ```
> CREATE INDEX ix_orders_open ON orders (CASE WHEN status = 'OPEN' THEN customer_id END);
>
> SELECT order_id
> FROM   orders
> WHERE  CASE WHEN status = 'OPEN' THEN customer_id END = :c;
> ```
>
> Oracle 12c added partial indexing for partitioned tables, with indexing switched on or off per partition.

## 8.4 Finding index candidates

Every engine can point you at candidates: tables read mostly by full scans, the statements behind those scans, and in some engines the optimizer's own suggestions. Treat all of them as leads to investigate, not as indexes to create.

POSTGRESQL **Tables read by large sequential scans**

Find the tables first, then the statements behind them in `pg_stat_statements`.

```
SELECT relname, seq_scan, seq_tup_read, idx_scan,
       seq_tup_read / NULLIF(seq_scan, 0) AS avg_rows_per_seq_scan
FROM   pg_stat_user_tables
WHERE  seq_scan > 0
ORDER  BY seq_tup_read DESC
LIMIT  20;
```

MYSQL **Statements that ran without a usable index**

```
SELECT query, exec_count, no_index_used_count, no_good_index_used_count,
       rows_examined_avg, rows_sent_avg
FROM   sys.statements_with_full_table_scans
ORDER  BY no_index_used_count DESC
LIMIT  20;
```

SQL SERVER **Missing index suggestions**

The optimizer records the indexes it wished it had:

```
SELECT TOP (20)
       d.statement AS table_name,
       d.equality_columns, d.inequality_columns, d.included_columns,
       s.user_seeks, s.avg_user_impact,
       s.user_seeks * s.avg_total_user_cost * (s.avg_user_impact / 100.0) AS est_benefit
FROM   sys.dm_db_missing_index_details d
JOIN   sys.dm_db_missing_index_groups g      ON g.index_handle = d.index_handle
JOIN   sys.dm_db_missing_index_group_stats s ON s.group_handle = g.index_group_handle
WHERE  d.database_id = DB_ID()
ORDER  BY est_benefit DESC;
```

### Treat the suggestions as leads, not orders

They ignore existing indexes and each other, often overlap, list columns in table order rather than the best order, include every column a query touched, and disappear at restart. Consolidate them into a few well-designed indexes, and check each against the queries it is meant to serve.

ORACLE **Heavy cached statements with full table scans**

```
SELECT DISTINCT p.sql_id, p.object_owner, p.object_name, s.executions, s.buffer_gets
FROM   v$sql_plan p
JOIN   v$sqlstats s ON s.sql_id = p.sql_id AND s.plan_hash_value = p.plan_hash_value
WHERE  p.operation = 'TABLE ACCESS' AND p.options = 'FULL'
  AND  p.object_owner NOT IN ('SYS', 'SYSTEM')
ORDER  BY s.buffer_gets DESC;
```

**Test new indexes safely.** Where the engine supports invisible indexes, create the index invisible first. It is maintained but ignored by the optimizer unless a session asks for it, so you can test it without changing anyone else's plans.

ENGINE NOTES **Invisible and hypothetical indexes**

- **PostgreSQL:** no invisible indexes. The hypopg extension creates hypothetical indexes that only `EXPLAIN` can see, so you can test before building anything.
- **SQL Server:** no invisible indexes. Test on a copy with production-like data.

**MySQL (8.0+)**

```
ALTER TABLE orders ADD INDEX ix_orders_cust_status (customer_id, status) INVISIBLE;
SET SESSION optimizer_switch = 'use_invisible_indexes=on';
ALTER TABLE orders ALTER INDEX ix_orders_cust_status VISIBLE;
```

**Oracle (11g+)**

```
CREATE INDEX ix_orders_cust_status ON orders (customer_id, status) INVISIBLE;
ALTER SESSION SET optimizer_use_invisible_indexes = TRUE;   -- test in this session only
ALTER INDEX ix_orders_cust_status VISIBLE;                  -- when you're satisfied
```

## 8.5 Finding unused and duplicate indexes

Unused indexes still cost writes, memory, storage, backup time and replication bandwidth.

POSTGRESQL **Indexes never scanned**

```
-- Non-unique indexes never scanned since statistics were last reset
SELECT s.schemaname, s.relname, s.indexrelname, s.idx_scan,
       pg_size_pretty(pg_relation_size(s.indexrelid)) AS index_size
FROM   pg_stat_user_indexes s
JOIN   pg_index i ON i.indexrelid = s.indexrelid
WHERE  s.idx_scan = 0
  AND  NOT i.indisunique
ORDER  BY pg_relation_size(s.indexrelid) DESC;
```

MYSQL **Unused and redundant indexes**

```
-- sys schema views (since the server started)
SELECT * FROM sys.schema_unused_indexes;
SELECT * FROM sys.schema_redundant_indexes;
```

SQL SERVER **Indexes with no reads since the last restart**

```
-- Nonclustered indexes with no reads since the last restart
SELECT OBJECT_NAME(i.object_id) AS table_name, i.name AS index_name,
       u.user_seeks, u.user_scans, u.user_lookups, u.user_updates
FROM   sys.indexes i
LEFT   JOIN sys.dm_db_index_usage_stats u
       ON u.object_id = i.object_id AND u.index_id = i.index_id AND u.database_id =
DB_ID()
WHERE  OBJECTPROPERTY(i.object_id, 'IsUserTable') = 1
  AND  i.type_desc = 'NONCLUSTERED'
  AND  i.is_primary_key = 0 AND i.is_unique_constraint = 0
  AND  ISNULL(u.user_seeks + u.user_scans + u.user_lookups, 0) = 0
ORDER  BY u.user_updates DESC;
```

ORACLE **Index usage tracking**

On Oracle 12.2 and later, `DBA_INDEX_USAGE` records index use automatically (by sampling, by default, so treat "never used" as a strong hint rather than proof). On 11g, `ALTER INDEX ... MONITORING USAGE` sets a used/not-used flag that the index owner can read in `V$OBJECT_USAGE`.

WATCH OUT **Before you drop an index**

- Check how long the counters have been collecting. A server restarted yesterday knows nothing about month-end jobs.
- Usage on read replicas is usually not counted on the primary, and often not visible from it at all.
- Keep indexes that enforce primary keys and uniqueness, and indexes that support foreign keys, even if no query "uses" them.
- Make the index invisible first where the engine supports it, or keep its definition ready to recreate, and watch the top statements for a full business cycle.

**Duplicates.** An index on `(a)` is usually redundant when `(a, b)` exists, unless it is unique, much narrower for a very hot query, or supports a foreign key on its own.

## 8.6 Build and maintain indexes without hurting production

Building an index on a busy table can block writes for as long as the build runs, unless you build it online. Every engine has an online option, but the edition rules and the failure behavior differ.

ENGINE NOTES **Online index builds**

**PostgreSQL**

It can't run inside a transaction block, and if it fails it leaves an `INVALID` index that you must drop.

```
CREATE INDEX CONCURRENTLY ix_orders_cust ON orders (customer_id, status) INCLUDE
(total_amount);
```

#### MySQL

InnoDB online DDL:

```
ALTER TABLE orders ADD INDEX ix_orders_cust (customer_id, status), ALGORITHM=INPLACE,
LOCK=NONE;
```

#### SQL Server

`ONLINE = ON` needs Enterprise Edition; resumable index creation arrived in 2019.

```
CREATE INDEX ix_orders_cust ON dbo.orders (customer_id, status) INCLUDE (total_amount)
    WITH (ONLINE = ON, RESUMABLE = ON);
```

#### Oracle

Online index builds are an Enterprise Edition feature.

```
CREATE INDEX ix_orders_cust ON orders (customer_id, status) ONLINE;
```

**Rebuilding indexes on a schedule because of "fragmentation" is mostly wasted work** on modern SSD and SAN storage, and it generates large amounts of log. Rebuild for a reason, such as wasted space after heavy deletes or a measurable gain in reads:

ENGINE NOTES **When a rebuild is worth it**

- **PostgreSQL:** B-tree bloat after heavy updates or deletes; `REINDEX CONCURRENTLY` (12+). Good autovacuum settings prevent most of it.
- **MySQL:** `OPTIMIZE TABLE` rebuilds the table and its indexes; rarely needed outside big purges.
- **SQL Server:** low page density wastes memory and I/O; rebuild or reorganize selectively. Remember that a rebuild also refreshes that index's statistics with a full scan, which is often the real reason a rebuild "fixed" something.
- **Oracle:** rebuilds are rarely needed; consider `COALESCE` or `SHRINK SPACE` after large purges.

## 8.7 Common indexing pitfalls

- Over-indexing write-heavy tables: every index adds work, log and memory to each write.
- Overlapping indexes such as `(a)`, `(a, b)` and `(a, b, c)` on the same table.
- The right columns in the wrong order, or a low-cardinality flag indexed on its own.
- Functions and implicit conversions that make the index invisible to the query.
- Unindexed foreign keys.
- Random keys (UUIDv4) as the clustered key; prefer a sequential key or UUIDv7.
- Creating every missing-index suggestion as-is.
- Index hints that outlive the reason they were added.

# 9 Optimizing procedure code

Stored procedures, functions, triggers and packages (PL/SQL, T-SQL, PL/pgSQL and MySQL stored programs) concentrate a lot of database time in a few places. They also hide it: a procedure appears as one call, while it runs thousands of statements inside. The core principle is the same in every engine: **the SQL engine works in sets, so let it**. Procedural loops that issue one statement per row multiply round trips, context switches, log writes and lock acquisitions.

## 9.1 The patterns that cost the most

| Pattern | Why it hurts | Fix |
|---|---|---|
| Row-by-row loops ("RBAR") that update or insert one row at a time | One statement execution, and often one round trip, per row | A single set-based `INSERT ... SELECT`, `UPDATE` with a join, or `MERGE` |
| A query inside a loop (the N+1 pattern) | N extra executions, each cheap, together expensive | Join once and loop over the result, or do the whole job in SQL |
| `COUNT(*)` to test whether something exists | Counts every match when one would do | `EXISTS`, or fetch the first row only |
| Dynamic SQL built by concatenating values | Hard parse or compile per call, plan cache pollution, SQL injection risk | Dynamic SQL with bind parameters |
| `COMMIT` inside a loop | A log flush, and a wait for it, on every commit (the commit waits in section 5.7) | Commit per batch, or once at the end, as the business rules allow |
| An exception handler per row | Expensive setup per row, and errors hidden from the caller | Validate in SQL first; use bulk error handling |
| Scalar functions called for each row of a query | Per-row execution and context switches; in some engines, plans that can't run in parallel | Inline the logic, or use the engine's function inlining and caching features (section 9.4) |
| Cursors where a set operation would do | Row-at-a-time processing | Set-based SQL; if a cursor is truly needed, the cheapest cursor type |
| Long transactions around slow work | Locks held while the code waits on something else | Short transactions; never wait on users or external services while holding locks |

## 9.2 Set-based rewrites

The rewrite is nearly always the same: find the loop, find the condition that drives it, and express the whole job as one statement. The engine notes show the pattern in each procedural language.

POSTGRESQL **PL/pgSQL: a loop over a lookup table becomes UPDATE … FROM**

```
-- Before: one UPDATE per loyalty tier row
CREATE OR REPLACE FUNCTION apply_discounts() RETURNS void LANGUAGE plpgsql AS $$
DECLARE r record;
BEGIN
  FOR r IN SELECT customer_id, discount_pct FROM loyalty_tiers LOOP
    UPDATE orders
    SET    total_amount = total_amount * (1 - r.discount_pct / 100.0)
    WHERE  customer_id = r.customer_id AND status = 'OPEN';
  END LOOP;
END $$;
```

```
-- After (assuming one tier row per customer)
UPDATE orders o
SET    total_amount = o.total_amount * (1 - t.discount_pct / 100.0)
FROM   loyalty_tiers t
WHERE  t.customer_id = o.customer_id AND o.status = 'OPEN';
```

MYSQL **The same rewrite as a multi-table UPDATE**

```
UPDATE orders o
JOIN   loyalty_tiers t ON t.customer_id = o.customer_id
SET    o.total_amount = o.total_amount * (1 - t.discount_pct / 100.0)
WHERE  o.status = 'OPEN';
```

SQL SERVER **T-SQL: a cursor becomes one statement**

```
-- Before: one UPDATE per row
DECLARE @id int;
DECLARE c CURSOR LOCAL FAST_FORWARD FOR
    SELECT order_id FROM dbo.orders
    WHERE  status = 'SHIPPED' AND shipped_at < DATEADD(day, -30, SYSUTCDATETIME());
OPEN c;
FETCH NEXT FROM c INTO @id;
WHILE @@FETCH_STATUS = 0
BEGIN
    UPDATE dbo.orders SET status = 'CLOSED' WHERE order_id = @id;
    FETCH NEXT FROM c INTO @id;
END;
CLOSE c;
DEALLOCATE c;
```

```
-- After: one statement does the same work
UPDATE dbo.orders
SET    status = 'CLOSED'
WHERE  status = 'SHIPPED' AND shipped_at < DATEADD(day, -30, SYSUTCDATETIME());
```

ORACLE **PL/SQL: when per-row logic is unavoidable, bulk it**

The best version is still a single `UPDATE`. When each row needs PL/SQL logic, `BULK COLLECT` and `FORALL` move rows between the PL/SQL and SQL engines in batches instead of one at a time:

```
DECLARE
  TYPE t_ids IS TABLE OF orders.order_id%TYPE;
  l_ids t_ids;
  CURSOR c IS
    SELECT order_id FROM orders
    WHERE  status = 'SHIPPED' AND shipped_at < SYSDATE - 30;
BEGIN
  OPEN c;
  LOOP
    FETCH c BULK COLLECT INTO l_ids LIMIT 1000;
    EXIT WHEN l_ids.COUNT = 0;
    -- ...per-row PL/SQL logic on l_ids here...
    FORALL i IN 1 .. l_ids.COUNT
      UPDATE orders SET status = 'CLOSED' WHERE order_id = l_ids(i);
  END LOOP;
  CLOSE c;
  COMMIT;   -- once, after the loop: committing inside it while the cursor is open risks
ORA-01555
END;
/
```

## 9.3 Dynamic SQL, done safely

Dynamic SQL is fine. Dynamic SQL with values concatenated into the text is not: it compiles a new statement for each value and opens the door to SQL injection. Bind the values.

Identifiers (table and column names) can't be bound. Validate them against the data dictionary or a fixed list, and quote them with the engine's quoting function, listed in the engine note.

ENGINE NOTES **Bound dynamic SQL**

### PostgreSQL (PL/pgSQL)

```
EXECUTE 'SELECT count(*) FROM orders WHERE customer_id = $1' INTO cnt USING p_cust;
```

### MySQL

```
PREPARE s FROM 'SELECT COUNT(*) FROM orders WHERE customer_id = ?';
EXECUTE s USING @cust;
DEALLOCATE PREPARE s;
```

### SQL Server (T-SQL)

```
EXEC sys.sp_executesql
     N'SELECT COUNT(*) FROM dbo.orders WHERE customer_id = @c',
     N'@c int', @c = @cust;
```

### Oracle (PL/SQL)

```
EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM orders WHERE customer_id = :1' INTO l_cnt USING
p_cust;
```

**Quoting identifiers**

- **PostgreSQL:** `quote_ident()` or `format('%I', ...)`.
- **MySQL:** `sys.quote_identifier()` in the `sys` schema.
- **SQL Server:** `QUOTENAME()`.
- **Oracle:** `DBMS_ASSERT`, for example `DBMS_ASSERT.SQL_OBJECT_NAME` or `ENQUOTE_NAME`.

## 9.4 Functions called from SQL

A function in a `SELECT` list or `WHERE` clause can run once per row, and it hides its own work from the optimizer.

Three things help in every engine: tell the engine what the function does (deterministic or not, reads data or not), let it inline or cache the function where it can, and best of all, express the logic in SQL.

ENGINE NOTES **Functions called from SQL**

- **PostgreSQL:** Declare volatility correctly (`IMMUTABLE`, `STABLE`, `VOLATILE`): it decides whether the planner can evaluate a function once, use it in an index expression, or must call it for every row. Simple SQL-language functions can be inlined into the calling query.
- **MySQL:** Declare `DETERMINISTIC` and the data access characteristic (`READS SQL DATA`, `NO SQL`) accurately; they affect binary logging and optimization.
- **SQL Server:** Scalar user-defined functions used to run per row and force serial plans. SQL Server 2019 (compatibility level 150) can inline many of them automatically; check `is_inlineable` in `sys.sql_modules`. Multi-statement table-valued functions are opaque to the optimizer; inline table-valued functions are not, so prefer those.
- **Oracle:** Each call from SQL to PL/SQL is a context switch. Options: `PRAGMA UDF` or a `WITH FUNCTION` clause (12c+) to cut the switch cost; `DETERMINISTIC` and the function result cache for repeated inputs; wrapping the call as a scalar subquery, `(SELECT f(x) FROM dual)`, so Oracle can cache results for repeated values within a query.

## 9.5 Parameter sensitivity in procedures

Code inside procedures is compiled once and reused, so the first parameter values seen can decide the plan for everyone. When data is skewed, one customer with two orders and one with two million share a plan that suits only one of them.

The remedies are the same everywhere: recompile infrequent, highly variable calls; let the engine keep more than one plan; or give very different cases their own code path or statement.

ENGINE NOTES **Parameter sensitivity in procedures**

- **PostgreSQL:** PL/pgSQL statements are prepared. The first five executions use custom plans, after which PostgreSQL may switch to a generic plan if it isn't estimated to be worse. `plan_cache_mode` (12+) can force one behavior, and `EXECUTE` (dynamic SQL) always plans fresh.
- **MySQL:** Statements in stored programs are re-optimized on each execution, so sniffing is rarely the issue; stale statistics and missing histograms are more common causes.
- **SQL Server:** Parameter sniffing. Options: `OPTION (RECOMPILE)` on the statement for infrequent, highly variable calls; `OPTIMIZE FOR` a representative value or `UNKNOWN`; Parameter Sensitive Plan optimization (2022, compatibility level 160) caches several plans for one statement; Query Store hints to apply a fix without changing code (2022).
- **Oracle:** Bind peeking with adaptive cursor sharing (11g+), which can create separate child cursors for different value ranges once it notices the problem. Histograms on the skewed column are what make it work.

**SQL Server: recompile one statement on every call**

```
-- Only for infrequent, highly variable calls
CREATE OR ALTER PROCEDURE dbo.orders_by_status @status varchar(20)
AS
SELECT order_id, customer_id, total_amount
FROM   dbo.orders
WHERE  status = @status
OPTION (RECOMPILE);
```

## 9.6 Errors, logging and transactions

- **Bulk error handling.** Divert bad rows instead of failing a whole batch: validate in SQL first, or use the engine's error handling for bulk loads.
- **Logging tables.** Writing a log row per processed row, in the same transaction, doubles the work and the locking. Log per batch, or buffer messages and write them once. Log rows that must survive the caller's rollback need a transaction of their own.
- **Transaction design.** Keep transactions short. Touch tables in a consistent order across procedures to avoid deadlocks. Never hold locks while waiting for a user, a file or a web service. For work queues, use reads that skip locked rows (section 4.2) so workers don't queue behind each other. Retry deadlock victims in the application.

ENGINE NOTES **Bulk errors and independent logging**

- **PostgreSQL:** `COPY ... WITH (ON_ERROR ignore)` skips rows with data type errors (17+). There are no autonomous transactions in core.
- **MySQL:** `INSERT IGNORE` turns many errors into warnings and skips the bad rows. Check `SHOW WARNINGS`, because it hides problems too.
- **SQL Server:** no row-level error logging for DML, so validate in a staging table and wrap each batch in `TRY...CATCH`. Table variables keep their rows after a rollback, which makes them a simple buffer for log messages.

- **Oracle:** DML error logging diverts bad rows to an error table instead of failing the whole statement, and `FORALL ... SAVE EXCEPTIONS` does the same for bulk binds. Autonomous transactions make logging independent of the caller's commit or rollback; use them for logging only.

**Oracle: DML error logging**

```
EXEC DBMS_ERRLOG.CREATE_ERROR_LOG('ORDERS_STAGE');   -- creates ERR$_ORDERS_STAGE

INSERT INTO orders_stage (order_id, customer_id, order_date, total_amount)
SELECT order_id, customer_id, order_date, total_amount FROM ext_orders
LOG ERRORS INTO err$_orders_stage ('load 2026-11-01') REJECT LIMIT UNLIMITED;
```

## 9.7 Instrument your code so time can be attributed

Monitoring can only tell you which feature is slow if the code says who it is. This costs almost nothing and pays off in every incident.

Set an application or module name on every connection, and an action for each unit of work. Every engine has a place for them, and monitoring tools pick them up.

ENGINE NOTES **Naming the work**

**PostgreSQL**

```
-- Visible in pg_stat_activity and in log lines (%a in log_line_prefix)
SET application_name = 'billing:close_month';
```

**MySQL**

Connectors can send connection attributes (such as `program_name`) that appear in `performance_schema.session_connect_attrs`.

**SQL Server**

```
-- Set "Application Name=billing" in the connection string, and add detail per call
EXEC sys.sp_set_session_context @key = N'action', @value = N'close_month';
```

**Oracle**

```
-- Visible in V$SESSION and in V$SQL (MODULE, ACTION), without any pack
BEGIN
  DBMS_APPLICATION_INFO.SET_MODULE(module_name => 'billing', action_name =>
'close_month');
END;
/
```

## 9.8 Profile procedure code

Every engine can show which statements inside a routine take the time, and most can show the totals for each routine.

ENGINE NOTES **Profiling routines**

- **PostgreSQL:** `track_functions = 'pl'` fills `pg_stat_user_functions` with calls and time per function; `pg_stat_statements.track = 'all'` records the statements inside functions; `auto_explain.log_nested_statements = on` logs their plans.
- **MySQL:** `performance_schema.events_statements_summary_by_program` gives totals per stored program, and the statement history tables show the statements inside.
- **SQL Server:** Query Store captures statements inside procedures; `sys.dm_exec_procedure_stats` and `sys.dm_exec_trigger_stats` give totals per object; Extended Events (`sp_statement_completed`) shows statements one by one.
- **Oracle:** `DBMS_HPROF`, the hierarchical profiler (11g+, free), shows time per subprogram, including SQL; SQL trace with TKPROF shows every statement a procedure runs. For statements already in the cursor cache, `V$SQL.PROGRAM_ID` and `PROGRAM_LINE#` name the PL/SQL unit and the line that issued each one, so you can rank a package's statements by cost without tracing anything.

REINDB **Where ReinDB helps**

The Stored code report reads procedure and function source on all four engines and flags the patterns in this chapter: row-by-row loops, a query per row, `COUNT` used for existence checks, SQL built by concatenation, an exception block per row, and more. It links the slow statements inside a routine back to the lines that run them, sketches a set-based rewrite where one applies, and writes a commented fix script. Pre-check runs the same rules on a procedure you paste in before it ships.

# 10 Materialized views and partitioning

Both techniques trade extra design and maintenance for faster reads and easier data management. Both are frequently misused: materialized views as a substitute for fixing a query, partitioning as a magic speed switch. Used for the right reasons, they are two of the most effective tools in this guide.

## 10.1 Materialized views: precomputed answers

A materialized view (MV) stores the result of a query as a table and refreshes it on a schedule or as the base data changes.

**Use one when:**

- An expensive aggregation or join is read far more often than its base data changes.
- Readers can accept data that is a little stale (minutes or hours), or the engine can keep it current cheaply.
- Dashboards and reports hit the same summary again and again.
- You need a local copy of remote or slowly changing reference data.

**Avoid one when** the base data changes constantly and readers need it exact; when the real problem is a missing index or a bad query; or when the refresh would cost more than the reads it saves.

### Support differs widely by engine

| Engine | What you get |
|---|---|
| PostgreSQL | `CREATE MATERIALIZED VIEW` with full refresh only. `REFRESH ... CONCURRENTLY` avoids blocking readers but requires a unique index on the MV. No incremental refresh in core (third-party extensions exist) and no query rewrite: queries must name the MV |
| MySQL | No materialized views. Use summary tables maintained by scheduled events, triggers or your ETL |
| SQL Server | Indexed views: a view with `SCHEMABINDING` and a unique clustered index, maintained synchronously with every change to the base tables. The optimizer matches them automatically only in Enterprise Edition; other editions must reference the view with `WITH (NOEXPAND)`. Many restrictions (no outer joins, deterministic expressions only, `COUNT_BIG(*)` required with `GROUP BY`, specific `SET` options) |
| Oracle | The fullest implementation: complete or fast (incremental) refresh, on demand or on commit; materialized view logs to track changes; query rewrite, so the optimizer can answer queries against base tables from an MV automatically; `DBMS_MVIEW.EXPLAIN_MVIEW` to see why fast refresh isn't possible; real-time materialized views (12.2+). Query rewrite requires Enterprise Edition |

## Examples

### POSTGRESQL A materialized view refreshed without blocking readers

```
CREATE MATERIALIZED VIEW mv_sales_by_day AS
SELECT order_date, region, sum(total_amount) AS sales, count(*) AS cnt
FROM   orders
GROUP  BY order_date, region;

CREATE UNIQUE INDEX ON mv_sales_by_day (order_date, region);   -- required for
CONCURRENTLY
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_sales_by_day;
```

### MYSQL A summary table refreshed by a scheduled event

```
-- A summary table refreshed every hour for the last two days
CREATE TABLE sales_by_day (
  order_date DATE          NOT NULL,
  region     VARCHAR(20)   NOT NULL,
  sales      DECIMAL(14,2) NOT NULL,
  cnt        BIGINT        NOT NULL,
  PRIMARY KEY (order_date, region));
```

```
CREATE EVENT refresh_sales_by_day
  ON SCHEDULE EVERY 1 HOUR
  DO
    REPLACE INTO sales_by_day (order_date, region, sales, cnt)
    SELECT order_date, region, SUM(total_amount), COUNT(*)
    FROM   orders
    WHERE  order_date >= CURRENT_DATE - INTERVAL 2 DAY
    GROUP  BY order_date, region;
```

### SQL SERVER An indexed view

```
-- An indexed view.
-- total_amount must be NOT NULL: an indexed view can't SUM a nullable expression.
CREATE VIEW dbo.v_sales_by_day
WITH SCHEMABINDING
AS
SELECT order_date, region, SUM(total_amount) AS sales, COUNT_BIG(*) AS cnt
FROM   dbo.orders
GROUP  BY order_date, region;
GO
CREATE UNIQUE CLUSTERED INDEX ix_v_sales_by_day ON dbo.v_sales_by_day (order_date,
region);
GO
-- Outside Enterprise Edition, ask for the view explicitly:
SELECT order_date, sales FROM dbo.v_sales_by_day WITH (NOEXPAND) WHERE region = 'WEST';
```

### ORACLE A fast-refreshable summary with query rewrite

```
-- Fast-refreshable daily summary with query rewrite
CREATE MATERIALIZED VIEW LOG ON orders
  WITH ROWID, SEQUENCE (order_date, region, total_amount) INCLUDING NEW VALUES;
```

```
CREATE MATERIALIZED VIEW mv_sales_by_day
  BUILD IMMEDIATE
  REFRESH FAST ON DEMAND
  ENABLE QUERY REWRITE
AS
SELECT order_date, region,
       SUM(total_amount)   AS sales,
       COUNT(total_amount) AS cnt_amount,   -- fast refresh of SUM needs COUNT of the same
column
       COUNT(*)            AS cnt
FROM   orders
GROUP  BY order_date, region;
```

```
EXEC DBMS_MVIEW.REFRESH('MV_SALES_BY_DAY', method => 'F');
```

### Materialized view pitfalls

- **Refresh cost and locking.** A full refresh of a large MV can take longer than the queries it saves, and some refresh modes block readers.
- **Stale data surprises.** Make the refresh time visible to users ("data as of 06:00").
- **On-commit refresh slows writers.** Views maintained inside every transaction add work to each change on the base tables, and can become a source of contention and deadlocks on busy tables.
- **Growing change logs.** Where incremental refresh reads a change log, a view that stops refreshing keeps the log from being purged, and it grows without limit.
- **Missing statistics and indexes.** An MV is a table: gather statistics on it and index it for the queries that read it.
- **Dependency chains.** MVs built on MVs must refresh in order; a failure in the middle leaves an inconsistent set.

## 10.2 Partitioning: one table, many pieces

Partitioning splits a table (and usually its indexes) into pieces by a key: ranges of dates, lists of values, or a hash.

**What it gives you:**

- **Partition pruning:** queries that filter on the partition key read only the relevant partitions.
- **Cheap data lifecycle:** drop, truncate, detach or exchange a whole partition in seconds instead of running a huge `DELETE`.
- **Maintenance per partition:** gather statistics, rebuild indexes, compress or move one partition at a time; old partitions can live on cheaper storage.
- **Partition-wise joins and parallelism** for large analytic queries.

> **WATCH OUT** **Partitioning is mainly a manageability feature**
>
> It only speeds up queries that filter on the partition key. Queries that don't must visit every partition (and every local index), which can make them slower than before. Partitioning a table to "make it faster", without checking which predicates the top statements use, is one of the most common and expensive mistakes in this guide.

## What each engine offers

| Engine | Notes |
|---|---|
| PostgreSQL | Declarative partitioning (10+): range, list, and hash (11+); a default partition (11+); indexes on the parent cascade to partitions (11+); foreign keys referencing partitioned tables (12+); `DETACH PARTITION ... CONCURRENTLY` (14+). Primary keys and unique constraints must include the partition key. Execution-time pruning (11+) helps prepared statements. The `pg_partman` extension automates creating and retiring partitions |
| MySQL | Range, list, hash and key partitioning, plus `RANGE COLUMNS` and `LIST COLUMNS`. Every unique key, including the primary key, must include all partitioning columns. Partitioned InnoDB tables can't have foreign keys, in either direction. Up to 8,192 partitions. `ALTER TABLE ... EXCHANGE PARTITION` and `DROP PARTITION` are fast |
| SQL Server | A partition function and a partition scheme; available in all editions since 2016 SP1. Keep indexes aligned with the table. `ALTER TABLE ... SWITCH` moves a whole partition in or out instantly; `TRUNCATE TABLE ... WITH (PARTITIONS (...))` (2016+) empties one; up to 15,000 partitions per table |
| Oracle | **A separately licensed option of Enterprise Edition.** Range, list, hash and composite partitioning; interval partitioning (11g) creates new partitions automatically; reference partitioning (11g). 12c added partial indexes and online partition moves; 12.2 added online conversion of a regular table to a partitioned one, auto-list partitioning and filtered partition maintenance; 19c added hybrid partitioned tables. Local indexes are partitioned with the table; global indexes need maintenance when partitions are dropped (`UPDATE INDEXES`; asynchronous global index maintenance in 12c+) |

> **CLOUD** **Partitioning on managed services**
>
> - **Oracle:** included in the OCI Base Database High Performance and Extreme Performance tiers, in Exadata Database Service and in Autonomous Database. On RDS for Oracle it comes only with your own Enterprise Edition license and the Partitioning option.
> - **PostgreSQL:** most managed services offer `pg_partman`, with `pg_cron` to schedule it, so partitions can be created and retired without a server of your own.

## Examples: monthly partitions by order date

### POSTGRESQL Declarative range partitioning

```
CREATE TABLE orders_p (
  order_id     bigint        NOT NULL,
  order_date   date          NOT NULL,
  customer_id  bigint        NOT NULL,
  total_amount numeric(12,2),
  PRIMARY KEY (order_id, order_date)
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_p_2026_11 PARTITION OF orders_p
  FOR VALUES FROM ('2026-11-01') TO ('2026-12-01');

-- Retire an old month without a big DELETE (CONCURRENTLY needs PostgreSQL 14+)
ALTER TABLE orders_p DETACH PARTITION orders_p_2025_10 CONCURRENTLY;
```

### MYSQL RANGE COLUMNS partitioning

```
CREATE TABLE orders_p (
  order_id     BIGINT        NOT NULL,
  order_date   DATE          NOT NULL,
  customer_id  BIGINT        NOT NULL,
  total_amount DECIMAL(12,2),
  PRIMARY KEY (order_id, order_date)
)
PARTITION BY RANGE COLUMNS (order_date) (
  PARTITION p2026_10 VALUES LESS THAN ('2026-11-01'),
  PARTITION p2026_11 VALUES LESS THAN ('2026-12-01'),
  PARTITION pmax     VALUES LESS THAN (MAXVALUE)
);

ALTER TABLE orders_p DROP PARTITION p2026_10;
```

### SQL SERVER A partition function and scheme

```
CREATE PARTITION FUNCTION pf_month (date)
    AS RANGE RIGHT FOR VALUES ('2026-09-01', '2026-10-01', '2026-11-01', '2026-12-01');
CREATE PARTITION SCHEME ps_month AS PARTITION pf_month ALL TO ([PRIMARY]);

CREATE TABLE dbo.orders_p (
    order_id     bigint        NOT NULL,
    order_date   date          NOT NULL,
    customer_id  int           NOT NULL,
    total_amount decimal(12,2) NULL,
    CONSTRAINT pk_orders_p PRIMARY KEY CLUSTERED (order_date, order_id)
) ON ps_month (order_date);
```

ORACLE **Interval partitioning**

```
-- Interval partitioning creates each month's partition on first insert
CREATE TABLE orders_p (
  order_id     NUMBER       NOT NULL,
  customer_id  NUMBER       NOT NULL,
  order_date   DATE         NOT NULL,
  total_amount NUMBER(12,2),
  CONSTRAINT pk_orders_p PRIMARY KEY (order_id)   -- a global index (see the pitfalls
below)
)
PARTITION BY RANGE (order_date)
INTERVAL (NUMTOYMINTERVAL(1, 'MONTH'))
(PARTITION p_start VALUES LESS THAN (DATE '2024-01-01'));
```

### Choosing the partition key and size

- Choose the column that **most important queries filter on** and that **defines the data lifecycle**. For transactional data that is nearly always a date.
- Size partitions so a typical query touches one or a few, and so the number of partitions stays manageable: tens to low thousands, not tens of thousands. Daily partitions for seven years of data is usually too many; monthly is often right.
- Create future partitions ahead of time (or let the engine or a scheduled job do it), and monitor that the job runs. A missing partition means failed inserts, or rows piling into a default partition.

### Partitioning pitfalls

- A key that queries don't filter on, so nothing is pruned.
- Too many tiny partitions, which inflates planning time and metadata.
- Unique constraints that must now include the partition key, which changes what "unique" means.
- Maintenance of global (non-partitioned) indexes when partitions are dropped or exchanged.
- **Statistics on partitioned tables.** Table-level statistics can need a rescan of every partition, or may not be kept up to date automatically (see the engine note).
- Functions or type conversions on the partition key in predicates, which defeat pruning just as they defeat indexes.
- A hot current partition that takes all the writes, while old partitions sit idle. That is normal, but size storage and indexes for it.

ENGINE NOTES **Statistics on partitioned tables**

- **PostgreSQL:** autovacuum analyzes each partition but **not the partitioned parent**, so run `ANALYZE` on the parent yourself after significant changes.
- **MySQL:** `ALTER TABLE ... ANALYZE PARTITION` refreshes statistics for the partitions you name.
- **SQL Server:** incremental statistics (`STATISTICS_INCREMENTAL = ON`, 2014+) update per partition instead of rescanning the whole table.
- **Oracle:** incremental statistics (11g+) keep global statistics current without rescanning every partition.

## 10.3 Using them together

Partitioning and materialized views combine well for time-series data:

- **Partition the summary too.** A daily or monthly summary partitioned by date can be refreshed one recent partition at a time.
- **Load by swapping.** Build the new period's data or summary in a separate table, then swap it into the partitioned table in one quick metadata operation.

> ENGINE NOTES **Refreshing and loading by partition**
>
> - **PostgreSQL:** `ATTACH PARTITION` adds a table you loaded separately. Create a matching `CHECK` constraint on it first, and the attach skips its validation scan.
> - **MySQL:** `ALTER TABLE ... EXCHANGE PARTITION` swaps a loaded table in.
> - **SQL Server:** switch-in loading. Build the new period's summary in a staging table, then `SWITCH` it into the partitioned table in one instant metadata operation.
> - **Oracle:** Partition Change Tracking (PCT) lets a materialized view over a partitioned table refresh only the parts affected by changed partitions, and keeps query rewrite working for the partitions that are still fresh. `EXCHANGE PARTITION` swaps a loaded table in.

## 10.4 Which tool for which problem?

| Tool | Solves | Costs | Don't use it for |
|---|---|---|---|
| Index | Finding few rows fast; ordering; covering hot queries | Write overhead, storage | Queries that read most of the table |
| Materialized view or summary table | Repeated expensive aggregations and joins | Refresh work, staleness, storage | Data that must always be exact and changes constantly |
| Partitioning | Data lifecycle, maintenance per piece, pruning on the key | Design constraints, more objects to manage, sometimes a license | Speeding up queries that don't filter on the key |
| Columnstore or columnar tier | Scans and aggregates over very many rows | Slower single-row access, load patterns | OLTP lookups |
| Archiving or tiering (Chapter 4) | Storage, backup time, maintenance windows | Moving data, cross-tier queries | Data that is still hot |

# 11 Settings: match the workload, then keep it there

Configuration rarely turns a bad design into a good one, but a bad setting can cripple a good design. Defaults are written for a generic machine and a generic workload, and they are often years behind the hardware you run on. This chapter covers how to match settings to the workload, the kinds of settings that matter in every engine (with each engine's own in the notes), and how to stop them drifting afterwards.

## 11.1 Principles

1. **Know the workload class first.** OLTP: many short transactions, a small hot working set, high concurrency, commit latency matters. Analytics: large scans, big sorts and hashes, parallelism, memory per query. Mixed: protect the OLTP side from the reports, with replicas or resource governance.
2. **Judge each setting against the right moment.** Caches are sized for the **average** working set. Per-operation memory, connection limits and log sizes are sized for the **peak**. Judging a peak setting against an average day is how systems fall over at month end.
3. **Change one setting at a time,** and measure at the same hour on the following days.
4. **Prefer scoped overrides to global changes.** If one report needs more sort memory, give it to that report, not to every session.
5. **Use the engine's own advice** (advisory views, checkpoint logs, wait statistics) before rules of thumb from the internet.
6. **On managed services, change the provider's parameter settings,** not the instance. Some parameters are static and need a reboot, and some are controlled by the provider (see the cloud note in section 11.3).

## 11.2 The settings that matter, by category

The names differ, but every engine asks you to make the same decisions:

| Category | What you decide | Judge against |
|---|---|---|
| Data cache | How much memory holds data and index pages | Average |
| Memory per operation | Sort and hash memory for each step of a query, multiplied by how many run at once | Peak |
| Connections | The maximum number of sessions, and whether a pool sits in front | Peak |
| Log and checkpoint sizing | How much log space absorbs write bursts before the engine must flush or switch | Peak |
| Durability | Whether each commit waits for its log write to reach storage | Not load-dependent |
| Parallelism | When a query goes parallel, and how many workers it may use | Peak |

| Category | What you decide | Judge against |
|---|---|---|
| Optimizer cost model and features | Storage cost assumptions, the optimizer version and adaptive features | Not load-dependent |
| Background work | Statistics gathering, cleanup and flushing rates | Peak |
| Temporary space | How much memory internal temporary results get before they spill, and where they spill to | Peak |
| Timeouts and limits | Statement and idle-transaction timeouts, and caps on runaway sessions | Peak |
| Instrumentation | What the engine measures and keeps (Chapter 2) | Not load-dependent |

## 11.3 Starting points by engine

The engine notes list the settings that matter most in each engine, with a starting point for each and the moment to judge it against. Starting points are where you begin measuring, not where you stop.

**POSTGRESQL Key settings**

| Setting | Controls | Starting point | Judge against |
|---|---|---|---|
| `shared_buffers` | PostgreSQL's own buffer cache | About 25% of RAM on a dedicated server (the documented starting point); PostgreSQL also relies on the OS cache | Average |
| `effective_cache_size` | The planner's estimate of total cache (allocates nothing) | 50 to 75% of RAM | Average |
| `work_mem` | Memory per sort or hash operation, per plan node, per parallel worker | Modest globally (the default is 4 MB); raise per role or session for reports; make sure peak connections times operations fits in RAM | Peak |
| `maintenance_work_mem` | Memory for vacuum and index builds | Much larger than `work_mem`, for example 1 GB or more on big servers | Peak |
| `max_connections` | Maximum connections, each a separate process | Keep it modest and put a connection pooler such as PgBouncer in front | Peak |
| `random_page_cost` | The planner's relative cost of random reads | 1.1 to 2 on SSD storage (the default of 4 assumes spinning disks) | Not load-dependent |
| `max_wal_size`, `checkpoint_timeout`, `checkpoint_completion_target` | How often checkpoints happen and how evenly they spread | Large enough that checkpoints are triggered by time, not by WAL volume (check with `log_checkpoints = on`) | Peak |

| Setting | Controls | Starting point | Judge against |
|---|---|---|---|
| `autovacuum_vacuum_scale_factor`, `autovacuum_analyze_scale_factor` | When a table is vacuumed and analyzed | Lower them per table for large tables, so they don't wait for 10 to 20% of rows to change | Not load-dependent |
| `autovacuum_vacuum_cost_limit`, `autovacuum_max_workers` | Vacuum throughput | Raise on fast storage when vacuum falls behind | Peak |
| `idle_in_transaction_session_timeout`, `statement_timeout` | Stuck and runaway sessions | Set them, at least per application role | Peak |
| `jit` | Just-in-time compilation for expensive queries | Consider turning it off for OLTP if compilation time shows up in short queries | Not load-dependent |

### MYSQL Key settings (InnoDB)

| Setting | Controls | Starting point | Judge against |
|---|---|---|---|
| `innodb_buffer_pool_size` | The data and index cache | Commonly 50 to 75% of RAM on a dedicated server | Average |
| `innodb_redo_log_capacity` (8.0.30+), or `innodb_log_file_size` | Redo space; too small forces checkpoints and stalls writes | Large enough that peak write bursts don't force flushing; a common rule of thumb is about an hour of peak redo | Peak |
| `innodb_flush_log_at_trx_commit`, `sync_binlog` | Durability of each commit | 1 and 1 for full durability. A value of 2 for the first trades up to about a second of transactions on an OS crash for speed | Not load-dependent |
| `innodb_io_capacity`, `innodb_io_capacity_max` | Background flushing rate | Match what the storage can really sustain | Peak |
| `max_connections` | Connection limit | Modest, with pooling in the application | Peak |
| `tmp_table_size`, `max_heap_table_size`, `temptable_max_ram` (8.0) | In-memory internal temporary tables before they go to disk | Raise if many temporary tables spill to disk | Peak |
| `table_open_cache` | Cached open table handles | Large enough that `Opened_tables` stops climbing quickly | Average |
| `long_query_time` | Slow query log threshold | 1 second or lower | Not load-dependent |

### SQL SERVER Key settings

| Setting | Controls | Starting point | Judge against |
|---|---|---|---|
| `max server memory (MB)` | Memory for the buffer pool and caches | Leave headroom for the OS and anything else on the host; the default is effectively unlimited | Average |

| Setting | Controls | Starting point | Judge against |
|---|---|---|---|
| `max degree of parallelism` | Maximum processors per parallel query | Follow Microsoft's guidance, based on logical processors per NUMA node (often 8, never more than 16); setup suggests a value since 2019 | Peak |
| `cost threshold for parallelism` | When a query is considered for a parallel plan | The default of 5 is low for modern hardware; many teams start between 25 and 50 and adjust | Peak |
| `optimize for ad hoc workloads` | Stores a stub for single-use ad hoc plans | On, for applications that send many unparameterized statements | Average |
| tempdb data files | Allocation contention in tempdb | Equal-sized files: one per logical core up to 8, then add in groups of 4 if contention remains | Peak |
| `READ_COMMITTED_SNAPSHOT` | Readers no longer block writers, and the reverse | On for most OLTP applications, after testing; row versions live in tempdb (or in the database itself with accelerated database recovery) | Peak |
| Query Store options | History and plan forcing | `READ_WRITE`, capture mode `AUTO`, storage sized for your retention | Not load-dependent |
| Compatibility level | Optimizer behavior (cardinality estimator, intelligent query processing) | The newest level you have tested; turn on Query Store first so regressions can be caught and fixed | Not load-dependent |
| File growth | Size of each autogrowth | Fixed sizes in MB, not percentages; instant file initialization for data files | Peak |
| Database scoped configuration | Per-database `MAXDOP`, `LEGACY_CARDINALITY_ESTIMATION`, `PARAMETER_SNIFFING` | Change per database when one application needs it, instead of server-wide | Not load-dependent |

### ORACLE Key settings

| Setting | Controls | Starting point | Judge against |
|---|---|---|---|
| `SGA_TARGET`, `PGA_AGGREGATE_TARGET` | Automatic shared memory management and the PGA target | Size with the free advisory views `V$SGA_TARGET_ADVICE` and `V$PGA_TARGET_ADVICE` | SGA: average; PGA: peak |
| `MEMORY_TARGET` | Automatic memory management of SGA and PGA together | Avoid it for large SGAs on Linux, where it can't use HugePages; prefer `SGA_TARGET` plus `PGA_AGGREGATE_TARGET` with HugePages | Not load-dependent |
| `PGA_AGGREGATE_LIMIT` (12c+) | A hard cap on total PGA | Review the default; sessions are terminated when it is exceeded | Peak |
| `PROCESSES`, `SESSIONS` | Connection limits | Peak plus pool growth; check high-water marks in `V$RESOURCE_LIMIT` | Peak |

| Setting | Controls | Starting point | Judge against |
|---|---|---|---|
| `OPEN_CURSORS`, `SESSION_CACHED_CURSORS` | Cursor caching | Raise when parse counts are high and soft parses dominate | Average |
| `CURSOR_SHARING` | Replacement of literals with binds | `EXACT`; `FORCE` only as a stopgap for applications that don't use binds | Not load-dependent |
| Redo log size and number of groups | Log switch frequency and checkpoint pressure | A common rule of thumb is a switch every 15 to 30 minutes at peak; `log file switch (checkpoint incomplete)` waits mean the logs are too small or too few | Peak |
| `FILESYSTEMIO_OPTIONS` | Asynchronous and direct I/O on file systems | `SETALL` on most Linux file systems (test first; not relevant for ASM) | Not load-dependent |
| `OPTIMIZER_ADAPTIVE_PLANS`, `OPTIMIZER_ADAPTIVE_STATISTICS` (12.2+) | Adaptive optimizer features | The defaults (`TRUE` and `FALSE`) are sensible; 12.1's single `OPTIMIZER_ADAPTIVE_FEATURES` setting caused many regressions | Not load-dependent |
| `OPTIMIZER_FEATURES_ENABLE` | Optimizer behavior by release | Leave it at the current release; set it back only temporarily to contain an upgrade regression | Not load-dependent |
| `STATISTICS_LEVEL` | Instrumentation level | `TYPICAL` | Not load-dependent |
| `DB_FILE_MULTIBLOCK_READ_COUNT` | Multiblock read size | Leave it unset, so Oracle chooses | Not load-dependent |

ORACLE **11g and 19c**

Many settings behave differently across releases. `PGA_AGGREGATE_LIMIT` doesn't exist in 11g, so a runaway PGA consumer can push an 11g server into swapping. The adaptive optimizer features arrived in 12c. Multitenant (CDB and PDB) changes where some parameters can be set. Appendix B lists the differences that matter most for tuning.

CLOUD **Settings on managed services**

- **Parameter groups and flags.** Changes go through the provider, and static ones need a reboot, often in a maintenance window. Some defaults are formulas based on the instance size, such as `{DBInstanceClassMemory*3/4}` for the InnoDB buffer pool on Amazon RDS, so a resize changes them for you. Others are fixed and can't be changed at all.
- **Aurora** replaces the storage layer, so it fixes or ignores settings that belonged to it, such as redo log sizing and many checkpoint settings.
- **Azure SQL Database and Oracle Autonomous Database** manage most settings themselves. You choose a service tier or a number of CPUs, then tune with database scoped configurations (Azure) or the service you connect through (Autonomous).

## 11.4 Scoped overrides for the outliers

Give the outliers what they need, and only them. Every engine lets you change settings for a narrower scope than the whole server: a user or role, a session or transaction, or a single statement.

For whole workloads, use resource governance where the engine offers it, so reporting users can't starve the transactional ones of CPU, parallelism or memory. Check which editions include it for your version.

> ENGINE NOTES **Scoped settings and resource governance**
>
> **PostgreSQL**
>
> ```
> -- More sort memory for the reporting role, or for one transaction
> ALTER ROLE reporting SET work_mem = '256MB';
> BEGIN;
> SET LOCAL work_mem = '1GB';
> -- ...the one big report...
> COMMIT;
> ```
>
> There is no built-in resource governance; per-role settings such as `statement_timeout` and `work_mem`, and a separate replica for reports, do the job.
>
> **MySQL**
>
> ```
> -- A setting for one statement only (8.0+)
> SELECT /*+ SET_VAR(sort_buffer_size = 16777216) */ customer_id, SUM(total_amount)
> FROM   orders GROUP BY customer_id ORDER BY 2 DESC;
> ```
>
> Resource groups (8.0+) set thread priority and CPU affinity for a class of work.
>
> **SQL Server**
>
> ```
> -- Cap one query's memory grant
> SELECT customer_id, SUM(total_amount) AS total
> FROM   dbo.orders GROUP BY customer_id
> OPTION (MAX_GRANT_PERCENT = 10);
> ```
>
> Resource Governor caps CPU, memory grants and parallelism per workload group. It was an Enterprise Edition feature until SQL Server 2025 added it to Standard Edition.
>
> **Oracle**
>
> `ALTER SESSION` changes settings for one session, and the `OPT_PARAM` hint changes an optimizer parameter for one statement. Database Resource Manager (Enterprise Edition) caps CPU, parallelism and sessions per consumer group.

## 11.5 Keeping it there

Settings drift. A parameter changed during an incident never gets changed back, an upgrade brings new defaults, an instance is resized and memory settings stay where they were. Treat configuration like code.

- **Keep the approved configuration in version control:** parameter files, parameter groups (through Terraform or CloudFormation), `sp_configure` and database scoped configuration scripts.
- **Compare live settings against it on a schedule,** and alert on differences.

- **Re-evaluate after every** version upgrade, instance resize, major workload change or doubling of data.
- **Record every change** with the reason, the person, and the before and after values.

Each engine can list what has been changed from its defaults. Save the output and compare it over time:

ENGINE NOTES **Settings changed from the defaults**

### PostgreSQL

```
SELECT name, setting, unit, source
FROM   pg_settings
WHERE  source NOT IN ('default', 'override')
ORDER  BY name;
```

### MySQL

```
-- Where each variable's value came from (8.0+)
SELECT v.variable_name, g.variable_value, v.variable_source, v.set_time
FROM   performance_schema.variables_info v
JOIN   performance_schema.global_variables g ON g.variable_name = v.variable_name
WHERE  v.variable_source <> 'COMPILED'
ORDER  BY v.variable_name;
```

### SQL Server

```
-- There is no "default" column, so save and compare snapshots
SELECT name, value_in_use FROM sys.configurations ORDER BY name;
SELECT name, value FROM sys.database_scoped_configurations ORDER BY name;
```

### Oracle

```
SELECT name, value, ismodified
FROM   v$parameter
WHERE  isdefault = 'FALSE'
ORDER  BY name;
```

WATCH OUT **Settings traps**

- Copying "best" settings from a blog post written for a different version or workload.
- Changing five settings at once and keeping all five because things got better.
- Giving the database 100% of RAM and leaving nothing for the OS, the file cache or the backup agent.
- Huge connection limits in place of a connection pool.
- Trading durability for speed in production, such as turning off log flushing or fsync where data loss is unacceptable.
- Carrying settings across an upgrade without checking what the new version's defaults already do.

**REINDB** **Where ReinDB helps**

ReinDB's settings review judges each setting against Live, Average or Peak, using the hourly history it already keeps, and gives a recommendation with a commented fix script. On Amazon RDS, Aurora and other managed services, the fix says which parameter-group value to change instead of `ALTER SYSTEM`. For individual statements and routines that need more memory or a different setting, it suggests scoped session-level settings rather than global ones.

# 12 Baselining: our approach and why it matters

"Is this normal?" is the first question in every performance incident, and the one most teams can't answer quickly. A baseline answers it. It is a description of what normal looks like for a particular database, at a particular time, so you can tell a real change from the usual rhythm of the day.

## 12.1 Why baselines matter

- **Averages lie.** A database that is busy at 09:00 and idle at 03:00 has a daily average that describes neither. A load of 4 active sessions is alarming at 03:00 and perfectly normal at 09:00.
- **Static thresholds are either noisy or blind.** Set "CPU above 80%" and you get paged every busy morning; set it at 95% and you miss the slow climb that matters.
- **Baselines turn opinions into evidence.** "It feels slower" becomes "this statement takes 3.2 times its usual time for this hour, since Tuesday 14:00".
- **They prove a fix worked,** by comparing the same hours before and after the change.
- **They catch regressions early,** before a statement climbs into the top ten.
- **They feed capacity planning** with a trend instead of a guess.

## 12.2 Kinds of baseline

| Kind | What it compares against | Good for |
|---|---|---|
| Static threshold | A fixed number | Hard limits: disk full, connections at the maximum, replication lag beyond the recovery point objective |
| Rolling window | The last N hours or days | Sudden changes in steady workloads |
| Seasonal | The same hour of the day (and optionally the same day of the week) over recent weeks | Most real workloads, which follow daily and weekly cycles |
| Event | A named period before a release, an upgrade or a parameter change | Proving that a change helped, or finding what it broke |

## 12.3 What to baseline

- **Load:** database time, or average active sessions, per hour.
- **Throughput:** executions, transactions and commits per second.
- **Per statement:** time per execution, executions per hour, logical reads per execution, rows per execution, and the plan in use.

- **Waits:** time by wait class.
- **I/O and memory:** physical reads, temp usage and spills.
- **Log volume:** redo, WAL or transaction log generated per hour.
- **Sessions and contention:** active sessions, sessions idle inside an open transaction, and blocked sessions; lock wait time; deadlocks.
- **Capacity:** database and table sizes and their growth.
- **Replication:** lag in seconds and bytes.

## 12.4 Our approach in ReinDB

ReinDB is built around a seasonal baseline, because that is what matches how real databases behave.

1. **Measure every hour from the engine's own statistics.** ReinDB reads the cumulative statement and wait statistics each engine already keeps (`pg_stat_statements`, `performance_schema`, Query Store or the plan cache, and Oracle's V$ views) and stores the differences between runs. Nothing is installed in the database, and it reads only.
2. **Define normal as the median and spread over the last 14 days at the same hour of the day.** Monday's 09:00 is compared with the previous two weeks of 09:00s, not with last night.
3. **Flag what drifts from normal, ranked by the extra time it costs.** A statement that is ten times slower but adds one second per hour ranks below one that is 20% slower and adds an hour of database time.
4. **Track each flag's life.** Flags are NEW when they first appear, ONGOING while the drift continues and RESOLVED when it ends, so a recurring problem looks different from a one-off.
5. **Explain with evidence.** Each flag shows what drifted, by how much, since when, and the extra time per hour, with the statement's plan history and the analysis behind it.

Why these choices:

- **The median, not the mean.** One bad hour doesn't redefine normal; outliers barely move a median.
- **Spread, not just a center.** Statements with naturally noisy timings need a bigger change before they are flagged, so steady statements can be flagged on smaller changes. That keeps the noise down without hiding real drift.
- **Fourteen days.** Two full weekly cycles: long enough to be stable, short enough to adapt as the data grows.
- **Hour of the day.** Daily cycles dominate most workloads. For strong weekly patterns, such as a Monday batch, compare with the same hour last week as well, which the "Vs baseline" report and the Baselines page both offer.
- **Ranked by extra time.** It focuses attention where fixing something actually returns capacity.

## 12.5 Build a simple seasonal baseline yourself

If you are building your own repository, the core is small. Snapshot the cumulative statement statistics every hour, store the differences, and compute the usual value for each statement and hour of the day. The

example is written for PostgreSQL, and the engine note covers the functions that differ elsewhere; the idea works in any engine.

POSTGRESQL **The core of a seasonal baseline**

```
-- One row per statement per hour (the differences between two hourly snapshots)
CREATE TABLE stmt_hourly (
  hour_start   timestamptz      NOT NULL,
  statement_id text             NOT NULL,
  calls        bigint           NOT NULL,
  total_ms     double precision NOT NULL,
  PRIMARY KEY (statement_id, hour_start)
);
```

```
-- Usual time per call for each statement and hour of the day, over the 14 days before
today
WITH hist AS (
  SELECT statement_id,
         -- hour of the day in the time zone the workload follows
         extract(hour FROM hour_start AT TIME ZONE 'America/Denver') AS hod,
         total_ms / NULLIF(calls, 0)                                 AS ms_per_call
  FROM   stmt_hourly
  WHERE  hour_start >= date_trunc('day', now()) - interval '14 days'
    AND  hour_start <  date_trunc('day', now())
    AND  calls > 0
)
SELECT statement_id, hod,
       percentile_cont(0.5) WITHIN GROUP (ORDER BY ms_per_call) AS median_ms,
       percentile_cont(0.9) WITHIN GROUP (ORDER BY ms_per_call) AS p90_ms,
       count(*)                                                 AS samples
FROM   hist
GROUP  BY statement_id, hod;
```

ENGINE NOTES **Percentiles and time zones elsewhere**

- **MySQL:** there are no percentile functions; rank the rows with `ROW_NUMBER()` and `COUNT()` window functions and pick the middle one. `CONVERT_TZ()` converts time zones once the time zone tables are loaded.
- **SQL Server:** `PERCENTILE_CONT` works only as a window function (with `OVER`), while `APPROX_PERCENTILE_CONT` (2022+) is an aggregate. `AT TIME ZONE` converts time zones (2016+).
- **Oracle:** `PERCENTILE_CONT ... WITHIN GROUP` works as an aggregate, as in the example, and `FROM_TZ` with `AT TIME ZONE` converts time zones.

Then flag a statement in the current hour when its time per call is well above its usual range (for example above both 1.5 times the median and the 90th percentile), **and** the extra time it costs (the excess per call multiplied by the calls) is above a floor that matters to you, such as a minute per hour. Require a minimum number of samples before you trust a baseline. A robust alternative to percentiles is the median absolute deviation (MAD): scaled by 1.4826, it estimates the standard deviation without letting a few bad hours distort it, so "more than three MADs above the median" makes a sturdy test.

> NOTE **Time zones and daylight saving**
>
> Store timestamps in UTC, but compute the hour of the day in the time zone the workload follows. If users work 9 to 5 in Denver, their 09:00 peak moves by an hour in UTC twice a year, and a UTC-only baseline will flag it.

## 12.6 Baselines around changes

- Take, or name, a baseline before every deploy, upgrade, parameter change and index change. "Before release 4.2" is a baseline worth keeping.
- After the change, compare the same hours of comparable days, not the hour after against the hour before.
- Annotate dashboards with deploys and changes, so drift lines up with its cause at a glance.

## 12.7 Baseline pitfalls

- **Normal isn't good.** A baseline says what is usual, not what is acceptable. A database that has been slow for a month has a slow baseline. Pair baselines with targets (Chapter 1).
- **Counter resets.** Restarts, statistics resets and plan cache eviction make one interval look strange. Detect resets (a counter that goes down) and skip those intervals.
- **Calendar effects.** Month end, quarter end, holidays and campaigns break the pattern. Compare them with the same event last time, not with an ordinary week.
- **Slow drift.** Growth shifts normal gradually, and a 14-day window follows it. That is why capacity and growth need their own trend lines and projections, separate from drift detection.
- **Too little history.** A baseline built from two days is a guess. Wait for enough samples before acting on the flags.

> REINDB **Baselines without building anything**
>
> ReinDB starts the hourly baseline as soon as a database is added. Database-wide flags start after five hourly runs, and statement flags after a statement has appeared in three. The Baselines page gives one-click comparisons (today against yesterday, today against a week ago, this week against last, and against usual) and lets you save any range as a named baseline, such as "Before release 4.2". The "Vs baseline" report compares any period with the 14 days before it, the previous period, the same time last week or a custom range.

# 13 Ongoing observability

Tuning fixes today's problem. Observability keeps tomorrow's from reaching users. The goal is simple to state: see problems before users do, with enough history to explain them, and make the right data visible to the right people.

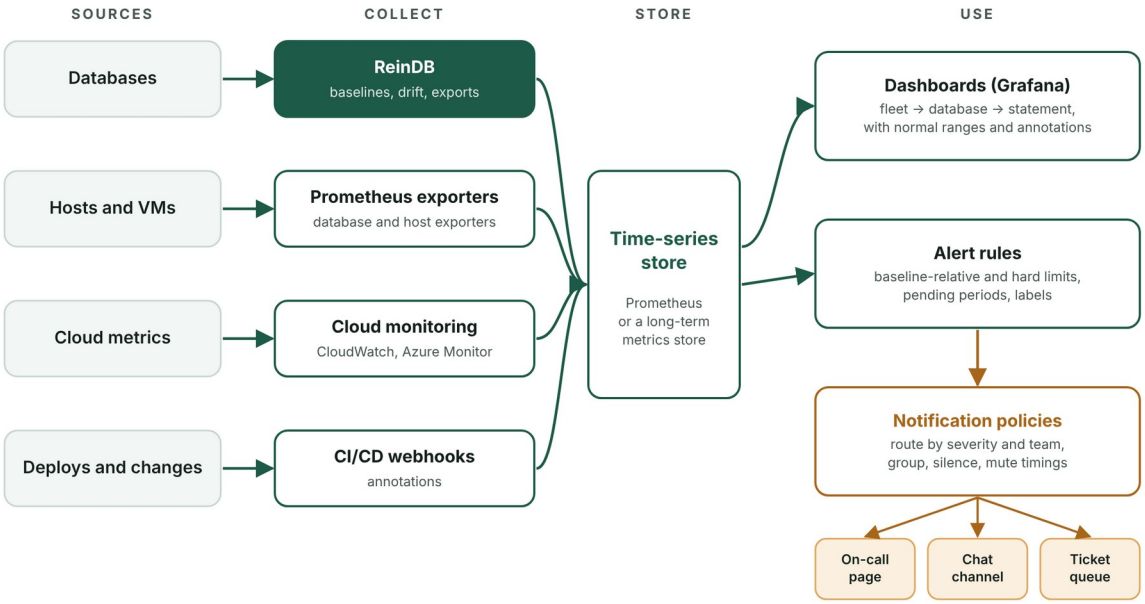
## 13.1 What to track

| Layer | Signals | Why |
|---|---|---|
| Experience | Application latency and error rates; timeouts | The symptom users feel; from your APM or service metrics |
| Database load | Database time or average active sessions against CPU cores; throughput; connections (active, idle, idle in transaction) | Is the database busier than usual for this hour? |
| Statements | Top statements by total time; regressions against baseline; new heavy statements; plan changes | The most common cause of change |
| Waits | Time by wait class over time | What kind of problem it is |
| Contention | Blocking time and chains; deadlocks; long transactions | Intermittent timeouts and stalls |
| Resources | CPU; memory; storage latency for reads and writes; IOPS and throughput against provisioned limits; network | The platform underneath (USE method) |
| Durability and availability | Replication lag; age of the last good backup; log, WAL or archive status; failover readiness | Problems that are invisible until they are disasters |
| Capacity | Storage used and days until full; growth by table; identity and sequence headroom (an `INT` key nearing 2.1 billion); file and autoextend limits; log and archive space | Outages you can see coming weeks ahead |
| Maintenance | Statistics age; cleanup lag and dead rows; failed maintenance jobs | Slow decay that turns into bad plans |
| Configuration and versions | Drift from approved settings; patch level against the rest of the fleet | Unplanned change, and known bugs |
| Security signals | Failed login spikes; privilege changes | Not performance, but the same pipeline can carry them |

ENGINE NOTES **Capacity and maintenance signals worth tracking**

- **PostgreSQL:** transaction ID age (`age(datfrozenxid)`), autovacuum lag and dead rows, and replication slots that hold WAL: an inactive slot keeps WAL until the disk fills.
- **MySQL:** purge lag (the history list length) and binary log disk use against `binlog_expire_logs_seconds`.
- **SQL Server:** why the transaction log can't be reused (`log_reuse_wait_desc` in `sys.databases`), file growth limits, and tempdb size.
- **Oracle:** tablespace and autoextend limits, fast recovery area usage (`V$RECOVERY_AREA_USAGE`), and the archived log destination.

## 13.2 The shape of a monitoring environment




*A monitoring environment: databases and hosts feed collectors and exporters, which feed a time-series store; dashboards and alert rules read from it, and alerts are routed to on-call, chat and ticket tools.*

- **Collectors and exporters** read the databases and hosts: Prometheus exporters (`postgres_exporter`, `mysqld_exporter`, Oracle's open-source database observability exporter, community SQL Server exporters), `node_exporter` and `windows_exporter` for hosts, cloud metrics such as Amazon CloudWatch, and database monitoring tools such as ReinDB.
- **A time-series store** keeps the numbers: Prometheus, or a long-term store behind it, or your cloud's metrics service.
- **Dashboards** (Grafana is the common choice) turn the numbers into views for different audiences.
- **Alert rules** evaluate the numbers continuously and send notifications through routing rules to on-call tools, chat and tickets.

- **Annotations** from your deploy pipeline and change process mark when things changed, on every chart.

Keep everything on one clock (UTC), with consistent database names and labels across tools, so a database is called the same thing on every dashboard and in every alert.

> CLOUD **The provider's monitoring is one more source**
>
> - **Amazon CloudWatch Database Insights** replaced Performance Insights, which reached end of life on July 31, 2026. Standard mode is free with 7 days of history. Advanced mode adds longer retention, fleet-wide views, execution plans for some engines and lock analysis for PostgreSQL.
> - **Azure and Google Cloud** offer Query Performance Insight (Azure SQL Database) and Query Insights (Cloud SQL) on top of their platform metrics.
> - **Grafana** has data sources for Amazon CloudWatch, Azure Monitor and Google Cloud Monitoring, so the platform limits from Chapter 5 (IOPS, throughput, credits) can sit on the same dashboards as the database's own measurements.

## 13.3 Exporting to Grafana

There are three common ways to get database performance data into Grafana:

1. **Exporters scraped by Prometheus.** Live, fine-grained metrics with little effort. Exporters give you raw counters, though: baselines, statement history and drift detection are up to you.
2. **A SQL data source.** Grafana can query PostgreSQL, MySQL, SQL Server and other databases directly. Point it at a monitoring repository (your own hourly snapshot tables, for example), never at production tables during busy hours.
3. **Files from a monitoring tool.** Scheduled exports of already-processed measurements (baselines, flags, top statements), loaded through a plugin or a data lake.

> REINDB **ReinDB exports for Grafana (Pro and Enterprise)**
>
> Scheduled exports copy ReinDB's own measurements, never table data, every hour, day or week. Choose databases and datasets (system measures, top statements, waits, open flags, sizes, replication and sessions) and a format: CSV or JSON Lines per dataset, or a Prometheus textfile of the latest values for `node_exporter`'s textfile collector. Files land in a folder you choose (self-hosted) or your organization's storage (hosted), and each export has a private pull link that Grafana, `curl` or a cron job can read.

**The Prometheus textfile route** gives Grafana ReinDB's baselines and flags next to your other metrics:

1. Run `node_exporter` with the textfile collector pointed at a directory, for example `--collector.textfile.directory=/var/lib/node_exporter/textfile`.
2. Create a ReinDB export in the Prometheus format with an hourly schedule.
3. Self-hosted: write the export into that directory. Hosted: have a cron job fetch the file through the export's pull link. Either way, write to a temporary file and rename it, so `node_exporter` never reads half a file.

4. Build panels and alert rules on the `reindb_*` metrics, for example `reindb_open_flags`, `reindb_system`, `reindb_statement_elapsed_seconds`, `reindb_wait_average_sessions`, `reindb_database_size_bytes`, `reindb_replication_lag_seconds` and `reindb_last_sample_timestamp_seconds`.

```
# Prometheus alert rules on ReinDB metrics (illustrative thresholds)
groups:
  - name: reindb
    rules:
      - alert: ReinDBHighDriftFlags
        expr: reindb_open_flags{severity="HIGH"} > 0
        for: 15m
        labels:
          severity: ticket
        annotations:
          summary: "{{ $labels.database }} has open HIGH drift flags"
          runbook_url: "https://wiki.example.com/runbooks/database-drift"
      - alert: ReinDBMeasurementsStale
        expr: time() - reindb_last_sample_timestamp_seconds > 3 * 3600
        for: 10m
        labels:
          severity: page
        annotations:
          summary: "No new measurement for {{ $labels.database }} in 3 hours"
```

**The CSV and JSON Lines route** suits trend dashboards and data lakes: read the files with a Grafana plugin that loads CSV or JSON over HTTP (such as the Infinity data source), or land them in object storage and query them through a lake engine that Grafana already uses.

## 13.4 Designing dashboards people actually use

- **Build in layers:** a fleet overview, then one dashboard per database, then a statement drill-down. Link them so a click moves down a layer with the same time range.
- **Show normal on every chart.** A band for the usual range at that hour turns a line into an answer. A spike inside the band is just Monday.
- **Annotate changes:** deploys, parameter changes, index builds, failovers.
- **Give each panel a question it answers.** "Is load higher than usual?" earns its place. "Buffer cache hit ratio" usually doesn't.
- **Use consistent units and time zones,** and label units in titles.
- **Build for each audience:** on-call needs "what is wrong right now"; DBAs need statement and wait detail; development teams need their own application's statements (filtered by application or module name, see Chapter 9); managers need service levels and capacity.
- **Keep the top of each dashboard sparse.** The first screen should answer "is anything wrong?" in five seconds.

A per-database dashboard that works well, from top to bottom: health and open flags; load against its usual band; waits by class (stacked); top statements (a table with time, calls, time per call and change against usual); contention (blocked sessions, longest wait, oldest transaction); capacity (size, growth, days until full); replication and backups.

## 13.5 Alerts: what to set and how to manage them

**Principles that keep alerting useful:**

- **Page only for things that are urgent, actionable and real.** Everything else goes to a ticket queue or a daily digest. If an alert fires and nobody needs to do anything, it shouldn't page.
- **Alert on symptoms and imminent failures,** not every cause: user-visible latency, the database being unreachable, a disk about to fill, replication about to break the recovery point objective.
- **Use baseline-relative conditions for load and latency,** and absolute conditions for hard limits.
- **Use a pending period** ("for 15 minutes") to avoid flapping on brief spikes.
- **Give every alert** an owner, a severity, a short description of the impact, a runbook link and a dashboard link.
- **Group and deduplicate** related alerts (by cluster or service) so one incident is one notification.
- **Silence during planned maintenance,** with an expiry, and test the whole path (rule, routing, notification) regularly.
- **Review alerts monthly.** Delete or fix any alert that fired without leading to action. Alert fatigue is how real pages get ignored.

In Grafana Alerting, those principles map directly onto its building blocks: **alert rules** (a query, a condition and a pending period), **labels** (severity, team, service), **contact points** (where notifications go), **notification policies** (routing by label, grouping and timing), and **silences** and **mute timings** for maintenance windows.

**A starter alert catalog:**

| Alert | Condition (adjust to your environment) | Severity | First action |
|---|---|---|---|
| Database unreachable | Connection or measurement failing for 2 to 5 minutes | Page | Check host, service and network; fail over if needed |
| Load far above usual | Load at least 3 times the usual for this hour, for 15 minutes | Page if users are affected, otherwise ticket | Top statements, waits and blocking now |
| New high-severity regression | A new HIGH drift flag | Ticket or chat | Open the statement and its plan history |
| Blocking | A session blocked for more than 5 minutes, or more than 5 waiters behind one blocker | Page | Identify the blocker; contact its owner; end it if necessary |
| Replication lag | Lag above the recovery point objective, for 5 minutes | Page | Replica health, long transactions, network |
| Storage | 85% used (ticket); 95% used, or fewer than 7 days until full (page) | Ticket, then page | Growth by table; purge, extend or tier |
| Backup age | No successful backup for more than 26 hours | Ticket, or page for critical systems | Fix the job; take a backup |

| Alert | Condition (adjust to your environment) | Severity | First action |
|---|---|---|---|
| Long transaction | An open transaction older than 1 hour | Ticket | Find the session; it holds back cleanup of old row versions and grows undo or version stores |
| Identity or sequence headroom | Above 80% of the type's maximum | Ticket | Plan the change to a larger type well ahead |
| Deadlocks | More than a few per hour, or any for critical transactions | Ticket | Read the deadlock graph; fix the access order |
| Measurements stale | No new monitoring data for 3 hours | Page (to the monitoring owner) | Fix the collector; you are flying blind |

ENGINE NOTES **Engine-specific alerts worth adding**

- **PostgreSQL:** transaction ID age. Page when `age(datfrozenxid)` passes about 1 billion. PostgreSQL stops accepting writes as transaction IDs approach wraparound, at a little over 2 billion, so the alert sits well before that point. Find what blocks freezing (long transactions, abandoned replication slots, forgotten prepared transactions) and vacuum the oldest tables.
- **MySQL:** a history list length that keeps growing for hours (purge can't keep up, usually behind a long transaction), and binary log disk use.
- **SQL Server:** a transaction log that can't be reused for a long time (`log_reuse_wait_desc` other than `NOTHING`), and Query Store switching itself to read-only when it runs out of space.
- **Oracle:** fast recovery area above 85%, tablespaces close to their limits, and archiver errors, which stop all changes until the archived log destination has room again.

## 13.6 Keeping it optimal: a review rhythm

| When | Review |
|---|---|
| Daily (5 minutes) | Open flags and new regressions; anything degraded or down; failed jobs and backups |
| Weekly | Top statements and their trend; new heavy statements; this week against last; storage growth; blocking and deadlock history |
| Monthly | Alert review (remove noise, add what was missed); settings drift; unused and duplicate indexes; capacity forecast; forced plans and hints |
| Quarterly | Version and patch levels across the fleet; disaster recovery test; baseline sanity; data tiering and retention review |
| After every release | The same hours before and after, for the statements the release touched |

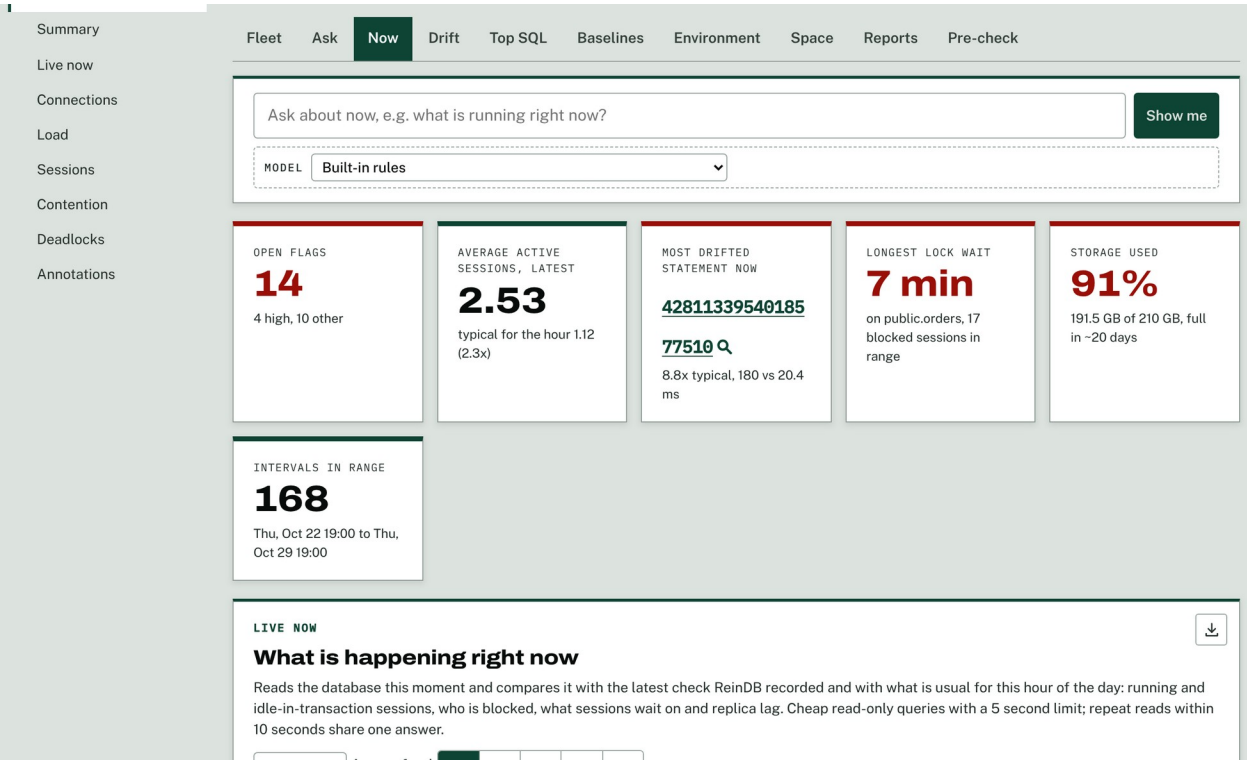
Give each review an owner and a place to record what was found. Postmortems should end with a new check, dashboard panel or alert, so each incident makes the system better at catching the next one.

> REINDB **Where ReinDB helps**
>
> The Fleet page shows every database on one screen, worst first, with down and degraded databases (and the reason) highlighted at the top, plus version and patch level and a "behind" marker when another database of the same major version runs a newer patch. Alerts go to phones for new regressions, and the monitor's exit code and command hook can send mail, open a ticket or page someone. A daily email, HTML reports, CSV, JSON and Prometheus exports, a REST API and audited read-only MCP tools connect ReinDB to everything else.

# 14 How ReinDB fits in

ReinDB is Bandlei's read-only drift monitoring for PostgreSQL, MySQL, SQL Server and Oracle, self-managed or on Amazon RDS and Aurora. It learns what normal looks like for every database, flags what drifts from it, answers plain-English questions with charts, and writes the fix as a script your DBA can test. It never changes your database.

It was built by database practitioners to do the repetitive parts of this guide every hour, for every database, without anyone having to remember.




*The Now page: open flags, load against what is usual for the hour, the most drifted statement, the longest lock wait and storage used, with Live now below. Demo database, synthetic data.*

## 14.1 ReinDB at each step of this guide

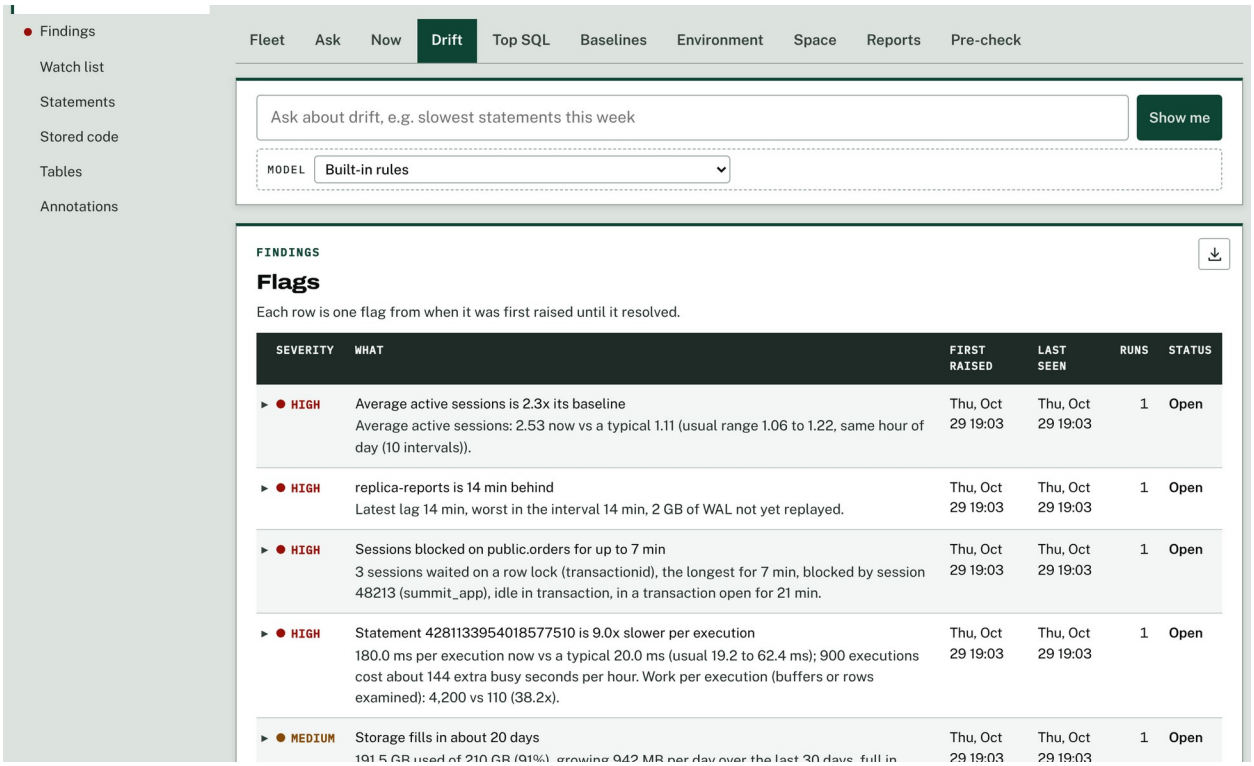
| Guide chapter | What ReinDB does for you |
|---|---|
| 1. How to start | The Now page and the health check report cover most of the first-hour checklist: flags, load against usual, the worst statement, lock waits, storage, version and patch level, settings |
| 2. Toolkit | One read-only login per database with the smallest grants, from a script per engine. Reads each engine's own statement statistics, needs no host access on managed services, and on Oracle needs no Diagnostics or Tuning Pack |
| 3. Data profile | Database size every run and table sizes daily, with growth and days-until-full projections; busiest tables; Top SQL |

| Guide chapter | What ReinDB does for you |
|---|---|
| 4. Platform choices | Table activity, size history and storage projections: the evidence for keeping, partitioning or tiering a table |
| 5. What to look for | Drift flags ranked by the extra database time each costs; time split into CPU, I/O wait and lock wait per statement; wait profile over time; Live now for current sessions, blocking, waits and replica lag |
| 6. Tuning procedure | Statement pages with plan history and analysis (disk reads, temp spills, rows examined per row returned, full scans, N+1 patterns, misestimates on Oracle); fix scripts with an undo for each step; Pre-check before deploying |
| 7 and 8. Joins, predicates and indexes | Index suggestions from plans; unused-index findings where the engine records usage; Pre-check flags risky query and DDL changes |
| 9. Procedure code | The Stored code report on all four engines: row-by-row loops, queries per row, concatenated SQL and other patterns, with the statements inside each routine and a set-based rewrite sketch |
| 11. Settings | Settings review judged against Live, Average or Peak, with a commented fix script, parameter-group instructions on managed services, and scoped session-level settings for the statements that need them |
| 12. Baselining | An automatic 14-day, hour-of-day baseline for every database and statement; one-click comparisons; saved baselines such as "Before release 4.2"; the "Vs baseline" report |
| 13. Observability | The Fleet page; phone alerts; exit code and command hook for mail, tickets or pages; scheduled exports for Grafana (CSV, JSON Lines or Prometheus textfile); REST API and read-only MCP tools |

## 14.2 A quick tour

- **Fleet:** every database on one page, worst first. Down and degraded databases are listed at the top with the reason; each tile shows engine, version, uptime, load against usual and the newest measurement. Pro and Enterprise add groups (production, staging, a customer) and side-by-side comparison of one engine's databases.
- **Ask:** type a question such as "show drift for the last week as a graph" or "which queries use orders this week?" and get a chart or table with a caption computed from the measurements. A model only interprets the question and picks what to show; it never invents the numbers. Built-in rules work with no model at all, and you can bring your own OpenAI-compatible model.
- **Now:** current activity against what is usual for this hour, Live now, long-running sessions, blocking and deadlocks.
- **Drift:** what got slower. Flags, statement drift, the watch list, stored code and the busiest tables.
- **Top SQL:** the biggest consumers over any range, with when each statement first and last appeared.
- **Baselines:** one-click comparisons and saved, named baselines.
- **Environment:** waits, CPU, I/O, storage and growth, replication, and the database review with settings.
- **Space:** where the space goes and how fast it grows (Oracle and PostgreSQL).

- **Reports:** workload, top SQL, waits, I/O, blocking, capacity, stored code and "Vs baseline" reports over any range, printable as PDF, plus reports of your own.
- **Pre-check:** paste a query, procedure or DDL change before it ships. ReinDB asks the database for the plan only (nothing is executed) and returns Pass, Review or Risky, with the production statement it would replace, its calls per hour, a suggested rewrite or index, and the quietest hour to deploy.




*The Drift page: each flag from when it was first raised until it resolves, with severity, what drifted, the usual range for that hour and the extra time it costs. Demo database, synthetic data.*

## 14.3 Getting started

1. **Choose where ReinDB runs.** Hosted by Bandlei; onsite on Linux or Windows; or in your own AWS or Google Cloud account. Onsite and in-cloud editions run as one container or Python service, and nothing is installed in the database. For networks where ReinDB may not connect to the database at all, a small local collector can measure the database and send the results instead.
2. **Prepare each database.** Run the grant script for its engine to create the read-only login, and make sure the engine's statement statistics source is on (see the engine note below). Optionally, allow Analyze to read plans (an extra grant described in the user guide) so it can suggest indexes from them. For the hosted service, allow ReinDB's address through your firewall or security group.
3. **Add the database** in the web app, choose the encryption setting, and test the connection.
4. **Let the baseline form.** The first measurement interval needs two runs, so allow an hour or two. Database-wide flags start after five hourly runs, statement flags after a statement has appeared in three runs, and the baseline is complete after 14 days.

5. **Set up alerts.** Install the app on your phone from the browser for notifications of new regressions, and set quiet hours. Self-hosted, wire the command hook to your mail, ticketing or paging tool.
6. **Connect your observability stack** (Pro and Enterprise): create a scheduled export for Grafana, as described in Chapter 13.
7. **Adopt the routine below.**

> ENGINE NOTES **What each engine needs before you add it**
>
> - **PostgreSQL:** `pg_stat_statements` in `shared_preload_libraries`, and the extension created in the database.
> - **MySQL:** `performance_schema`, which is on by default.
> - **SQL Server:** Query Store, recommended for statement history that survives restarts.
> - **Oracle:** nothing extra. ReinDB uses V$ views only and needs no Diagnostics or Tuning Pack.

## 14.4 A week with ReinDB

| When | What to do in ReinDB |
|---|---|
| Every morning | Glance at Fleet and Now, or read the daily email. Open any new HIGH flag, click the statement, read where its time goes and its plan history |
| Before a release | Pre-check the changed queries, procedures and DDL; save a baseline named for the release |
| After a release | Compare the release baseline with the same hours since; check the watch list |
| Weekly | Run the "Vs baseline" report for this week against last; review Top SQL and storage growth |
| Monthly | Review settings against Peak; check the Fleet page for databases behind on patches; read the Stored code report for the busiest routines |
| During an incident | Live now for sessions, blockers and waits; Ask for "what changed since yesterday?"; download the fix script and test it |

## 14.5 Built for strict security boundaries

- Read-only sessions with each engine's smallest grant. ReinDB never runs a fix; it writes scripts for your change control.
- No passwords in files: the operating system keyring or a per-database environment variable onsite, and encrypted credentials in the hosted service.
- Literals are redacted from statement text before anything reaches a model, and you can choose any approved OpenAI-compatible model, or none.
- Every question and tool call is written to an audit log.
- TLS with certificate verification to every database.
- Payment cards are handled by Stripe only.

## 14.6 Plans

The first database is free, with no time limit. Paid plans (Starter, Pro and Enterprise) add more databases and people and phone alerts; Pro and Enterprise add fleet groups and comparison, scheduled exports for Grafana and Excel import. See reindb.com for current plans and pricing.

> REINDB **Start free**
>
> Add your first database at reindb.com and see its baseline take shape over the first day. Questions, or want a walkthrough for your team? Write to inquiries@bandlei.com.

# Appendix A. Checklists

## A.1 First hour on an unfamiliar database

- ☐ Version, patch level, edition and platform recorded
- ☐ Time since the last restart known (it limits what the counters cover)
- ☐ Size, growth and storage headroom for data, log or WAL, temp and archive destinations
- ☐ Last good backup confirmed; replica lag checked
- ☐ Top wait classes and the time model read
- ☐ Top statements by total time listed
- ☐ Blocking chains, long transactions and sessions idle inside one checked
- ☐ Statistics freshness and maintenance job history checked
- ☐ Recent deployments, parameter changes and incidents collected
- ☐ Settings far from defaults listed

## A.2 Statement tuning

- ☐ Goal stated as a number, with the hours it applies to
- ☐ Baseline recorded: executions, time per execution, logical reads per execution, rows, plan identifier
- ☐ Full text and representative parameter values captured, including skewed values
- ☐ Actual plan captured with runtime row counts
- ☐ First large gap between estimated and actual rows found
- ☐ One hypothesis written in one sentence
- ☐ Least invasive fix chosen (rewrite, statistics, index, plan pin, setting, schema)
- ☐ Tested on production-like data with several parameter values; logical reads compared
- ☐ Rollback script written and tested
- ☐ Deployed at a quiet hour; owners told
- ☐ Verified against the baseline at the same hours on following days
- ☐ Change, reason and numbers recorded

## A.3 Index change

- ☐ The queries the index serves are listed, with their share of database time
- ☐ Column order follows equality, then range, then order, then covering columns
- ☐ Overlap with existing indexes checked; redundant ones identified
- ☐ Write overhead considered for the table's insert, update and delete rate

- ☐ Built online or concurrently; invisible first where the engine supports it
- ☐ Top statements on the table re-checked for plan changes afterwards
- ☐ For drops: counters cover a full business cycle; not used for keys, uniqueness, foreign keys or replicas

## A.4 Settings change

- ☐ Workload class and the right moment (average or peak) identified
- ☐ Current value, proposed value and reason recorded
- ☐ Scoped override considered before a global change
- ☐ Restart or reboot requirement known (static parameters, parameter groups)
- ☐ One change at a time; measured at the same hours on following days
- ☐ Approved configuration in version control updated

## A.5 Monitoring starter set

- ☐ Statement history source on for every database (Chapter 2)
- ☐ Least-privilege monitoring login on every database
- ☐ At least 14 days of hourly history kept
- ☐ Dashboards: fleet overview, per database, statement drill-down, with normal ranges shown
- ☐ Deploys and changes annotated on dashboards
- ☐ Alerts: unreachable, load far above usual, blocking, replication lag, storage, backup age, stale measurements
- ☐ Each alert has an owner, severity, runbook link and dashboard link
- ☐ Monthly alert review on the calendar

## A.6 Running on a managed service

- ☐ License model known (included or brought) and what it allows: edition, packs and options
- ☐ Monitoring login works without SYSDBA, sysadmin or superuser rights (section 2.5)
- ☐ Parameter groups or flags kept in version control; static parameters and reboot needs known
- ☐ Storage IOPS, throughput and burst credits monitored against their limits
- ☐ Instance class limits known: CPU credits, network and log throughput
- ☐ Extensions, plugins and options you rely on are on the provider's list
- ☐ Provider monitoring tier, retention and cost known
- ☐ Logs and trace files reachable through the console or API
- ☐ Maintenance windows and automatic minor upgrades annotated on dashboards

# Appendix B. Version notes

## B.1 Oracle 11g against 19c

Oracle 11g (11.2) is long past the end of its support, but plenty of it still runs. 19c is the long-term support release most estates have moved to. The differences below change how you tune.

| Area | 11g Release 2 | 19c |
|---|---|---|
| Top-N and paging syntax | `ROWNUM` around an ordered subquery | `FETCH FIRST n ROWS ONLY` and `OFFSET` (12c+) |
| Architecture | Non-CDB only | Multitenant (CDB and PDB); V$ views carry `CON_ID`, and a PDB user sees only its PDB. Non-CDB is deprecated but still supported in 19c |
| Adaptive optimization | Cardinality feedback | Adaptive plans (on by default), statistics feedback and SQL plan directives; adaptive statistics off by default from 12.2 |
| Histograms | Frequency and height-balanced | Adds top-frequency and hybrid histograms (12c), built when statistics are gathered with `AUTO_SAMPLE_SIZE` (the default) |
| Statistics for bulk loads | Gather after the load | Online statistics gathering for `CREATE TABLE AS SELECT` and direct-path loads into empty tables (12c) |
| Real-time statistics, high-frequency statistics, automatic indexing, SQL Quarantine | Not available | New in 19c, but licensed only on Exadata: Enterprise Edition on Exadata, Exadata Database Service (including Oracle Database@Azure, Google Cloud and AWS) and Autonomous Database. Not on Base Database Service or Amazon RDS (section 2.6) |
| PGA control | `PGA_AGGREGATE_TARGET` is a target only | `PGA_AGGREGATE_LIMIT` hard cap (12c) |
| Index usage | `ALTER INDEX ... MONITORING USAGE`, visible to the index owner in `V$OBJECT_USAGE` | `DBA_INDEX_USAGE`, collected automatically (12.2+) |
| Partitioned indexes | Full indexes only | Partial indexes switched on or off per partition (12c) |
| Online operations | Online index builds and some redefinition | Online partition moves (12c), online table moves and online conversion to partitioned (12.2) |
| Key generation | Sequence plus trigger | Identity columns (12c) |
| Identifier length | 30 bytes | 128 bytes (12.2) |
| PL/SQL called from SQL | Context switch per call | `PRAGMA UDF` and `WITH FUNCTION` reduce the cost (12c) |
| Approximate aggregates | Not available | `APPROX_COUNT_DISTINCT` (12c) and approximate percentiles (12.2) for fast estimates over large data |

| Area | 11g Release 2 | 19c |
|---|---|---|
| Plan display | `DBMS_XPLAN` with `ALLSTATS LAST` | Adds `+ADAPTIVE` (12c) and a hint usage report (19c) that shows which hints were used or ignored |

## B.2 SQL Server

| Version | Changes that matter for tuning |
|---|---|
| 2016 | Query Store; live query statistics; with SP1, partitioning, compression and columnstore in all editions, and `CREATE OR ALTER` |
| 2017 | Automatic plan correction; adaptive joins and memory grant feedback (batch mode); interleaved execution for multi-statement functions; per-query wait statistics in Query Store; resumable online index rebuilds |
| 2019 | Scalar UDF inlining; table variable deferred compilation; batch mode on rowstore; row-mode memory grant feedback; `OPTIMIZE_FOR_SEQUENTIAL_KEY`; last actual plan (`LAST_QUERY_PLAN_STATS`); accelerated database recovery; resumable online index creation |
| 2022 | Parameter Sensitive Plan optimization; Query Store on by default for new databases, and Query Store hints; degree-of-parallelism and cardinality estimation feedback; granular permissions such as `VIEW SERVER PERFORMANCE STATE`; Parquet and Delta files in S3-compatible object storage |
| 2025 | Optimized locking (needs accelerated database recovery); optional parameter plan optimization for catch-all predicates; cardinality estimation feedback for expressions; degree-of-parallelism feedback on by default; the `ABORT_QUERY_EXECUTION` hint to block a known bad query; tempdb space resource governance; Query Store on readable secondaries |

Many of the newer optimizer features depend on the database compatibility level, not just the server version. Upgrading the server without raising the compatibility level leaves them switched off. SQL Server 2016 left extended support in July 2026.

## B.3 PostgreSQL

| Version | Changes that matter for tuning |
|---|---|
| 10 | Declarative partitioning; `CREATE STATISTICS` for correlated columns; the `pg_monitor` role |
| 11 | Hash partitioning and default partitions; `INCLUDE` columns in indexes; JIT compilation; procedures that can `COMMIT` |
| 12 | `REINDEX CONCURRENTLY`; `plan_cache_mode`; most-common-value extended statistics; common table expressions inlined by default (add `MATERIALIZED` to keep the old behavior) |
| 13 | `pg_stat_statements` renames its timing columns (`total_exec_time`) and adds planning time; incremental sort; B-tree deduplication |
| 14 | `DETACH PARTITION CONCURRENTLY`; query identifiers computed in core (`compute_query_id`); `pg_stat_wal` |
| 15 | `MERGE`; cumulative statistics kept in shared memory instead of a statistics collector process |

| Version | Changes that matter for tuning |
|---|---|
| 16 | `pg_stat_io`; `EXPLAIN (GENERIC_PLAN)` |
| 17 | Incremental backups; much more memory-efficient vacuum; `EXPLAIN` options `SERIALIZE` and `MEMORY` |
| 18 | Asynchronous I/O; B-tree skip scan; planner statistics kept through `pg_upgrade` |
| 19 (release candidate, October 2026) | `NOT IN` turned into an anti-join when no NULLs are possible; parallel autovacuum workers; `EXPLAIN (ANALYZE, IO)`; the `pg_stat_lock` view; generic and custom plan counts in `pg_stat_statements`; `log_lock_waits` on by default |

PostgreSQL 13 has been out of support since November 2025, and 14 reaches end of life on November 12, 2026.

## B.4 MySQL

| Version | Changes that matter for tuning |
|---|---|
| 5.7 | Optimizer hints; JSON; the `sys` schema included. End of life since October 2023 |
| 8.0 | Common table expressions and window functions; histograms; invisible and descending indexes; functional indexes and skip scan (8.0.13); hash joins and `EXPLAIN ANALYZE` (8.0.18); instant `ADD COLUMN`; the query cache removed; `innodb_redo_log_capacity` (8.0.30). End of life since April 2026 |
| 8.4 LTS | Several InnoDB defaults changed (for example, the adaptive hash index is off by default and `innodb_io_capacity` is higher); review settings carefully when upgrading |
| 9.7 LTS | Released April 2026. The hypergraph join optimizer and the telemetry component are now in the Community edition. Turn the hypergraph optimizer on for one session (`optimizer_switch`) and compare plans before using it more widely |

Check the vendor's current support status for any release you run.

# Appendix C. Glossary

| Term | Meaning |
|---|---|
| Average active sessions (AAS) | Database time divided by elapsed time: how many sessions were working or waiting, on average. Compare it with the number of CPU cores |
| Baseline | A description of normal for a database at a given time, used to tell real change from routine variation |
| Cardinality | The number of rows an operation returns, or the number of distinct values in a column |
| Clustering factor | How closely a table's physical row order follows an index's order; it decides how expensive index range scans are |
| Covering index | An index that contains every column a query needs, so the table isn't read |
| Database time | Total time sessions spend working or waiting in the database (CPU plus non-idle waits) |
| Drift | A measurable departure from a database's or statement's own normal behavior |
| Hard parse | Compiling a new statement and choosing its plan, as opposed to reusing a cached one |
| Histogram | Statistics describing how values are distributed in a column, used for skewed data |
| Implicit conversion | A type conversion the engine adds on its own; on a column, it usually disables the index |
| Keyset pagination | Paging by remembering the last key and seeking past it, instead of using `OFFSET` |
| Logical read | A read of a block or page from memory; the most stable measure of a statement's work |
| Median absolute deviation (MAD) | A robust measure of spread: the median of the distances from the median |
| Partition pruning | Skipping partitions that can't contain matching rows, based on predicates on the partition key |
| Plan regression | A statement getting slower because the optimizer chose a different, worse plan |
| RBAR | "Row by agonizing row": procedural, one-row-at-a-time processing where a set operation would do |
| Sargable | A predicate that can use an index to find matching rows directly |
| Seasonal baseline | A baseline that compares each hour with the same hour on previous days |
| Selectivity | The fraction of rows a predicate matches |
| Wait event | A named reason a session is waiting instead of working: I/O, locks, commits, network |
| Watermark | The boundary value that separates data in one tier (or one load) from the next |

# Appendix D. Sources and further reading

Vendor documentation is the final word on behavior and licensing for your exact version.

- Oracle Database Licensing Information User Manual (19c): docs.oracle.com/en/database/oracle/oracle-database/19/dblic/
- Oracle Database Performance Tuning Guide (19c): docs.oracle.com/en/database/oracle/oracle-database/19/tgdba/
- Oracle Database SQL Tuning Guide (19c): docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/
- Oracle Database PL/SQL Language Reference (19c), Bulk SQL and Bulk Binding: docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/bulk-sql-and-bulk-binding.html
- Oracle Base Database Service: oracle.com/database/base-database-service/
- Oracle Autonomous Database FAQ: oracle.com/database/technologies/datawarehouse-bigdata/adb-faqs.html
- Amazon RDS for Oracle licensing options: docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Oracle.Concepts.Licensing.html
- Amazon RDS for Oracle, Statspack: docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Appendix.Oracle.Options.Statspack.html
- Amazon RDS for Oracle, AWR reports: docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Appendix.Oracle.CommonDBATasks.AWR.html
- Amazon CloudWatch Database Insights: docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Database-Insights.html
- Microsoft Learn, What's new in SQL Server 2025: learn.microsoft.com/en-us/sql/sql-server/what-s-new-in-sql-server-2025
- Microsoft Learn, Azure SQL Database server roles: learn.microsoft.com/azure/azure-sql/database/security-server-roles
- Microsoft Learn, Monitor performance by using the Query Store: learn.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store
- Microsoft Learn, Prerequisites for minimal logging in bulk import: learn.microsoft.com/en-us/sql/relational-databases/import-export/prerequisites-for-minimal-logging-in-bulk-import
- Microsoft Learn, Use table-valued parameters: learn.microsoft.com/en-us/sql/relational-databases/tables/use-table-valued-parameters-database-engine
- Microsoft Learn, sys.dm_os_wait_stats: learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-os-wait-stats-transact-sql
- PostgreSQL documentation, pg_stat_statements: postgresql.org/docs/current/pgstatstatements.html

- PostgreSQL documentation, The Cumulative Statistics System: postgresql.org/docs/current/monitoring-stats.html
- PostgreSQL documentation, Table Partitioning: postgresql.org/docs/current/ddl-partitioning.html
- PostgreSQL documentation, Populating a Database: postgresql.org/docs/current/populate.html
- MySQL Reference Manual, MySQL Performance Schema: dev.mysql.com/doc/refman/8.4/en/performance-schema.html
- MySQL Reference Manual, Restrictions and Limitations on Partitioning: dev.mysql.com/doc/refman/8.4/en/partitioning-limitations.html
- MySQL Reference Manual, Optimizing INSERT Statements: dev.mysql.com/doc/refman/8.4/en/insert-optimization.html
- Trino documentation: trino.io/docs/current/
- Apache Iceberg documentation: iceberg.apache.org/docs/latest/
- Grafana Alerting documentation: grafana.com/docs/grafana/latest/alerting/
- Brendan Gregg, The USE Method: brendangregg.com/usemethod.html
- Google, Site Reliability Engineering, "Monitoring Distributed Systems": sre.google/sre-book/monitoring-distributed-systems/
- Cary Millsap and Jeff Holt, *Optimizing Oracle Performance* (O'Reilly, 2003), the origin of Method R

ReinDB

FROM BANDLEI

## Let ReinDB run the loop every hour

This guide is a method: measure, compare with normal, find the biggest consumer, explain it, fix it and verify. ReinDB runs that loop for PostgreSQL, MySQL, SQL Server and Oracle, self-managed or on Amazon RDS and Aurora.

- **Read-only, least-privilege access.** On Oracle it uses V$ views only, with no Diagnostics or Tuning Pack needed.
- **Learns normal for every database and hour,** and flags what drifted, ranked by the extra time it costs.
- **Explains each flag with evidence** and writes the fix as a commented script with an undo, for your team to review.
- **Pre-checks a change before it ships,** using EXPLAIN only, and suggests the quietest hour to deploy.
- **Fits your stack:** scheduled exports for Prometheus and Grafana on the Pro and Enterprise plans.

**Your first database is free, with no time limit.**

Start at **reindb.com**

Questions, or a walkthrough for your team? Write to inquiries@bandlei.com

Bandlei, Inc. · Evergreen, Colorado · bandlei.com