

SQL Database Types:

An Introduction and Course Roadmap



Three axes: engine family, deployment model, SQL dialect



Four families: self-managed, cloud-managed, distributed SQL, embedded



Why it matters: dialect skills, analytics tooling, cost, lock-in



Default in 2026: a managed PostgreSQL variant



Roadmap: foundations → families → dialects → workload fit → selection → practice



Foundations

Families

Dialects

Workload fit

Selection

Practice

Core Relational Concepts Still Underpin Every SQL Engine

Learn the foundation that every SQL database type shares — from PostgreSQL to Spanner.



Tables, rows, keys, and foreign keys define relationships; modeling makes joins meaningful



ACID transactions guarantee atomicity, consistency, isolation, and durability across engines



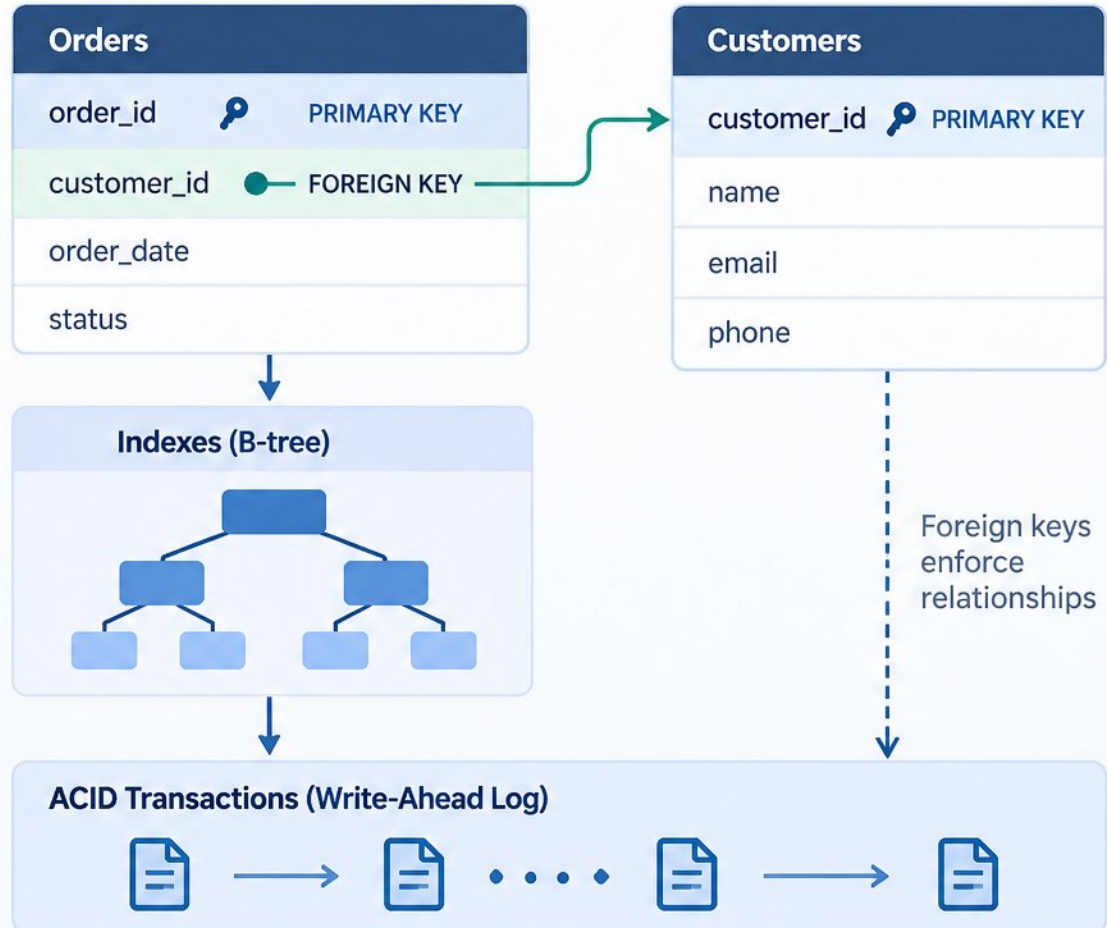
Normalize to 3NF first; denormalize only with a stated reason



Constraints (NOT NULL, CHECK, UNIQUE, FK) enforce correctness at the storage layer



Indexes trade read speed for write cost; missing foreign key indexes are a silent performance bug



Row-store



Column-store



Distributed



Embedded

Traditional Relational Engines: The Self-Managed Incumbents

TRUSTED
ENGINES
LASTING
FOUNDATIONS



PostgreSQL

LICENSE	BSD/MIT-style
CURRENT RELEASE	PostgreSQL 18 at 18.6; 19 in beta
END OF SUPPORT	PostgreSQL 14 support ends 12 November 2026



MySQL

LICENSE	GPLv2
CURRENT RELEASE	MySQL now on CalVer: 26.10 Early Access alongside 9.7 and 8.4 LTS
END OF SUPPORT	MySQL 8.0 in Sustaining Support since <u>21 April 2026</u>



SQL Server

LICENSE	
CURRENT RELEASE	
END OF SUPPORT	



Oracle

LICENSE	
CURRENT RELEASE	
END OF SUPPORT	

- PostgreSQL 18 at 18.6; 19 in beta; five-year support per major
- MySQL now on CalVer: 26.10 Early Access alongside 9.7 and 8.4 LTS
- MySQL 8.0 in Sustaining Support since 21 April 2026
- PostgreSQL 14 support ends 12 November 2026 — put dates in your calendar
- Licensing: PostgreSQL permissive (BSD/MIT-style) vs MySQL GPLv2
- Strengths: maturity, tooling, skills. Limits: single-primary writes, operational toil

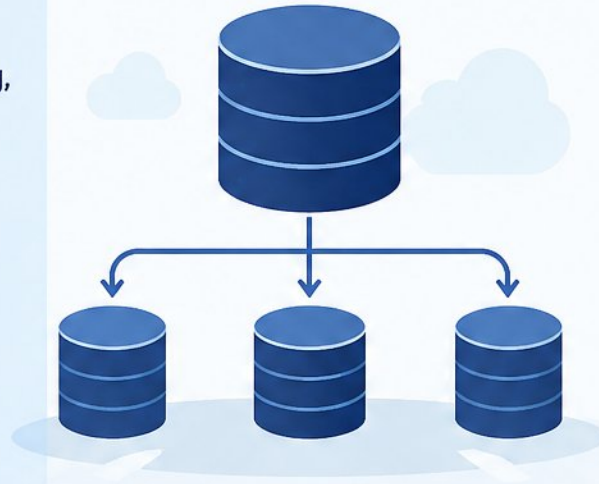


Cloud-Managed and DBaaS: What You Are Actually Buying

Managed Provider Handles



Managed services handle provisioning, patching, backups, PITR, failover, and replica wiring



You Still Own



You still own schema design, queries, tuning, and the monthly bill



Three Ecosystems



AWS RDS/Aurora



Azure SQL plus
PostgreSQL Flexible Server



GCP Cloud SQL/AlloyDB/Spanner



Cloud-native engines (Aurora, AlloyDB) add custom storage and columnar speed on PostgreSQL/MySQL compatibility



Trade-offs: lower operational toil versus cost, egress, portability, and provider-specific lock-in



Managed almost always wins for small teams; self-host only with a defensible reason.

Distributed SQL and NewSQL:

Horizontal Scale with Relational Semantics

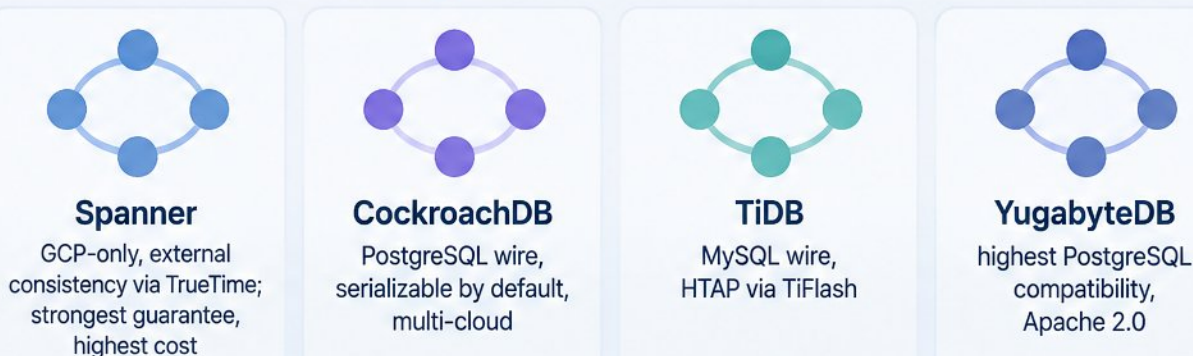


- 1 Shared-nothing nodes replicate each range via Raft or Paxos consensus
- 2 Spanner: GCP-only, external consistency via TrueTime; strongest guarantee, highest cost
- 3 CockroachDB: PostgreSQL wire, serializable by default, multi-cloud
- 4 TiDB: MySQL wire, HTAP via TiFlash; YugabyteDB: highest PostgreSQL compatibility, Apache 2.0
- 5 PACELC reality: the daily trade-off is latency versus consistency
- 6 Reach for it only when writes exceed one primary or must stay consistent across regions

Shared-Nothing Cluster



Multi-Region Placement (Examples)



PACELC Reality: Latency vs Consistency

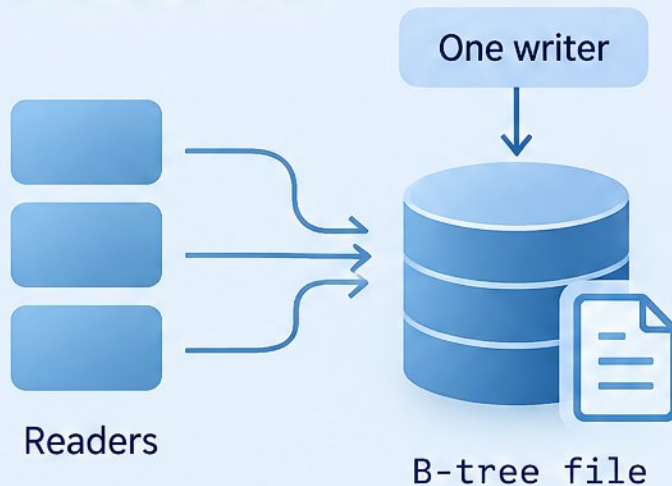


Embedded and Local Engines: SQLite and DuckDB

- Both are in-process, serverless, single-file databases
- Complements, not competitors; each sits at a different pipeline layer
- SQLite: row-oriented OLTP, ACID, WAL mode, one writer plus readers
- DuckDB: columnar OLAP, queries CSV/Parquet/JSON directly, no load step
- DuckDB aggregates 1M rows 10-200x faster than SQLite
- Neither replaces multi-writer servers: SQLite serializes writes; DuckDB single read-write process

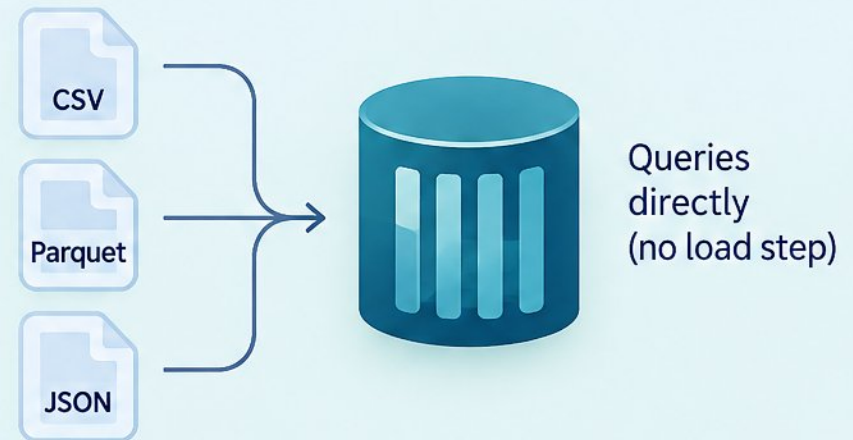
SQLite

Row-oriented OLTP



DuckDB

Columnar OLAP



SQL Dialects and Compatibility: Reading and Porting Queries

SAME IDEA
DIFFERENT
SYNTAX



Feature	PostgreSQL	MySQL	SQL Server	Oracle
Row limiting	<code>SELECT * FROM users LIMIT 10;</code>	<code>SELECT * FROM users LIMIT 10;</code>	<code>SELECT TOP 10 * FROM users;</code>	<code>SELECT * FROM users FETCH FIRST 10 ROWS ONLY;</code>
Concatenation	<code>SELECT 'A' 'B';</code>	<code>SELECT CONCAT('A', 'B');</code>	<code>SELECT 'A' + 'B';</code>	<code>SELECT 'A' 'B';</code>
NULL fallback	<code>SELECT COALESCE(name, 'N/A');</code>	<code>SELECT IFNULL(name, 'N/A');</code>	<code>SELECT ISNULL(name, 'N/A');</code>	<code>SELECT NVL(name, 'N/A');</code>
Date functions	<code>SELECT CURRENT_DATE;</code>	<code>SELECT CURDATE();</code>	<code>SELECT CAST(GETDATE() AS DATE);</code>	<code>SELECT SYSDATE;</code>
Auto-increment	id SERIAL PRIMARY KEY	id AUTO_INCREMENT PRIMARY KEY	id INT IDENTITY PRIMARY KEY	id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY
Identifier quoting	<code>SELECT "user" FROM "order";</code>	<code>SELECT `user` FROM `order`;</code>	<code>SELECT [user] FROM [order];</code>	<code>SELECT "user" FROM "order";</code>

- ANSI/ISO standard exists, but no vendor implements it fully
- Row limiting: LIMIT / TOP / FETCH FIRST / ROWNUM
- Concatenation: || / + / CONCAT
- NULL fallback: COALESCE / IFNULL / ISNULL / NVL
- Oracle DATE stores time; MySQL ignored CHECK before 8.0.16
- Prefer ANSI forms; verify collation and casting manually



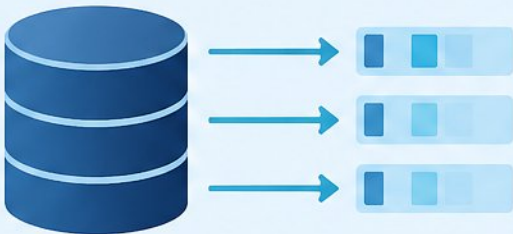
Workload Fit: OLTP, OLAP, and HTAP Explained

- **OLTP:** many short transactions on a few rows by key (1–10 ms target)
- **OLAP:** few queries scanning millions to billions of rows (seconds to minutes)
- Row store for point access, columnar for wide scans — up to 46x faster aggregation
- Access pattern, not data size, decides the engine
- HTAP works below ~100 GB warm analytical data; then CDC-separated engines win
- Co-locating analytics moved worst-case writes from 38 ms to 332 ms

OLTP

Many short transactions
on a few rows by key

1–10 ms target



OLAP

Few queries scanning
millions to billions of rows

Seconds to minutes

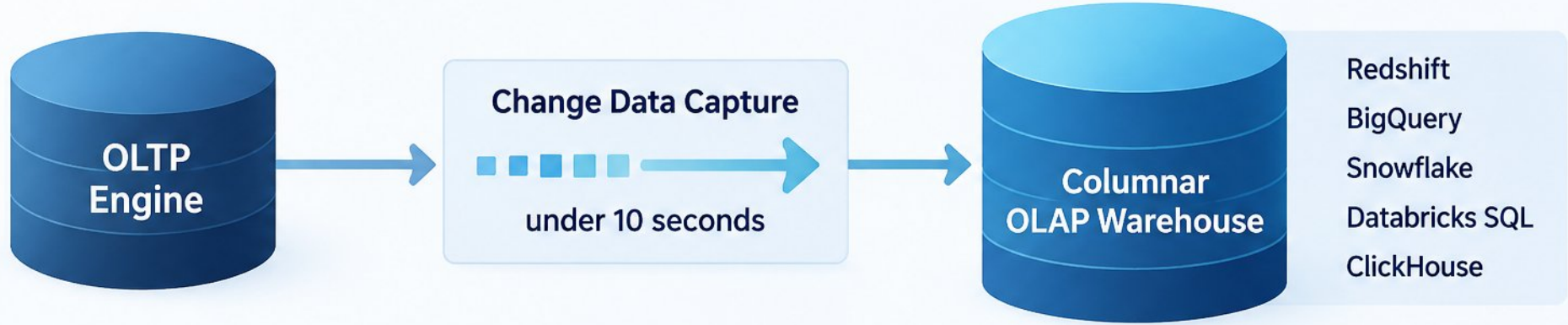


Co-locating Analytics and Write Latency

Worst-case writes



Analytical Engines and the Composed Data Stack



Same architecture
at local scale



- Cloud columnar warehouses: Redshift, BigQuery, Snowflake, Databricks SQL, ClickHouse
- 2026 default: OLTP plus columnar OLAP, joined by CDC under 10 seconds
- Zero-ETL (Aurora→Redshift, Fabric Mirroring) makes the pipeline operationally cheap
- Converged HTAP stays a compromise: storage, I/O, and latency pull opposite ways
- DuckDB and SQLite repeat the row/column split at local scale
- Above ~100 GB warm analytical data, split into dedicated engines

Deployment Models and Serverless Options Compared



self-hosted



cloud VM



managed
instance



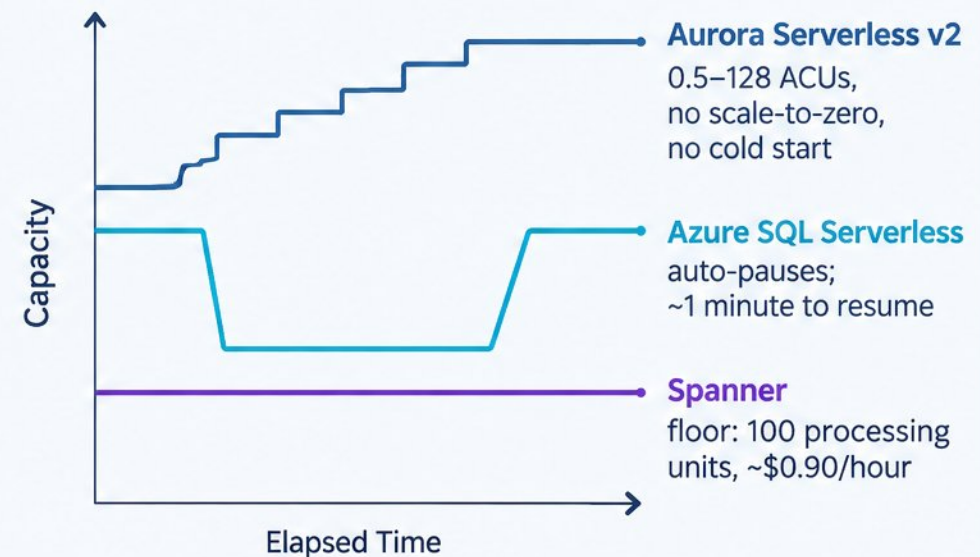
serverless



embedded
in-process

- **Five shapes:** self-hosted, cloud VM, managed instance, serverless, embedded in-process
- **Aurora Serverless v2:** 0.5–128 ACUs, no scale-to-zero, no cold start
- **Azure SQL Serverless** auto-pauses; ~1 minute to resume
- **Spanner floor:** 100 processing units, ~\$0.90/hour
- **A database on a cloud VM is not managed** — patching, backups, failover are still yours
- **Billing models:** per-instance, scale-to-zero, per-request, resource-based — match traffic shape first
- **Egress and cross-AZ transfer** often exceed compute for chatty apps

Scaling Behavior Over Time



Lower Latency

Tail latency spike

Latency versus Consistency

Stronger Consistency

Commit-wait



Selection Criteria and a Practical Decision Framework



Weigh candidates across five dimensions: workload, scalability, consistency, operations, cost



2026 default: start with managed PostgreSQL; make alternatives argue you off it



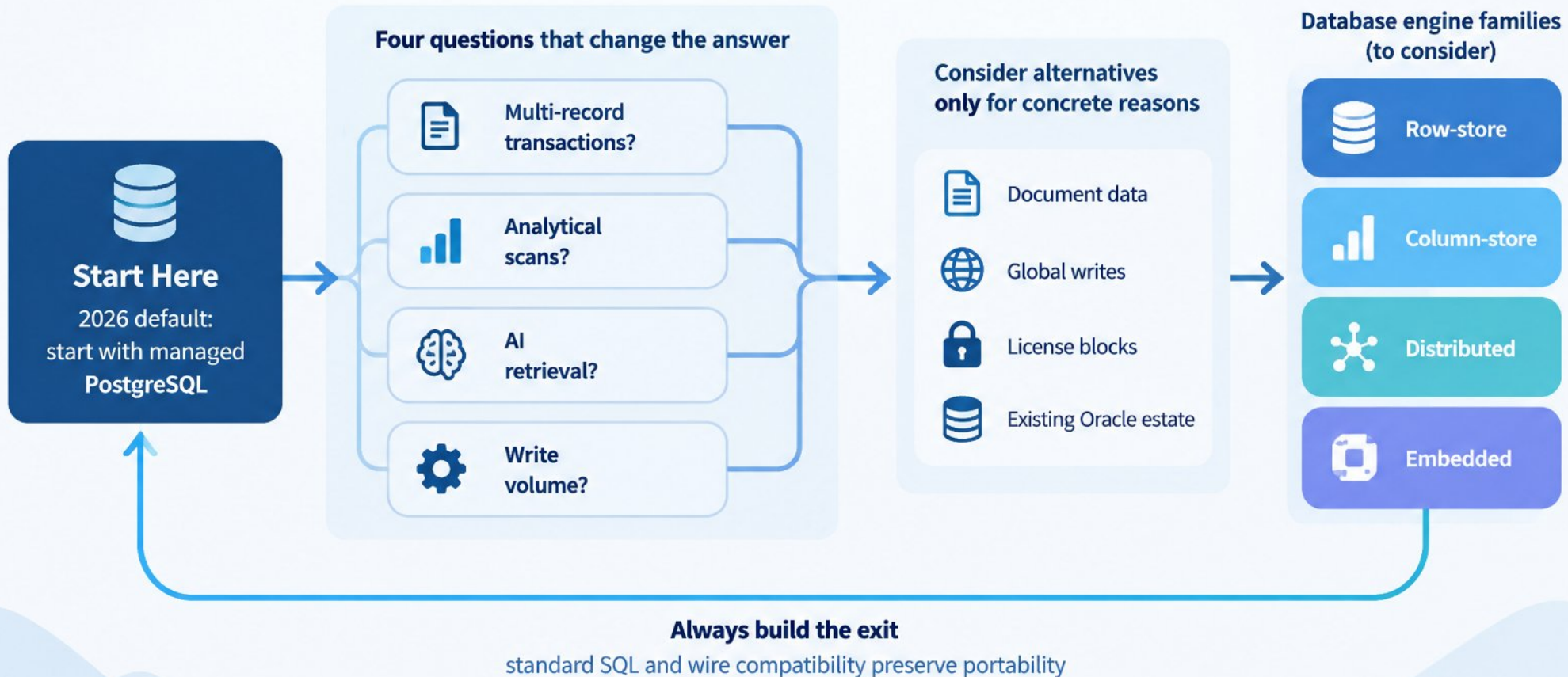
Four questions that change the answer: multi-record transactions, analytical scans, AI retrieval, write volume



Deviate only for concrete reasons: document data, global writes, license blocks, existing Oracle estate



Always build the exit: standard SQL and wire compatibility preserve portability



Scenario Walkthroughs: Startup, Analytics Team, and Enterprise

Startup



- Early-stage, under 100 GB: managed PostgreSQL on any major cloud; don't over-optimize

Analytics Team



- Mid-scale with analytics: Postgres for transactions, columnar engine via CDC, Redis caching; watch egress

Enterprise



- Enterprise governance: polyglot persistence portfolio with a system of record and paved roads

Global Active-Active



- Global active-active: distributed SQL (Spanner, CockroachDB, Aurora DSQL) or sharded MySQL—only if justified



- Regulated/air-gapped: Azure SQL Managed Instance, or self-managed PostgreSQL with platform expertise

Hands-On Practice: Running the Same Query Across Engines



Sandbox options: DuckDB via pip install, SQLite in Python, PostgreSQL in a container, cloud free tiers



Exercise: load one dataset into SQLite and DuckDB; compare GROUP BY and window-function timings



Port a PostgreSQL query (LIMIT, ||, NOW()) to SQL Server (TOP, +, GETDATE()) and Oracle (FETCH FIRST, ||, SYSDATE)



Run EXPLAIN ANALYZE before shipping: watch sequential scans, stale statistics, large nested loops

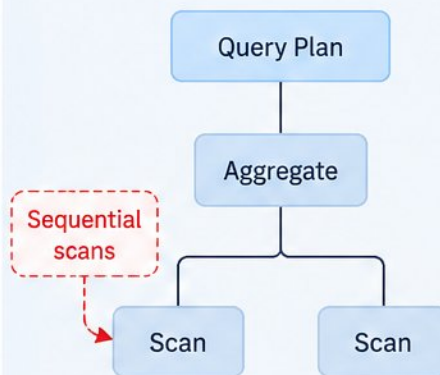


If a GROUP BY on your OLTP primary takes seconds, move that report to a columnar engine

```
SELECT region, COUNT(*)  
FROM sales  
GROUP BY region;
```

```
SELECT region, amount,  
SUM(amount) OVER (PARTITION BY region)  
FROM sales;
```

SQLite



DuckDB



PostgreSQL

```
LIMIT  
||  
NOW()
```



SQL Server

```
TOP  
+  
GETDATE()
```



Oracle

```
FETCH FIRST  
||  
SYSDATE()
```

Common Pitfalls and Best Practices for Each Audience

SAME DATA. DIFFERENT PERSPECTIVES. STRONGER OUTCOMES.



Developer

- Mistakes that fail silently
- Parameterized queries



Analyst

- EXPLAIN ANALYZE
- Columnar reporting targets



PM

- End-of-support dates in planning documents



Key SQL and Data Practices (All Audiences)

- `SELECT *` in app code; filter aggregates with `HAVING`, not `WHERE`
- Compare `NULL` with `IS NULL`; `COUNT(*)` $\neq$ `COUNT(column)`
- Money as `DECIMAL`, never `FLOAT`; use junction tables, not CSV columns
- Index foreign keys (Postgres won't); avoid functions on indexed columns
- Parameterized queries, least privilege, timeouts, `CREATE INDEX CONCURRENTLY`

PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/cda1ed2d/public