

DATABASE SYSTEMS

Model facts precisely, execute and cost queries, then test data correctness under concurrency, failure, distribution, and hostile access.

OFFICIAL DOCUMENTATION

November 2026

CONTENTS

DATA CONTRACTS, RELATIONS, AND SQL

Database abstraction: stable contracts over changing storage

Relational model: facts, keys, and constraints

Relational algebra: derive a query before SQL

SQL querying: specify rows, then test edge cases

SCHEMA DESIGN AND SQL PRACTICE

Functional dependencies: state what determines a fact

Normalization: decompose without losing facts

Schema review: turn requirements into constraints

SQL lab: build and verify an allocation report

STORAGE, ACCESS, AND EXECUTION

Storage Engines

Indexes

Query Execution and Optimization

TRANSACTIONS AND RECOVERABLE EXECUTION

Transactions and Isolation

Concurrency Control

Transaction Trace Lab

Recovery and Logging

DATA AT SCALE AND SYNTHESIS

Distributed Data

Replication and Partitioning

Database Security

Database Capstone

COURSE PRACTICE

Database Systems Practice Workshop

Data contracts, relations, and SQL

SECTION DETAIL

Database abstraction: stable contracts over changing storage

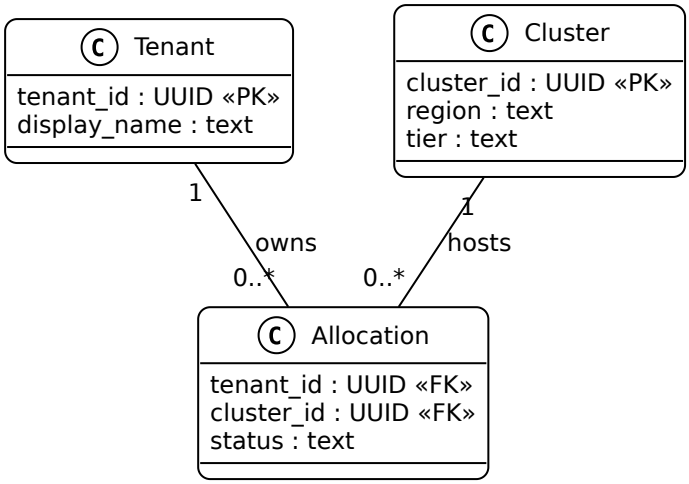
An application asks, “Which services are running across our clusters?” It should not need to know whether the database found those rows through a heap scan, an index, or a partition. It *does* need to know what one output row means, which rows it may see, and whether its requested order is guaranteed.

DEFINITION 1.1.1 **Database abstraction contract**

An abstraction contract states the meaning and permitted observations of data at one boundary. An external contract is what a client can request and observe; a logical contract is relations, attributes, and constraints; a physical contract is the layout and access methods that implement them.

One domain, three descriptions

Start with a cluster-allocation domain. The diagram is conceptual: it says a tenant can have many compute allocations and a cluster can host many allocations. It deliberately says nothing about files or indexes.



The same domain acquires a different, sharper meaning at each layer.

Layer	Example representation	Question it answers	Deliberately not promised
External	`CurrentAllocation(display_name, region, tier)` view	What may a reporting client read?	Base-table names or access path
Logical	`Tenant`, `Cluster`, `Allocation` with keys and foreign keys	What facts exist and how are they related?	Page layout or index choice
Physical	Heap, B-tree on `(tenant_id, cluster_id)`, partitions by region	How are logical rows stored and found?	A different result for the same logical query

> ***Worked example.*** *The relationship in the diagram becomes* `Allocation(tenant_id, cluster_id, status)`*. Its grain is* ***one tenant's allocation on one cluster****. Its primary key can be* `(tenant_id, cluster_id)` *if a tenant has at most one active allocation per cluster. Both columns are foreign keys. A client-facing view may join the three relations and omit identifiers, but it must preserve that stated grain.*

INVARIANT 1.1.2 **Physical data independence**

Physical data independence means that if only the physical design changes, every permitted logical query and constraint has the same meaning. Cost, lock timing, and an undocumented incidental row order may change; specified semantics may not.

A view is an interface, not decoration

Suppose a reporting client needs "one row per active allocation." This is an external contract; it needs more than a column list.

Contract field	Decision for `CurrentAllocation`
Grain	One active allocation for one tenant on one cluster
Authority	`Allocation.status = 'active'` is the source of truth for active
Duplicates	Impossible if `(tenant_id, cluster_id)` is unique; otherwise report them as a data error
`NULL`	`display_name` may be absent; `tenant_id` and `cluster_id` may not be absent
Ordering	None unless the client explicitly requests `ORDER BY region, display_name`
Freshness and access	Reads see the database's transaction visibility rules and the caller's permissions

WARNING

A join does not choose the grain for you. Joining `Allocation` to a table with one row per telemetry metric event can turn one allocation into many output rows. A correct-looking query can still violate the view's "one row per allocation" promise.

> ***Counterexample.*** *Replacing* `Cluster.region` *with a nullable free-text column may be a logical-schema change, not merely a physical one. A view that promised a non-null, validated region code now has a broken contract even if every query still parses.*

Declarative request, physical plan

SQL describes a requested result. The DBMS can choose a sequential scan, an index scan, a hash join, a sort, or another operator tree to produce it. Those choices are cost decisions, not promises made by ordinary SQL text.

DEFINITION 1.1.3 **Logical result vs. physical plan**

The logical result is the rows, values, duplicates, and any explicitly requested ordering prescribed by the query semantics and the database constraints. A plan is one implementation of that result.

> ***Worked example.*** *For a tenant dashboard, the query predicate is* `tenant_id = ? AND created_at >= ?`*. A composite index on* `(tenant_id, created_at, order_id)` *may avoid scanning unrelated tenant rows. The proof obligation is two-part: compare query results and constraint outcomes before/after the index, then measure latency, writes, and plan behavior on representative data.*

Do not overstate the boundary. Adding an index can raise write cost. Partitioning can complicate uniqueness or cross-partition operations. A query rewrite that is safe for an inner join can be wrong for an outer join. A physical change preserves *logical* behavior only within the documented dialect, transaction, and authorization semantics.

LOGICAL RESULT VERSUS PHYSICAL PLAN

A team adds a composite index on (tenant_id, created_at) to accelerate dashboard filtering. The rows and columns returned are identical. Evaluate each claim:

Answer: The physical access path changed while the logical query semantics and result set remain invariant.; Index maintenance increases write and storage costs on subsequent insert, update, and delete operations.

An index is a physical access method that preserves logical query semantics while altering read, write, and storage trade-offs.

Constraints make the logical layer enforceable

Keys, foreign keys, `CHECK` constraints, and `UNIQUE` constraints place facts at the database boundary so every client is checked by the same rule.

DEFINITION 1.1.4 **Referential integrity constraint**

Referential integrity means that if `Allocation.tenant_id` is non-null and refers to `Tenant(tenant_id)`, then a matching tenant row must exist when the foreign-key constraint is checked. A deferred foreign key postpones that check; it does not waive it at commit.

Worked example. *Two web requests both see one available compute slot and both attempt to allocate a worker. An application-side "count first, then insert" check can race. The schema can at least enforce uniqueness of a tenant's allocation; a capacity rule requires an atomic transaction design, which is a later topic. The abstraction lesson's point is the boundary: a constraint is checked centrally, while a client-only observation is not.*

Change evidence: preserve meaning, measure cost

Use two separate checklists for a physical redesign.

Correctness evidence	Operational evidence
Representative query results, including duplicates and `NULL` cases	Latency and throughput under realistic read/write mix
Keys, foreign keys, checks, and authorization still hold	Plan changes, memory use, and write amplification
Required ordering and pagination remain explicit	Migration duration, rollback, backup, and recovery behavior
Old and new views agree on a fixed test dataset	Lock waits and error rates during deployment

> **WARNING**
>
> A database rollback can undo rows it controls. It cannot automatically unsend an email or reverse an API call. Connect such effects through an explicit idempotent protocol, such as an outbox, when later transaction material introduces that pattern.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.1.1: Physical Indexing and Output Ordering

Verification recall

The platform database administrator adds an index on `Allocation(cluster_id, tenant_id)`. A report asks for the count of active allocations per cluster but contains no `ORDER BY`. Which observations must be unchanged, and which observation must not be inferred from a successful test run?

Separate the requested result from the way rows were found.

The count and the set of grouped clusters must be unchanged if the index is correct. The chosen plan, page reads, and emitted row sequence need not be unchanged. Because the query has no `ORDER BY`, a test must not treat the observed output order as part of its contract.

Exercise 1.1.2: Schema Migration and View Grain Preservation

Construction application

An existing view exposes `display_name`, `region`, and `tier`. The team wants to split `Cluster` into `Datacenter(region, campus)` and `Cluster(cluster_id, region, tier)`. State a compatible view definition and name one test case that catches a broken migration.

The client should still see the same grain and column meanings. Look for a cluster whose datacenter lookup is missing.

The compatible view still has one row per active allocation and obtains `campus` by joining `Cluster` to `Datacenter` through the validated region. Test an allocation whose cluster exists but whose datacenter row is accidentally absent after migration: an inner join would silently drop the client row. Compare old and new view results, including this case, rather than comparing table counts alone.

Exercise 1.1.3: Declarative Client Contract Specification

Formalisation application

A management dashboard wants a view called `MyAllocations`. It must show each authenticated tenant only their active cluster allocations, omit terminated allocations, and display a stable time order. Specify the four contract clauses required before physical optimization.

Use grain, authority, access, and ordering; avoid naming an index.

One sound contract is: (1) one row per active allocation for the authenticated tenant; (2) `Allocation.status = 'active'` determines inclusion; (3) authorization binds the authenticated tenant to `Allocation.tenant_id` and does not trust a client-supplied identifier; (4) rows are ordered by allocation timestamp and then `cluster_id`. An index might help later, but it is not part of this client contract.

Exercise 1.1.4: Table Partitioning Correctness and Operational Trade-Offs

Generalisation transfer

Your team proposes partitioning `Allocation` by `region` because reports usually filter by region. Produce a design note specifying the logical rows and constraints that must remain true, one query differential test, and two operational measurements that could reveal an unfavorable trade-off.

A partition key is physical. Start from the allocation relation's grain and foreign keys, then name costs that are not logical errors.

The note should preserve one allocation per permitted `(tenant_id, cluster_id)` pair and every checked foreign-key relationship. It can compare the old and new `CurrentAllocation` view over boundary regions, plus test that a tenant cannot read another tenant's allocation. Useful operational measurements include cross-region query cost, insert/delete cost, lock or migration duration, and backup/restore behavior. A slower write path can reject the proposal even when every logical result matches.

References & Further Reading

- E. F. Codd, “A Relational Model of Data for Large Shared Data Banks” (IBM Research, 1970)
- Selinger et al., “Access Path Selection in a Relational Database Management System” (IBM Research, 1979)
- PostgreSQL: Constraints

SECTION DETAIL

Relational model: facts, keys, and constraints

"Store cluster allocations" is not a schema. First decide what one row is allowed to assert. Then make the database reject row combinations that would contradict that assertion.

DEFINITION 1.2.1 **Relation schema, instance, and grain**

A relation schema names attributes and their domains, such as `Allocation(tenant_id, cluster_id, status)`. A relation instance is the current collection of tuples permitted by that schema and its constraints. The **grain** states what one tuple represents.

Build one small schema carefully

We will model a tenant allocating compute resources in a cluster during a deployment lifecycle.

Relation	Grain	Candidate key	Required checks
`Tenant(tenant_id, display_name)`	One identified tenant	`tenant_id`	`tenant_id` is non-null and unique
`Cluster(cluster_id, cluster_name, region, capacity)`	One managed compute cluster	`cluster_id`	`capacity >= 0`
`Allocation(tenant_id, cluster_id, status)`	One tenant's current resource allocation on one cluster	`(tenant_id, cluster_id)`	Both IDs refer to existing rows; `status` is an allowed value

The three relations and their identifying keys, with every reference from `Allocation` to its parents:

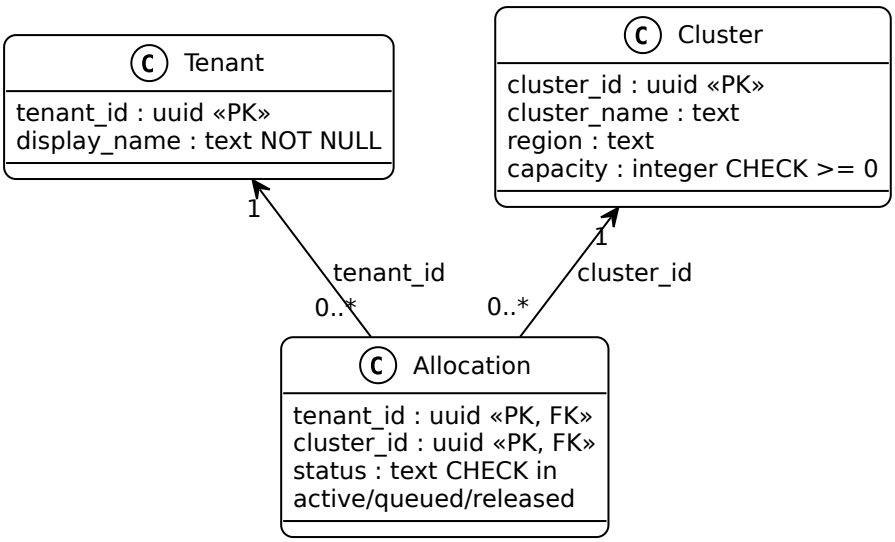


Figure 1. The allocation grain: one `Allocation` tuple asserts one tenant's current resource state on one `Cluster`. The composite key `(tenant_id, cluster_id)` makes two rows for the same pair illegal regardless of `status`, and the two foreign keys confine each allocation to existing parent rows.

> **EXAMPLE**
>
> The tuple `(t17, c3, 'active')` in `Allocation` says one fact: tenant `t17` holds an active allocation on cluster `c3`. If a later requirement says that a tenant may release and reallocate multiple times, this relation no longer has the right grain. Add an event identifier or timestamped allocation-history relation rather than quietly inserting duplicate current rows.

Worked example: declare the schema, then watch it reject rows

Given `Tenant = {(t17, 'Acme Corp')}` and `Cluster = {(c3, 'prod-east', 'us-east-1', 30)}`, declare the schema and attempt four inserts:

```
CREATE TABLE Allocation (
  tenant_id text NOT NULL REFERENCES Tenant(tenant_id),
  cluster_id text NOT NULL REFERENCES Cluster(cluster_id),
  status text NOT NULL
    CHECK (status IN ('active', 'queued', 'released')),
  PRIMARY KEY (tenant_id, cluster_id)
);

-- Input rows, one statement per attempt:
INSERT INTO Allocation VALUES ('t17', 'c3', 'active');    -- accepted
INSERT INTO Allocation VALUES ('t17', 'c3', 'released');  -- rejected:
duplicate key (t17, c3)
INSERT INTO Allocation VALUES ('t99', 'c3', 'active');    -- rejected:
no Tenant row t99
INSERT INTO Allocation VALUES ('t17', 'c9', 'active');    -- rejected:
no Cluster row c9
```

The second statement fails with a unique-violation on the composite primary key: changing `status` does not create a new current-allocation fact, because `status` is not part of the key. The third and fourth fail foreign-key checks because each `NOT NULL` reference must match an existing key value in its parent. Which statement fails *first* for the second row—the key check or a check constraint—is not fixed by the relational model; a constraint set only has to reject the row.

Prediction: what is illegal?

Assume `Allocation` has primary key `(tenant_id, cluster_id)`, and `tenant_id` and `cluster_id` are non-null foreign keys.

tenant_id	cluster_id	status
t17	c3	active
t17	c3	released
t99	c3	active

Which rows can coexist if only `t17` and `c3` exist?

Check the composite key first, then each foreign key.

Only the first row `(t17, c3, 'active')` can exist. The second row duplicates the existing candidate key `(t17, c3)`, while the third references absent tenant `t99` in violation of foreign key integrity.

Keys identify, foreign keys connect

DEFINITION 1.2.2 **Superkeys, candidate keys, and primary keys**

A superkey is a set of attributes that uniquely identifies a tuple in every legal instance. A candidate key is a minimal superkey: remove any one of its attributes and it no longer identifies tuples. A primary key is a chosen candidate key. (Ramakrishnan & Gehrke, *Database Management Systems*, 3rd ed., §3.2.1.)

For `Allocation`, `(tenant_id, cluster_id)` is a candidate key under the stated rule. `(tenant_id, cluster_id, status)` is a superkey but not a candidate key because `status` is unnecessary for identity. `(tenant_id, status)` is not even a superkey: a tenant may hold the same status across two different clusters, so it does not uniquely identify a tuple.

INVARIANT 1.2.3 **Constraint check point and deferrability**

A declared constraint must hold at its check point. By default SQL checks each constraint **immediately**, after every statement that could violate it. A constraint declared `DEFERRABLE INITIALLY DEFERRED` is checked at **commit** instead; the transaction may transiently violate it. Deferring never weakens the constraint—it only changes *when* the database checks it.

Worked example (deferred check point). *Constraint: "`Cluster.region` is referenced by allocations and each cluster keeps a `standby_cluster_id` foreign key to a backup cluster." Swapping two clusters' `standby_cluster_id` values with two `UPDATE` statements transiently breaks referential integrity if the constraint is immediate. Declaring it `DEFERRABLE INITIALLY DEFERRED` lets both updates commit only if,* ***at commit****, every referenced `standby_cluster_id` exists. Committing with the violation still fails—deferral moves the check, never removes it.*

Counterexample (UNIQUE is not "no duplicates" in general). *The SQL standard leaves `UNIQUE` treatment of multiple `NULL`s to the implementation; PostgreSQL's default (`NULLS DISTINCT`) treats `NULL`s as mutually distinct, so `UNIQUE(email)` admits unlimited rows with `email IS NULL`. Only PostgreSQL 15's `UNIQUE NULLS NOT DISTINCT` collapses them. Claiming "`UNIQUE` forbids duplicates" without saying how the vendor treats `NULL` is a falsifiable error a reviewer can check against the vendor's own constraint documentation.*

Counterexample. *`display_name` might look unique in a three-row sample, but two tenants can share it. An observed sample pattern is not an architectural invariant. Declare a `UNIQUE` constraint only when the domain truly requires uniqueness, and say which `NULL` treatment you are relying on.*

IDENTITY VERSUS CONNECTION

An allocation row stores tenant_id and cluster_id forming a composite primary key (tenant_id, cluster_id). Evaluate each claim:

Answer: The composite pair uniquely identifies one allocation at the stated grain, while each component serves as a foreign key.; A column may simultaneously serve as a foreign key reference and participate in a composite primary key.

Key identity and referential connection are distinct constraints; composite keys routinely contain foreign-key attributes, and referential actions govern cascade or restrict behavior.

Relation semantics versus everyday SQL tables

The mathematical relational model treats a relation as a set of tuples. SQL is deliberately more complicated in practice.

Question	Pure set relation	Typical SQL behavior to account for
Can equal tuples appear twice?	No	Tables and `SELECT` results can contain duplicate row values unless a constraint or `DISTINCT` removes them
Is there a built-in row order?	No	No order is guaranteed without `ORDER BY`
Can an attribute be missing?	Classical model has a value in its domain	SQL has `NULL`; predicates can evaluate to `UNKNOWN`

WARNING 1.2.4 Three-valued predicate logic and NULL semantics

`NULL` is not an ordinary value. In a `WHERE` condition, only `TRUE` retains a row. `FALSE` and `UNKNOWN` both remove it. Therefore `WHERE gateway_ip <> '10.0.0.1'` does not retain a row whose `gateway_ip` is `NULL`: the comparison is `UNKNOWN`, not `TRUE`.

Worked example. *Suppose two allocation rows project to the same* `region` *after joining with clusters.* `SELECT region ...` *can return the region twice;* `SELECT DISTINCT region ...` *returns it once. Neither query is universally "more relational"—the right one depends on whether the client wants distinct regions or allocation occurrences.*

From requirements to a legal instance

Schema review starts by separating facts from presentation.

EXAMPLE

`TenantClusterAllocation(tenant_name, cluster_name, region, capacity, status)` mixes a cluster fact (`capacity`) with an allocation fact (`status`). If 30 tenants allocate resources on the cluster, the same capacity appears 30 times and can disagree. Put capacity in `Cluster`; put status in `Allocation`; create a view when a screen needs both.

INVARIANT 1.2.5 **Constraint scope and enforcement boundaries**

Every constraint has a scope. A key constrains rows of one relation. A foreign key constrains a reference across relations. A view controls a presented shape. None automatically proves an aggregate policy such as "at most capacity allocated."

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.2.1: Candidate Key and Referential Integrity Validation

Verification recall

Consider `Tenant(tenant_id, display_name)`, `Cluster(cluster_id, cluster_name, region, capacity)`, and `Allocation(tenant_id, cluster_id, status)` with primary key `(tenant_id, cluster_id)`. Suppose only tenant `t17` and cluster `c3` exist in the database. Examine these candidate rows: `(t17, c3, 'active')`, `(t17, c3, 'released')`, and `(t99, c3, 'active')`. Which rows can legally coexist?

Check the composite key first, then each foreign key.

The first row can exist. The second conflicts with the same composite primary key, even though its `status` differs. The third violates the `tenant_id` foreign key because `t99` has no matching `Tenant` row. A history requirement needs a different key and grain; changing `status` alone does not create a new current-allocation fact.

Exercise 1.2.2: Narrowest Constraint Domain Selection

Construction application

For each requirement, specify the narrowest SQL constraint that directly enforces it: (1) A cluster cannot have negative capacity; (2) A tenant cannot appear twice in the same current allocation; (3) Every allocation names an existing cluster; (4) A status must be `active`, `queued`, or `released`.

Use a check for a predicate, a key for identity, and a foreign key for a cross-relation reference.

Use `CHECK (capacity >= 0)`, a primary or unique key on `(tenant_id, cluster_id)`, a foreign key from `Allocation.cluster_id` to `Cluster.cluster_id`, and a check constraint or controlled ENUM status domain. These constraints do not by themselves enforce multi-row capacity limits; that requires transaction-aware isolation.

Exercise 1.2.3: Set vs. Bag Projection Semantics

Description application

If an `Allocation` relation contains two different tenants allocated on cluster `prod-east`, what does projecting onto `cluster_name` produce under pure relational set semantics versus SQL bag semantics? Explain how to align the result grain with application intent.

Projection removes attributes. Decide whether equal remaining tuples collapse.

With set semantics, the projection has one tuple `prod-east`. With bag semantics, it has two occurrences of `prod-east`. The source relation's grain was one allocation; a result whose grain is one cluster should normally use `DISTINCT` deliberately rather than inherit duplicate behavior accidentally.

Exercise 1.2.4: Multi-Pool Cluster Allocation Schema Specification

Generalisation transfer

A cloud infrastructure platform offers the same cluster tier across multiple resource pools in one region (`prod-tier` in `us-east-1`, pools `pool-a` and `pool-b`). Design a revised `Cluster` and `Allocation` schema. State the grain and candidate key of each relation, provide one row combination your constraints reject, and describe how allocation queuing priority is formally modeled.

Cluster name is no longer enough to identify a resource pool. Keep the allocation fact separate from the cluster pool fact.

One sound solution is `Cluster(cluster_id, tier, region, pool, capacity)` with candidate key `cluster_id` and unique constraint `UNIQUE(tier, region, pool)`, plus `Allocation(tenant_id, cluster_id, status)` with composite key `(tenant_id, cluster_id)`. The schema rejects duplicate allocations for the same tenant in one pool and rejects non-existent clusters. Queue ordering requires an explicit ranking attribute such as `queue_rank` with a partial uniqueness constraint.

References & Further Reading

- E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13(6), pp. 377–387 (1970) — relational vocabulary, relations vs. implementation. ACM DOI · IBM Research page
- Ramakrishnan & Gehrke, *Database Management Systems*, 3rd ed. (McGraw-Hill, 2003) — §3.2.1 superkeys and candidate keys; §3.1 relation schema and instance; §3.4 integrity constraints over instances.
- PostgreSQL, "Constraints," *Current Documentation* ch. 5 — unique constraints (including the `NULLS DISTINCT` default), check constraints, foreign keys, and deferrable check points.
- PostgreSQL, "Queries," *Current Documentation* ch. 7 — `DISTINCT` duplicate elimination, sorting rows (`ORDER BY`), and `LIMIT` without `ORDER BY`.
- PostgreSQL, "Comparison Functions and Operators" — `NULL` comparisons and `IS NULL`.

SECTION DETAIL

Relational algebra: derive a query before SQL

Relational algebra gives a compact way to answer “what rows?” before a SQL dialect, a query plan, or a UI obscures the question. Its greatest practical use is not memorizing symbols. It is exposing a missing condition, an incompatible difference, or a changed result grain early.

DEFINITION

An algebra expression takes one or more relations as input and produces another relation. Name the output schema and grain after every nontrivial operator; this is how you notice an accidental duplication or lost attribute.

The operators we will use

DEFINITION 1.3.1 **Set-relational operators**

For a relation R and predicate θ:

$$\sigma\theta(R) = \{t \in R \mid \theta(t)\}$$

- **Selection** $\sigma\theta(R)$ keeps tuples satisfying θ.
- **Projection** $\pi_{A,B}(R)$ keeps named attributes and removes duplicate tuples under set semantics.
- **Rename** $\rho_{new_name \leftarrow old_name}(R)$ changes a relation or attribute name so an expression is well-formed.
- **Product** $R \times S$ pairs every tuple of R with every tuple of S.
- **Join** $R \bowtie_{R.a=S.a} S$ is a selected product with compatible joined attributes.
- **Difference** $R - S$ keeps tuples in R that are not in S; R and S must have compatible schemas.

Trace a query, not just its notation

Use these fixed relations for region `us-east`.

`Node`:

`node_id`	`tier`
worker-01	edge
worker-02	core
worker-03	edge

`RequiredService`:

service_id
S101
S201

`Deployment`:

node_id	service_id	region
worker-01	S101	us-east
worker-01	S201	us-east
worker-02	S101	us-east
worker-02	S201	eu-west
worker-03	S101	us-east

> ***Worked example.*** *"Which nodes are hosting service S101 in us-east?" starts with selection, then projection:*

$$\pi_{node_id}\left(\sigma_{service_id=\text{'S101'} \wedge region=\text{'us-east'}}(Deployment)\right) = \{\text{worker-01}, \text{worker-0}$$

The selected relation still has grain "one deployment." The projected result has grain "one node that meets the condition." That change is intentional.

Predict the intermediate result

What is the schema and tuple set of

$$\pi_{node_id, service_id}\left(\sigma_{region=\text{'us-east'}}(Deployment)\right)?$$

Selection leaves all attributes in place; projection then drops only `region`.

Its schema is `(node_id, service_id)` and its tuples are `(worker-01, S101)`, `(worker-01, S201)`, `(worker-02, S101)`, and `(worker-03, S101)`. The result says that a node hosts a service during the chosen regional scope; it no longer states the region, so do not reuse it for a multi-region question without rebuilding the scope.

Make the same operations visible

The playground contains a separate fixed deployment fixture. Use it to perform one selection, compare bag versus set projection, and inspect an equijoin. Before each step, say whether you expect its output grain to be one deployment, one service, or a joined pair.

EXECUTE AN ALGEBRA OPERATOR OVER VISIBLE RELATIONS

The static comparison keeps the input relations beside the result. The join has one output tuple for each deployment because `Service.service_id` is unique in this fixture.

DEPLOYMENT

node_id	service_id	region
worker-01	S101	us-east
worker-02	S101	us-east
worker-03	S202	eu-west
worker-01	S202	eu-west

SERVICE

service_id	name
S101	Auth API
S202	Billing Worker

DEPLOYMENT ⋈ DEPLOYMENT.SERVICE_ID = SERVICE.SERVICE_ID SERVICE

node_id	service_id	region	name
worker-01	S101	us-east	Auth API
worker-02	S101	us-east	Auth API
worker-03	S202	eu-west	Billing Worker
worker-01	S202	eu-west	Billing Worker

Static state: the equijoin carries each Deployment.service_id to the matching Service row and appends the service name; compare the input grain with the joined-pair grain.

WARNING

This visual model intentionally excludes `NULL`, outer joins, nonunique join keys, ordering, and optimizer behavior. It teaches a clean set-oriented core; SQL introduces those additional semantics later.

"All required services" is a missing-pair problem

The word **all** often tempts people into a vague division operator. Derive it by enumerating what should exist and subtracting what is absent.

EXAMPLE

Find nodes that host every required service in `us-east`. First make every expected node/service pair. Then remove pairs that have no matching deployment.

$$
\begin{aligned}
Deployed &= \pi_{node_id, service_id}(\sigma_{region='\text{us-east}'}(Deployment)) \\
Expected &= \pi_{node_id}(Node) \times RequiredService \\
Missing &= Expected - Deployed \\
Answer &= \pi_{node_id}(Node) - \pi_{node_id}(Missing)
\end{aligned}
$$

`Expected` and `Deployed` both have schema `(node_id, service_id)`, so the difference is defined. For the data above, `Missing` contains `(worker-02, S201)`, `(worker-03, S201)`; `Answer` is `{worker-01}`.

INVARIANT

Before taking a union, intersection, or difference, verify compatible schemas and attribute meanings. Two columns both named `id` are not automatically compatible if one is a node identifier and the other is a service identifier.

Apply the derivation

A query specification requires finding nodes that host every required service **and** belong to the `edge` tier. Where should the tier condition go: in `Deployed`, `Expected`, or the final answer?

The tier condition restricts which nodes are required to satisfy all pairs, not what counts as a matching deployment.

Restrict the candidate nodes before forming expected pairs:

$$Eligible = \sigma_{tier=\text{'edge'}}(Node), \qquad Expected = \pi_{node_id}(Eligible) \times RequiredSe$$

Then use the same `Missing` and final subtraction pattern with `Eligible`. Filtering only the final answer can be correct for a simple output filter, but it unnecessarily constructs missing pairs for nodes that were never candidates and obscures the requirement.

A rewrite needs stated assumptions

Algebra helps explain why an optimizer can sometimes move work earlier.

PROPOSITION 1.3.2 **Safe inner-join selection pushdown**

For set relations, an inner join, and a two-valued predicate θ_R that mentions only attributes of R:

$$\sigma\theta_R(R \bowtie_{R.a=S.a} S) = \sigma\theta_R(R) \bowtie_{R.a=S.a} S.$$

A joined tuple survives the left expression exactly when its `R` component satisfies θ_R. Because the predicate reads no `S` attribute, filtering `R` first removes exactly those same joined tuples. The statement does not claim the two expressions have equal cost, nor does it cover outer joins or SQL's three-valued predicate behavior.

∎

> ***Counterexample.*** *With a left outer join, moving a predicate about the right relation from* `WHERE` *into a pre-join filter can preserve unmatched left rows that the original* `WHERE` *condition removes. For example, if* `R` *has one row and* `S` *has none,* `WHERE S.flag = 'Y'` *removes the null-extended row after the join. The same condition inside the right input still lets the left outer join emit it. The expressions are not interchangeable.*

WARNING

SQL normally uses bag semantics and three-valued logic. An algebra law stated for sets and two-valued predicates is a reasoning aid, not permission to apply it blindly to a SQL query.

Check a proposed rewrite

An engineer changes

$$\sigma_{R.region = \text{'us-east'}}(R \bowtie_{R.node_id = S.node_id} S)$$

to

$$\sigma_{R.region = \text{'us-east'}}(R) \bowtie_{R.node_id = S.node_id} S.$$

List the three facts you would check before accepting the rewrite.

Match the proposition's assumptions one by one.

Check that the join is inner, the predicate reads only `R` attributes, and the intended semantics support the stated selection law (including how duplicate rows and `NULL` values are handled in the actual SQL dialect). Also confirm that `region` means the same thing at this point in the expression; an alias or prior rename can change that fact.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.3.1: Intermediate Schema and Tuple Projection

Description recall

Given relation schema $Deployment(node_id, service_id, region)$ and selection $\sigma_{region=\text{'us-east'}}(Deployment)$, determine the schema and result grain of $\pi_{node_id, service_id}\left(\sigma_{region=\text{'us-east'}}(Deployment)\right)$. Explain why dropping $region$ affects subsequent multi-region queries.

Selection preserves all attributes; projection then retains only node_id and service_id.

The output schema is $(node_id, service_id)$ with tuples representing nodes actively hosting a service in us-east. Because $region$ is omitted from the projected schema, the output relation loses geographic context and cannot be directly intersected with other regions without reapplying a region qualifier.

Exercise 1.3.2: Inner-Join Pushdown Precondition Check

Verification application

An engineer proposes rewriting $\sigma_{R.region=\text{'us-east'}}(R \bowtie_{R.node_id=S.node_id} S)$ into $\sigma_{R.region=\text{'us-east'}}(R) \bowtie_{R.node_id=S.node_id} S$. Referring to Proposition 1.3.2, verify whether this rewrite is valid under two-valued set semantics, and identify one condition under which it fails for outer joins.

Check predicate attribute dependency and outer-join null extension semantics.

Under two-valued set semantics, the rewrite is valid because the selection predicate references only attributes of R and the join is an inner equijoin. For a left outer join where the predicate references S instead (e.g. $S.active = \text{'Y'}$), pushing the predicate into S retains non-matching R tuples with NULL padding, whereas the post-join filter eliminates them.

Exercise 1.3.3: Universal Division via Difference Formulation

Construction application

Let $Node(node_id, tier)$, $RequiredService(service_id)$, and $Deployment(node_id, service_id, region)$ be set relations. Using relational algebra operators from Definition 1.3.1, construct an expression that returns all nodes in the `edge` tier that host every service in $RequiredService$ in region `us-east`.

Restrict candidate nodes first before constructing the Cartesian product of expected pairs.

Define candidate edge nodes and expected pairs:

$$
\begin{aligned}
Eligible &= \sigma_{tier=\text{'edge'}}(Node) \\
Expected &= \pi_{node_id}(Eligible) \times RequiredService \\
Deployed &= \pi_{node_id, service_id}\left(\sigma_{region=\text{'us-east'}}(Deployment)\right) \\
Missing &= Expected - Deployed \\
Answer &= \pi_{node_id}(Eligible) - \pi_{node_id}(Missing)
\end{aligned}
$$

Filtering candidates before forming the product avoids generating missing pairs for unrelated tiers.

Exercise 1.3.4: Constrained Core-Service Deployment Derivation

Generalisation transfer

Let $Node(node_id)$, $Service(service_id)$, $Deployed(node_id, service_id, region)$, and $Core(service_id)$ be set relations. Derive an algebra expression for nodes that host every core service in `us-east` but host no non-core service in that region. State the compatibility of schemas in every difference operation.

Derive universal deployment of Core, then subtract the set of nodes with non-core deployments.

First isolate us-east deployments: $Active = \pi_{node_id, service_id}(\sigma_{region='\text{us-east}'}(Deployed))$. Compute nodes hosting all core services:

$$ExpectedCore = \pi_{node_id}(Node) \times Core$$
$$AllCore = \pi_{node_id}(Node) - \pi_{node_id}(ExpectedCore - Active)$$

Next isolate non-core services: $NonCore = \pi_{service_id}(Service) - Core$ (both schema $(service_id)$). Find nodes with any non-core service: $HasNonCore = \pi_{node_id}(Active \bowtie NonCore)$. The final answer is $AllCore - HasNonCore$. Both operands have schema $(node_id)$.

References & Further Reading

- E. F. Codd, “A Relational Model of Data for Large Shared Data Banks” (IBM Research, 1970)
- Ramakrishnan and Gehrke, *Database Management Systems*: open-access contents
- PostgreSQL: `SELECT`

SECTION DETAIL

SQL querying: specify rows, then test edge cases

SQL is concise enough to hide a wrong question. Before typing, say the output grain: "one row per service pool in us-east-1, including pools with zero active allocations." That single sentence determines whether a filter belongs in `ON`, `WHERE`, or `HAVING`.

DEFINITION

This lesson uses PostgreSQL `SELECT` semantics. A query's clauses define a logical result; the optimizer may use a different physical execution order to compute it.

Follow the logical pipeline

For a common grouped query, reason in this order.

Stage	Question answered	Typical clause
Build candidate rows	Which source rows and matches are available?	`FROM ... JOIN ... ON ...`
Remove individual rows	Which candidate rows satisfy a condition?	`WHERE`
Form groups	Which remaining rows belong together?	`GROUP BY`
Remove groups	Which computed groups qualify?	`HAVING`
Shape output	Which expressions become columns?	`SELECT`
Present a sequence	What order is promised to the client?	`ORDER BY`

> **WARNING**
>
> This is a semantic trace, not an execution-plan trace. A database may push a filter into a scan or rearrange inner joins when it can preserve the same result. Never infer a result order without `ORDER BY`.

Build a zero-safe allocation report

Use this service allocation fixture.

`ServicePool`:

`pool_id`	`service_code`	`region`
10	S101	us-east-1
11	S201	us-east-1
12	S301	us-east-1
13	S101	us-west-2

`Allocation`:

`client_id`	`pool_id`	`status`
Ada	10	active
Bo	11	suspended

The report must include service pools 11 and 12 with count zero. Its grain is **one service pool in us-east-1**, not one joined allocation row.

Execute and predict

The runner seeds exactly this fixture in SQLite. State the three result rows before running the starting query, then move `AND a.status = 'active'` into `WHERE` and run it again. The difference is the lesson.

ZERO-SAFE ALLOCATION REPORT

Predict which service pools survive, including zero counts, before executing the query.

```
SELECT p.service_code,
     COUNT(a.client_id) AS active_count
FROM ServicePool AS p
LEFT JOIN Allocation AS a
ON a.pool_id = p.pool_id
AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
ORDER BY p.service_code;
```

The screen version executes this query against the fixture in SQLite. Predict the output before checking it in an engine.

The analogous PostgreSQL statement is:

```sql
SELECT p.service_code,
       COUNT(a.client_id) AS active_count
FROM ServicePool AS p
LEFT JOIN Allocation AS a
  ON a.pool_id = p.pool_id
 AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
ORDER BY p.service_code;
```

> ***Worked example.*** `ON` *decides which allocation rows match each service pool. The status condition keeps only active matches, while the left join still emits null-extended rows for pools with no such match.* `WHERE` *then removes the other-region pool 13.* `COUNT(a.client_id)` *counts only non-null matching IDs, yielding* `S101 → 1`, `S201 → 0`, *and* `S301 → 0`.

Predict the bug

Move `AND a.status = 'active'` from `ON` to `WHERE`. Which service pool rows disappear, and why?

A left join creates NULLs for rows with no qualifying right-side match; `WHERE` runs afterward.

Service pools 11 and 12 disappear. Pool 11's only matching allocation is `suspended`; pool 12 is null-extended because it has no allocation. Neither makes `a.status = 'active'` true in `WHERE`. The condition turns this part of the left join into inner-join-like behavior for that predicate.

OUTER JOIN PREDICATE PLACEMENT AND NULL SURVIVAL

A report must retain every target-region service pool from ServicePool while counting only matching Allocation rows where memory_gb >= 70. Evaluate each assertion regarding predicate placement in a LEFT JOIN query:

Answer: Placing 'a.memory_gb >= 70' in the ON clause preserves service pools with no matching allocations while suppressing allocations below 70 GB.; Placing 'p.region = us-east-1' in the WHERE clause discards service pools from other regions after joining.

Conditions in ON determine which right rows qualify as matches, leaving unmatched left rows null-extended. Conditions in WHERE filter the final combined rows. Aggregating with COUNT(column) ignores nulls, while COUNT(*) counts the row itself.

RECONSTRUCT THE ZERO-SAFE REPORT

```
SELECT p.service_code,
     COUNT(a.client_id) AS active_count
FROM ServicePool AS p
LEFT JOIN Allocation AS a
ON a.pool_id = p.pool_id
AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
ORDER BY p.service_code
```

`COUNT(a.client_id)` ignores the NULL-extended right side and therefore yields zero for an empty service pool; `COUNT(*)` would count the left row. The status predicate belongs in `ON` so unmatched pools survive. `GROUP BY` fixes the output grain at one row per service pool rather than one row per allocation.

See predicate placement change the result

The visual model fixes `worker-01` at 87 GB, `worker-02` at 40 GB, and `worker-03` with no allocations. Set the threshold to 50 GB. Predict whether `worker-03` survives before trying both placements.

PLACE A LEFT-JOIN PREDICATE; INSPECT WHICH HOSTS SURVIVE

With a threshold of 50 GB, the two placements answer different questions. Compare the result tables before deciding where the predicate belongs.

Predicate in `ON`: preserve every host

host	service	memory_gb
worker-01	auth-svc	87
worker-02	NULL	NULL
worker-03	NULL	NULL

Predicate in `WHERE`: filter after NULL extension

host	service	memory_gb
worker-01	auth-svc	87

Static state: memory_gb ≥ 50 in ON preserves worker-01, worker-02, and worker-03; the same predicate in WHERE leaves only worker-01 because the NULL-extended and nonqualifying rows are filtered afterward.

INVARIANT

In a **`LEFT JOIN`**, a predicate in **`ON`** limits right-side matches while retaining every left-side row. A predicate in **`WHERE`** filters the post-join rows, including null-extended rows. Put it where the report's inclusion rule says it belongs.

Counterexample. *"Put every filter in `WHERE`" fails for the zero-safe report. "Put every filter in `ON`" also fails when the condition should remove left rows, such as `p.region = 'us-east-1'`. Clause placement follows row survival, not a style rule.*

Apply the row-survival test

The infrastructure team wants **all** us-east-1 service pools, but only allocations of 70 GB or more should be shown when an allocation exists. Where do `p.region = 'us-east-1'` and `a.memory_gb >= 70` belong?

Ask whether each condition describes a left row that may survive and a right row that may match.

`p.region = 'us-east-1'` belongs in `WHERE` because an other-region service pool should disappear. `a.memory_gb >= 70` belongs in `ON` if a us-east-1 pool must still appear when it has no qualifying allocation. This preserves the left relation's required rows while changing which right rows qualify as matches.

Groups answer a different question from rows

`WHERE` filters rows before aggregation; `HAVING` filters a group after an aggregate has been computed.

```
SELECT p.service_code,
       COUNT(a.client_id) AS active_count
FROM ServicePool AS p
LEFT JOIN Allocation AS a
  ON a.pool_id = p.pool_id
 AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
HAVING COUNT(a.client_id) >= 1;
```

> ***Worked example.*** *This version has the same group grain as the zero-safe report but removes groups with zero active allocations at* `HAVING`*. Moving* `COUNT(a.client_id) >= 1` *to* `WHERE` *is invalid because* `WHERE` *is evaluated before the count exists.*

Trace a result

For the fixture above, what rows survive the `HAVING` version? Then state what one output row means.

Reuse the three group counts from the earlier report.

Only `S101, 1` survives. Each output row means "one service pool in us-east-1 whose active allocation count is at least one." It does not mean one client, because the grouping collapsed allocation rows.

Anti-joins: prefer a NULL-safe question

To ask for clients with no active allocation in region us-east-1, write the absence condition directly.

```sql
SELECT c.client_id
FROM Client AS c
WHERE NOT EXISTS (
  SELECT 1
  FROM Allocation AS a
  JOIN ServicePool AS p ON p.pool_id = a.pool_id
  WHERE a.client_id = c.client_id
    AND a.status = 'active'
    AND p.region = 'us-east-1'
);
```

> ***Worked example.*** *The correlated condition asks whether an active allocation row exists for this client. If none exists,* `NOT EXISTS` *is true. The query does not depend on a subquery list having no* `NULL` *values.*

WARNING

`NOT IN (subquery)` can behave unexpectedly if the subquery returns `NULL`: comparisons can become `UNKNOWN`, so no outer row is retained. Use `NOT EXISTS` for an anti-join unless you have proved the list is non-null and understand the intended semantics.

Prediction with a NULL

Suppose a data-quality bug produces an allocation row with `client_id = NULL`. Why can `WHERE c.client_id NOT IN (SELECT client_id FROM Allocation)` return no clients, while the `NOT EXISTS` query above can still answer the intended question?

Evaluate `c.client_id <> NULL` using SQL three-valued logic.

The `NOT IN` condition is equivalent to a chain of inequalities. One comparison to `NULL` becomes `UNKNOWN`, so the whole predicate may fail to become true for every client. `NOT EXISTS` tests only whether a correlated row with equal non-null ID and active status exists; an unrelated null ID does not make that condition true.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.4.1: ON versus WHERE Predicate Placement Semantics

Description recall

Explain how moving an outer join filter predicate from the `ON` clause to the `WHERE` clause changes query semantics for null-extended rows. Give a concrete example where a service pool with zero allocations is erroneously discarded.

A LEFT JOIN produces NULL values for right attributes when no match occurs. WHERE filters after that generation.

In a `LEFT JOIN`, conditions in the `ON` clause decide whether right-side rows match, preserving all left rows. If the filter condition is moved to `WHERE`, it is evaluated on the joined result where right attributes are `NULL`. Comparisons against `NULL` evaluate to `UNKNOWN` in SQL's three-valued logic, which `WHERE` treats as false, eliminating unmatched left-side rows and inadvertently converting the outer join into an inner join.

Exercise 1.4.2: Anti-Join NULL Semantics: NOT EXISTS vs. NOT IN

Verification application

Suppose an `Allocation` table contains a single row with `client_id = NULL`. Verify why `WHERE c.client_id NOT IN (SELECT client_id FROM Allocation)` returns zero clients, while a correlated `NOT EXISTS` query correctly identifies clients without allocations.

Evaluate the logical expansion of NOT IN with a NULL in the candidate set.

`c.client_id NOT IN (a1, a2, ..., NULL)` expands to `(c.id <> a1) AND (c.id <> a2) AND ... AND (c.id <> NULL)`. In SQL, `c.id <> NULL` yields `UNKNOWN`. Because `TRUE AND UNKNOWN` is `UNKNOWN`, the entire predicate evaluates to `UNKNOWN` for every client, discarding all rows. In contrast, `NOT EXISTS` checks whether the subquery returns any row; a row with `NULL` matching condition `a.client_id = c.client_id` evaluates to `UNKNOWN` and is not returned, leaving `NOT EXISTS` true.

Exercise 1.4.3: Grouped Aggregation with Post-Filter Specification

Construction application

Write a PostgreSQL query against `ServicePool(pool_id, service_code, region)` and `Allocation(client_id, pool_id, status)` that returns the service code and active allocation count for all us-east-1 service pools having strictly more than 10 active allocations.

Use WHERE to filter service pools by region, ON for allocation status, and HAVING for aggregate thresholds.

```sql
SELECT p.service_code,
       COUNT(a.client_id) AS active_count
FROM ServicePool AS p
JOIN Allocation AS a
  ON a.pool_id = p.pool_id
 AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
HAVING COUNT(a.client_id) > 10
ORDER BY p.service_code;
```

Because the threshold requires `> 10` allocations, an inner join is sufficient. `GROUP BY` includes `pool_id` to guarantee uniqueness per service pool.

Exercise 1.4.4: Zero-Safe Service Pool Report with Computed Boolean Status

Generalisation transfer

Construct a query returning `(service_code, active_count, is_full)` for all service pools in region `us-east-1` with `capacity >= 1`. Pools with zero allocations must appear with count 0 and `is_full = false`. Explain how the grain is preserved.

Preserve unmatched service pools with LEFT JOIN and count non-null client IDs.

```sql
SELECT p.service_code,
       COUNT(a.client_id) AS active_count,
       COUNT(a.client_id) >= p.capacity AS is_full
FROM ServicePool AS p
LEFT JOIN Allocation AS a
  ON a.pool_id = p.pool_id
 AND a.status = 'active'
WHERE p.region = 'us-east-1'
  AND p.capacity >= 1
GROUP BY p.pool_id, p.service_code, p.capacity
ORDER BY p.service_code;
```

The grain is one positive-capacity service pool in us-east-1. Placing `a.status = 'active'` in `ON` allows pools without active allocations to survive as null-extended rows, which `COUNT(a.client_id)` aggregates to 0.

References & Further Reading

- PostgreSQL: `SELECT`
- PostgreSQL: Table Expressions and Joins
- PostgreSQL: Queries

Schema design and SQL practice

SECTION DETAIL

Functional dependencies: state what determines a fact

A functional dependency is a design claim: once you know one fact, another fact is forced. It is not an observation that happened to hold in three rows of a spreadsheet. That distinction lets us ask whether a relation stores one fact once or repeats it under many keys.

DEFINITION 2.1.1 **Functional dependency**

For a relation schema R and attribute sets $X, Y \subseteq R$, the functional dependency

$$X \to Y$$

holds when every **legal** instance of R has this property: any two tuples that agree on X also agree on Y. X is the determinant; Y is determined by it.

Begin with a domain rule, not a table pattern

DEFINITION

A domain rule describes which future instances are legal. It is the evidence for a functional dependency; a repeated pattern in a current export is not.

Assume a cloud platform assigns exactly one node IP and one network port to each service instance identified by `(cluster_id, service_name, instance_id)`.

$$(cluster_id, service_name, instance_id) \to (node_ip, port)$$

> ***Worked example.*** *In* `Instance(cluster_id, service_name, instance_id, node_ip, port)`*, the stated scheduling rule makes* `(cluster_id, service_name, instance_id)` *determine* `node_ip` *and* `port`*. It does* ***not*** *say that* `port` *determines* `node_ip`*: different instances can use the same port number across different nodes.*

> ***Counterexample.*** *A sample in which every row has a different* `service_fqdn` *does not prove* $service_fqdn \to service_id$*. The domain name may be unmapped, aliased, or dynamically routed unless the infrastructure configuration declares and enforces it as a unique identifier. Dependencies come from the legal domain, ideally backed by a key or explicit policy.*

Prediction: spot the unsupported dependency

The first three rows of a telemetry table all show `datacenter = 'us-east'` and `operator = 'ops-team-a'`. May you infer $datacenter \to operator$? What extra domain rule would make it valid?

A dependency must survive every future legal row, not merely the current sample.

No. Another service or cluster can be maintained by another operator in the same datacenter. The dependency would be valid only under a much stronger rule such as “the platform assigns exactly one operator to every cluster in every datacenter,” which is not the intended domain.

Compute closure as a trace

Attribute closure answers: starting with attributes X, what attributes can the dependency set force? Let

$$F = \{A \to B,\ B \to C,\ CD \to E\}.$$

To calculate AD^+, begin with $\{A, D\}$ and add only a right side whose left side is already present.

Step	Available attributes	Dependency enabled	New attribute
Start	A, D	—	—
1	A, D	$A \to B$	B
2	A, B, D	$B \to C$	C
3	A, B, C, D	$CD \to E$	E

So $AD^+ = \{A, B, C, D, E\}$. The order of enabled steps can vary; the final closure cannot, provided you keep applying dependencies until none adds an attribute.

Each step preserves equality on the starting attributes. If two tuples agree on `A`, $A \to B$ forces agreement on `B`; agreement on `B` forces `C`; agreement on both `C` and `D` forces `E`. Thus any two tuples agreeing on `A,D` agree on every attribute in the closure.

Try the closure tool

Choose seed attributes, fire each dependency only when its determinant is available, and compare your trace to the displayed closure. Use `A,D` after trying a smaller seed such as `A`.

BUILD AN ATTRIBUTE CLOSURE ONE JUSTIFIED STEP AT A TIME

For seed `{A, D}`, apply A → B, B → C, then CD → E.

AD⁺ = `{A, B, C, D, E}`

Static state: AD is a key for the five-attribute universe because the closure reaches every attribute.

INVARIANT

X is a superkey for schema R exactly when X^+ contains every attribute of R under the assumed dependency set. It is a candidate key only if no proper subset of X is also a superkey.

Apply closure to a real key question

Let `TaskAllocation(task_id, service_name, cluster_id, node_ip, port)` have the dependency

$$(service_name, cluster_id) \rightarrow (node_ip, port).$$

Is `(task_id, service_name, cluster_id)` a candidate key under this dependency set? Show the closure and the minimality check.

The proposed attributes already include three columns; use the dependency to add the remaining two, then try removing one attribute at a time.

Its closure starts with `task_id, service_name, cluster_id` and adds `node_ip, port`, so it is a superkey. Removing `task_id` cannot identify which individual task allocation is meant. Removing `service_name` or `cluster_id` prevents the given dependency from identifying the assigned node and port. Under the stated dependency set, it is a candidate key. A different domain rule, such as an immutable UUID, can produce an additional candidate key.

Derive dependencies without hand-waving

Three sound inference rules are enough for many small derivations.

Rule	Form	Example
Reflexivity	If $Y \subseteq X$, then $X \to Y$	$AB \to A$
Augmentation	If $X \to Y$, then $XZ \to YZ$	$A \to B$ implies $AD \to BD$
Transitivity	If $X \to Y$ and $Y \to Z$, then $X \to Z$	$A \to B, B \to C$ implies $A \to C$

> ***Worked example.*** *From $A \to B$ and $B \to C$, derive $A \to C$ by transitivity. With $A \to B$, derive $AD \to BD$ by augmentation. In the earlier closure, these steps explain why `A,D` can eventually enable $CD \to E$—first it gains `B`, then `C`.*

> **WARNING**
>
> Do not reverse a dependency. $course_code \to title$ does not imply $title \to course_code$ unless the domain makes titles unique. Direction records a claim about equality, not a visual arrow between columns.

Check a derivation

Given $F = \{A \to B, B \to C, CD \to E\}$, can $D \to E$ be derived? Can $AD \to E$ be derived?

Compute the two closures separately. A dependency can fire only when all determinant attributes are available.

$D^+ = \{D\}$, so $D \to E$ cannot be derived. AD^+ gains `B` from $A \to B$, then `C` from $B \to C$, so it contains `C,D` and gains `E` from $CD \to E$. Therefore $AD \to E$ is derivable.

Reduce a dependency set before using it

A minimal cover gives a compact equivalent set of dependencies. It is a tool for reasoning, not a claim that the compact version is easier for every reader to understand.

EXAMPLE

Start with

$$F = \{A \to BC,\ B \to C,\ AB \to C\}.$$

Split the right side: $A \to B$ and $A \to C$. Then $A \to C$ follows from $A \to B$ and $B \to C$, so remove it. `AB → C` follows from $B \to C$ because `B` is already part of `AB`, so remove it too. A minimal cover is $\{A \to B, B \to C\}$.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.1.1: Attribute Closure Computation

Construction application

Given schema $R(A, B, C, D, E)$ and functional dependency set $F = \{A \to B, B \to C, CD \to E\}$, compute the attribute closures D^+ and $(AD)^+$. Conclude which of the dependencies $D \to E$ or $AD \to E$ is logically implied by F.

Apply Armstrong's axioms or the attribute closure algorithm iteratively until no new attributes can be added.

$D^+ = \{D\}$ because no dependency in F has determinant subset $\{D\}$; therefore $D \to E$ is not implied. For $(AD)^+$: start with $\{A, D\}$; $A \to B$ adds B, yielding $\{A, B, D\}$; $B \to C$ adds C, yielding $\{A, B, C, D\}$; $CD \to E$ adds E, yielding $\{A, B, C, D, E\}$. Since $E \in (AD)^+$, $AD \to E$ is logically implied by F.

Exercise 2.1.2: Armstrong Axiom Soundness and Reversal Fallacy

Verification recall

An engineer observes that in the current service catalog snapshot every `display_name` is associated with exactly one `service_id`. Explain why inferring $display_name \to service_id$ from this snapshot is unsound under Definition 2.1.1.

Functional dependencies express invariant constraints across all legal instances, not coincidental properties of a single snapshot.

A functional dependency is a semantic assertion that must hold across all legal instances admitted by the domain. Observing unique display names in one snapshot does not preclude registering distinct services under the same human-readable label (e.g. `Edge Proxy`) in future regional or versioned deployments. Inferring $display_name \rightarrow service_id$ confuses temporary instance data with an invariant domain constraint.

Exercise 2.1.3: Canonical Minimal Cover Derivation

Construction application

Compute a canonical minimal cover for $F = \{A \rightarrow BC,\ B \rightarrow C,\ AB \rightarrow C\}$. Show each reduction step: right-hand side decomposition, extraneous attribute elimination, and redundant dependency removal.

Decompose into singleton right sides, test determinants for extraneous attributes, then test each FD for redundancy.

1. Decompose right-hand sides: $\{A \to B, A \to C, B \to C, AB \to C\}$.
2. Check extraneous attributes: In $AB \to C$, since $B \to C$ holds, A is extraneous; the dependency reduces to $B \to C$ (a duplicate).
3. Check redundant dependencies: In $\{A \to B, A \to C, B \to C\}$, computing A^+ under $\{A \to B, B \to C\}$ yields $\{A, B, C\}$, showing $A \to C$ is transitive and redundant. The minimal cover is $\{A \to B, B \to C\}$.

Exercise 2.1.4: Multi-Region Service Endpoint Dependency Redesign

Generalisation transfer

A routing table currently claims $(service_name, environment) \to host_ip$. The platform introduces multi-region deployments (e.g. production traffic in `us-east` on `10.0.1.5` and in `eu-west` on `10.0.2.8`). Provide a counterexample instance refuting the dependency, specify a revised relation schema with grain 'one regional service endpoint', and state its valid functional dependency.

A single service now has multiple regional deployments. The determinant must incorporate the region.

Counterexample tuples: `(auth-svc, prod, us-east, 10.0.1.5)` and `(auth-svc, prod, eu-west, 10.0.2.8)`. Both share $(service_name, environment)$ but differ in $host_ip$, violating the functional dependency. Revised schema: $RegionalEndpoint(service_name, environment, region, host_ip)$ with grain 'one regional endpoint for a deployed service'. Assuming a service has exactly one primary ingress host per region in an environment, the valid dependency is $(service_name, environment, region) \rightarrow host_ip$.

References & Further Reading

- Ramakrishnan and Gehrke, *Database Management Systems*: schema-refinement chapters in the open-access contents
- Ramakrishnan and Gehrke: Functional Dependencies slides
- PostgreSQL: Constraints

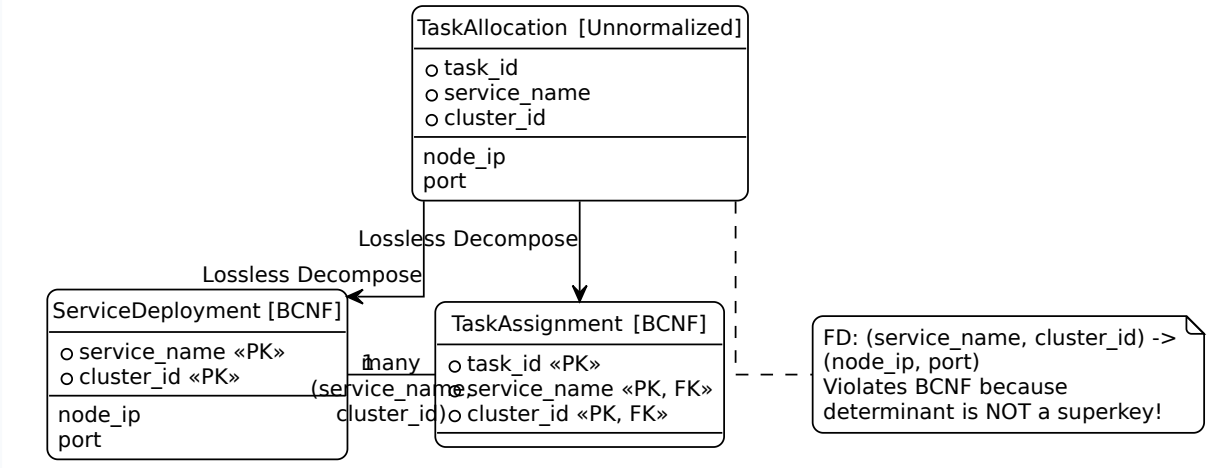
SECTION DETAIL

Normalization: decompose without losing facts

Normalization is not a ritual of splitting tables. It asks whether one tuple stores several facts whose values can change independently. If it does, a dependency tells us where the repetition comes from and a lossless decomposition tells us whether we can separate the facts safely.

DEFINITION

A relation has an update anomaly when one logical fact is stored in multiple tuples and those copies can disagree. Normalization uses dependencies to move each independently determined fact to a relation whose key actually determines it.



See the repeated fact before naming a normal form

Suppose one relation records both scheduled tasks and cluster deployment facts:

```
TaskAllocation(task_id, service_name, cluster_id, node_ip, port)
```

The grain is **one task execution assigned to a deployed service on a cluster**. The infrastructure scheduling rule says

$$(service_name, cluster_id) \rightarrow (node_ip, port).$$

`task_id`	`service_name`	`cluster_id`	`node_ip`	`port`
t-101	auth-svc	us-east-1	10.0.1.5	8080
t-102	auth-svc	us-east-1	10.0.1.5	8080
t-103	billing-svc	us-east-1	10.0.1.12	8443

> ***Worked example.*** *`(task_id, service_name, cluster_id)` is a candidate key under the stated rules, yet `node_ip` and `port` depend only on `(service_name, cluster_id)`. Moving `auth-svc` to another node requires editing rows `t-101` and `t-102`. A missed edit creates an inconsistent state about the same service deployment.*

Predict the anomaly

`auth-svc` migrates to node `10.0.1.9` before any task is scheduled. Which operation is awkward in this relation, and what data could be lost if the last `billing-svc` task finishes and its row is deleted?

The relation's grain begins with a task execution, but the deployment fact is about a cluster service.

Inserting the service deployment is awkward because there is no `task_id` to form a legal tuple. Deleting the last `billing-svc` task also deletes the only record of its assigned node and port. These are insertion and deletion anomalies; they reveal that a service deployment fact has been forced into a task-grain relation.

Use the normal forms as tests

Form	Useful test
1NF	Each attribute value is modeled as one value in its chosen domain; repeating groups are made explicit relations when they represent repeated facts.
2NF	Every non-prime attribute is fully dependent on every candidate key; no non-prime fact depends on a proper subset of a composite candidate key.
3NF	For every nontrivial $X \to A$, either X is a superkey or A is prime (belongs to a candidate key).
BCNF	For every nontrivial $X \to Y$, X is a superkey.

> **WARNING**
>
> "Atomic" does not mean "never split a value." A postal address may be a single value for one application and several independently queried facts for another. Choose domains from the operations and constraints you must support.

In `TaskAllocation`, `(service_name, cluster_id)` is not a superkey because it does not determine `task_id`. Its dependency to `node_ip, port` therefore violates BCNF. Because those attributes are non-prime and depend on a proper subset of the composite key, it also violates 2NF.

Recheck the determinant

Trace the dependency inside the actual normalization example rather than changing fixtures.

Start	Dependency applied	Closure after the step	Missing attribute
`(service_name, cluster_id)`	`(service_name, cluster_id) → (node_ip, port)`	`(service_name, cluster_id, node_ip, port)`	`task_id`

In `TaskAllocation`, the service deployment determinant cannot add a task identifier.

For `TaskAllocation`, `(service_name, cluster_id)^+` adds `node_ip` and `port`, not `task_id`; it is not a superkey. That is exactly the BCNF failure. The functional-dependencies lesson's explorer is the place to practice the general closure algorithm; this table keeps the proof tied to the relation being decomposed.

Decompose, then prove the join is lossless

Split the relation on the violating dependency:

```
ServiceDeployment(service_name, cluster_id, node_ip, port)
TaskAssignment(task_id, service_name, cluster_id)
```

`ServiceDeployment` has one tuple per service deployed to a cluster. `TaskAssignment` has one tuple per task scheduled on that deployment. A foreign key from `(service_name, cluster_id)` in `TaskAssignment` to the key of `ServiceDeployment` records the intended connection.

PROPOSITION 2.2.1 **Binary lossless-join criterion**

For a decomposition of relation R into R_1 and R_2, the decomposition is lossless with respect to F when

$$(R_1 \cap R_2) \to R_1 \quad \text{or} \quad (R_1 \cap R_2) \to R_2$$

is implied by F^+. In words: the shared attributes determine one whole side.

For every proposed split, verify a three-part evidence record: the two projected relations, their natural join on the shared attributes, and either (a) equality with the original legal instance under the stated dependencies or (b) a concrete spurious tuple. A lossless claim without the shared-attribute closure and this counterexample test is incomplete. The join test is a finite witness for the supplied instance; the criterion above is the dependency-level argument that generalizes it.

> ***Worked example.*** *The shared attributes are* `(service_name, cluster_id)`*. They determine all of* `ServiceDeployment` *by the deployment dependency* `(service_name, cluster_id) -> (node_ip, port)`*, so joining* `TaskAssignment` *back to* `ServiceDeployment` *does not invent extra task/deployment combinations. The decomposition is lossless under the stated rule.*

> ***Counterexample.*** *Let* $R(A, B, C)$ *contain* `(a1,b1,c1)` *and* `(a2,b1,c2)`*. Decompose it into* $R_1(A, B)$ *and* $R_2(B, C)$ *without any dependency that makes* `B` *determine either side. Joining the projections produces four tuples, including spurious* `(a1,b1,c2)` *and* `(a2,b1,c1)`*. Splitting columns alone is not a proof.*

Work the criterion

For the `TaskAllocation` split, name R_1, R_2, their intersection, and the dependency that satisfies the criterion.

VERIFY THE LOSSLESS-JOIN CRITERION

For a relation R(A, B, C) decomposed into R1(A, B) and R2(B, C), evaluate which statements regarding join losslessness and functional dependencies are correct:

Answer: The decomposition is lossless if and only if the shared attribute set determines either R1 or R2 under F+.; If the intersection fails to determine either component, the natural join may produce invalid spurious tuples.

Binary decomposition losslessness requires the intersection of attributes to functionally determine at least one of the decomposed schemas.

Put the service-deployment relation on the side determined by the shared pair.

Take $R_1 = ServiceDeployment(service_name, cluster_id, node_ip, port)$ and $R_2 = TaskAssignment(task_id, service_name, cluster_id)$. Their intersection is $(service_name, cluster_id)$, and $(service_name, cluster_id) \to R_1$ follows from $(service_name, cluster_id) \to (node_ip, port)$ plus reflexivity. The criterion therefore proves the binary decomposition lossless.

BCNF is not the only goal

To decompose a schema S on a BCNF-violating dependency $X \to Y$, form:

$$S_1 = X \cup Y, \qquad S_2 = S - (Y - X).$$

DEFINITION

A BCNF decomposition keeps the determinant X in its second relation. Writing only $S - Y$ can accidentally remove attributes needed to join the pieces back and is not the BCNF decomposition rule.

BCNF removes every dependency whose determinant is not a superkey. It can sacrifice local enforcement of another dependency. 3NF is often chosen when lossless decomposition **and** dependency preservation matter more than removing every BCNF anomaly.

> ***Worked example.*** *Let $R(A, B, C)$ have $F = \{AB \to C, C \to B\}$. Candidate keys are* `AB` *and* `AC`*; all attributes are prime, so $C \to B$ satisfies 3NF but violates BCNF because* `C` *is not a superkey. Decomposing on $C \to B$ yields* `CB` *and* `AC`*. The original dependency $AB \to C$ is not contained in either relation, so enforcing it may require a join or another design decision.*

Choose the trade-off

Your team can enforce dependencies with single-relation checks during every write, but a BCNF decomposition would require checking one dependency across a join. What design question should be documented before choosing 3NF or BCNF?

THE BCNF VS 3NF ARCHITECTURE DECISION

Consider schema Address(Street, City, Zip) with FDs: {(Street, City) -> Zip, Zip -> City}. Candidate keys are (Street, City) and (Street, Zip). Notice Zip -> City violates BCNF because Zip is not a superkey, but satisfies 3NF because City is prime. Decomposing into R1(Zip, City) and R2(Street, Zip) achieves BCNF.

Evaluate the trade-offs of decomposing Address into BCNF versus retaining 3NF:

Reference decision: Remaining in 3NF preserves (Street, City) -> Zip as a local single-table check without cross-table join enforcement.

3NF guarantees lossless joins and dependency preservation, making writes cheap to validate locally. BCNF eliminates all FD-induced redundancy but may sacrifice dependency preservation.

There is no automatic winner: name the anomaly risk and the cost of enforcement.

Document the business cost of inconsistent redundant facts, the dependency that would no longer be locally enforceable under BCNF, the write path that must enforce it, and the expected query/join cost. Choose 3NF when preserving that dependency locally is important and its residual redundancy is understood; choose BCNF when the stronger anomaly protection outweighs that enforcement cost.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.2.1: Binary Decomposition Losslessness Verification

Verification application

A relation schema $TaskAllocation(task_id, service_name, cluster_id, node_ip, port)$ is decomposed into $ServiceDeployment(service_name, cluster_id, node_ip, port)$ and $TaskAssignment(task_id, service_name, cluster_id)$. Using Proposition 2.2.1, verify whether this decomposition is lossless under functional dependency $(service_name, cluster_id) \rightarrow (node_ip, port)$.

Find the shared attributes between ServiceDeployment and TaskAssignment, then compute their attribute closure.

The intersection of attributes is $ServiceDeployment \cap TaskAssignment = \{service_name, cluster_id\}$. Under the given functional dependency, $\{service_name, cluster_id\}^+ = \{service_name, cluster_id, node_ip, port\}$, which equals the entire attribute set of $ServiceDeployment$. By the binary lossless-join criterion, because the intersection functionally determines one of the decomposed relations, the decomposition is guaranteed to be lossless.

Exercise 2.2.2: Spurious Tuple Construction on Arbitrary Split

Construction application

Let $R(A, B, C)$ be decomposed into $R_1(A, B)$ and $R_2(B, C)$ under empty dependency set $F = \emptyset$. Construct a concrete 2-tuple instance of R whose natural join $R_1 \bowtie R_2$ produces spurious tuples, and identify the spurious combinations.

Choose two tuples that share a B value but have distinct A and C values.

Let $R = \{(a_1, b_1, c_1), (a_2, b_1, c_2)\}$. The projections are $R_1 = \{(a_1, b_1), (a_2, b_1)\}$ and $R_2 = \{(b_1, c_1), (b_1, c_2)\}$. Computing $R_1 \bowtie R_2$ yields four tuples: $\{(a_1, b_1, c_1), (a_1, b_1, c_2), (a_2, b_1, c_1), (a_2, b_1, c_2)\}$. The tuples (a_1, b_1, c_2) and (a_2, b_1, c_1) are spurious tuples not present in R.

Exercise 2.2.3: BCNF vs. 3NF Architectural Trade-Off Analysis

Description recall

Explain why schema $Address(Street, City, Zip)$ with dependencies $(Street, City) \rightarrow Zip$ and $Zip \rightarrow City$ is in 3NF but not BCNF. What engineering trade-off is made when deciding whether to decompose to BCNF?

Check candidate keys and whether every determinant is a superkey or every determined attribute is prime.

Candidate keys are $(Street, City)$ and $(Street, Zip)$. In $Zip \rightarrow City$, Zip is not a superkey, violating BCNF. However, $City$ is part of candidate key $(Street, City)$ (i.e. prime), so it satisfies 3NF. Decomposing to BCNF produces $(Zip, City)$ and $(Street, Zip)$, eliminating redundancy but destroying dependency preservation: enforcing $(Street, City) \rightarrow Zip$ now requires an expensive join assertion across both tables.

Exercise 2.2.4: Lossless Normalization of Multi-Region Service Deployments

Generalisation transfer

A microservice deployment schema is given by $RegionalDeployment(task_id, service_name, cluster_id, region, endpoint_port, node_ip)$ with dependencies $(service_name, cluster_id) \rightarrow$

$node_ip$ and $(service_name, cluster_id, region) \rightarrow endpoint_port$. Propose a BCNF decomposition into three relations with stated grains, candidate keys, and lossless join justifications.

Separate service placement facts, regional routing endpoint facts, and task executions iteratively.

Decompose into three relations:

1. $ServiceHost(service_name, cluster_id, node_ip)$ with key $(service_name, cluster_id)$ and grain 'one node assignment for a service cluster'.
2. $RegionalIngress(service_name, cluster_id, region, endpoint_port)$ with key $(service_name, cluster_id, region)$ and grain 'one regional ingress port for a cluster service'.
3. $TaskExecution(task_id, service_name, cluster_id, region)$ with key $(task_id, service_name, cluster_id, region)$ and grain 'one task execution allocated to a regional service'. Each step is lossless: $ServiceHost$ intersects the remainder on $(service_name, cluster_id)$, which determines $ServiceHost$; $RegionalIngress$ intersects $TaskExecution$ on $(service_name, cluster_id, region)$, which is a candidate key of $RegionalIngress$.

References & Further Reading

- Ramakrishnan and Gehrke, *Database Management Systems*: schema-refinement chapters in the open-access contents
- Ramakrishnan and Gehrke: Functional Dependencies slides
- PostgreSQL: Constraints

SECTION DETAIL

Schema review: turn requirements into constraints

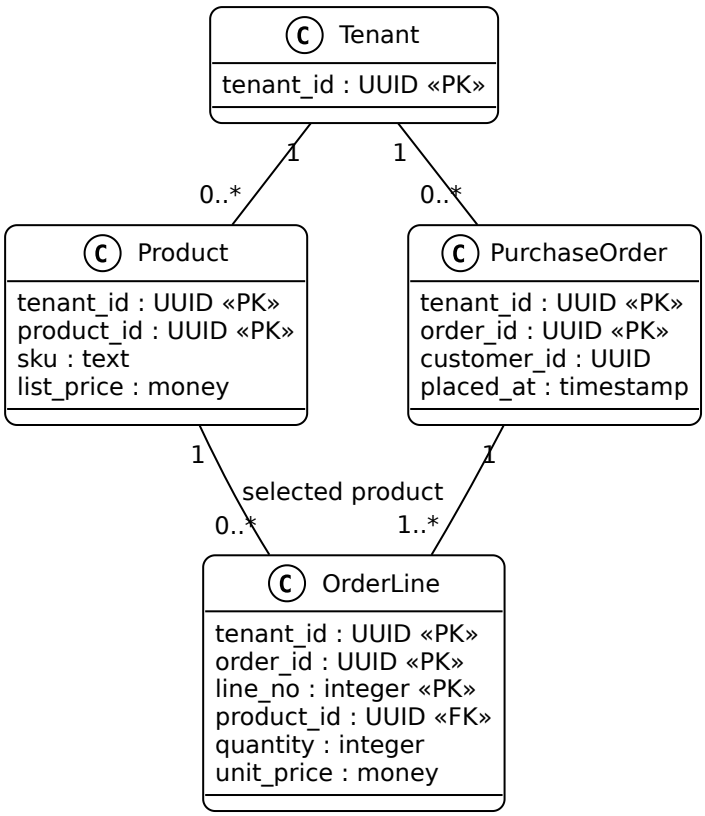
A schema review is a testable argument, not a naming review. For every relation, ask: what is one row? What identifies it? Which other row must exist? Which value is a historical fact rather than a lookup? Then write the smallest constraint or test that would catch a violation.

DEFINITION

A schema review maps domain rules to database-enforceable constraints, documents rules that cannot be enforced locally, and supplies adversarial examples for both. It reviews facts and failure modes before it reviews syntax.

Review one multi-tenant order design

An online marketplace gives every seller a tenant. Product and order identifiers are tenant-scoped, so a foreign key must carry the tenant identity too.



Relation	Grain	Key	Core reference or rule
`Product(tenant_id, product_id, sku, list_price)`	One catalog product for one tenant	`(tenant_id, product_id)`	`sku` may need `UNIQUE(tenant_id, sku)` if the business calls it a tenant-local identifier
`PurchaseOrder(tenant_id, order_id, customer_id, placed_at)`	One placed order for one tenant	`(tenant_id, order_id)`	Tenant-scoped customer reference, if customers are tenant-scoped
`OrderLine(tenant_id, order_id, line_no, product_id, quantity, unit_price)`	One product line in one order	`(tenant_id, order_id, line_no)`	Composite reference to its order and selected product; `quantity > 0`

> ***Worked example.*** *`unit_price` belongs on `OrderLine` when it records the price actually agreed at purchase time. `Product.list_price` is a current catalog fact and may change tomorrow. Replacing `unit_price` with a lookup from `Product` changes historical orders retroactively; it is not a harmless normalization.*

Predict the tenant leak

Tenant A has `product_id = p9`; tenant B also has `product_id = p9`. An order line for tenant A references only `product_id = p9`. What incorrect fact can the database accept if the foreign key is only `OrderLine.product_id → Product.product_id`?

The identifier is scoped by tenant, so the product ID alone is ambiguous.

The order line can be associated with tenant B's product even though the order belongs to tenant A. The database sees a matching product ID but cannot prove tenant consistency. Make `(tenant_id, product_id)` a referenced key of `Product` and reference both columns from `OrderLine`.

KEEP THE TENANT IN THE PRODUCT REFERENCE

```
FOREIGN KEY (tenant_id, product_id)
REFERENCES Product (tenant_id, product_id)
```

A composite foreign key must name a candidate key with the same tenant-scoped meaning in Product.

INVARIANT

If identifiers are tenant-scoped, every relationship that uses one must establish tenant equality as part of its key or foreign key. Filtering by tenant in application code is not an equivalent integrity constraint.

Walk the review questions in order

Review question	Evidence to request	A failure it catches
What is one row?	A grain sentence for each relation	A line item table that also stores order-wide shipping state
What identifies it?	Candidate keys and whether they are stable	A display label silently used as a primary key
What must exist first?	Foreign keys and deletion/update policy	An order line with no order or wrong-tenant product
Which values are derived or historical?	Source of truth and update rule	Repricing past orders by joining current catalog price
Which rules span rows?	Transaction/procedure/test design	Two writes that independently pass an aggregate capacity rule
How will it change?	Backfill, compatibility, rollback, and verification plan	A non-null column added before legacy rows are populated

WARNING

A `CHECK` constraint validates the row being written; it is not a general substitute for a rule involving arbitrary other rows. “An order total equals the sum of its lines” or “stock never drops below zero” needs a design that coordinates reads and writes, not a wish written as a local check.

Review a mixed-grain proposal

A teammate proposes this shortcut:

```
OrderLine(tenant_id, order_id, line_no, product_id, quantity,
          order_status, placed_at, customer_email)
```

Name two facts with the wrong grain. Where should each live instead?

A line is one product occurrence; ask whether each value is the same for every line of an order.

`order_status` and `placed_at` are facts about the whole order, so they belong in `PurchaseOrder`. `customer_email` is normally a customer fact, so it belongs in a tenant-scoped customer relation or an explicitly justified immutable order-contact snapshot. Repeating order-level values on every line causes update anomalies and makes a changed line capable of contradicting its own order.

Turn dependencies into constraints and tests

DEFINITION

A dependency hypothesis is useful in a review only when it names a business rule that can be enforced by a key, tested by adversarial rows, or explicitly owned by an application procedure.

For this design, the following dependencies express useful review hypotheses:

$$(tenant_id, product_id) \to (sku, list_price)$$

$$(tenant_id, order_id, line_no) \to (product_id, quantity, unit_price).$$

They justify the relation keys only if the product and line identifiers really are stable in the business domain. A natural key such as SKU may also be unique per tenant, but only after spelling out capitalization, reuse, and retirement policy.

> ***Counterexample.*** *Adding* `UNIQUE(sku)` *globally because the current export has no duplicates can reject two independent sellers' legitimate use of the same SKU. The requirement is usually tenant-local uniqueness, which has a different key and different error behavior.*

Make an adversarial test set

Write four insert attempts that should not all succeed:

1. a duplicate line number in one order;
2. a zero or negative quantity;
3. a line that names a missing order;
4. a line for tenant A that names tenant B's product ID.

Map each attempt to a distinct constraint; the last one needs the composite key.

The duplicate is rejected by `PRIMARY KEY (tenant_id, order_id, line_no)`. The invalid quantity is rejected by `CHECK (quantity > 0)`. The missing order is rejected by `FOREIGN KEY (tenant_id, order_id) REFERENCES PurchaseOrder`. The cross-tenant product is rejected only when the product foreign key is `(tenant_id, product_id) REFERENCES Product(tenant_id, product_id)`. These tests establish different invariants; one successful happy-path order proves none of them.

Review a change before the migration is written

The product team wants to allow a line to have no catalog product after a product is retired, while preserving the original line description and price.

> ***Worked example.*** *This is not just "make* `product_id` *nullable." Decide first whether a line must retain a historical product snapshot, whether product deletion is forbidden, or whether retirement is represented by a status rather than deletion. If the relationship becomes optional, update the foreign-key nullability, the reporting contract, and tests for null product references together.*

REMARK **Operational rollout deferred to migration planning**

A schema review should identify, but not guess, the rollout method: how existing lines are backfilled, whether reads tolerate a temporary dual representation, what rollback preserves, and how a large index or constraint build affects live traffic. Those operational choices require deployment tooling and production workload metrics.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.3.1: Composite Key Multi-Tenant Isolation

Verification application

In a multi-tenant order schema, an engineer declares `OrderLine(tenant_id, order_id, line_no, product_id, quantity)` with foreign key `FOREIGN KEY (product_id) REFERENCES Product(product_id)`. Verify why this foreign key fails to prevent cross-tenant data corruption, and specify the correct multi-column constraint.

The foreign key must reference both the tenant identifier and the product identifier.

Because `product_id` is only unique within a tenant, referencing `Product(product_id)` alone permits an order in tenant A to reference a catalog product belonging to tenant B. The constraint must enforce multi-tenant scoping: `FOREIGN KEY (tenant_id, product_id) REFERENCES Product(tenant_id, product_id)`, backed by a matching composite candidate key on `Product`.

Exercise 2.3.2: Order Line–Item Grain and Constraint Definition

Construction application

Specify SQL table definitions for `PurchaseOrder` and `OrderLine` with composite tenant keys. Include primary keys, referential integrity foreign keys, and a check constraint ensuring line quantities are strictly positive.

Include tenant_id in primary and foreign keys of both relations.

```sql
CREATE TABLE PurchaseOrder (
  tenant_id UUID NOT NULL,
  order_id BIGINT NOT NULL,
  ordered_at TIMESTAMPTZ NOT NULL,
  PRIMARY KEY (tenant_id, order_id)
);

CREATE TABLE OrderLine (
  tenant_id UUID NOT NULL,
  order_id BIGINT NOT NULL,
  line_no INT NOT NULL,
  product_id BIGINT NOT NULL,
  quantity INT NOT NULL CHECK (quantity > 0),
  PRIMARY KEY (tenant_id, order_id, line_no),
  FOREIGN KEY (tenant_id, order_id) REFERENCES
PurchaseOrder(tenant_id, order_id)
);
```

Exercise 2.3.3: Product Retirement Historical Snapshot Modeling

Description recall

When a catalog product is retired, existing order lines must preserve the purchase price and product description at the time of sale. Explain why making `product_id` nullable without capturing historical price attributes violates relational schema design principles.

Distinguish between the current catalog state and an immutable transaction line snapshot.

An order line records an immutable historical financial event. If `product_id` is set to NULL or points to an updated/deleted product row, historical reports either lose identity or recompute totals using altered prices. The schema must store historical unit price and description directly on `OrderLine`, or retain soft-deleted product records with active status indicators.

Exercise 2.3.4: Discount Code vs. Applied Discount Schema Review

Generalisation transfer

Review a proposed `OrderDiscount(tenant_id, order_id, code, amount)` relation. Determine whether its grain represents a discount definition, an applied order discount, or both. State candidate keys, foreign keys, and identify two business rules that must be clarified before finalizing the schema.

Separate coupon definition rules from the event of applying a coupon to a purchase order.

The grain represents one applied discount on a purchase order. The candidate key is `(tenant_id, order_id, code)` if each code can apply at most once per order, with foreign key `(tenant_id, order_id) REFERENCES PurchaseOrder(tenant_id, order_id)`. The general definition of codes belongs in a separate `Discount(tenant_id, code, percentage, valid_until)` relation. Before finalizing, clarify: (1) whether multiple instances of the same discount code can be applied to an order, and (2) whether `amount` is an immutable financial snapshot or recomputed dynamically upon return or line adjustment.

References & Further Reading

- PostgreSQL: Constraints
- PostgreSQL: `CREATE TABLE`
- Ramakrishnan and Gehrke, *Database Management Systems*: schema-refinement chapters

SECTION DETAIL

SQL lab: build and verify an allocation report

Every task uses the same fixed fixture and expected edge cases. The embedded runner executes read-only SQLite in the browser, so a wrong join or a misplaced predicate produces a real result rather than a plausible hand trace. SQLite and PostgreSQL both support aggregate `FILTER`, but a SQLite result is not evidence of PostgreSQL-specific behavior. Use a disposable local PostgreSQL database for constraint error text and PostgreSQL plans.

Evidence path	What it establishes	What it does not establish
In-browser SQLite runner	executable join, grouping, and `NOT EXISTS` result semantics for this fixture	PostgreSQL-specific planner, MVCC, constraint error text, or isolation behavior
PostgreSQL 18 fixture	PostgreSQL syntax, constraints, `FILTER`, and an observed `EXPLAIN` plan	behavior of other engines or every workload

Record the engine and version with every observed result. In the runner, `SELECT sqlite_version();` reports the bundled SQLite version. Do not use SQLite output as evidence that a PostgreSQL-specific statement or plan executed.

DEFINITION

A lab result contract states the output grain, inclusion rules, ordering, and test rows before execution. A query is not complete because it returns plausible rows on a happy-path dataset.

Lab brief and evidence checklist

Artifact	What to include	Evidence of success
Zero-safe allocation report	Contract, query, predicted table	Service pools with zero active allocations remain
Anti-join	Contract, query, predicted table	A client with only an other-region allocation is correctly included
Constraint probes	Four deliberately invalid writes	Each is rejected by the intended key, foreign key, or check
Plan observation (optional)	`EXPLAIN` output and one cautious claim	A measured plan fact, not a promise that all data will use that plan

> **WARNING**
>
> The browser runner is read-only and sandboxed to this fixture; it is not a production client. Do not run lab statements against a shared or production schema, and do not treat SQLite acceptance as proof of PostgreSQL behavior.

Create the fixed fixture

The browser runner below loads this fixture into SQLite automatically. To test PostgreSQL-specific behavior, load these statements into a disposable PostgreSQL database; the tables also support hand tracing.

```
CREATE TABLE client (
  client_id    integer PRIMARY KEY,
  display_name text NOT NULL
);

CREATE TABLE service_pool (
  pool_id      integer PRIMARY KEY,
  service_code text NOT NULL,
  region       text NOT NULL,
  capacity     integer NOT NULL CHECK (capacity >= 0)
);

CREATE TABLE allocation (
  client_id integer NOT NULL REFERENCES client(client_id),
  pool_id   integer NOT NULL REFERENCES service_pool(pool_id),
  status    text NOT NULL CHECK (status IN ('active', 'queued',
'suspended')),
  memory_gb integer CHECK (memory_gb IS NULL OR memory_gb BETWEEN 0 AND
100),
  PRIMARY KEY (client_id, pool_id)
);

INSERT INTO client (client_id, display_name) VALUES
  (1, 'Ada'), (2, 'Bo'), (3, 'Cy'), (4, 'Dee');

INSERT INTO service_pool (pool_id, service_code, region, capacity)
VALUES
  (10, 'S101', 'us-east-1', 2),
  (11, 'S201', 'us-east-1', 1),
  (12, 'S301', 'us-east-1', 2),
  (13, 'S101', 'us-west-2', 2);

INSERT INTO allocation (client_id, pool_id, status, memory_gb) VALUES
  (1, 10, 'active', 87),
  (2, 10, 'suspended', 40),
  (3, 11, 'queued', NULL),
  (4, 13, 'active', 93);
```

Relation	Fixture facts
`client`	Ada, Bo, Cy, and Dee exist
`service_pool`	S101, S201, S301 are in us-east-1; S101 also has an older us-west-2 pool
`allocation`	Ada has active allocation in S101; Bo is suspended; Cy is queued; Dee has active allocation only in us-west-2

Execute the fixture

The runner creates the tables and rows above in SQLite. Predict the warm-up report before pressing **Run query**, then change the query and explain every difference.

SERVICE ALLOCATION FIXTURE

Count active allocations per us-east-1 service pool. Predict three rows, including the zero counts, before running.

```
SELECT p.service_code,
     COUNT(a.client_id) AS active_allocations
FROM service_pool AS p
LEFT JOIN allocation AS a
ON a.pool_id = p.pool_id
AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
ORDER BY p.service_code;
```

The screen version executes this query against the fixture in SQLite. Predict the output before checking it in an engine.

INVARIANT

`allocation` has grain “one client’s status in one service pool.” Its composite primary key forbids two simultaneous rows for the same client/pool pair. The foreign keys forbid an allocation for an unknown client or service pool.

Warm-up prediction

Before writing SQL, list the us-east-1 service pools and the count of rows whose status is `active`. Do not count `suspended` or `queued` rows.

Filter the allocation rows by status first, then keep every current-region service pool.

The desired report has S101 with count 1, S201 with count 0, and S301 with count 0. The fixed fixture has no active row for S201 or S301. This table is the oracle for the first task, not an optional afterthought.

Task 1: keep service pools with zero active allocations

Result contract: one row per service pool in `us-east-1`, including pools with no active allocations; columns are `pool_id`, `service_code`, `active_allocations`, and `is_full`; ordered by service code then pool ID.

Start with this skeleton. Decide where the status condition belongs before running it.

COMPLETE THE MATCHING AND GROUPING CONTRACT

```
SELECT p.pool_id,
     p.service_code,
     COUNT(a.client_id) AS active_allocations,
     COUNT(a.client_id) >= p.capacity AS is_full
FROM service_pool AS p
LEFT JOIN allocation AS a
ON a.pool_id = p.pool_id
AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code, p.capacity
ORDER BY p.service_code, p.pool_id
```

`COUNT(a.client_id)` counts only real matches, so a null-extended service pool reports zero; compare that count with capacity for `is_full`. The status predicate belongs in `ON`: moving it to `WHERE` removes the pools this report must include. `GROUP BY` states the one-row-per-pool grain.

```sql
SELECT p.pool_id,
       p.service_code,
       COUNT(a.client_id) AS active_allocations,
       COUNT(a.client_id) >= p.capacity AS is_full
FROM service_pool AS p
LEFT JOIN allocation AS a
  ON a.pool_id = p.pool_id
 AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code, p.capacity
ORDER BY p.service_code, p.pool_id;
```

Worked example. *The left join emits each us-east-1 service pool even when no active right row exists.* ***`COUNT(a.client_id)`*** *ignores the null-extended value, so it produces zero for S201 and S301;* ***`COUNT(*)`*** *would instead count the left row and be wrong for this contract.*

Expected `Task 1` result (SQLite evaluates boolean comparisons such as `>=` as `0` / `1`, whereas PostgreSQL returns `false` / `true`):

`pool_id`	`service_code`	`active_allocations`	`is_full`
10	S101	1	0
11	S201	0	0
12	S301	0	0

Break it on purpose

Move `a.status = 'active'` into `WHERE`, then predict the result. Why is S201 absent even though it has an allocation row?

`WHERE` filters after NULL extension and after the right-side status is visible.

Only S101 remains. S201's only match has status `queued`, so the `WHERE` predicate is false; S301 has a null-extended status, so the predicate is `UNKNOWN`. Both are removed. The query has changed from "all service pools, count active matches" to "service pools with at least one active match."

> ***Counterexample.*** *A query can be syntactically valid, return a count, and still fail the report contract. A test dataset containing only service pools with active allocations would not expose this* `ON` / `WHERE` *bug; zero and nonqualifying cases must be seeded deliberately.*

Task 2: find clients with no active allocation in region us-east-1

Result contract: one row per client who has no `active` allocation in a us-east-1 service pool; ordered by client ID.

Complete the absence test, then compare your result with the fixture.

RECONSTRUCT THE CORRELATED ABSENCE TEST

```
SELECT c.client_id, c.display_name
FROM client AS c
WHERE NOT EXISTS (
SELECT 1
FROM allocation AS a
JOIN service_pool AS p ON p.pool_id = a.pool_id
WHERE a.client_id = c.client_id
  AND a.status = 'active'
  AND p.region = 'us-east-1'
);
```

`NOT EXISTS` states the required absence directly and is NULL-safe. The correlated equality is what binds the subquery to one outer client; without it the subquery answers a global question instead of a per-client one.

```sql
SELECT c.client_id, c.display_name
FROM client AS c
WHERE NOT EXISTS (
  SELECT 1
  FROM allocation AS a
  JOIN service_pool AS p ON p.pool_id = a.pool_id
  WHERE a.client_id = c.client_id
    AND a.status = 'active'
    AND p.region = 'us-east-1'
)
ORDER BY c.client_id;
```

> ***Worked example.*** *Bo is included because suspended is not active. Cy is included because queued is not active. Dee is included because the only active row is in us-west-2. Ada is excluded because the correlated subquery finds the active us-east-1 S101 allocation.*

Expected `Task 2` result:

`client_id`	`display_name`
2	Bo
3	Cy
4	Dee

> **WARNING**
>
> Do not replace this query with `NOT IN (SELECT client_id FROM allocation)` without a precise proof about NULL and the omitted region/status conditions. `NOT EXISTS` states the required correlated absence directly.

Change the requirement

The platform policy changes "no active allocation" to "no active **or queued** allocation." Which predicate changes, and which expected row disappears?

<details>
<summary></summary>

The region boundary is unchanged. Only the accepted allocation states change.

</details>

<details>
<summary></summary>

Change the correlated condition to `a.status IN ('active', 'queued')`. Cy disappears because Cy is queued in S201 for us-east-1. Bo remains because suspended is still excluded; Dee remains because the only row is in us-west-2.

</details>

Task 3: test the schema, not just the queries

For each proposed write, predict the intended rejection. In a local PostgreSQL lab, issue each in its own transaction and roll it back after the expected error.

Proposed fact	Constraint that should reject it
Add a second `(1, 10, 'active', 91)` row	Composite primary key on `(client_id, pool_id)`
Add `(1, 999, 'active', 91)`	Foreign key to `service_pool`
Add `(1, 11, 'active', -4)`	Memory allocation range check
Add `(1, 11, 'active', 101)`	Memory allocation range check

```
BEGIN;
INSERT INTO allocation (client_id, pool_id, status, memory_gb)
VALUES (1, 999, 'active', 91);
-- PostgreSQL reports a foreign-key violation here.
ROLLBACK;
```

INVARIANT

Constraint tests are part of the schema's behavior specification. A successful `SELECT` cannot prove that invalid state is impossible; each rule needs at least one adversarial write that targets it.

Diagnose a missing rule

The infrastructure team now says "a full service pool may not gain another active allocation." Can you express that correctly with the existing row-local `CHECK` constraints? State what is missing from the lab design.

Capacity depends on multiple allocation rows, but a CHECK sees its own row's values.

No. `capacity` and the count of active allocation rows live in different rows/relations, so an ordinary row-local check cannot safely compare them. The design needs a transaction-aware allocation operation or another coordinated integrity mechanism, plus concurrency tests. This lab intentionally stops before transaction isolation and does not pretend a check constraint solves it.

Task 4: make a cautious plan observation (optional)

Run the zero-safe report with PostgreSQL's read-only plan command:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT p.pool_id,
       p.service_code,
       COUNT(a.client_id) AS active_allocations
FROM service_pool AS p
LEFT JOIN allocation AS a
  ON a.pool_id = p.pool_id
 AND a.status = 'active'
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
ORDER BY p.service_code, p.pool_id;
```

Record exactly one observation that the output supports, such as "on this four-pool fixture, PostgreSQL chose a sequential scan." Do **not** conclude that a sequential scan is always best or that the plan proves semantic correctness.

REMARK **Optional extension on testing an index hypothesis**

In a disposable lab only, add an index after recording the baseline, rerun the same `EXPLAIN`, and compare the actual rows as well as the plan. For a tiny fixture PostgreSQL may rationally keep a sequential scan. The experiment teaches that a physical change is a cost hypothesis, while the report contract is a correctness obligation.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.4.1: Conditional Aggregation with Status Filtering

Construction application

Write a PostgreSQL query against `service_pool(pool_id, service_code, region)` and `allocation(client_id, pool_id, status)` that reports `(service_code, active_count, queued_count)` for all us-east-1 service pools, preserving pools with zero allocations. Use standard SQL aggregate `FILTER` clauses.

Use a LEFT JOIN from service_pool to allocation and aggregate using FILTER (WHERE ...).

```sql
SELECT p.service_code,
       COUNT(*) FILTER (WHERE a.status = 'active') AS
active_count,
       COUNT(*) FILTER (WHERE a.status = 'queued') AS
queued_count
FROM service_pool AS p
LEFT JOIN allocation AS a
  ON a.pool_id = p.pool_id
WHERE p.region = 'us-east-1'
GROUP BY p.pool_id, p.service_code
ORDER BY p.service_code;
```

`COUNT(*) FILTER (WHERE ...)` evaluates conditions over right-side matches while preserving empty service pools with count 0.

Exercise 2.4.2: Capacity Check Constraint Scope Limitation

Verification application

A developer attempts to enforce a service pool capacity limit by writing `CHECK ((SELECT COUNT(*) FROM allocation a WHERE a.pool_id = pool_id) <= capacity)` inside a table definition. Verify why standard SQL rejects or fails to guarantee this constraint under concurrent transactions.

Check constraints are evaluated strictly on the single modified row and cannot safely query dynamic foreign tables.

Standard SQL table check constraints evaluate predicates over the immediate columns of the mutated row; subqueries in `CHECK` constraints are either explicitly disallowed by RDBMS engines or susceptible to race conditions under non-serializable isolation levels, where concurrent commits can exceed capacity without seeing each other's inserts.

Exercise 2.4.3: Query Plan Execution vs. Correctness Semantics

Description recall

Explain why observing a sequential scan in `EXPLAIN ANALYZE` on a four-row fixture does not indicate an optimization defect or justify adding an index.

Compare disk page fetch overhead with sequential cache hits on small tables.

For small relations occupying a single disk page, a sequential scan reads the single page directly into cache without paying the additional overhead of traversing an index B-tree. The optimizer correctly minimizes cost based on catalog statistics. Physical plans reflect cost models, not relational correctness contracts.

Exercise 2.4.4: Multi-Status Allocation Reporting with Null Preservation

Generalisation transfer

Design test cases for the multi-status allocation query. Identify two adversarial seed rows that expose bugs in: (1) moving status checks to `WHERE`, and (2) using unfiltered `COUNT(*)`.

Provide a service pool with only suspended allocations, and a service pool with zero allocations.

1. Seed a service pool with only a `suspended` allocation: moving status checks into `WHERE` removes this pool entirely instead of reporting counts `(0, 0)`.
2. Seed a service pool with zero allocations: using unfiltered `COUNT(*)` reports count 1 due to the null-extended joined row, whereas `COUNT(*) FILTER (...)` or `COUNT(a.client_id)` correctly yields 0.

References & Further Reading

- PostgreSQL 18: `SELECT`
- PostgreSQL 18: Constraints
- PostgreSQL 18: `EXPLAIN`
- PostgreSQL 18: Table Expressions and Joins

Storage, access, and execution

SECTION DETAIL

Storage Engines

An order update changes `paid_at` from `NULL` to a timestamp. The SQL-level fact is tiny; the physical system must find a page, keep it resident while a worker changes it, preserve a stable reference from indexes, and survive a power loss between the write and the acknowledgement. A storage engine is the machinery that makes those obligations compatible.

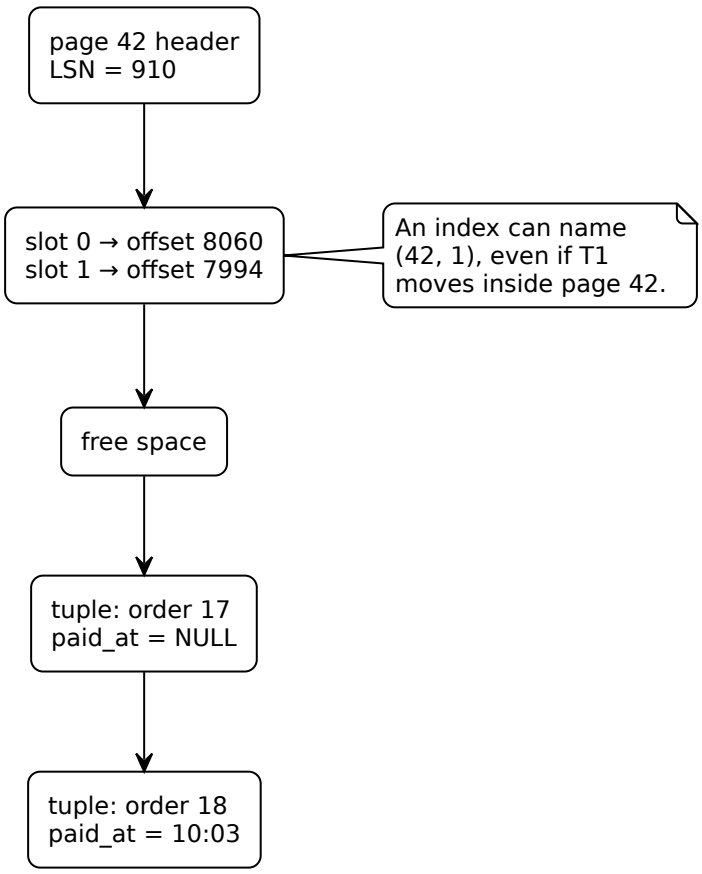
A page I/O is one page read or write; it is the model's unit of work. A scan over N pages is $O(N)$ I/Os. A balanced tree with fan-out B reaches a key in about $O(\log_B N)$ page reads. A hash table is useful for equality lookup under its distribution assumptions, not for ordered range traversal. Treat these as named approximations: each comparison states its page, cache, and workload assumptions.

Pages preserve location while records move

DEFINITION

A **page** is the unit the engine reads, caches, writes, and recovers. A **record identifier** is usually `(page_id, slot_id)`, not an application key such as an order number.

A slotted page grows its slot directory from one end and tuple payloads from the other. A slot points to a tuple's current offset. Moving a variable-length tuple *within the page* changes the slot, so an index entry that names the slot remains valid. Moving it to another page requires a forwarding record or index maintenance. The distinction explains why a physical identifier is unsuitable as a public business identifier.



The header records structural facts such as free-space boundaries and, in a WAL-based engine, the most recent log sequence number reflected by the page. PostgreSQL's current format is a concrete example: it has a fixed page header, line pointers, free space, items, and special space. Other engines choose different fields; the indirection principle is the transferable idea.

Prediction — a tuple grows

Page 42 has 40 bytes of free space. Slot 1 points to a 24-byte tuple. Updating the tuple makes it 56 bytes. Which action preserves an index entry `(42, 1)`: overwrite in place, change only the slot, or copy to page 43 without further work?

Ask whether the record identifier names bytes or an indirection entry.

Overwriting is impossible: the tuple needs 32 more bytes. If another 56-byte region exists within page 42 after compaction, the engine can copy the tuple there and change slot 1's offset; `(42, 1)` still resolves. Copying it to page 43 without a forwarding record or index update leaves the old identifier dangling. The missing idea is stable indirection, not a larger page.

A buffer frame has three different protections

INVARIANT

A worker may use a page only while its frame is pinned; a frame may be evicted only when its pin count is zero; a dirty update must have a recovery-safe path before the frame is reused.

The buffer pool maps page identifiers to memory frames. Fetching a page pins a frame, possibly waits for I/O, and returns a frame that is initialized. A short latch protects the in-memory page structure while the worker examines or changes slots. A transaction lock or MVCC rule governs which *logical* rows a transaction may observe or modify across statements. These scopes differ deliberately.

Mechanism	Protects	Typical lifetime	Failure if omitted
Pin	frame residency	until caller releases the page	eviction turns a live pointer into a stale reference
Latch	bytes and metadata in one frame	a short critical section	concurrent slot edits corrupt page structure
Transaction lock / version rule	logical row or predicate	until commit, abort, or validation	a correct page can still contain a lost update

Worked trace — two fetches and one eviction

1. Worker A fetches page 42; the page table installs frame 3 and A pins it.
2. Worker B asks for page 42 while the read is in flight; B joins that in-flight fetch rather than issuing a second read.
3. A latches frame 3, changes slot 1, marks it dirty, releases the latch, and later unpins it.
4. Eviction may choose frame 3 only after **all** pins are gone. It first satisfies the log condition below, writes the dirty page if necessary, then removes the page-table mapping.

I/O can take orders of magnitude longer than metadata updates. A global latch held during I/O serializes unrelated page lookups. A correct design protects the page table briefly, represents the in-flight state, and lets other fetches wait on that page rather than on the entire pool.

Application — scan pollution

A pool holds 100 pages. A hot OLTP working set uses pages 1–80 repeatedly. An analyst sequentially scans 10,000 distinct pages using naive LRU. Predict the result for the next OLTP burst.

A page referenced once by the scan looks recent to LRU.

The scan continually inserts one-time pages at the most-recent end and eventually evicts much of pages 1–80. The next OLTP burst faults those hot pages again. A scan-resistant policy can give sequential pages low retention priority or use a separate ring, but it cannot ignore the need to pin pages while a scan consumes them.

Layout is a workload hypothesis

EXAMPLE

A fraud screen fetches one order and its 25 attributes; a finance report sums two numeric columns over 200 million orders. The first benefits from row locality; the second from reading and compressing only the needed columns.

Layout	Favors	Pays for	Boundary case
Row store	point reads, full-row writes, transactional updates	unnecessary bytes during narrow analytics	wide rows can waste cache and I/O
Column store	scans of few attributes, aggregation, compression	rewriting or locating one full mutable row	late materialization must still reconstruct rows when needed
Log-structured / LSM	sustained append-heavy writes and immutable segments	compaction, read amplification, tombstone retention	a point read needs an index/filter across levels

Compression changes the I/O/CPU balance. Dictionary, run-length, delta, and bit-packed encodings exploit repeated or ordered values; they are not free. A column segment that compresses well may be expensive to update one row at a time. An LSM engine writes new versions sequentially, then compacts segments; it must retain a delete tombstone until every relevant snapshot and replica can no longer interpret an older value.

Transfer — a changed workload

An event table receives 50,000 inserts/s, later serves a 90-day aggregate, and also supports one support-agent lookup by event ID. Choose a primary layout and name the auxiliary structure that prevents the lookup from becoming a full scan.

Separate the write path, analytical path, and point-lookup path instead of seeking one universally best layout.

An append-friendly log/LSM or time-partitioned row store fits the ingest path; columnar segments or a columnar replica can serve the aggregate after data becomes stable. A B-tree or hash-like key directory on `event_id` supplies the support lookup. The decision is conditional: if support lookups become dominant and updates are frequent, a row-oriented primary store may become the simpler choice.

Columnar execution changes the unit of work

DEFINITION

A row store keeps the fields of one logical row adjacent; a column store keeps values of one attribute adjacent across rows. The layout changes which bytes a scan must read, which encodings are cheap, and how updates are applied.

A finance query that sums `amount` over 200 million orders touches one or two attributes. In a row store with a 200-byte row, a full scan still reads every attribute for every row: roughly 40 GB. In a column store the same query reads the `amount` column and perhaps `region`, and each column segment can be compressed independently: dictionary encoding for repeated region names, delta encoding for ordered timestamps, run-length encoding for low-cardinality status. The saving is not a better algorithm; it is a different unit of I/O.

A vectorized executor can process batches instead of invoking each operator once per row. Amortizing dispatch and processing contiguous values can improve CPU efficiency and sometimes expose SIMD operations; the gain depends on operator, data, and implementation. The MonetDB/X100 design uses this approach. Vectorized execution and columnar storage are complementary choices, not the same mechanism, and compression can add decoding work.

Question	Row store answer	Column store answer
Read one full order	one page, all attributes together	reconstruct the row from several column segments
Sum one numeric column	read all attributes, discard most	read only that segment, often compressed
Update one field	in-place or new tuple version	append a delta or rewrite a segment
Compress repeated values	limited by row heterogeneity	strong per-column encodings
Late materialization	unnecessary	fetch remaining columns only for qualifying rows

Columnar layouts pay for their scan advantage on updates and point reads. A single-row update may touch many segments, and a delete becomes a deletion vector or a rewritten segment. That is why analytical engines separate ingest, compaction, and query paths, and why a columnar layout is a workload hypothesis rather than a universal upgrade.

Application — change the workload

A fraud service changes from nightly two-column scans to frequent updates and lookups of individual orders with all 25 attributes. Does the original columnar advantage still determine the primary layout?

Compare the bytes and structures touched by a point operation, not only the scan's previous throughput.

Not by itself. A row-oriented primary with an order-ID index can fetch and update one record without reconstructing it from multiple column segments. Retain a columnar analytical replica only if measured scan demand justifies replication and freshness costs. Compare update rate, point latency, scan frequency, and lag rather than assuming either layout is always superior.

COLUMNAR LAYOUT AND LATE MATERIALIZATION

An analytical query evaluates SELECT AVG(revenue) FROM orders WHERE status = 'delivered' AND date >= '2026-01-01'. The table has 40 columns. Evaluate which statements regarding columnar storage and execution are correct:

Answer: Scanning and filtering only compressed status and date column segments avoids reading the remaining 37 unreferenced attributes from disk.; Late materialization evaluates predicates to form qualifying row-id bitsets before fetching matching revenue segment values.

Column stores optimize analytical queries by reading only referenced columns and using late materialization to minimize payload retrieval. In contrast, high-frequency point updates incur high write amplification across columnar segments.

A log makes a dirty page recoverable

THEOREM 3.1.1 **Write-Ahead Logging Protocol Invariant**

Under write-ahead logging (WAL), the log record describing an update to a database page must be written to non-volatile storage before the modified data page itself may be flushed to disk. Furthermore, a transaction commit acknowledgement requires the corresponding commit log record to be durable.

For the `paid_at` update, write a log record with LSN 911, change the in-memory page, mark its page LSN 911, then flush the log through 911 before flushing page 42. After a crash, recovery compares the durable log with page LSNs: a page missing a logged update can be redone; an uncommitted update can be undone or made invisible according to the system's recovery design. Writing the page first risks a durable change with no durable explanation.

> **WARNING**
>
> A checksum detects accidental corruption probabilistically; it does not authenticate a page against a malicious writer. Backups and replicas help only when their integrity is independently verified.

Prediction — crash boundary

The log is durable through LSN 910. Page 42 has been changed in memory to page LSN 911, but neither the log record 911 nor the page is flushed. The machine loses power. What may recovery expose?

Check whether any durable record of LSN 911 reached non-volatile storage.

Nothing from LSN 911 may be exposed, because neither its log record nor its page image was durable. If the page had reached storage before the log record, recovery could find an unexplained update and could not rely on WAL redo/undo reasoning. The safe result follows from the ordering condition, not from hoping the device flushes in a convenient order.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.1.1: Write-Ahead Logging Flushing Order Verification

Verification application

A storage engine flushes dirty data page 42 (containing uncommitted update with page LSN 911) to persistent disk while the WAL buffer is durable only through LSN 910. Referring to Theorem 3.1.1, identify why this flush violates the write-ahead logging invariant and what failure mode ensues if power is lost immediately afterward.

Examine whether recovery can reconstruct the transaction status of changes on page 42.

The flush violates Theorem 3.1.1 because the data page modification reached disk before its explanatory log record was durable. If power is lost immediately afterward, recovery encounters a page containing an uncommitted modification without a durable undo log record, leaving persistent storage in an unrecoverable, inconsistent state.

Exercise 3.1.2: Slotted-Page Architecture and Item Pointer Indirection

Description recall

Describe how a slotted-page architecture uses an array of item pointers (line pointers) at the start of the page to decouple index pointers from physical byte offsets during record updates.

Indexes store (page_number, slot_index) pairs rather than raw byte offsets.

In a slotted page, indexes reference a tuple by its tuple ID `(page_number, slot_index)`. The slot array grows forward from the page header while variable-length tuple records are packed from the end of the page upward. When a tuple is updated or defragmented within the page, only its offset in the slot array changes; index references to the stable slot index remain completely untouched.

Exercise 3.1.3: Row vs. Column I/O Calculation on Filtered Scans

Construction application

Consider a table with 1,000,000 rows and 50 fixed-width 8-byte columns (400 bytes per row, 20 rows per 8 KB page, total 50,000 pages). An analytical query computes the average of one column for rows matching a predicate on another column, with 1% selectivity. Compare the raw I/O volume between a full row-store table scan and a column store using late materialization (assuming 8-byte uncompressed column segments).

Row store must read all 50,000 pages. Column store reads only the filter and target columns.

1. Row store: Scans all 50,000 pages = $50{,}000 \times 8\ \text{KB} = 400\ \text{MB}$.
2. Column store: The filter column requires $1{,}000{,}000 \times 8\ \text{B} = 8\ \text{MB}$ (1,000 pages). The matching 1% (10,000 rows) requires reading target column segments for those positions ($10{,}000 \times 8\ \text{B} = 80\ \text{KB}$, spanning at most 1,000 pages if contiguous or fetched selectively). Total columnar I/O is at most $8\ \text{MB} + 8\ \text{MB} = 16\ \text{MB}$, a $25\times$ reduction in data transferred.

Exercise 3.1.4: Layout Selection under Shifting Analytical–OLTP Workloads

Generalisation transfer

An engineering team migrates a financial fraud system from batch nightly reporting (2–column scans over 50 columns) to real-time transactional scoring (1,000 updates/sec touching all 50 attributes by customer ID). Contrast the physical trade-offs of retaining a pure columnar store versus switching to a row-oriented primary engine with an asynchronous columnar replica.

Analyze write amplification and lock granularity for point updates across 50 separate column files.

A pure columnar engine suffers extreme write amplification under 1,000 point updates/second: each single-row update must modify or append to 50 independent column segments, incurring heavy I/O, lock contention, and fragmentation. Switching to a row-oriented primary engine isolates each update to a single slotted page with minimal write amplification and microsecond index lookups. Analytical queries can be offloaded to an asynchronous columnar replica (e.g. via change data capture), decoupling write throughput from scan optimization.

References & Further Reading

- Stonebraker, M., et al. (2005). *C-Store: A Column-oriented DBMS*. VLDB, 553–564. (Column layout, compression, and read-optimized projections.)
- Abadi, D. J., Madden, S. R., & Hachem, N. (2008). *Column-Stores vs. Row-Stores: How Different Are They Really?* ACM SIGMOD, 967–980. (Late materialization, compression, and the limits of a row-store simulation.)
- Boncz, P. A., Zukowski, M., & Nes, N. J. (2005). *MonetDB/X100: Hyper-Pipelining Query Execution*. CIDR. (Vectorized, cache-conscious execution.)
- PostgreSQL: Database Page Layout
- PostgreSQL: Write-Ahead Logging
- CMU 15–445/645 Database Systems

SECTION DETAIL

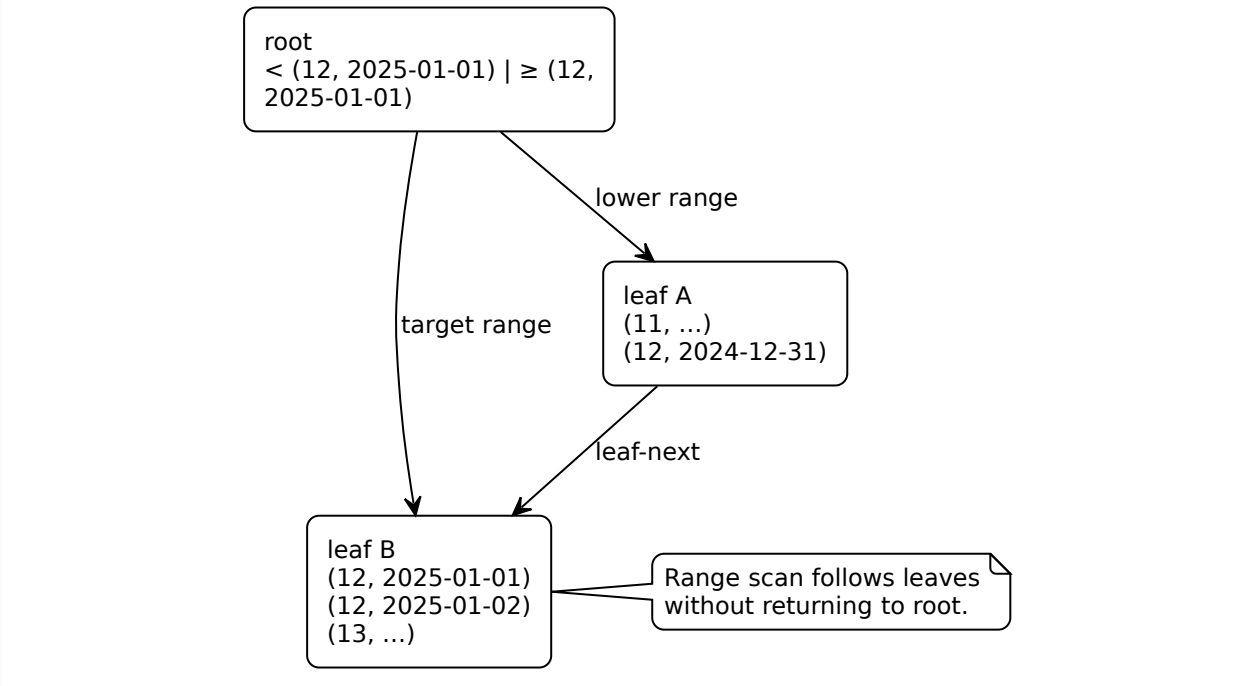
Indexes

A support query asks for the newest 50 failed deliveries for tenant 12. A table scan is correct, yet reading millions of unrelated rows is not a design. An index can make the request selective, but every new delivery and status update must also maintain its entries. The central question is not “should this column be indexed?” but “which ordered search space does this workload need, and what update work buys it?”

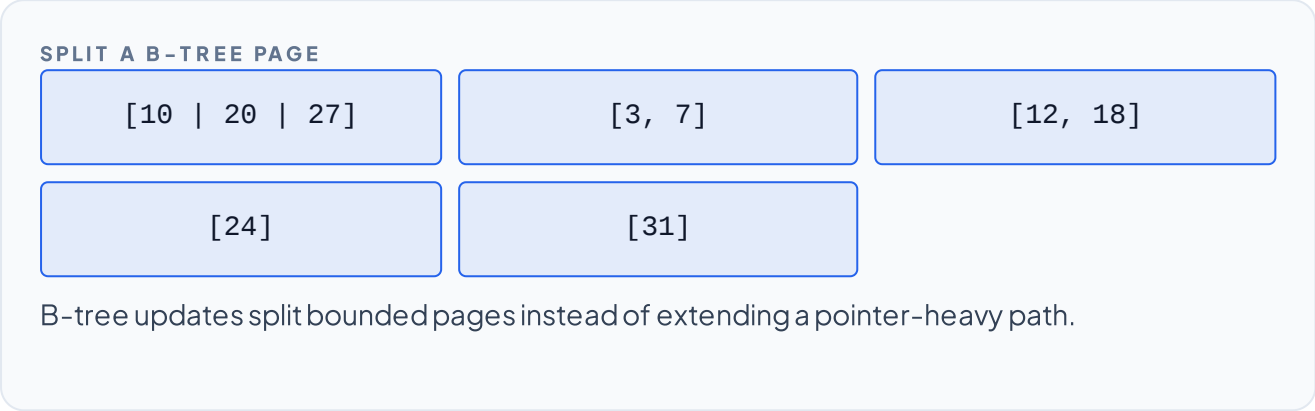
B+ tree navigation is an ordering proof

DEFINITION

A B+ tree keeps keys sorted in page-sized nodes. An internal separator partitions key ranges; leaves contain search keys and tuple references and are linked for ordered scans. Balance means every leaf has the same depth.



To find `(12, 2025-01-02)`, compare it with the root separator, choose leaf B, then search that leaf. If B is full on insertion, split it into two sorted leaves and propagate a separator upward. The split changes navigation, not query meaning. A correct implementation ensures every key remains in exactly one leaf and every separator directs search to a child whose range contains the key.



B-tree updates split bounded pages instead of extending a pointer-heavy path.

Use the playground to insert its shown keys until the leaf divides. Before advancing, state which half contains the new key and which separator the parent needs. The visual models a leaf split only; it does not model disk latches, deletion merge policy, or vendor page format.

Prediction — separator choice

Leaves after an insert must be `[4, 7]` and `[9, 13]`. Which value can be a separator if the parent routes keys less than the separator left and all other keys right?

Use the smallest key on the right leaf under this comparison convention.

`9` works: 4 and 7 route left; 9 and 13 route right. `7` would route 7 right even though it remains left, unless the comparator convention were changed consistently. Separator values are routing contracts, not decorative summaries.

Composite keys have a left-to-right geometry

> **INVARIANT**
>
> An index on `(tenant_id, created_at, id)` is ordered first by tenant, then by time *within an equal tenant*, then by `id` within an equal tenant and time.

Query shape	Can `(tenant_id, created_at, id)` navigate directly?	Why
`tenant_id = 12 ORDER BY created_at, id`	Yes	equality fixes the leading group; suffix order is already present
`tenant_id = 12 AND created_at >= t`	Yes	starts at one tenant/time boundary and scans forward
`created_at >= t` for all tenants	Usually no	matching times are scattered across tenant groups
`tenant_id = 12 ORDER BY id`	Not generally	`created_at` intervenes before `id` order

This is lexicographic order, not a slogan about "leading columns." It also explains why a query returning most of a table can prefer a sequential scan: an index may locate every tuple but require many scattered heap reads. Clustering, cache state, visibility, and selectivity decide the trade.

Worked example — an access path proposal

For:

```
SELECT id, created_at, error_code
FROM delivery
WHERE tenant_id = $1 AND status = 'failed'
ORDER BY created_at DESC, id DESC
LIMIT 50;
```

an index such as `(tenant_id, status, created_at DESC, id DESC) INCLUDE (error_code)` can align equality predicates with output order and may avoid a sort. It is useful only if failed rows per tenant are selective enough and the added write/space cost is justified. If `status` changes often, the same

index can amplify a hot write path.

> **REMARK** **Conditions for index-only scans**
>
> All selected values may be present in the index, but an MVCC engine can still need a heap or visibility-map check before returning a tuple. An index-only scan is a plan opportunity, not a guarantee that heap I/O is zero.

Derived structures must preserve query semantics

> **WARNING**
>
> An index is not the logical source of truth. It is correct only if visible rows required by its predicate have entries and each entry resolves to a row that still satisfies its indexed condition.

A partial index stores only rows satisfying a fixed predicate, for example `WHERE status = 'failed'`. The optimizer may use it only when it can establish that the query condition implies the partial predicate. An expression index on `lower(email)` similarly requires matching normalization, collation, and expression semantics. A raw `email` index does not automatically answer a transformed predicate.

Hash indexes target equality lookups; they do not yield the ordered leaf traversal required by a range. A Bloom filter answers a narrower question: it may safely say "definitely absent," but "possibly present" still requires another access path. Treating a filter's positive result as a row location is a logical bug.

Transfer — a subtle uniqueness requirement

Each tenant may register an email once, case-insensitively. Tenants may share the same spelling. Propose the key and state two details that could invalidate a naive implementation.

The uniqueness domain contains both tenant identity and the agreed normalization rule.

Use a unique index or constraint over `(tenant_id, lower(email))` only after specifying the normalization/collation contract. The naive `UNIQUE(email)` wrongly forbids different tenants; `UNIQUE(tenant_id, email)` can allow case variants when the product treats them as equal. Unicode normalization and whether NULL emails are allowed must be specified as well, because DBMS NULL uniqueness behavior varies.

Maintenance cost belongs in the design

EXAMPLE

Inserting one delivery row may insert one heap record plus entries in the primary key, foreign-key support, status/time, and audit indexes. Updating an indexed status can remove and add an entry even when the business update looks small.

Proposal	Read benefit	Write and recovery cost	Decision evidence
key index on `delivery(id)`	direct point lookup	one entry per insert; split/vacuum work	point lookup frequency and uniqueness need
wide covering support index	fewer heap visits for support query	larger entries, cache pressure, status-update work	`EXPLAIN (ANALYZE)` plus write benchmark
partial failed-delivery index	small hot subset	predicate transitions maintain membership	failed-row fraction and query implication

Measure representative reads and writes, including warm and cold-cache assumptions. A plan estimate is evidence about a model, not proof: compare estimated and actual cardinalities, then inspect skew or stale statistics before adding an index merely to silence a slow example.

Prediction — reject an index

A monthly archive table is append-only, queried once nightly by `WHERE created_at >= current_date - interval '30 days'`, and 95% of rows match. Should a B-tree on `created_at` be the default answer?

Evaluate the I/O cost of random heap lookups when almost the entire table matches.

Not automatically. The predicate returns almost the entire relation, so a sequential scan can be cheaper than index navigation plus widespread tuple access. Time partitioning, zone maps/segment metadata, or a sequential aggregate may fit the actual workload better. The decisive evidence is selectivity and physical locality, not the existence of a date predicate.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.2.1: Covering Indexes and Visibility Map Prerequisites

Description recall

Explain why an index-only scan on a covering index `(tenant_id, status) INCLUDE (error_code)` may still perform random heap page reads in PostgreSQL MVCC execution.

Consider transaction isolation and tuple visibility flags.

Under multi-version concurrency control (MVCC), index tuples do not store transaction commit status or row creation/expiry timestamps (`xmin` , `xmax`). To return an attribute directly from the index without visiting the heap, the engine checks the table's visibility map. If the target heap page is not marked all-visible to all active transactions, the engine must fetch the underlying heap page to verify tuple visibility.

Exercise 3.2.2: Index Access Path Rejection on Low Selectivity

Verification application

A query scans an archive table where predicate `created_at >= NOW() - INTERVAL '30 days'` matches 95% of total rows. Verify why the query planner rejects an existing unclustered B-tree index on `created_at` in favor of a sequential scan.

Compare the cost of sequential page reads against random single-tuple heap page fetches.

With an unclustered B-tree index matching 95% of rows, fetching matching tuples through the index incurs random I/O for almost every page in the heap, reading nearly every heap page multiple times. A sequential table scan reads each heap page sequentially exactly once with hardware prefetching, resulting in substantially lower total cost despite evaluating non-matching rows.

Exercise 3.2.3: Partial and Expression Index Specification

Construction application

Specify PostgreSQL DDL for: (1) a partial index that optimizes queries looking up only deliveries in `status = 'failed'`, and (2) an expression index that enforces case-insensitive email uniqueness scoped by tenant.

Use a WHERE clause in CREATE INDEX for partial indexes, and apply lower() for expression indexes.

```
-- Partial index for failed deliveries
CREATE INDEX idx_delivery_failed
  ON delivery (tenant_id, created_at DESC, id DESC)
  WHERE status = 'failed';

-- Scoped case-insensitive unique constraint
CREATE UNIQUE INDEX idx_user_tenant_email_ci
  ON app_user (tenant_id, lower(email));
```

Exercise 3.2.4: Case-Insensitive Multi-Tenant Unique Constraint Design

Generalisation transfer

An engineering team must guarantee that no tenant can register duplicate user emails case-insensitively, while distinct tenants may legitimately register identical email addresses. Contrast `UNIQUE (email)`, `UNIQUE (tenant_id,`

`email)`, and `UNIQUE (tenant_id, lower(email))` against this requirement.

Check cross-tenant isolation and case normalization.

1. `UNIQUE (email)`: Incorrect. Prevents different tenants from registering the same email address, violating multi-tenant data isolation.
2. `UNIQUE (tenant_id, email)`: Incomplete. Permits a single tenant to register both `User@example.com` and `user@example.com`, violating case-insensitivity.
3. `UNIQUE (tenant_id, lower(email))`: Correct. Scopes uniqueness to the tenant and normalizes case uniformly, satisfying both isolation and case-insensitive identity invariants.

References & Further Reading

- PostgreSQL: Indexes
- PostgreSQL: Multicolumn Indexes
- PostgreSQL: Index-Only Scans and Covering Indexes
- CMU 15–445/645 Database Systems

SECTION DETAIL

Query Execution and Optimization

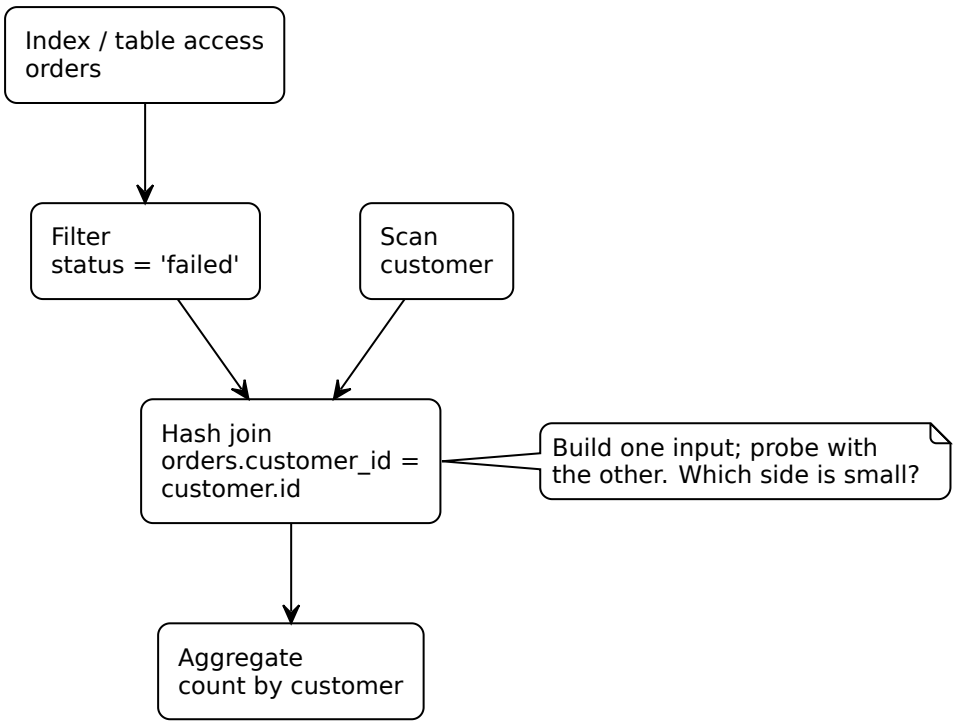
`SELECT` says what rows are required, not how to obtain them. For a tenant's failed deliveries, a database may scan a table, probe an index, hash a small relation, sort before aggregation, or stream groups in order. All correct plans return the same logical bag under the same SQL semantics; their intermediate cardinalities, memory pressure, and I/O differ radically.

Selectivity is the fraction of rows that survive a predicate: a 5% predicate on 10,000 rows first estimates 500 rows. For two predicates, multiplying selectivities is an **independence assumption**, not a fact about correlated data. Tree height, hash access, and page I/O use the same bounded models introduced in storage; every estimate below names what it ignores.

An operator tree carries relations upward

DEFINITION

A physical plan is an operator tree: each node consumes one or more streams of rows, produces a stream with a defined schema and multiplicity, and pays a resource cost under stated assumptions.



Read the tree bottom-up. The `orders` access produces rows; the filter reduces them; the join attaches customer attributes; the aggregate changes the grain to one row per customer. “Hash join” alone does not establish quality. Building a hash table on 10,000 filtered orders can fit memory; building one on 500 million customers may spill or fail the cost budget.

Prediction — join result grain

`orders` has one row per order and `order_line` has one row per line. A join on `order_id` followed by `SUM(line.amount)` has what grain immediately before the aggregate?

A one-to-many join repeats the order fields once per matching line.

The joined stream has one row per matching order line, not one per order. Grouping by customer then sums line amounts at customer grain. If an order-level amount is joined and summed after this join, it is duplicated once per line; aggregate it at order grain first or use a different construction.

Cost begins with cardinality, not a magic constant

PROPOSITION 3.3.1 **A first selectivity estimate**

For a relation R with $|R|$ rows and a predicate with selectivity s, a simple estimate is $|\sigma_p(R)| \approx s|R|$. For an equijoin, an independence-based first estimate is often $|R \bowtie S| \approx |R||S| / \max(V(R,a), V(S,a))$, where $V(R,a)$ is the distinct-count estimate. These are models to test, not laws of the data.

Suppose `orders` contains 10,000,000 rows and 1% are failed. The filter estimate is 100,000 rows. If 80% of failures belong to one enterprise tenant, treating `tenant_id` and `status` as independent can underestimate that tenant's result by nearly two orders of magnitude. That error can turn a nested loop chosen for a tiny outer input into repeated expensive probes.

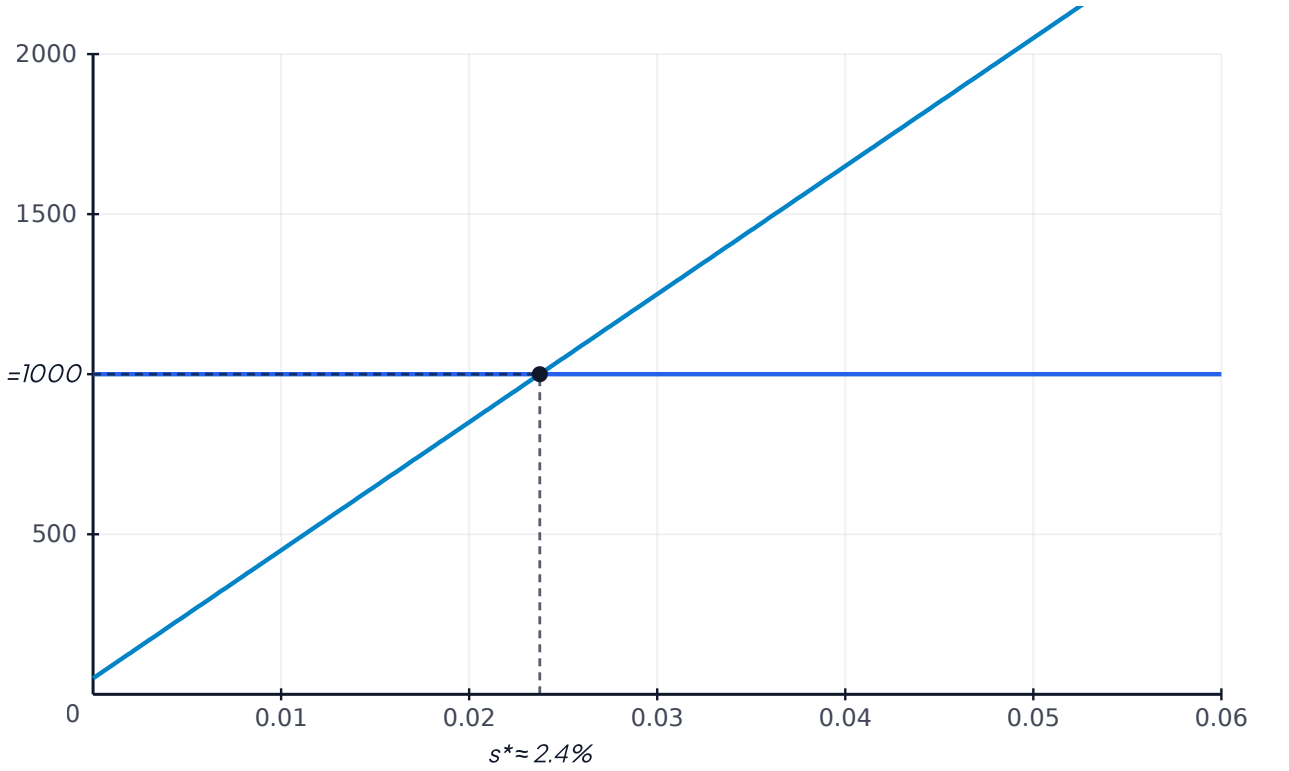


Figure 3.3.1: Estimated I/O cost as a function of predicate selectivity s. The unclustered index scan dominates for small s but exceeds the full sequential table scan at the crossover threshold $s^* \approx 2.4\%$.

CHANGE SELECTIVITY; INSPECT THE PLAN'S I/O MODEL

With 1,000 Orders pages and a 5% filter, 50 clustered pages qualify. Compare the three candidate plans under the same stated assumptions.

Sequential scan + hash join	1,100 I/Os
Clustered index selection + hash join	153 I/Os
Clustered index selection + nested loops	5,053 I/Os

Static state: low selectivity lets a clustered index avoid most Orders pages; the index-assisted hash plan costs 153 I/Os, versus 1,100 for the scan/hash plan and 5,053 for this nested-loops model.

Vary selectivity and page count, then predict before revealing a plan. The model exposes a bounded scan/index/join trade-off; it does not claim a real DBMS uses its exact coefficients. Its value is making the hidden assumption visible: a selective access path becomes worse when the matching set approaches a scan.

Worked example — two plans

Assume a 1,000-page `orders` table, a 20-page `customer` table, and a predicate that returns 2 orders. A nested-loop plan that probes an index for two customers may be excellent. If stale statistics estimate 2 but actual rows are 200,000, the same plan can perform 200,000 probes; a hash join with a 20-page customer build may dominate. The logical query did not change. The cardinality claim did.

> **REMARK** **Distinguishing statistical error from algorithm selection**
>
> Compare `EXPLAIN` estimates with `EXPLAIN ANALYZE` actual rows at the first divergent node. If the estimate is wrong, inspect stale statistics, skew, correlation, parameter-sensitive values, and predicate semantics. If estimates are sound but the cost model still chooses poorly, measure its I/O, memory, and cache assumptions before forcing a plan.

Physical algorithms expose different failure modes

> **EXAMPLE**
>
> A merge join needs both inputs ordered on the join key; it can exploit matching indexes or pay to sort. A hash join needs a build side that fits its memory budget or partition/spill logic. A nested loop needs a small outer input or a cheap indexed inner probe.

Operator choice	Strong case	Dangerous assumption	Observable warning
Sequential scan	high selectivity loss / most pages needed	index navigation is always faster	index plan visits most heap pages randomly
Index scan	few qualifying rows, useful order	predicate matches an index range	actual rows approach table cardinality
Hash join	modest build input, equality join	build fits memory	batches/spills or unexpectedly huge build
Merge join	inputs ordered or cheap to sort	order cost is recovered later	both inputs sort only for one join
Nested loop	tiny outer plus indexed inner	outer cardinality is truly tiny	probes explode after a misestimate

Application — predicate placement changes both semantics and cost

An outer join must retain every customer, even those with no failed order. Where does `order.status = 'failed'` belong: in `WHERE` after a left join or in the join condition / filtered right input?

Place it in the right input or `ON` condition. A `WHERE order.status = 'failed'` filter after the left join removes null-extended rows and turns the result into an inner-join-like result. The optimizer may push a semantically valid filter down, but only after preserving outer-join null behavior. Correct cardinality starts with correct semantics.

The optimizer searches a space of equivalent plans

DEFINITION

Query optimization selects a physical plan from the plans that compute the same logical result. Join order, access path, and operator implementation interact combinatorially, so an optimizer needs both a search strategy and a cost model; neither is a proof that the chosen plan is globally best.

System R's Selinger-style optimizer builds left-deep join trees bottom-up. For each subset of relations it keeps the cheapest plan for each interesting order, then extends that set by one relation and discards dominated alternatives. With n relations there are 2^n subsets, so dynamic programming avoids enumerating $n!$ orders, but the retained plan is only as good as its cardinality estimates and cost model.

Worked search. Let A have 100 rows, B 1,000 rows, and C 10 rows. Assume all relevant equality joins exist, $|A \bowtie B| = 50$, $|B \bowtie C| = 100$, and use the deliberately simplified cost $|L| + |R|$ per hash join, with both inputs scanned once and all intermediate results fitting memory. Ignore output production, ordering, and access-path differences.

Candidate	First intermediate	First join cost	Second join cost	Total
$(A \bowtie B) \bowtie C$	50 rows	1,100	60	1,160
$(B \bowtie C) \bowtie A$	100 rows	1,010	200	1,210
$(A \bowtie C) \bowtie B$	unknown	110	$1,000 + \lvert A \bowtie C \rvert$	$1,110 + \lvert A \bowtie C \rvert$

The first two plans differ because both the first join's input work and the intermediate size matter. The third is cheaper than 1,160 only if $|A \bowtie C| < 50$ under these assumptions. In a real engine, selectivity estimates, memory, useful ordering, and physical access paths can reverse this result. A hash join may build on the smaller input; the intermediate size is not automatically the build-side size.

Volcano/Cascades optimizers search top-down instead. A **memo** stores equivalence classes of logical expressions; transformation rules generate equivalent logical alternatives, implementation rules produce physical alternatives, and branch-and-bound pruning discards alternatives whose lower bound exceeds the best known cost. This can explore bushy trees that a left-deep-only search excludes and can track required physical properties, at the cost of more complex rule interaction and search control. System R also tracked interesting orders; this is not unique to Cascades. Engines choose different search and cost strategies.

> **WARNING**
>
> When a plan is bad, name which of three failures you are diagnosing: the optimizer never generated the useful equivalent plan, it generated and pruned it using a wrong cost, or it estimated the inputs incorrectly. "The optimizer is dumb" diagnoses none of them.

SEARCH STRATEGY VERSUS COST MODEL DIAGNOSIS

A query plan selects a nested loop with index probe. The planner estimates 50 outer rows, but execution observes 50,000 outer rows, causing severe probe degradation. Evaluate which diagnostic conclusions are sound:

Answer: The cost model was applied to an input cardinality estimate that the actual data distribution severely contradicts.; Correlated predicates or data skew across tenant attributes can cause standard single-column statistics to misestimate selectivity.

When actual rows diverge sharply from estimates, inspect histograms, multivariate statistics, and predicate correlation before asserting optimizer engine defects.

Plans must be read as conditional evidence

WARNING

A low estimated cost is not elapsed time and a plan is not a proof that the query is correct. Cost units compare alternatives under a DBMS model; actual timing depends on cache state, concurrency, hardware, and configuration.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT customer_id, count(*)
FROM orders
WHERE tenant_id = 12 AND status = 'failed'
GROUP BY customer_id;
```

For each node, record estimated rows, actual rows, loops, buffer reads/hits, and whether the node changed the expected grain. Test an empty tenant, a tenant with skewed failed rows, and a fresh-cached versus warm-cached run. A forced index plan is a diagnostic experiment, not a general remedy.

Transfer — an estimate that lies politely

A table has independent-looking global statistics: 1% `status='failed'`, 1% `region='north'`. In north, however, 90% of rows have failed. A query filters both predicates. What should the optimizer or designer learn from a large estimate error?

The missing fact is correlation, not arithmetic competence.

Global independence predicts about 0.01% of rows, while the north subset predicts about 0.9% if north is 1% of the table. The response can include richer/multivariate statistics, a partition-aware estimate, a targeted index only if the workload supports it, and a regression query against the skew. Hard-coding one plan hides the data-distribution cause.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.3.1: Attribute Independence Assumption Failure in Cardinality Estimation

Verification application

Referring to Proposition 3.3.1, explain why estimating $|\sigma_{A=a \wedge B=b}(R)|$ as $s_A \cdot s_B \cdot |R|$ fails severely when A and B are strongly correlated, and describe how extended multivariate statistics repair the calculation.

Multiplying independent probabilities understates joint selectivity when values co-occur.

The product rule $s_A \cdot s_B$ assumes statistical independence. If attribute A implies or strongly concentrates attribute B (e.g. city and zip code), joint occurrences are vastly higher than their independent product, causing the optimizer to severely underestimate cardinality. Creating multivariate extended statistics (such as n-distinct or dependency statistics) provides joint distribution frequencies directly to the planner.

Exercise 3.3.2: Index vs. Sequential Scan Cost Crossover Analysis

Description recall

As visualized in the selectivity-cost crossover curve, describe the physical hardware reasons why an unclustered index scan becomes more expensive than a sequential heap scan once selectivity exceeds approximately 2% to 5%.

Compare sequential read throughput with random page lookup overhead.

Sequential table scans read contiguous disk blocks with sequential read-ahead buffering and predictable page cache replacement. An unclustered index scan must dereference heap tuple IDs randomly across table pages. Once selectivity exceeds a few percent, the number of random heap page fetches approaches or exceeds the total pages in the relation, making the sequential scan faster.

Exercise 3.3.3: Dynamic Programming Join Order Cost Calculation

Construction application

Given relations A (100 rows), B (1,000 rows), and C (10 rows), with $|A \bowtie B| = 50$, $|B \bowtie C| = 100$, and cost formula $\text{Cost}(L \bowtie R) = |L| + |R|$. Calculate and compare the total join cost of plan $(A \bowtie B) \bowtie C$ versus $(B \bowtie C) \bowtie A$.

Sum the cost of the first join and the subsequent join on the intermediate result.

1. Plan 1: First join $A \bowtie B$ has cost $100 + 1{,}000 = 1{,}100$, emitting 50 rows. Second join $(A \bowtie B) \bowtie C$ has cost $50 + 10 = 60$. Total cost = $1{,}100 + 60 = 1{,}160$.
2. Plan 2: First join $B \bowtie C$ has cost $1{,}000 + 10 = 1{,}010$, emitting 100 rows. Second join $(B \bowtie C) \bowtie A$ has cost $100 + 100 = 200$. Total cost = $1{,}010 + 200 = 1{,}210$. Plan 1 is cheaper by 50 units because its first join generates a smaller intermediate cardinality.

Exercise 3.3.4: Multivariate Statistics and Skewed Tenant Distribution Transfer

Generalisation transfer

A multi-tenant database contains 10,000,000 orders where tenant 42 owns 80% of all orders and 90% of all failed orders. Standard planner histograms treat `tenant_id` and `status` as uniformly distributed and independent. Predict the planner's access path choice under naive statistics versus observed execution, and propose the architectural resolution.

Contrast the cost of nested loops using index probes against hash joins under massive skew.

Under naive independence, the planner predicts $0.01 \times 0.01 \times 10{,}000{,}000 = 1{,}000$ rows, selecting a nested loop join with index lookup. In reality, tenant 42 generates over 7,000,000 rows, turning the index probe into an catastrophic loop of millions of random disk reads. The resolution is to collect multivariate tenant-frequency statistics (`CREATE STATISTICS`) or partition the relation by tenant so large tenants receive dedicated, accurate local statistics.

References & Further Reading

- Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). *Access Path Selection in a Relational Database Management System*. ACM SIGMOD, 23–34. (System R dynamic-programming join enumeration.)
- Graefe, G. (1995). *The Cascades Framework for Query Optimization*. IEEE Data Engineering Bulletin, 18(3), 19–29. (Rule-driven top-down search and memo.)
- PostgreSQL: Using EXPLAIN

- PostgreSQL: Planner Statistics
- CMU 15–445/645 Database Systems

Transactions and recoverable execution

SECTION DETAIL

Transactions and Isolation

Two buyers press “reserve” for the last seat. The application cannot treat two individually successful reads as proof that both reservations fit. It needs one unit of work whose committed outcome preserves `available_seats >= 0`, and it must say what a crash, competing transaction, or retry may observe.

ACID separates obligations that are often conflated

DEFINITION

A transaction is a unit with an all-or-nothing commit outcome. **Isolation** constrains concurrent observations; it does not infer every business rule from a schema.

Claim	Concrete question	What it does not promise
Atomicity	Can a crash expose only the debit, not the credit?	that the completed transfer was authorized
Consistency	Do commits preserve declared and encoded business invariants?	that an unexpressed rule is magically checked
Isolation	Which concurrent writes/read effects can this transaction see?	an absence of aborts or retries
Durability	After acknowledgement, which failures retain the commit?	survival of every regional or adversarial event

For a reservation, a row-local invariant can often be enforced by one conditional statement:

```
UPDATE seat_inventory
SET available = available - :requested
WHERE event_id = :event_id
  AND available >= :requested;
```

An affected-row count of zero means the predicate was false *at the protected update boundary*; it is not a successful reservation. This is stronger than `SELECT available`, computing in the application, then issuing an unconditional update.

Prediction — retry scope

A transaction inserts an order, reserves inventory, commits, then calls a payment provider. The network times out after the call. Should the client blindly repeat the whole sequence?

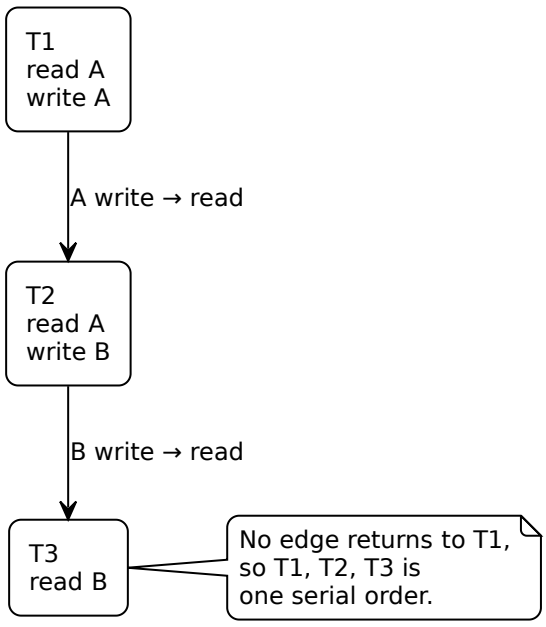
Separate a database rollback-safe action from an externally visible action that may already have occurred.

No. Retrying the database transaction before knowing the previous commit can duplicate inventory or orders; retrying the provider call can double-charge. Commit an outbox or an idempotency key with the database work, then make the provider operation retryable against that stable key. Isolation protects database histories; it does not make an external API atomic with them.

Histories can be tested with a precedence graph

THEOREM 4.1.1 **Conflict-serializability test**

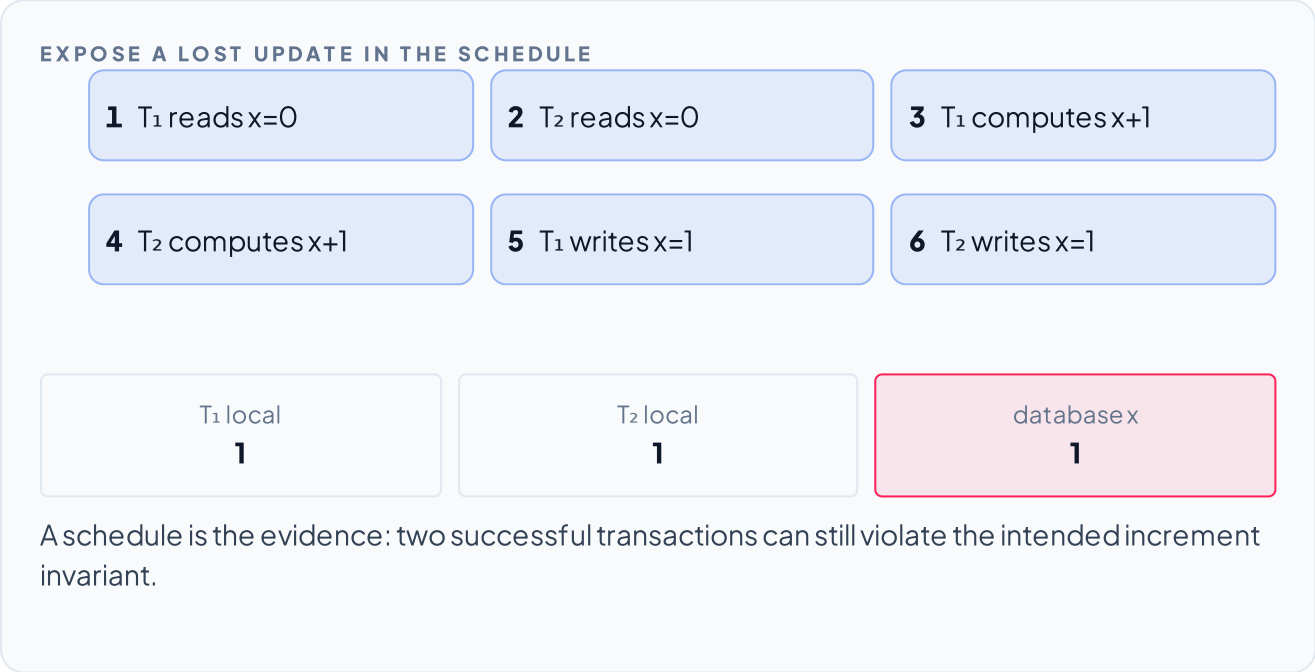
Build a graph with one node per transaction. Add $T_i \to T_j$ when an operation of T_i precedes and conflicts with an operation of T_j on the same item; a conflict has the same item and at least one write. The history is conflict-serializable exactly when this graph is acyclic.



Reads of the same item do not conflict. A write followed by a read, a read followed by a write, or two ordered writes does. An acyclic graph yields a topological order that preserves every conflict order. A cycle is stronger evidence than a suspicious final balance: it proves no serial ordering can retain all observed conflicts.

Worked trace — a lost update

Start with `available = 20`. T1 reads 20 and plans to subtract 7. T2 reads 20 and plans to subtract 7. T1 writes 13; T2 then writes 13. The intended serial result is 6, but the final result is 13: one subtraction disappeared. The writes create a cycle when each transaction's read/write dependencies are represented, so a serializable protocol must block, abort, or otherwise reject one path.

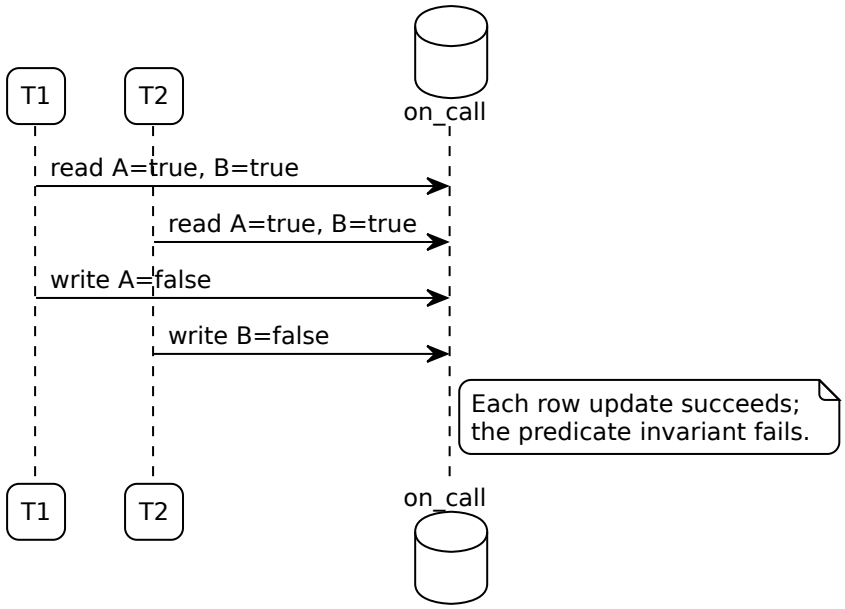


A schedule is the evidence: two successful transactions can still violate the intended increment invariant.

Use the schedule explorer to commit to the observed value before revealing its explanation. It models a read-modify-write lost update; it is not evidence that every database's named isolation levels behave identically.

Snapshot visibility has a known boundary

> ***Counterexample.*** *"At least one doctor remains on call" can fail even when each transaction reads a consistent snapshot. T1 reads doctors A and B and turns A off; T2 reads the same snapshot and turns B off. The writes are disjoint, both commit, and no doctor remains.*



Snapshot isolation often prevents dirty reads and many read anomalies while allowing readers and writers to proceed concurrently. It does **not** automatically serialize predicates over several rows. A serializable level may detect a dangerous dependency and abort one transaction; another repair is to lock/update a summary row that represents the invariant, or to encode a viable declarative constraint where possible.

Requirement	Reliable repair shape	Why a weaker shortcut fails
no duplicate email	unique constraint	`SELECT` then `INSERT` races
nonnegative stock in one row	conditional `UPDATE ... WHERE stock >= n`	a prior read becomes stale
at least one on-call doctor	serializable validation or protected summary/predicate	row-local updates are disjoint
price + order snapshot agree	same transaction and foreign-key/version rules	application-side sequencing can expose a prefix

Application — pick the conflict boundary

A tenant has a monthly credit limit across many invoices. Two requests each read remaining credit and add a different invoice. Name a repair and state the price it pays.

Maintain a tenant-month summary row and update it conditionally in the same transaction, or use serializable isolation with a retry on serialization failure. The summary row makes the aggregate invariant a shared write boundary but can become hot; serializable validation preserves flexibility but can abort more often under contention. A normal row lock on each new invoice does not protect the aggregate predicate.

Isolation labels require an implementation contract

WARNING

“Read committed,” “repeatable read,” and “serializable” are names, not a portable proof. Check the target DBMS’s documented semantics, error behavior, and replica/DDL caveats.

PostgreSQL, for example, documents Read Committed, Repeatable Read, and Serializable behavior, treats Read Uncommitted as Read Committed, and may report a serialization failure that the application must retry. That is an implementation example, not a license to generalize its exact snapshot boundary to another engine.

Transfer — design a test rather than trust the label

You are told a service uses “repeatable read.” Give the smallest experiment that reveals whether it protects the on-call invariant.

Run the two disjoint writes concurrently and retain outcome evidence, not just the final rows.

Create two on-call rows, begin two transactions at the claimed level, have both read the predicate, have each disable a different doctor, then commit. Record blocking, errors, commits, and the final predicate. If both commit with no doctor on call, the level does not protect this invariant; add a serializable/summary/predicate-based repair and rerun the same schedule.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.1.1: Conflict-Serializability Precedence Graph Construction

Verification application

Consider history $H = r_1(A)\ r_2(B)\ w_1(B)\ w_2(A)\ c_1\ c_2$. Using Theorem 4.1.1, construct the serialization graph and verify whether H is conflict-serializable.

Identify conflicting operation pairs on the same data item and draw directed edges.

The conflicting pairs are:

1. $r_2(B)$ precedes $w_1(B)$ on item B, inducing directed edge $T_2 \to T_1$.
2. $r_1(A)$ precedes $w_2(A)$ on item A, inducing directed edge $T_1 \to T_2$. The serialization graph contains the cycle $T_1 \to T_2 \to T_1$. By Theorem 4.1.1, because the graph contains a directed cycle, the history H is not conflict-serializable.

Exercise 4.1.2: Write Skew Anomaly under Snapshot Isolation

Description recall

Explain why snapshot isolation permits write skew in the classic hospital on-call problem (where at least one doctor must remain active) while preventing dirty reads, non-repeatable reads, and lost updates.

Each transaction reads the shared active count but modifies disjoint doctor rows.

Under snapshot isolation, transactions read from a consistent snapshot taken at start and check for write conflicts only on the specific rows they update (first-committer-wins). In write skew, transaction T_1 updates doctor A while T_2 updates doctor B. Because their write sets $\{A\}$ and $\{B\}$ are disjoint, no write-write conflict is detected, and both commit. However, their commit decisions depended on each other's state, violating the multi-row invariant.

Exercise 4.1.3: Multi-Invoice Aggregate Credit Limit Protection

Construction application

A business requires that the total of all unpaid invoices for a tenant cannot exceed credit limit L. Two concurrent transactions attempt to insert new invoices for tenant 12. Specify two alternative architectural mechanisms to prevent exceeding L under concurrent execution.

Consider explicit lock serialization on a parent row versus serializable snapshot isolation.

1. Explicit parent row lock: Each transaction begins by executing `SELECT * FROM Tenant WHERE tenant_id = 12 FOR UPDATE;`. This serializes all invoice additions for tenant 12, forcing each transaction to read the committed sum of prior transactions before inserting.
2. Serializable isolation: Run both transactions at `SERIALIZABLE` isolation. The database tracks predicate read dependencies and aborts one transaction with a serialization error (`40001`), requiring the application to retry.

Exercise 4.1.4: Empirical Isolation Level Validation Harness

Generalisation transfer

Design an automated test harness to empirically verify whether a database's declared `REPEATABLE READ` isolation level is vulnerable to write skew. Define the initial database fixture, the interleaved operations of two concurrent sessions, and the assertion that detects invariant violation.

Seed two doctors on call, have two concurrent threads each take one doctor off call, then assert final count.

1. Fixture: Insert `Doctor(id, on_call)` with rows `(1, true)` and `(2, true)`. Invariant: `COUNT(*) FILTER (WHERE on_call) >= 1`.
2. Schedule:
   - Session 1: `BEGIN ISOLATION LEVEL REPEATABLE READ;` reads active count (2).
   - Session 2: `BEGIN ISOLATION LEVEL REPEATABLE READ;` reads active count (2).
   - Session 1: `UPDATE Doctor SET on_call = false WHERE id = 1; COMMIT;`
   - Session 2: `UPDATE Doctor SET on_call = false WHERE id = 2; COMMIT;`
3. Assertion: Query `SELECT COUNT(*) FROM Doctor WHERE on_call = true;`. If count is 0, both committed and write skew occurred, proving the level is snapshot isolation rather than true serializability.

References & Further Reading

- PostgreSQL: Transaction Isolation
- PostgreSQL: Explicit Locking
- CMU 15–445/645 Database Systems

SECTION DETAIL

Concurrency Control

Two customers buy the last unit at the same instant. Each transaction reads `available = 1`, computes `1 - 1 = 0`, and writes `0`. Both commits succeed, the stock never becomes negative, and two units are sold from one. No individual statement is wrong; the interleaving is.

Step	T1	T2	Last written `available` (not necessarily committed)
1	read 1		1
2		read 1	1
3	write 0		0
4		write 0	0
5	commit	commit	0

The lost update is the motivating problem. Isolation names an allowed history; concurrency control is the mechanism that admits or rejects individual operations on the path to that history. A stock row can be protected with a lock, a version check, a timestamp rule, or serializable dependency tracking. Those mechanisms fail differently, so a system design must choose an error it can handle rather than merely demand "no races."

This schedule also shows why the repair must match the invariant. A conditional update `SET available = available - 1 WHERE available >= 1` prevents the negative count without an explicit read, while a plain read-then-write needs a lock or version check that spans both steps. The protocol is chosen from the invariant, not from a preference for one lock mode.

WHY TWO COMMITS ARE NOT ENOUGH

Both transactions in the lost-update schedule commit successfully without error. Evaluate which assertions regarding this concurrent execution are correct:

Answer: The concurrent execution sold two units from one in stock, producing a history that is not equivalent to any serial ordering.; Local transaction commit success under weak isolation does not guarantee preservation of global business invariants.

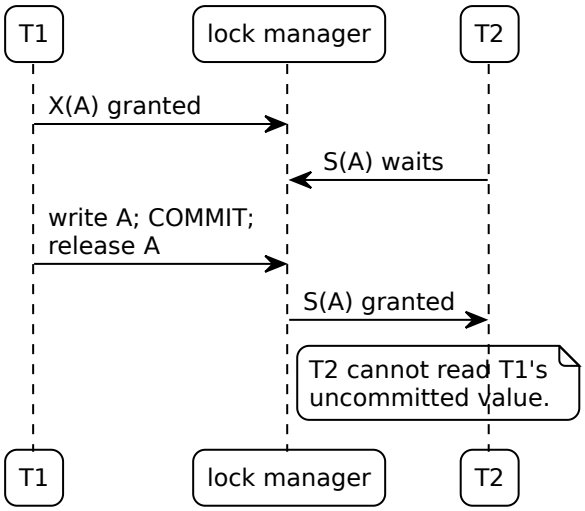
Each transaction's local operations were valid in isolation, but their interleaving produced a non-serializable history. Concurrency control must serialize operations, reject conflicting updates, or require atomic conditional expressions.

Strict two-phase locking blocks unsafe orderings

DEFINITION

In two-phase locking, a transaction acquires locks during a growing phase and releases them only during a shrinking phase. A **strict** variant keeps write locks until commit or abort, preventing other transactions from reading or overwriting uncommitted writes.

Requested lock	Existing shared	Existing exclusive
shared	compatible	wait
exclusive	wait	wait



Strictness is a recovery aid as well as a serializability tool: no committed transaction has observed a value that must later be undone. It can still reduce concurrency when a long transaction holds a hot row. A carefully bounded critical transaction is often more important than choosing an exotic lock mode.

Application — lock the fact, not the screen

A reservation flow reads inventory, calls a remote payment service for 10 seconds, then decrements inventory. Should it hold an exclusive database lock during the call?

A database lock protects a local invariant; remote latency makes the protected interval unbounded.

No. Holding the lock causes a slow external system to serialize unrelated reservations and can create deadlocks. First record an idempotent payment intent or outbox state, perform the remote call outside the inventory-critical transaction, then execute a short conditional reservation transaction. The repair is an explicit state machine, not a longer lock.

For one inventory row, the critical statement can be:

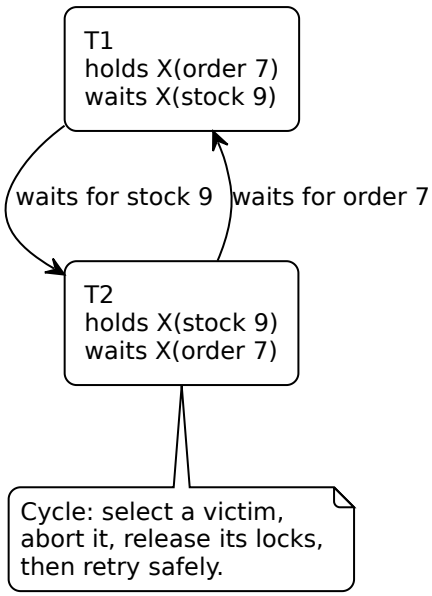
```
UPDATE inventory
SET available = available - :quantity
WHERE sku = :sku AND available >= :quantity;
```

The caller must inspect the affected-row count. `1` commits the reservation; `0` means the attempt lost the race or the request is no longer satisfiable and must not be treated as success. A serialization/deadlock error may be retried with the same idempotency key, bounded attempts, and backoff; a committed success must be returned from the idempotency record rather than applied twice. The predicate is necessary but not, by itself, evidence that an aggregate invariant spanning several stock rows is protected.

Deadlock is a graph, not “a slow query”

INVARIANT

A waits-for graph has an edge $T_i \to T_j$ when T_i waits for a lock held by T_j. A directed cycle means no member of that cycle can proceed without an abort or external intervention.



Detection periodically finds cycles and aborts a victim. Prevention imposes an ordering, such as always locking inventory before orders. Timeouts are simpler but can abort work that would have completed and may miss an operational root cause. Victim selection weighs age, work done, locks held, and rollback cost; it must never assume the transaction's external side effects can be undone by database rollback.

Prediction — break the cycle

T1 holds **warehouse/5** and waits for **sku/9**; T2 holds **sku/9** and waits for **warehouse/5**. Give one prevention and one detection response.

Prevention: define and enforce a global lock order, for example **(warehouse_id, sku)** sorted lexicographically, so both transactions request warehouse 5 before sku 9. Detection: record T1 → T2 and T2 → T1, abort a chosen victim, roll back its database work, then retry only if its operation uses an idempotency key or has no exposed external effect. Retrying without that boundary can duplicate messages or charges.

Protocols move the conflict check to different times

> ***Worked example.*** *A flash-sale counter has frequent writes to one row; an analytics dashboard reads millions of rows and rarely writes. One protocol need not optimize both workloads.*

Protocol	When it rejects/confirms conflict	Strength	Cost / failure mode
strict 2PL	lock request	clear exclusion and recoverable writes	waits and deadlocks
timestamp ordering	operation violates timestamp order	no deadlocks	restarts under hot contention
optimistic control	commit-time read/write-set validation	readers run freely when conflicts are rare	expensive wasted work on conflict
MVCC	version visibility plus write/serializability validation	readers often avoid writer blocking	old-version retention; write skew unless stronger validation

Timestamp ordering and optimistic control do not make conflicts disappear: they shift them into aborts. MVCC makes a snapshot readable, not necessarily serializable. A high-contention counter may prefer a single conditional update or a sharded/aggregated representation over a broad transaction that repeatedly aborts.

Worked example — snapshot isolation permits write skew. Two doctors are on call, and the invariant is "at least one doctor remains on call." T1 reads the on-call set $\{\text{Alice}, \text{Bob}\}$ and removes Alice; T2 concurrently reads the same set and removes Bob. Under snapshot isolation each transaction reads a consistent snapshot and writes a different row, so neither write conflicts with the other. Both commit, and the invariant is false. The conflict is on the predicate "the on-call set has size at least two," not on a single row. Repairs include serializable isolation with predicate or summary-row protection, a constraint enforced by a coordinated operation, or a design that makes the aggregate a versioned object. Retrying one transaction after commit is insufficient because neither transaction observed a write conflict.

REMARK **Why unconditional retry on serialization failure is unsafe**

The database transaction can be retried only if its inputs are still valid and it has not already produced an irreversible effect. Put an idempotency key in the committed database state, defer messages through an outbox, and make the consumer deduplicate. A retry loop without a limit, backoff, or idempotency boundary turns contention into duplication.

Granularity must match the invariant

WARNING

A row lock is not a predicate lock. Locking the rows already returned by an on-call predicate may not prevent another transaction from inserting or changing a row that changes the predicate's truth.

Invariant	Suitable shared boundary	Common false repair
one seat never sold twice	one inventory/seat row plus conditional update	read value in application, then unconditional write
no duplicate username	unique constraint	rely on an application mutex
at least one doctor on call	serializable predicate protection or summary row	lock only the doctor being changed
total monthly credit ≤ limit	tenant-month summary row or serializable validation	lock each newly inserted invoice

Transfer — predict contention before implementing

A service records one million page views per minute and must display an approximate dashboard every minute; it also records money transfers requiring exact balances. Which path may tolerate commutative or delayed aggregation, and which cannot?

Separate the semantic requirement from the shared data shape.

Page-view counters can be sharded, buffered, or eventually aggregated when a bounded delay/error is acceptable. Money transfers require an exact invariant at commit and need a transaction/constraint boundary that prevents overdraft. Applying exact locking to every page view wastes concurrency; applying approximate aggregation to balances changes the financial contract.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.2.1: Strict 2PL Deadlock Precedence Cycle Detection

Verification application

Consider two transactions under strict two-phase locking: T_1 holds an exclusive lock on row A and requests an exclusive lock on row B; T_2 holds an exclusive lock on row B and requests an exclusive lock on row A. Construct the wait-for graph, verify why this state is a deadlock, and describe how the engine resolves it.

Draw directed edges from the waiting transaction to the lock-holding transaction.

The wait-for graph contains directed edges: $T_1 \rightarrow T_2$ (since T_1 waits for B held by T_2) and $T_2 \rightarrow T_1$ (since T_2 waits for A held by T_1). The resulting cycle $T_1 \rightarrow T_2 \rightarrow T_1$ represents a circular wait where neither transaction can progress. The engine's deadlock detector identifies the cycle and aborts one transaction (the victim), rolling back its operations and releasing its locks so the other can proceed.

Exercise 4.2.2: Row-Level Lock Inadequacy for Predicate Constraints

Description recall

Explain why executing `SELECT * FROM Doctor WHERE on_call = true FOR UPDATE;` in PostgreSQL fails to protect against phantom inserts that violate predicate-level business invariants.

Row locks attach only to currently existing tuples returned by the query.

`SELECT ... FOR UPDATE` acquires row-level locks strictly on existing physical tuples matching the predicate at query execution time. It cannot lock nonexistent rows. A concurrent transaction can insert a brand new `Doctor` record that satisfies the predicate, committing without waiting on existing row locks. Preventing phantom violations requires true predicate locks (SSI), table-level locking, or explicit locks on a parent anchor row.

Exercise 4.2.3: Atomic Conditional Write for Inventory Reservation

Construction application

Write an atomic SQL update statement that decrements inventory item 42 by 1 while strictly guaranteeing that available stock never drops below zero, without requiring an explicit application-side lock or prior select.

Include the non-negative invariant directly in the WHERE clause and inspect row count.

```sql
UPDATE inventory
SET available = available - 1
WHERE item_id = 42
  AND available >= 1;
```

If the statement affects 1 row, the reservation succeeded atomically within single-statement execution. If it affects 0 rows, stock was exhausted, and the application safely rejects the purchase without race conditions.

Exercise 4.2.4: Workload-Aware Concurrency Control: Exact vs. Commutative Aggregation

Generalisation transfer

A social platform handles 100,000 post like increments per second and 500 bank balance transfers per second. Contrast how concurrency control should be architected for the like counter versus the bank balance.

Determine whether operation order matters and whether intermediate states violate invariant constraints.

Like increments are commutative ($+1$ can be applied in any order) and tolerate eventual consistency: they should be sharded into N counter slots per post or buffered in an in-memory queue and flushed in batches, avoiding single-row lock serialization. In contrast, bank balance transfers are non-commutative with zero error tolerance: an account must never overdraw, requiring strict serializable isolation or row-level `FOR UPDATE` serialization on account balances at commit time.

References & Further Reading

- PostgreSQL: Explicit Locking
- PostgreSQL: Transaction Isolation
- CMU 15-445/645 Database Systems

SECTION DETAIL

Transaction Trace Lab

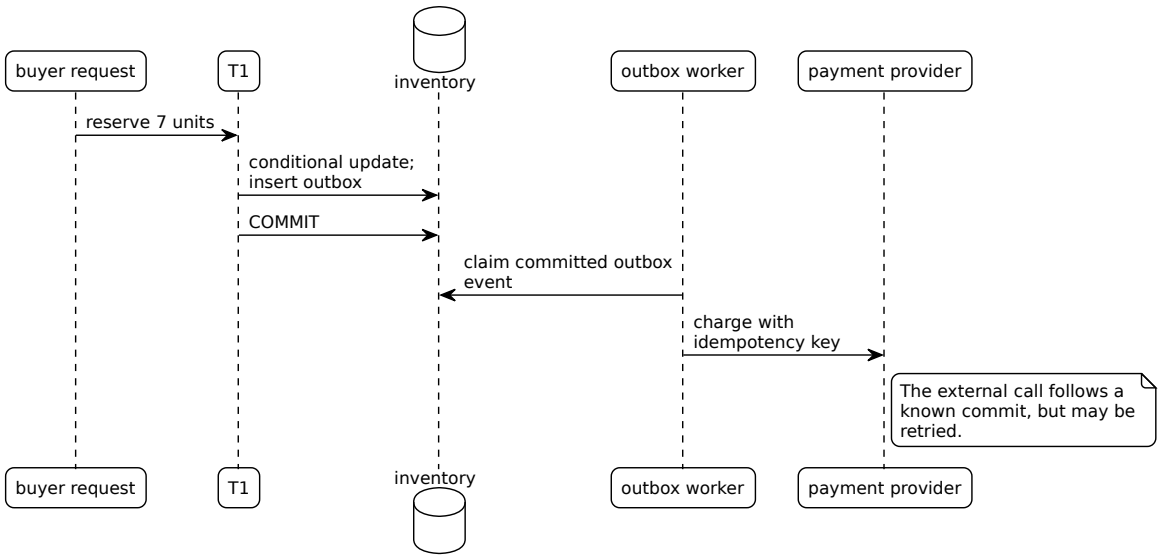
A trace is a proof artifact. A final table can look plausible after a lost update, a write skew, a deadlock victim, or a crash replay; the trace retains the operations, visibility rules, and failure boundary that distinguish them. Use the notation below before proposing a lock level or an ORM setting.

Record enough state to replay the decision

DEFINITION

A useful transaction trace records each transaction's ordered reads and writes, lock/version state, commit or abort outcome, durable-log boundary, and any effect sent outside the database.

Notation	Meaning	Question it answers
r1(x=20)	T1 read 20 from x	which version did T1 base its decision on?
w2(x=13)	T2 wrote 13	which operations conflict?
X1(x)	T1 holds an exclusive lock on x	why is T2 waiting or allowed through?
C1 / A1	T1 commits / aborts	may its writes become visible?
L=911 durable	log through LSN 911 is stable	what can recovery rely on after a crash?



The first question is always semantic: what must remain true? The second is operational: at what read, write, commit, or recovery point is that truth enforced? A trace missing either cannot diagnose the system.

Prediction — insufficient evidence

A support log says “stock became 13 after two requests for 7.” It contains only final row values. Name two distinct explanations that remain possible.

Final values discard order and visibility.

A lost update can produce 13 from initial stock 20 when both transactions read 20 and overwrite each other. A correctly serialized system could also produce 13 after one request succeeds and a compensating restock changes the value. The missing data are the read versions, statement outcomes, commit order, and any intervening events; final state alone is not a concurrency proof.

Trace a lost update without smuggling in oversell

EXAMPLE

Start with **stock = 20**. Two buyers each request 7. The intended serial outcome after two accepted requests is 6; using 20 keeps lost update distinct from a separate insufficient-stock rule.

Step	T1	T2	Stored stock	Consequence
1	read 20		20	T1 plans 13
2		read 20	20	T2 also plans 13
3	write 13; commit		13	one decrement recorded
4		write 13; commit	13	second decrement lost

A weak implementation reads stock in the application and executes an unconditional write. A row-local repair puts the condition and decrement in one SQL statement: **update stock only when stock is at least the requested amount**. The affected-row count distinguishes reservation from rejection. A lock or serializable protocol can also serialize the read-modify-write, but the conditional statement expresses the invariant directly.

Worked trace — complete the repair trace

Assume T1 performs the conditional decrement first and commits. T2 performs the same conditional decrement after T1. Fill in T2's affected-row outcome and final stock.

T1 changes 20 to 13. T2 sees 13, and its condition **stock >= 7** remains true, so it changes 13 to 6 and affects one row. If the initial stock were 10, T1 would leave 3 and T2 would affect zero rows. Conflating the 10–unit case with a lost update hides whether the design lost a write or correctly rejected an oversell.

EVIDENCE BEYOND THE FINAL VALUE

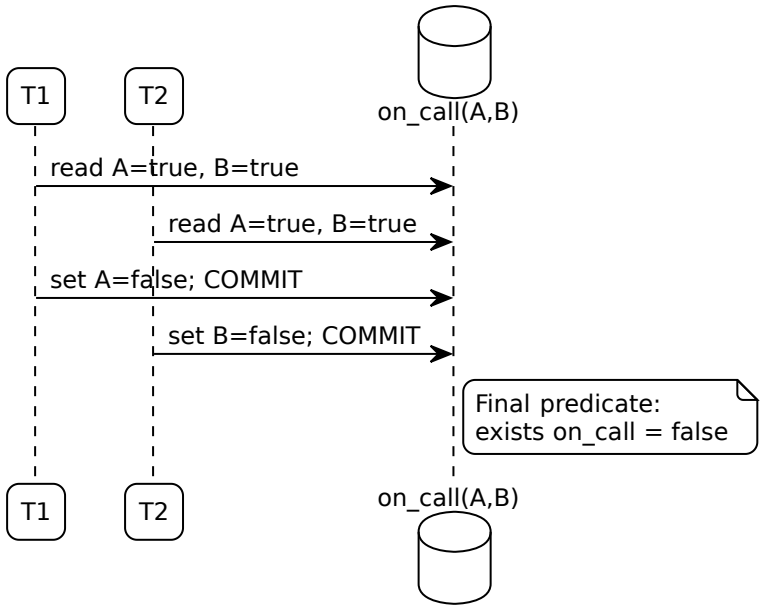
A support log shows stock = 13 after two requests for 7 units, starting from initial stock = 20. Evaluate which evidence sources reliably diagnose whether an anomaly occurred:

Answer: Read versions, statement row-count outcomes, and commit timestamps distinguish a lost update from a serialized retry.; Audit logs showing compensating supplier restocks explain why a serialized execution could legitimately terminate at stock 13.

Final database state discards order, visibility, and side effects. Diagnostic proof requires an execution trace showing statements, row counts, commit timestamps, and external compensations.

Trace write skew as a predicate failure

> ***Counterexample.*** *The invariant "at least one doctor remains on call" spans a set of rows. T1 changes doctor A; T2 changes doctor B. No row is written twice, yet both can jointly violate the predicate.*



A row-level optimistic version check can accept both writes because their write sets are disjoint. Repairs include serializable validation, predicate/range protection, or a single summary row updated as part of the rule. The trade-off is expected aborts or contention at the shared boundary.

Prediction — identify the false repair

Which repair is insufficient by itself: lock A only, lock both currently on-call rows, serializable transaction, or update a protected on-call-count row?

Locking A only is insufficient because T2 changes B. Locking both rows can work only if every transaction that could change membership follows the same discipline and range changes are also controlled; it is brittle. Serializable validation or a protected count/summary row gives a shared conflict boundary. The key gap is the predicate, not the absence of a row lock.

Trace waiting, deadlock, and a chosen abort

DEFINITION

A waits-for edge T1 → T2 means T1 waits for a lock held by T2. A cycle is a deadlock; deadlock detection breaks it by aborting at least one transaction.

Step	T1	T2	Waits-for graph
1	holds X(order 7)		none
2		holds X(stock 9)	none
3	waits for X(stock 9)		T1 → T2
4		waits for X(order 7)	T1 ↔ T2

REMARK **Victim selection is not a correctness proof**

Aborting T2 releases stock 9 and lets T1 proceed, but the policy should consider rollback work and age. More importantly, the aborted transaction may need a retry protocol that avoids repeating a notification or payment that already escaped the database boundary.

Transfer — order, crash, and webhook retry

Design the event order for: reserve stock, write an order, charge a provider, and send a confirmation. A crash can happen after any step; the provider can send the same webhook twice and out of order.

Make the database state transition and the retry key durable before depending on external delivery.

In one transaction, insert the order in a pending state, reserve stock conditionally, store a unique provider-event/idempotency key, and insert an outbox record; then commit. A worker invokes the provider with that key and records only legal monotonic payment transitions in later transactions. Deduplicate webhook IDs with a unique constraint, and send confirmation from a committed outbox. Recovery can replay unclaimed outbox records; it must not infer a payment from a network timeout.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.3.1: Differentiating Lost Updates from Serial Compensations

Description recall

Given a log indicating an inventory record transitions from 20 to 13 after two customer purchase requests for 7 units, describe what specific timeline artifacts are required to definitively prove whether a lost update occurred.

Inspect read timestamps, statement outcomes, and external audit logs.

To prove whether a lost update occurred, the timeline must establish: (1) whether transaction T_2 read the initial value 20 (lost update) or the committed value 13; (2) whether T_2's decrement statement actually affected a row or was rejected by a conditional check (`rows affected = 0`); and (3) whether an external supplier restock event (+7 units) occurred between two serialized sales. Scalar end-state alone is ambiguous.

Exercise 4.3.2: Write Skew Interleaving Trace Verification

Verification application

Consider transactions T_1 and T_2 operating under snapshot isolation on rows $A = \text{true}$ and $B = \text{true}$ with invariant $A \vee B$. Verify why schedule $r1(A, B)\ r2(A, B)\ w1(A = \text{false})\ w2(B = \text{false})\ c1\ c2$ violates the invariant despite zero lock or write-write conflicts.

Trace the disjoint write sets against the joint invariant predicate.

Under snapshot isolation, T_1 reads snapshot $\{A = \text{true}, B = \text{true}\}$ and decides it is safe to set $A = \text{false}$ because B is true. Concurrently, T_2 reads the identical snapshot and sets $B = \text{false}$ because A is true. Because T_1 modifies item A and T_2 modifies item B, their write sets $\{A\}$ and $\{B\}$ are completely disjoint. Both commit without error, leaving $A = \text{false}$ and $B = \text{false}$, directly violating $A \vee B$.

Exercise 4.3.3: Deadlock Wait-For Graph and Victim Selection Policy

Construction application

Construct a 4-step execution trace involving transactions T_1 and T_2 and resources `order_7` and `stock_9` that produces a circular wait under strict 2PL. Specify a heuristic for victim selection that minimizes aborted work.

Have each transaction acquire one resource and subsequently request the resource held by the other.

Trace:

1. T_1 acquires exclusive lock $X(\text{order_7})$.
2. T_2 acquires exclusive lock $X(\text{stock_9})$.
3. T_1 requests $X(\text{stock_9})$ and blocks ($T_1 \to T_2$).
4. T_2 requests $X(\text{order_7})$ and blocks ($T_2 \to T_1$). The wait-for graph has cycle $T_1 \leftrightarrow T_2$. A sound victim selection policy aborts the younger transaction or the transaction that has generated fewer write-ahead log bytes, minimizing rollback work and lock churn.

Exercise 4.3.4: Crash-Safe Payment and Webhook Event Choreography

Generalisation transfer

Design an idempotent, crash-safe transaction sequence that coordinates reserving local inventory, writing an order, calling an external payment gateway, and handling duplicate out-of-order webhook notifications.

Commit local state and outbox record before external network calls; deduplicate incoming webhooks.

1. Local transaction: In a single database transaction, insert `orders(id, status='pending')`, execute conditional stock reservation `UPDATE inventory SET stock = stock - 1 WHERE stock >= 1`, generate an idempotency key `k`, and insert an `outbox` task record with `k`. Commit.
2. Gateway invocation: A worker reads the outbox task and dispatches the payment charge with key `k`.
3. Webhook handling: Incoming payment webhooks carry key `k`. The endpoint attempts `INSERT INTO processed_events (event_id, idempotency_key) VALUES (...)`. If a unique-key collision occurs, it immediately returns HTTP 200 without reprocessing. Otherwise, it updates `orders` status to `paid` within an atomic transaction.

References & Further Reading

- PostgreSQL: Transaction Isolation
- PostgreSQL: Write-Ahead Logging
- CMU 15–445/645 Database Systems

SECTION DETAIL

Recovery and Logging

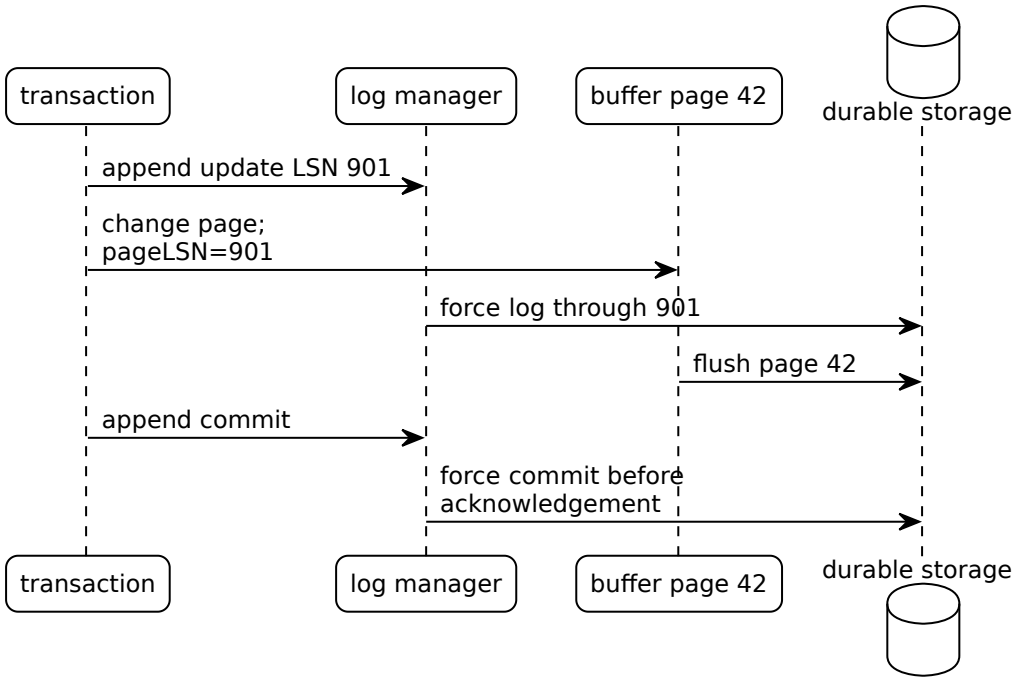
A commit response is a promise about a failure boundary. If a power loss follows an acknowledged order but precedes the page write, the order must still be recoverable. If a data page reaches storage before the log record explaining it, recovery loses its evidence. Logging is therefore a protocol for reconstructing a legal state, not an audit trail added after the fact.

A durable log prefix orders safe writes

INVARIANT

Before a changed data page reaches durable storage, the log records describing that page change must be durable. Before a client receives a durable-commit acknowledgement, the commit record must be durable under the stated configuration.

LSN	Transaction	Record	Durable before crash?
900	T1	begin	yes
901	T1	set stock 20 → 13	yes
902	T1	commit	yes
903	T2	begin	yes
904	T2	set stock 13 → 6	yes
905	T2	commit	no



The stable log prefix ending at 904 supports T1's commit. A crash before 905 becomes durable cannot honestly expose T2 as committed. A data page may contain an update before its transaction commits; recovery's transaction status decides whether to preserve, undo, or hide it. Page LSNs make redo idempotent: if a page already reflects LSN 901 or later, reapplying 901 is unnecessary.

Prediction — a page beats its log

A page reflecting LSN 911 reaches storage, but the log is durable only through 910. Why is this STEAL-without-WAL failure mode an error even if the update happens to be correct?

Recovery needs a durable explanation for every durable page effect.

The durable page contains an effect whose redo/undo record may be absent. After a crash, recovery cannot determine whether the change belonged to a committed transaction, reconstruct an overwritten older value, or repeat the operation safely on another page. WAL forbids this ordering; correctness is not established by observing one lucky final value.

Redo repeats history; undo removes losers

DEFINITION

A **winner** has a durable commit record at crash time. A **loser** began but lacks a durable commit. Recovery repeats durable changes that are missing from pages and removes or hides loser effects according to its log and visibility design.

CRASH BETWEEN LOG RECORDS; DERIVE REDO AND UNDO

POSSIBLE PAGE VALUES AT CRASH

`A ∈ {10, 7}`
`B ∈ {20, 30}`

A durable update log permits, but does not prove, its page write.

WINNER / LOSER

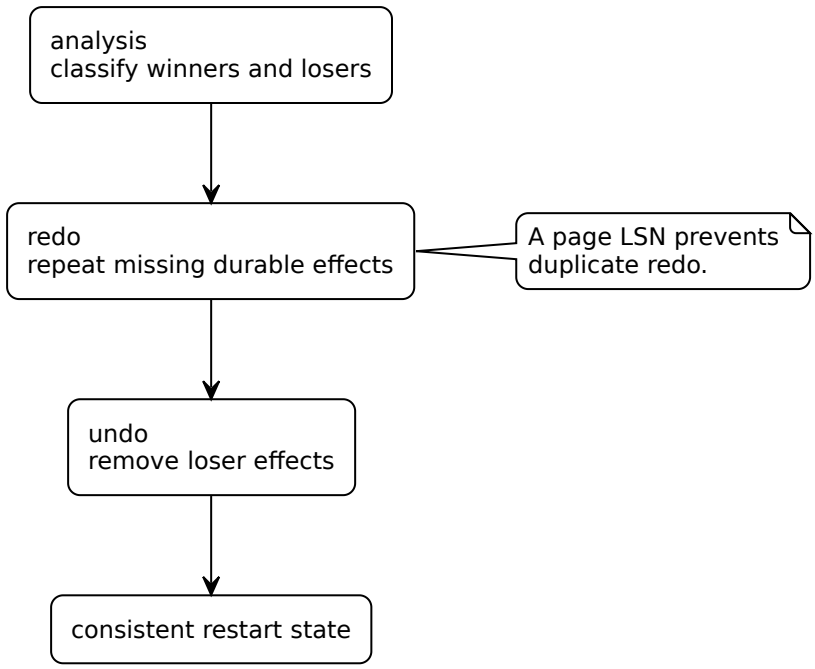
T1: winner — ensure A = 7
T2: loser — roll back B if present

AFTER RECOVERY

`A = 7`
`B = 20`

Static state: T1's commit record is durable, so recovery ensures its update is reflected if necessary. T2 has no commit record, so recovery rolls its update back if it reached a page.

Advance the durable prefix in the playground and state the committed state before revealing it. The model intentionally exposes one committed transaction and one unfinished transaction; it is a small reasoning model, not a complete implementation of checkpoints or compensation log records.



Worked trace — crash after page flush

T1 writes stock 20 → 13 at LSN 901 and commits at 902. The page flushed at LSN 901 before the crash. T2's update at 904 may be present in the buffer, but its data-page effect has not reached durable storage; T2 has no durable commit. Recovery sees page LSN 901, skips redundant redo of 901, and starts from a state that includes T1 but not T2. If T2's update had reached the page, recovery would undo or make it invisible because T2 is a loser.

> REMARK **Duality of redo and undo passes**
>
> A committed log record can survive while its changed page does not. Redo brings persistent pages up to the durable committed history. Undo handles the opposite case: a page can contain a loser update that reached storage before the crash. Neither pass substitutes for the other.

ARIES makes undo repeatable

The three-phase sketch above becomes a real recovery protocol only when an interrupted undo can itself survive a crash. ARIES adds **compensation log records** (CLRs) and a per-transaction **undo-next LSN** so that recovery never repeats an undo it already performed.

DEFINITION

A compensation log record (CLR) describes an undo action and carries an `undoNextLSN`, the next earlier record in that transaction's undo chain to examine. That record can be a begin record rather than another update. A CLR is redo-only: it can be replayed during redo but is not itself undone.

Worked trace. T2 begins at LSN 903, updates stock at 904, and crashes without a durable commit. During undo, recovery appends CLR 906 for the reversal with `undoNextLSN = 903` and applies the undo. WAL requires that CLR to be durable before its changed page is flushed. Suppose 906 is durable and recovery crashes again before finishing. On restart, redo can repeat 906 if its effect is missing from the page; undo resumes at 903 instead of undoing 904 again. If the CLR was not durable, recovery follows the durable log instead: WAL prevents the corresponding undo page effect from having been safely flushed first.

Phase	Reads	Produces	Key invariant
Analysis	log from the last checkpoint	transaction table, dirty-page table	every loser is identified before any undo
Redo	log forward from the smallest dirty-page `recLSN` found during analysis	pages brought to their logged history	page LSN skips changes already present
Undo	losers' log records backward	CLRs and restored pre-images	`undoNextLSN` advances exactly once per action

Analysis starts from the checkpoint designated by the master record and reconstructs transaction and dirty-page state. A fuzzy checkpoint records active transactions and dirty pages while work continues; the dirty-page table can contain a `recLSN` older than the checkpoint, so redo can start earlier. The correctness claim depends on WAL and the recovery protocol, not merely checkpoint frequency.

WARNING

A CLR is not an application-level compensating transaction. It repairs physical recovery state. Reversing a shipped order, a sent email, or a payment requires an idempotent business compensation and cannot be inferred from a database undo record.

WHY UNDO NEEDS ITS OWN LOG RECORD

Transaction T3 has updates at LSN 20 and LSN 30. Recovery writes a durable CLR for undoing 30 with undoNextLSN = 20, then crashes. Evaluate which recovery principles apply:

Answer: Upon restart, the undo pass follows undoNextLSN to LSN 20, avoiding redundant re-undo of the modification at LSN 30.; The CLR is a redo-only log record during the redo phase to bring the data page up to its logged post-undo state.

A durable CLR describes an undo action and points backward via undoNextLSN to the next undone record. Redo reapplies the CLR if needed, and undo resumes at undoNextLSN.

Recovery objectives require distinct artifacts

WARNING

A streaming replica improves failover capability; it is not automatically a historical backup. An accidental delete or logical corruption can replicate quickly to every replica.

Artifact	Main question answered	Cannot replace
WAL / transaction log	how to recover changes after a base state	a base copy once retention is exhausted
full or incremental backup	what durable base can be restored	recent changes without logs or later backups
replica	how quickly can service continue after a node loss	an independently retained historical restore point
checksum / invariant test	did restored bytes and domain facts survive?	the data copy itself

A stated recovery point objective (RPO) is acceptable data loss, measured in time or transactions. A recovery time objective (RTO) is acceptable restoration time. They are claims until a restore drill measures them. A backup file existing in object storage is not evidence that it restores, replays logs, starts, and passes business invariants.

Application — design a restore drill

The product claims RPO ≤ 5 minutes and RTO ≤ 60 minutes after a bad migration. State a drill that can disprove either number.

Capture a known workload and timestamps, take the documented base backup, retain the required WAL/archive interval, then introduce the bad migration. Restore into an isolated environment to a target time no more than five minutes before the declared incident boundary. Measure elapsed time through startup and validation, compare row checksums where appropriate, and test domain invariants such as tenant ownership and nonnegative stock. Failure to reach the time target disproves RPO; exceeding the measured recovery budget disproves RTO.

Corruption and accidental change need different responses

EXAMPLE

A checksum failure on one page calls for diagnosis, a known-good recovery source, and evidence that copies are independent. An accidental DELETE with a valid checksum calls for point-in-time recovery or controlled logical repair; checksum success says nothing about intended content.

Transfer — select the first safe action

A replica is healthy but has already replayed an accidental tenant-wide deletion. A nightly backup exists and WAL is retained. What is the first safe recovery plan?

Do not promote the replica merely because it is available; it has the same logical damage.

Preserve the primary, replica, backup, and WAL evidence; restore the backup plus WAL to an isolated target before the deletion; verify the tenant scope and invariants; then choose a controlled logical export/import or a promotion plan with client fencing. The replica can reduce outage time in a node-loss case, but here it is not an independent recovery source.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.4.1: ARIES Compensation Log Record and Idempotent Restart

Verification application

An active transaction T_1 modifies page P at LSN 100. During crash recovery, ARIES undo writes CLR 105 with `undoNextLSN = null`, updating page P to LSN 105. Before recovery completes, a second power outage occurs. Verify how the subsequent recovery restart handles CLR 105 and whether page P is undone again.

Check the repeating-history redo phase and CLR redo-only properties.

On the second restart, the analysis phase identifies CLR 105 in the log. During the repeating-history redo phase, if page P's page LSN on disk is less than 105, CLR 105 is reapplied to restore the page image. Because CLR 105 has `undoNextLSN = null`, the undo phase recognizes that all actions of T_1 are already fully reversed and does not attempt any further undo on page P. The process is strictly idempotent.

Exercise 4.4.2: Redo vs. Undo Duality in Write-Ahead Recovery

Description recall

Explain why write-ahead logging recovery requires both a forward redo pass and a backward undo pass, and why neither pass can substitute for the other.

Contrast unwritten committed pages with written uncommitted pages.

The redo pass is necessary because a transaction may commit while its dirty modified data pages remain in the volatile buffer pool without reaching disk (No-Force policy); redo replays logged updates to bring disk pages up to the committed state. The undo pass is necessary because dirty pages from active or uncommitted transactions may have been flushed to disk to free buffer frames (Steal policy); undo rolls back those uncommitted modifications to preserve transaction atomicity.

Exercise 4.4.3: Falsifiable RPO/RTO Disaster Recovery Drill Design

Construction application

An enterprise SLA specifies $\mathrm{RPO} \leq 5$ minutes and $\mathrm{RTO} \leq 30$ minutes. Specify a rigorous disaster recovery test drill protocol that either experimentally proves compliance or produces a falsifying measurement.

Establish a baseline workload with known timestamps, simulate catastrophic primary failure, and time restoration.

1. Baseline workload: Generate continuous synthetic transactions writing timestamped sequence rows.
2. Simulated catastrophe: At timestamp T_0, ungracefully terminate the primary node (`kill -9`) and sever network storage.
3. Restoration timing: At T_0, trigger restore automation using the latest base backup and WAL archives. Record elapsed wall-clock time until the recovered instance passes health checks and accepts read/write queries ($T_{\text{y}}^{\text{read}}$). If $T_{\text{y}}^{\text{read}} - T_0 > 30\ \text{min}$, the SLA fails RTO.
4. Data loss audit: Query the maximum transaction timestamp $T_{\text{t}}^{\text{las}}$ present in the restored database. If $T_0 - T_{\text{t}}^{\text{las}} > 5\ \text{min}$, the SLA fails RPO.

Exercise 4.4.4: Point-in-Time Recovery vs. Streaming Replication for Logical Corruption

Generalisation transfer

A developer mistakenly executes `DROP TABLE accounts;` on production. Within 100 milliseconds, the command is replicated to the hot standby replica. Contrast why promoting the streaming replica fails to recover the data, and prescribe the exact point-in-time recovery (PITR) procedure.

Physical replication propagates logical drop commands immediately. PITR halts replay prior to the drop.

A physical streaming replica faithfully executes every WAL record emitted by the primary. The `DROP TABLE` command is immediately applied to the replica, destroying the table there as well. Prescribed PITR procedure:

1. Preserve the WAL archive and base backup immediately; do not overwrite existing standby state.
2. Provision an isolated recovery server and restore the latest full physical base backup taken prior to the incident.
3. Configure recovery settings with `recovery_target_time` set to 1 second before the exact execution timestamp of the `DROP TABLE` statement (or target the transaction ID preceding the drop).
4. Replay WAL archives up to the target boundary, verify table integrity, and export the recovered `accounts` relation back to the production environment.

References & Further Reading

- Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwarz, P. (1992). *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*. ACM TODS, 17(1), 94–162. (Analysis/redo/undo, page LSNs, and compensation log records.)
- PostgreSQL: Write-Ahead Logging
- PostgreSQL: Continuous Archiving and Point-in-Time Recovery
- PostgreSQL: Backup and Restore
- CMU 15–445/645 Database Systems

Data at scale and synthesis

SECTION DETAIL

Distributed Data

A client sends “transfer $100” and times out. The service cannot infer failure: the request may have reached a leader, committed on a quorum, committed locally but not replicated, or never left the client. Distribution adds uncertain communication to already difficult transaction boundaries. Start with the fact that must remain true and the failures the system promises to handle.

A consistency claim must name its observer and failure

DEFINITION

A consistency contract states what operation may observe relative to which completed operations, under an explicit failure and communication model. “Eventually consistent” without a convergence target, stale-read bound, or conflict rule is not a useful contract.

Product fact	Unsafe observation	Plausible contract	Availability consequence
transfer never spends money twice	client sees two accepted debits	one ordered, durable decision plus idempotency	may reject/wait during leader loss
product search catches up	new item missing briefly	asynchronous indexed projection	stale result is visible and repairable
tenant authorization revocation	revoked user still reads data	strongly ordered or bounded-stale authorization path	read may wait/fail during partition
analytics counter	dashboard lags by a minute	convergent aggregate with stated window	value may be temporarily approximate

A network partition does not prove which side contains the newest value. A timeout is an ambiguous outcome, so a client needs an operation ID and a status lookup or idempotent retry rather than a second independent request.

Prediction — client action after timeout

A payment request with idempotency key K times out. Which reply is safer: “failed,” “succeeded,” or “unknown; check K”?

“Unknown; check K” is the honest reply until the durable system of record resolves it. Retrying with the same key can be safe if the server persists and deduplicates K; retrying with a new key can create a second charge. A false success misleads the client, while a false failure can trigger duplicate work.

Quorum arithmetic proves only intersection

PROPOSITION 5.1.1 **Read/write quorum intersection**

For N replicas, a read quorum R and write quorum W intersect when $R + W > N$. Two write quorums intersect when $2W > N$. These inequalities establish shared replica membership; they do not alone establish leader fencing, version ordering, durability of a particular implementation, or linearizability.

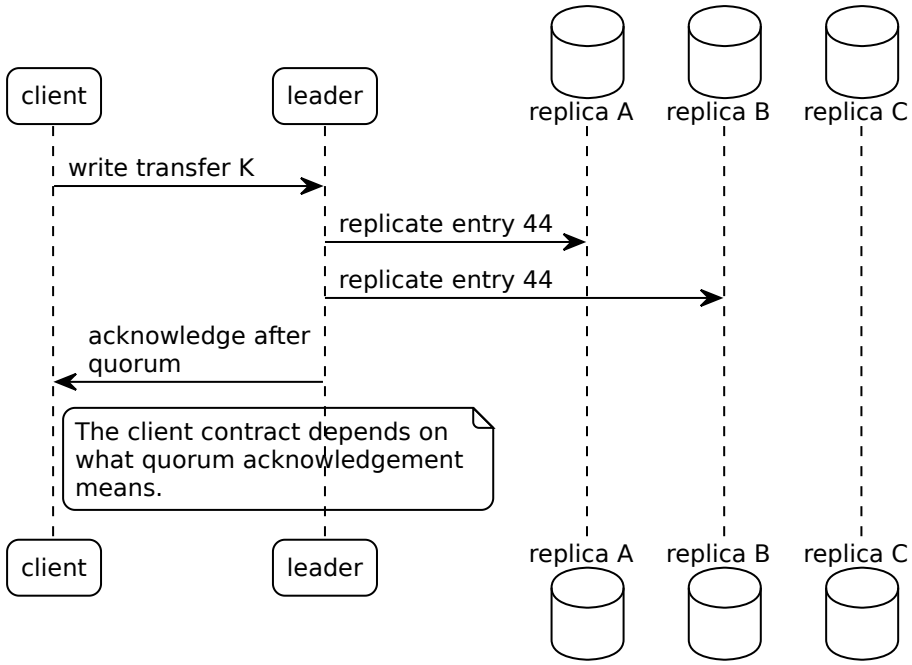
TEST A QUORUM CONTRACT AGAINST FAILURES



N = 3, R = 2, W = 2. Every read quorum intersects every write quorum, and one replica may be unavailable while reads and writes remain possible.

Static state: a three-replica, R=2/W=2 configuration provides quorum intersection and tolerates one unavailable replica for both operations.

Use the explorer to choose N, R, W, and failed replicas. Predict whether a read can obtain enough responses and whether its selected set must share a replica with a completed write. The model illustrates set intersection; a real protocol also needs a rule for choosing the newest valid version and for rejecting stale leaders.



Worked example — choose R and W

With N = 3 and W = 2, choose the smallest R that intersects every write quorum. Then state one thing the result still does not prove.

Use R + W > N, then list a protocol property outside set arithmetic.

R = 2, because 2 + 2 > 3. A read of two replicas intersects a completed two-replica write, but it still needs version/term rules to distinguish a newest value from a stale or divergent response. It also says nothing about whether a client received an acknowledgement before the intended data-loss boundary.

QUORUM INTERSECTION AND FRESHNESS

For an N = 5 node replica cluster, an architect configures read quorum R = 2 and write quorum W = 3. Evaluate which statements regarding quorum intersection and consistency are correct:

Answer: Because R + W = 5 <= N, a read of 2 nodes can miss all 3 nodes written by the previous write, returning stale data.; Monotonic read consistency is not guaranteed without client session stickiness or version tracking across queries.

Strict quorum intersection requires R + W > N. When R + W = N, read and write quorums may be disjoint, permitting stale reads unless additional version reconciliation is performed.

Coordination has a blocking and repair cost

EXAMPLE

An order service and a warehouse service each own a database. “Create order and reserve stock” can be treated as one distributed atomic decision, or as local durable steps with a compensating cancellation path. The observable failure behavior differs.

Design	Success rule	Partition / participant failure	Repair obligation
single database transaction	one local commit	one local failure domain	normal recovery
two-phase commit	coordinator collects prepare votes then decides	prepared participants can block awaiting decision	durable decision log and recovery protocol
saga / outbox	local commits advance an explicit state machine	temporary partial business state is visible	compensations, retries, and reconciliation
asynchronous projection	source event commits first	read model can lag	replay, watermark, and idempotent consumer

Two-phase commit is not “a transaction across anything”: participants must durably prepare, a coordinator records a decision, and failure can leave prepared participants unable to decide alone. A saga makes partial progress explicit. It is suitable only when domain-level compensation can restore an acceptable business state; reversing a shipment after delivery is not the same as undoing a row update.

Transfer — avoid a fake atomicity claim

A system writes a financial ledger in region X and sends a warehouse reservation request to region Y. Region Y is unreachable after the ledger commit. Choose a design and state what the customer can see.

Use a durable outbox/state machine: record a pending reservation with the ledger transaction, retry the Y request idempotently, and expose “payment authorized, fulfillment pending” until the reservation succeeds or a defined compensation/refund occurs. Claiming an atomic cross-region success after X commits is false without a coordinating protocol and its availability cost. The visible intermediate state is a product decision, not a technical accident.

Distributed commit needs a durable decision, not a wish

Two-phase commit (2PC) makes a set of participants agree on one outcome. In the **prepare** phase the coordinator asks each participant to durably record its vote and the resources it will need to commit. If every participant votes yes, the coordinator durably records **commit** and then tells them to commit; if any votes no, it records **abort**. A participant that has prepared must retain its locks and its ability to commit until it learns the decision.

Failure	What remains true	What the system cannot do alone
participant crashes after voting yes	its prepared state is durable	decide whether the coordinator chose commit or abort
coordinator crashes after recording commit	the decision is durable	notify unreachable participants until it recovers
coordinator crashes before recording a decision	no decision exists	force prepared participants to commit
network partition isolates a prepared participant	it cannot safely release locks	make progress without the decision

The blocking case is fundamental to 2PC: prepared participants may hold locks while they wait for the coordinator. Three-phase commit removes some blocking under stronger assumptions, but adds rounds and still assumes bounded or predictable failures. A saga avoids prepared locks by making intermediate state visible, at the cost of requiring domain compensation.

Consensus protocols such as Paxos and Raft replicate a **log**, not a collection of independent row copies. A write is committed when a leader has replicated it to a quorum and the entry is ordered in the log; a new leader must contain every committed entry. Leader terms or epochs fence stale leaders so an old leader cannot acknowledge a write after a new one has been elected.

> **INVARIANT**
>
> A quorum acknowledgement is evidence of replication, not of external consistency. The value a client reads may still depend on which replica answers and what ordering rule the client requires.

Spanner adds a physical ordering assumption: TrueTime exposes a bounded uncertainty interval for the local clock. A transaction waits out that uncertainty before committing, so timestamps assigned to committed transactions respect real-time order across regions. The result is **external consistency**, stronger than snapshot isolation, paid for with commit-wait latency and dependable clock infrastructure. Naming the assumption is part of the contract; a system without bounded clock uncertainty cannot copy the proof.

Transfer — choose the commit boundary

A cross-region transfer must never be visible as two independent halves, and the service cannot tolerate blocking during a coordinator failure. Which property must it give up, and what should it tell the client while a decision is pending?

A non-blocking design cannot also promise a single instantaneous cross-region commit without a stronger coordination substrate.

It must give up either non-blocking atomic commit or the claim of an instantaneous single decision. A durable outbox plus idempotent steps can keep serving, but the client sees an intermediate state such as "authorized, pending settlement." A consensus-backed ledger can provide an ordered decision, but it may wait for a quorum and must expose a truthful pending status on timeout. Claiming both non-blocking operation and atomic cross-region success without naming the coordination mechanism is false.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.1.1: Quorum Overlap Inequality Verification

Verification application

Referring to Proposition 5.1.1, evaluate a distributed cluster with $N = 7$ nodes configured with write quorum $W = 4$ and read quorum $R = 4$. Verify whether every read is guaranteed to observe at least one node containing the latest committed write.

Check the condition R + W > N.

Here $R + W = 4 + 4 = 8$. Because $8 > 7$ ($R + W > N$), by the Pigeonhole Principle the intersection of any read quorum of 4 nodes and any write quorum of 4 nodes contains at least $4 + 4 - 7 = 1$ node. Thus, every read quorum is mathematically guaranteed to contact at least one node that participated in the most recent successful write.

Exercise 5.1.2: CAP Theorem Partition Tolerance and Availability Boundaries

Description recall

Explain why a distributed data store experiencing a network partition between two data centers must choose between consistency (rejecting writes in the minority partition) and availability (accepting conflicting writes in both partitions).

Nodes in separated partitions cannot communicate state updates synchronously.

During a network partition, nodes in partition A cannot communicate with nodes in partition B. If the system chooses Availability, it must allow both partitions to accept writes independently, resulting in divergent states and violating linearizable Consistency. If the system chooses Consistency, nodes in the partition unable to contact a majority must refuse writes, sacrificing Availability for clients connected to that partition.

Exercise 5.1.3: Two-Phase Commit Coordinator Failure Analysis

Construction application

In a two-phase commit protocol across three database nodes, participant P_1 votes `YES` during the prepare phase. The coordinator crashes before logging or broadcasting the commit/abort decision. Explain why P_1 remains blocked holding locks and cannot unilaterally decide to abort or commit.

P1 does not know if other participants voted NO or if the coordinator already told another node to commit.

P_1 cannot unilaterally abort because the coordinator might have received `YES` from all nodes and committed the transaction before crashing (or communicated commit to P_2). P_1 cannot unilaterally commit because another participant might have voted `NO`, causing the coordinator to abort. Consequently, P_1 must remain in the `PREPARED` state holding locks until the coordinator recovers or an election-based cooperative termination protocol resolves the outcome.

Exercise 5.1.4: Asynchronous Saga and Outbox Pattern Design for Multi-Region Workflows

Generalisation transfer

An e-commerce order workflow requires debiting a customer's account in region `us-east` and reserving warehouse stock in region `eu-central`. Network latency and transient partitions make 2PC unacceptable. Design a resilient asynchronous saga using transactional outboxes and compensating actions to handle stock exhaustion.

Use local database transactions in each region with an explicit intermediate state machine.

1. Region `us-east`: In a single local transaction, record `order(id, status='pending')`, deduct balance, and write an `outbox` event `OrderPlaced`. Commit.
2. Message transport: An outbox worker reliably delivers `OrderPlaced` to an idempotent queue consumed by `eu-central`.
3. Region `eu-central`: Attempts stock reservation in a local transaction.
   - If stock is available: Mark reserved and emit `StockReserved`.
   - If stock is exhausted: Emit `StockFailed`.
4. Compensation: Upon consuming `StockFailed`, `us-east` executes a compensating transaction that refunds the customer's balance and transitions the order state to `cancelled`. Intermediate states (`pending`) are visible to users.

References & Further Reading

- Gray, J., & Lamport, L. (2006). *Consensus on Transaction Commit*. ACM TODS, 31(1), 133–160. (Two-phase commit, its blocking cases, and consensus-based commit.)
- Corbett, J. C., et al. (2012). *Spanner: Google's Globally-Distributed Database*. OSDI '12. (TrueTime, commit-wait, and external consistency.)
- In Search of an Understandable Consensus Algorithm (Raft)
- CMU 15–445/645 Database Systems
- PostgreSQL: Synchronous Replication

SECTION DETAIL

Replication and Partitioning

A replica that is one minute behind can be an excellent reporting target and a dangerous authorization target. A partitioned table can make retention cheap and still leave a tenant-hotspot unsolved. Replication and partitioning change where data lives and when it becomes visible; neither silently preserves every single-node guarantee.

Replication changes the meaning of acknowledgement

DEFINITION

Replication copies committed state or log records to other nodes. A write's durability claim depends on which copies must acknowledge before the client is told success and which failures are in scope.

Mode	Client acknowledgement	Read behavior	Primary risk
asynchronous	local durable primary commit	replicas can lag	acknowledged tail can be lost on failover
synchronous to one or more standbys	required standby acknowledgement as configured	some reads may still be stale	higher latency / reduced write availability
logical/event projection	source commit before consumer applies event	explicit lag and replay boundary	duplicate or delayed consumer work

A replica should report both replication position and lag relative to a named source. "Healthy" cannot mean only that its process is running. A read-after-write API must either route to a sufficiently caught-up replica, route to the writer, wait with a bound, or return a clearly stale contract.

Prediction — a false zero-loss claim

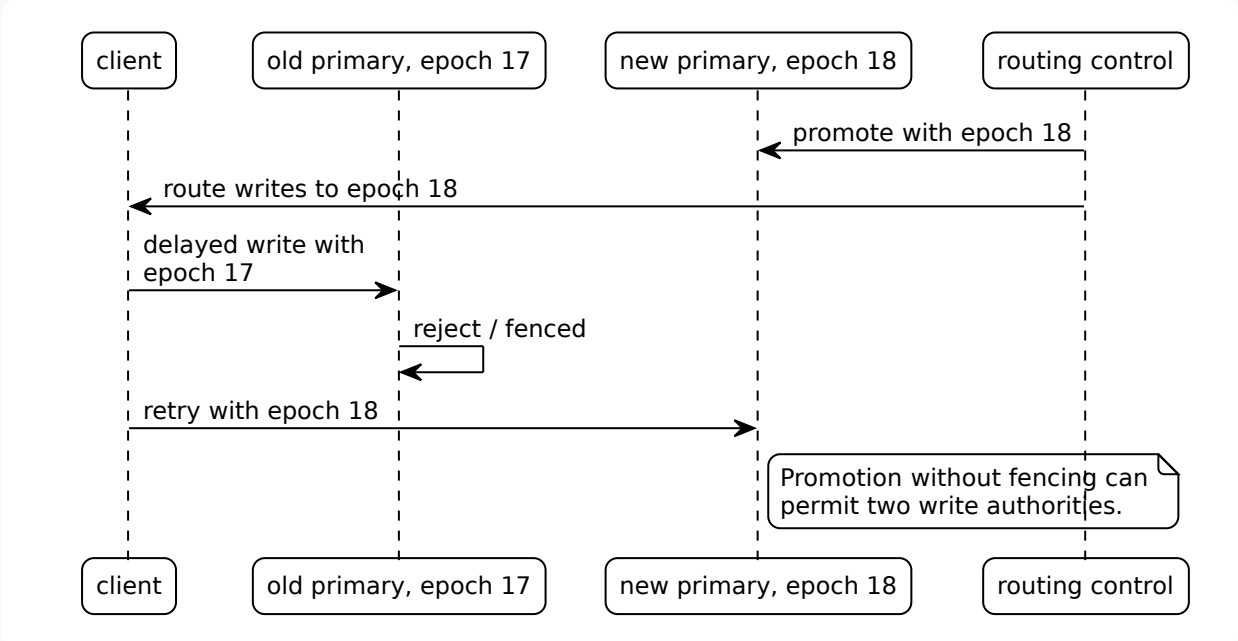
A primary acknowledges a write locally, then crashes before its asynchronous replica receives it. The replica is promoted. What happened to the client-observed write?

It can be lost despite the former primary's acknowledgement. The deployment promised local durable commit, not durable replication to the promoted node. Calling this "zero data loss" confuses storage durability with a failover durability contract. Synchronous acknowledgement, durable client retry records, or a revised product guarantee changes the claim and its latency/availability cost.

Promotion needs fencing against stale writers

INVARIANT

At most one authority may accept writes for a replication epoch. A promoted leader must reject requests carrying an old lease, term, or fencing token, and clients must stop reaching the former leader.



Failover is not finished when a process starts accepting traffic. Fencing prevents an isolated former primary from continuing to accept writes that later conflict with the new authority. The precise technique may be leases, terms, a strongly consistent configuration store, or a database-specific

mechanism; the required property is refusal of stale authority.

Application — read your own write

A client creates an order, then reads it from an asynchronous replica. Give two correct API behaviors.

Route the immediate read to the primary/source of truth, or attach a required log position/version and wait until the chosen replica confirms it has replayed that position within a timeout. Returning "not found" from a lagging replica as if the order never existed violates read-your-writes. The API can instead return an explicitly stale result if the product contract permits it.

WHY PROMOTION IS NOT ENOUGH

A standby replica is promoted to primary after network failure and accepts writes. The former primary remains online and receives delayed requests. Evaluate which mechanisms prevent split-brain data corruption:

Answer: Fencing tokens or epoch terms reject writes from the demoted primary by checking monotonically increasing leadership numbers.; Lease expiration guarantees that an isolated former primary cannot commit writes once its allocated time lease elapses.

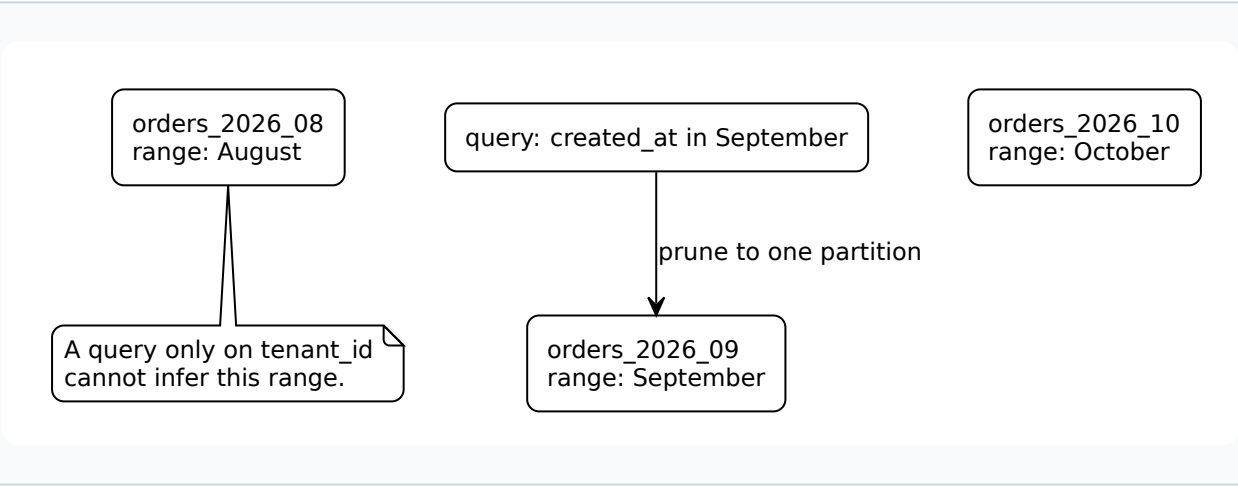
Promoting a new leader does not inherently disable the old leader. Split-brain prevention requires fencing tokens, generation epochs, or hardware STONITH to revoke write authority from stale nodes.

A partition key is a workload and operations decision

> ***Worked example.*** *Time partitions make dropping expired events cheap and can prune a query with a time predicate. They do not make a tenant-only query prune partitions, and they can concentrate today's write hotspot in one partition.*

Candidate key	Serves	Breaks or complicates	Evidence to collect
time range	retention, time-bounded scans	current-write hotspot; tenant-only query	date filters, retention cadence, ingest skew
tenant hash	tenant isolation and balanced ownership	cross-tenant reporting; tenant moves	tenant-size distribution, joins, reporting scope
geography	residency / latency boundary	global users and moving customers	legal domain, traffic origin, disaster plan
product/category	localized catalog access	uneven popularity and changing classification	hot-key distribution, reclassification rate

A partition is not automatically a shard. Local table partitions can remain on one database node; a shard adds independent placement, failure, and coordination boundaries. Global uniqueness and foreign keys become harder when the key omits the partition owner. Put tenant identity into keys where tenant isolation is a requirement, rather than assuming an application filter is always present.



Prediction — pruning counterexample

A table is partitioned monthly. A query filters only **tenant_id = 42** and lacks a time predicate. How many partitions can the planner discard from partition bounds alone?

None merely from month bounds: tenant identity gives no constraint on the partition key. An index can still help each partition, but the query may need to inspect all of them. Add an appropriate time predicate, choose a tenant-aligned partition/shard design when the workload warrants it, or maintain a separate routing structure with a documented correctness boundary.

Migration is a state transition, not a copy command

WARNING

Copying a tenant to a new shard and immediately switching traffic risks missing writes, duplicate events, and split reads. A migration needs a source of truth, an ordered cutover boundary, and reconciliation before deleting the old copy.

Phase	Required evidence
prepare	destination schema/version accepts the tenant; ownership routing is versioned
copy	snapshot position and checksum/count evidence are recorded
catch up	ordered changes after the snapshot replay idempotently
cut over	writes are fenced or redirected at one durable ownership epoch
verify	key counts, sampled hashes, invariants, and lag converge
retire	old copy is retained until rollback window and audit evidence expire

Transfer — choose the cutover test

Tenant 42 is moving to a new shard while orders continue to arrive. What single test is insufficient, and what must replace it?

A matching row count cannot prove every key/version and in-flight write moved exactly once.

A final row count alone is insufficient: it misses mismatched values, duplicate outbox events, and writes after the snapshot. Record a migration epoch, fence or route writes through that epoch, replay changes after a known position idempotently, then verify key sets, checksums or sampled hashes, and domain invariants before retirement. Keep a rollback decision rule rather than deleting the old copy on optimism.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.2.1: Split-Brain Prevention and Epoch-Based Leader Fencing

Verification application

In a high-availability primary-replica setup, a network partition isolates the active primary from the cluster monitor. The monitor promotes a standby to primary at epoch 2. Verify how a client request carrying epoch 1 sent to the old primary is safely rejected to prevent split-brain writes.

Examine monotonically increasing epoch tokens and lease validation.

Each mutation request carries the coordinator's current leadership epoch. If a request with epoch 1 arrives at the old primary, it either encounters an expired time-bounded leader lease or, upon attempting to append to persistent storage / consensus log, is rejected by storage fencing mechanisms that demand epoch ≥ 2. Because the cluster consensus quorum has already advanced to epoch 2, the stale primary cannot commit the transaction.

Exercise 5.2.2: Read-Your-Writes Consistency under Asynchronous Replication Lag

Description recall

Describe why a user who creates an account and is immediately redirected to a read replica may receive an HTTP 404 (Not Found) error under asynchronous replication, and propose two routing architectures that guarantee read-your-writes consistency.

Asynchronous replication decouples primary commit from replica log replay.

Under asynchronous replication, the primary confirms commits to the client before the WAL stream has been replayed on standby read replicas. If the client's subsequent read query routes to a replica lagging behind the commit LSN, the new account tuple is not yet visible. Architectural solutions:

1. Primary read routing: Route read requests immediately following user mutations to the primary database for a bounded duration (e.g. 5 seconds).
2. Session-level LSN wait: Record the primary's commit LSN in the client's session cookie; the read router forwards queries only to replicas that have confirmed replay past that recorded LSN.

Exercise 5.2.3: Workload-Driven Partition Key Selection: Time Range vs. Tenant Hash

Construction application

A SaaS platform with 50,000 tenants processes 50,000,000 order records per month. Contrast the operational trade-offs of range partitioning by `created_at` (monthly) versus hash partitioning by `tenant_id` (32 partitions) across query pruning and data retention workflows.

Analyze WHERE clause pruning and DROP TABLE efficiency.

1. Range by `created_at`: Ideal for data lifecycle retention; dropping an expired month is an $O(1)$ metadata operation (`DROP TABLE orders_2025_01`) with zero vacuum overhead. However, queries filtering only by `tenant_id` must scan all partitions, and all current-day writes concentrate into a single active partition.
2. Hash by `tenant_id`: Perfect for tenant isolation and query pruning on tenant-specific lookups; writes distribute evenly across all 32 partitions. However, dropping expired historical data requires expensive row-level `DELETE` scans across all 32 partitions.

Exercise 5.2.4: Zero-Downtime Live Shard Migration and Cutover Protocol

Generalisation transfer

Design a zero-downtime shard migration protocol to transfer a high-throughput tenant from an overloaded database shard A to a newly provisioned shard B. Detail the six sequential phases and evidence required before retiring shard A.

Follow prepare, copy, catch up, cut over, verify, and retire phases with epoch fencing.

1. Prepare: Verify destination shard B schema matches A, and initialize tenant routing configuration with epoch 1.
2. Initial copy: Take a consistent point-in-time snapshot of tenant rows on shard A at LSN L_0 and copy data to shard B.
3. Catch up: Enable CDC (change data capture) or logical replication to stream and apply all mutation records from L_0 forward onto shard B.
4. Cutover: Set tenant routing to read-only or brief pause; advance routing epoch to 2 pointing to shard B. Drain in-flight writes on A.
5. Verify: Execute reconciliation queries comparing row counts, cryptographic column checksums, and key sets between A and B.
6. Retire: Route all traffic to shard B under epoch 2. Retain frozen tenant data on shard A in read-only mode for 7 days before issuing drop.

References & Further Reading

- PostgreSQL: High Availability, Load Balancing, and Replication
- PostgreSQL: Table Partitioning
- In Search of an Understandable Consensus Algorithm (Raft)

SECTION DETAIL

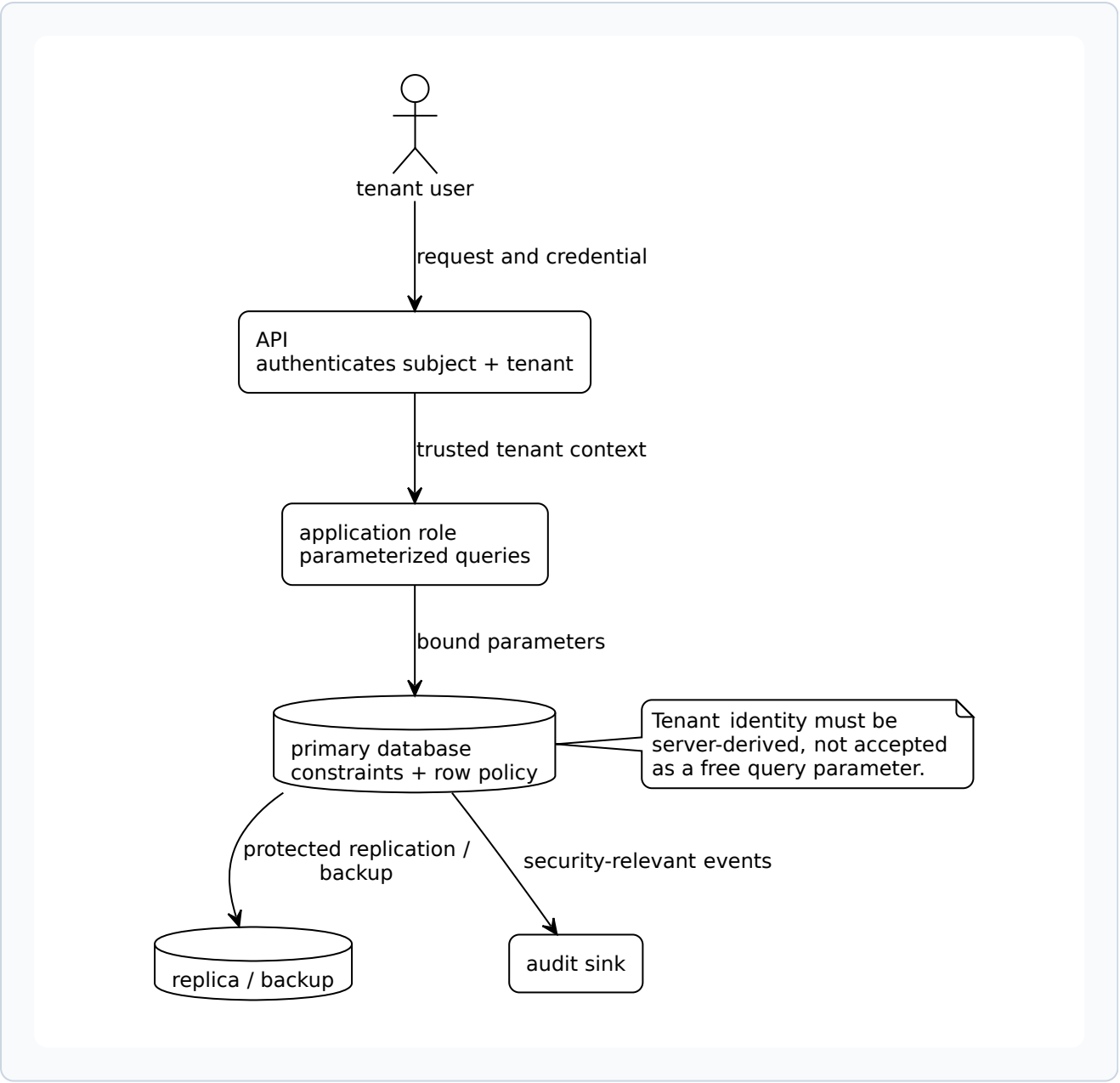
Database Security

A tenant filter in a web handler is not tenant isolation if an analyst endpoint, job worker, or direct SQL path can omit it. Database security begins by naming the data, the authority allowed to use it, and every transition where an attacker or defect can change that authority. The database is both a policy enforcement point and a high-value archive of credentials, history, backups, and replicas.

Put authority on the data flow

DEFINITION

A threat model names assets, trust boundaries, principals, entry points, and failure actions. A control is meaningful only if an untrusted path cannot bypass it.



The diagram has two authority claims: who is the caller, and which rows may this execution read or modify. The API should authenticate and derive tenant scope from a trusted session/token mapping. The database role should have only the permissions its code path requires. A database policy can add defense in depth, but it must be tested with the actual runtime role and with privileged bypass behavior explicitly understood.

Prediction — find the bypass and counterexample

An API filters every query by a client-supplied tenant ID. A scheduled export job connects with an owner-like database role and has no tenant predicate. Which path is the security boundary failure mode?

Both paths fail as a worked example of incomplete boundary design: the export job is an alternate path that can read every tenant, and the client-supplied tenant ID is not a trustworthy authority source. Derive tenant identity server-side, give the job a narrow role and explicit export scope, and enforce tenant ownership again in the database where practical. A correct public endpoint does not compensate for an overprivileged internal role.

Query construction and authorization solve different problems

WARNING

Parameter binding prevents attacker-controlled *values* from changing SQL structure. It does not decide whether an authenticated caller is authorized to request a row, and it cannot safely bind arbitrary table or column identifiers.

Need	Correct boundary	Common unsafe shortcut
value in a predicate	prepared statement / bound parameter	concatenate quoted input
user-selectable sort column	fixed allowlist mapped to known SQL fragments	bind an identifier as a value or concatenate it
tenant row access	server-derived tenant predicate and/or verified row policy	trust a browser-supplied tenant ID
migration changes	separate owner/migration role	let runtime API role alter schema
operational investigation	bounded support role plus audited procedure	share an administrator credential

A parameterized statement still needs an authorization condition. A perfect tenant predicate still needs binding so an injected value cannot alter the predicate. Security controls compose; their names are not interchangeable.

Application — repair a query API

A report endpoint accepts a status value and a sort field. It runs under the same role that can delete users. State the two independent repairs.

Separate SQL syntax handling from database authority.

Bind the status value as a parameter and map the sort field through a fixed allowlist of known columns/directions. Separately replace the broad runtime role with a least-privilege reporting role that can read only the approved view/table scope. Escaping input alone leaves destructive authority available; reducing the role alone leaves injection risk.

TWO CONTROLS, TWO JOBS

A reporting endpoint binds its filter values as parameterized inputs, but executes using a database role with full superuser privileges. Evaluate which security concerns remain unresolved:

Answer: The application role possesses destructive authority, allowing arbitrary table truncation if any secondary defect exists.; Parameterized queries cannot bind dynamic table or column identifiers (e.g. ORDER BY), leaving identifier injection possible.

Parameter binding separates data literals from query syntax trees, preventing SQL injection. It does not replace least-privilege role boundaries or protect against identifier injection.

Tenant isolation needs layered tests

INVARIANT

A request authenticated as tenant 12 must not read, write, infer, or mutate tenant 13's protected rows through ordinary APIs, background jobs, replicas, exports, or error paths.

Layer	Enforces	Test that can falsify it
application authorization	subject may act for tenant 12	change a URL/body tenant ID to 13
SQL query construction	input cannot rewrite a predicate	use hostile value syntax; verify query shape remains fixed
relational design	references do not cross tenant ownership	attempt a tenant-12 child pointing at tenant-13 parent
row-level policy	database filters rows for runtime role	issue a direct query as that role with no application predicate
audit/monitoring	sensitive access is discoverable	perform approved and denied access; inspect immutable event fields

Composite tenant-aware keys can make invalid cross-tenant references unrepresentable. Row-level security in PostgreSQL is an implementation feature whose policy enablement, session identity, table owner/superuser bypass, and role privileges need explicit tests. A policy that is never enabled for the runtime role is theater.

REMARK **Essential fields for security audit trails**

Record actor/service identity, tenant or scope, action, target identifier, outcome, correlation ID, time, and relevant policy/version context. Avoid logging raw secrets or unnecessary personal data. Audit events support investigation only if retention, access control, and tamper resistance are part of the design.

Protect the lifecycle, not only the primary table

EXAMPLE

TLS protects a database connection in transit; encryption at rest protects media under its key-management model; neither prevents an authorized but compromised application role from reading rows. Backups, replicas, logs, exports, and observability systems each need their own access boundary and retention rule.

Asset	Primary control	Recovery / detection evidence
runtime database credential	secret manager, rotation, narrow role	issuance/use record; revoke and validate replacement
customer rows	authorization, policies, constraints	denied-path tests; tenant-bound audit event
backup archive	independent access, encryption/key custody, retention	isolated restore with access logging
change history	append-oriented audit with protected writers	correlation across application and DB events
query traffic	TLS and certificate/endpoint validation	monitored failures and configuration review

Transfer — leaked runtime credential

A production read/write application credential appears in a build log. Put the first actions in order.

Prioritize credential containment, revocation, and scope analysis before considering data restore.

Contain first: revoke or rotate the exposed credential and stop the log's further distribution while preserving evidence. Identify its permissions and recent use, then inspect audit/query/access records for misuse and validate that the replacement uses a narrower role if possible. Rotate dependent connection pools and verify service behavior. A backup restore is not the first response unless data integrity evidence indicates destructive activity; encryption at rest does not invalidate a live credential.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.3.1: Parameter Binding vs. Dynamic Identifier Injection

Verification application

An endpoint constructs a query using: `f"SELECT * FROM items WHERE status = $1 ORDER BY {user_sort};"`. The developer asserts that because `$1` is parameterized, SQL injection is completely prevented. Verify why this claim is false, and demonstrate an exploit payload for `user_sort`.

Parameter binding cannot be used for SQL identifiers such as column names or sort directions.

Parameter binding separates data literals from query syntax trees; it cannot bind SQL keywords, table names, or column identifiers. If `user_sort` is interpolated directly into the query string, an attacker can pass `(CASE WHEN (SELECT 1 FROM users WHERE name='admin' AND substring(password,1,1)='a') THEN id ELSE name END)` or append stacked commands if multi-statement execution is enabled. Dynamic identifiers must be validated against a strict hardcoded allowlist.

Exercise 5.3.2: Separation of Runtime Role Privileges and Least Privilege

Description recall

Explain why connecting an application web server using a single all-powerful `postgres` superuser role violates least privilege, and outline a four-tier role separation model for production database systems.

Differentiate application web traffic, background workers, analytics, and schema migrations.

Using a superuser role grants unrestricted authority (`DROP TABLE` , bypass RLS, read write-ahead logs, execute system shell commands via extensions). If an injection or unauthorized access defect occurs, the entire database is compromised. Four-tier role model:

1. `app_user` : Limited `SELECT, INSERT, UPDATE, DELETE` strictly on necessary application tables; no DDL permissions.
2. `app_readonly` : `SELECT` permissions on reporting tables or views; used by analytics dashboards.
3. `migration_user` : DDL privileges (`CREATE, ALTER, DROP`) used exclusively during CI/CD deployment pipelines.
4. `admin` : Restricted administrative role reserved for DBA maintenance and emergency recovery.

Exercise 5.3.3: Row-Level Security Tenant Isolation Policy Definition

Construction application

Write PostgreSQL DDL to enable row-level security on `orders` and create a policy ensuring that the current session (via `current_setting('app.current_tenant_id')`) can select and insert only rows belonging to its authenticated tenant.

Enable RLS on the table and define a policy using USING and WITH CHECK clauses.

```
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation_orders ON orders
  FOR ALL
  TO app_user
  USING (tenant_id = current_setting('app.current_tenant_id',
true)::uuid)
  WITH CHECK (tenant_id =
current_setting('app.current_tenant_id', true)::uuid);
```

The `USING` clause restricts visible rows for reads, updates, and deletes; the `WITH CHECK` clause prevents inserting or updating rows with a foreign tenant ID.

Exercise 5.3.4: Incident Response Protocol for Compromised Database Credentials

Generalisation transfer

A continuous integration build log publicly exposes production read/write credentials for an orders database. Prescribe a 5-step incident response checklist to contain the breach, investigate potential data exfiltration, and re-establish secure operations.

Order actions: immediate revocation, credential rotation, log inspection, blast-radius audit, and post-mortem.

1. Immediate revocation: Terminate existing connection pool sessions and revoke or rotate the exposed database password immediately in secrets management.
2. Log suppression: Purge the public build log from version control and CI caches, invalidating web share links while preserving forensic copies internally.
3. Audit query logs: Query database audit logs and VPC connection logs for unauthorized client IP addresses connecting with the exposed username during the exposure window.
4. Data integrity verification: Run cryptographic checksums and domain invariant checks to detect unauthorized table modifications or exfiltration traces.
5. Least-privilege redeployment: Issue new credentials with least-privilege role scoping, and implement CI secrets scanning to prevent future pipeline credential leaks.

References & Further Reading

- OWASP: SQL Injection Prevention Cheat Sheet
- PostgreSQL: Row Security Policies
- NIST SP 800-57 Part 1 Rev. 5: Recommendation for Key Management

SECTION DETAIL

Database Capstone

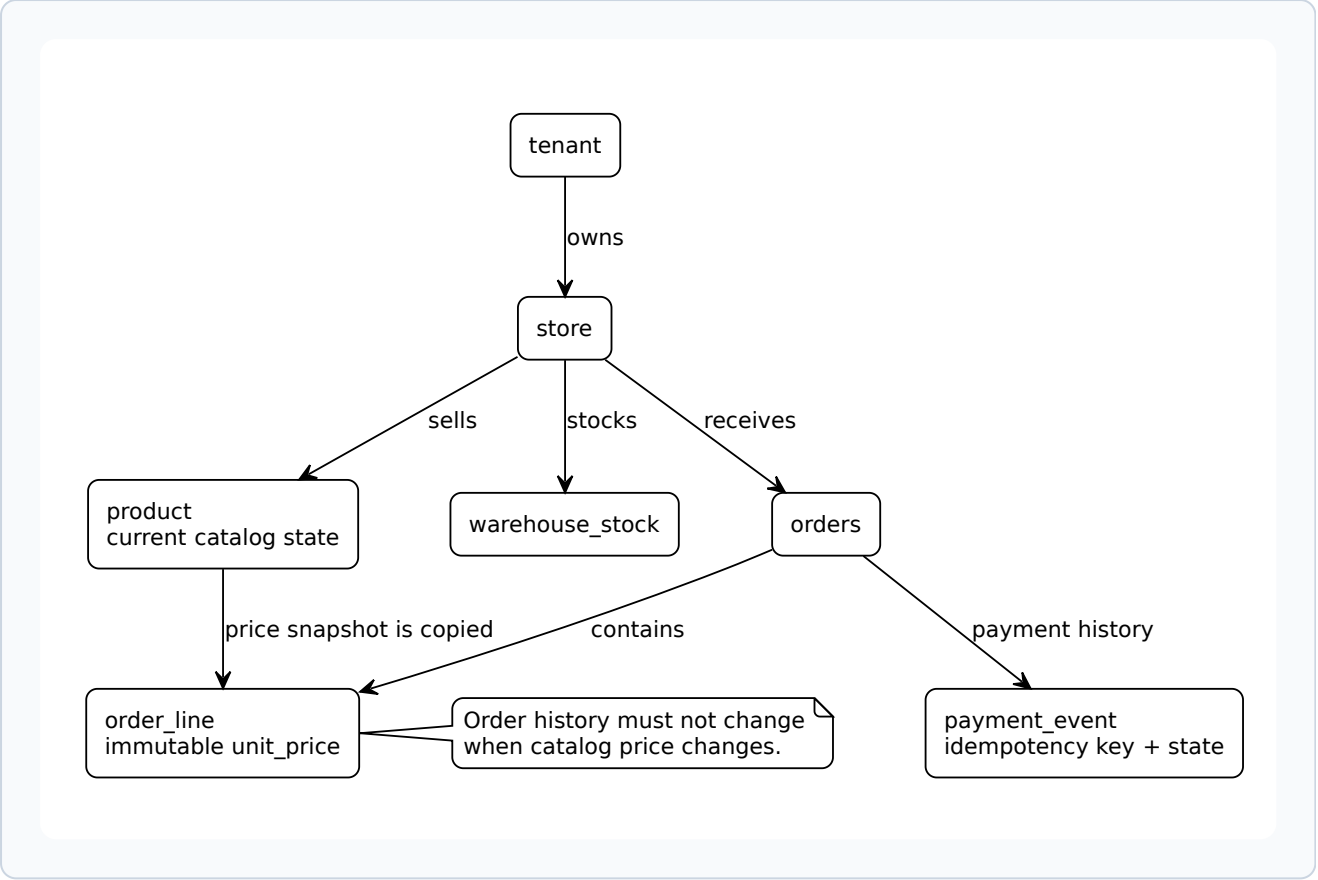
Build a database-backed marketplace for independent merchants. A merchant owns catalog items and warehouses; customers place orders; an order records immutable line prices; inventory is reserved before fulfillment; a provider sends duplicate and out-of-order payment webhooks. The target is not a collection of SQL snippets. It is an argument that the following facts survive normal operation and stated failures:

- a tenant cannot access another tenant's orders, products, inventory, or audit records;
- each order line retains the price accepted at checkout;
- inventory never becomes negative and one reservation is not counted twice;
- payment and fulfillment states advance only through legal, idempotent transitions; and
- an acknowledged order can be recovered, investigated, and measured.

Establish the data contract before selecting tables

INVARIANT

Every protected fact has one owner, one enforcement point, and one failure/reconciliation story. "The ORM handles it" does not identify any of those.



Write a one-page domain contract before DDL. It must name entity identity, tenant ownership, deletion policy, money representation, time zone, legal state transitions, service-level targets, and values that can be stale. For example, use integer minor units or an exact numeric representation for money; state whether a product may be soft-deleted after being ordered; and distinguish a current catalog price from a historical purchase price.

Prediction — find the missing fact

A draft schema stores order total and a foreign key to the current product price. A merchant changes a price after checkout. What report can now become false?

Historical revenue and line-item invoices can change retroactively if they derive accepted price from the mutable product row. Store the accepted unit price and currency on each order line, or reference an immutable price version. If an order total is stored for performance, declare whether it is derived or atomically maintained and test it against the line sum.

Make invalid relationships unrepresentable

EXAMPLE

Tenant-scoped composite keys make it impossible to attach a tenant-12 order line to a tenant-13 order when the foreign key includes both tenant identity and order identity.

```sql
CREATE TABLE orders (
  tenant_id      bigint NOT NULL,
  order_id       bigint GENERATED ALWAYS AS IDENTITY,
  customer_id    bigint NOT NULL,
  status         text NOT NULL,
  created_at     timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (tenant_id, order_id),
  CHECK (status IN ('pending', 'paid', 'cancelled', 'fulfilled'))
);

CREATE TABLE order_line (
  tenant_id       bigint NOT NULL,
  order_id        bigint NOT NULL,
  line_no         integer NOT NULL,
  product_id      bigint NOT NULL,
  quantity        integer NOT NULL CHECK (quantity > 0),
  unit_price_minor bigint NOT NULL CHECK (unit_price_minor >= 0),
  PRIMARY KEY (tenant_id, order_id, line_no),
  FOREIGN KEY (tenant_id, order_id)
    REFERENCES orders (tenant_id, order_id)
);

CREATE TABLE warehouse_stock (
  tenant_id   bigint NOT NULL,
  warehouse_id bigint NOT NULL,
  product_id  bigint NOT NULL,
  on_hand     integer NOT NULL CHECK (on_hand >= 0),
  PRIMARY KEY (tenant_id, warehouse_id, product_id)
);
```

The code is a starting artifact, not the full schema. Add product/customer ownership foreign keys, a tenant-scoped SKU uniqueness rule, state-transition enforcement, webhook-event uniqueness, and deletion policy. A row-local CHECK cannot safely enforce an aggregate rule over arbitrary sibling rows; identify the transaction or summary relation that carries that responsibility.

Application — prove one key choice

Give a counterexample that the composite order-line foreign key rejects but a foreign key on **order_id** alone could accept in a system where order identifiers are tenant-local.

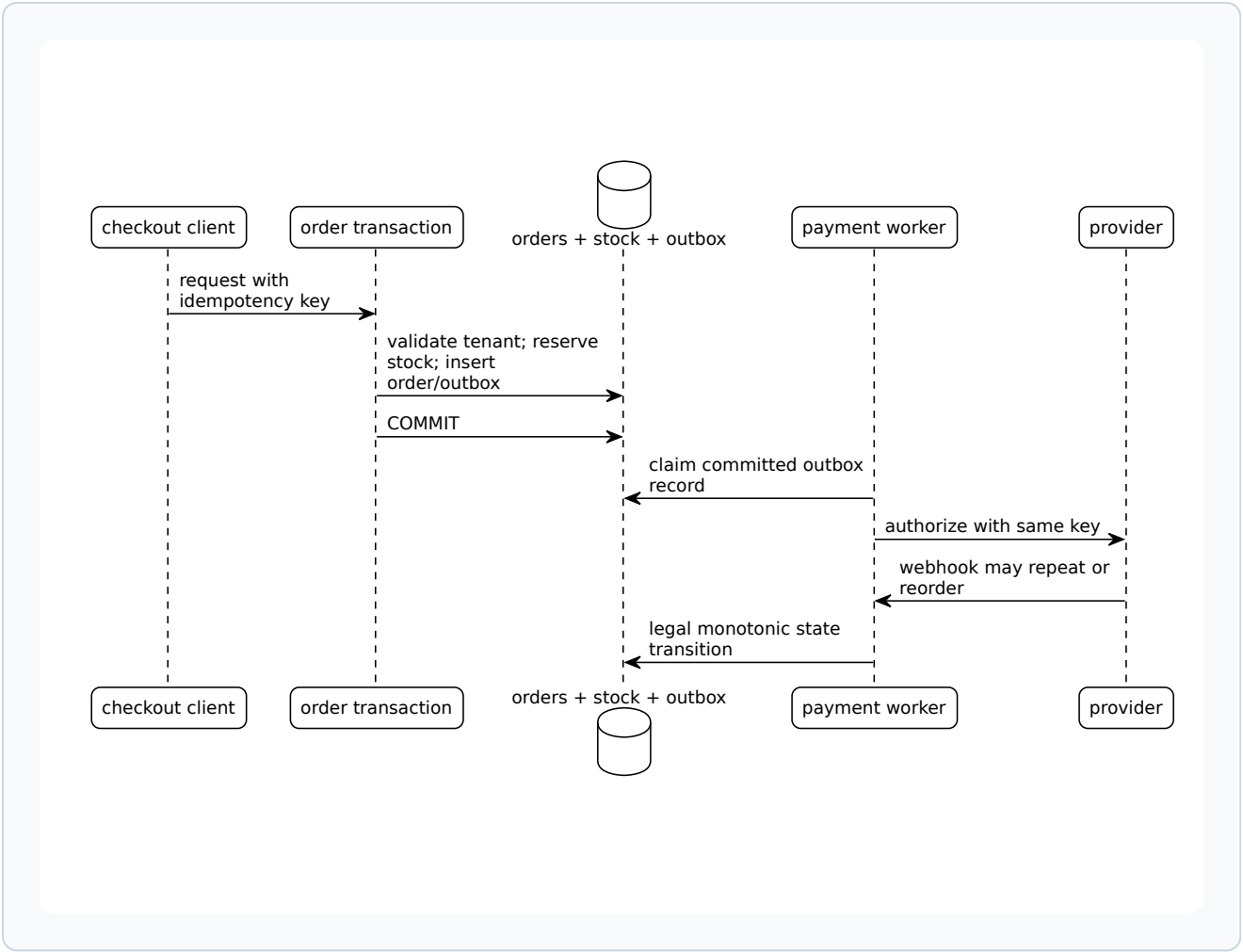
Construct the same order number in two tenants.

Let tenant 12 and tenant 13 both have order ID 9. A line with tenant 12 and order ID 9 must reference tenant 12's order, not whichever row an unconstrained ID lookup finds. The composite foreign key enforces that pair. A global surrogate ID can also work, but it does not remove the need to enforce tenant ownership on every related object and access path.

Design a short, recoverable order transaction

DEFINITION

The order transaction is the point where the system turns a customer request into durable local facts: order identity, immutable prices, inventory reservation, idempotency key, and an outbox event. It must not hold database locks while calling a payment provider.



The reservation must protect its own invariant. A conditional update of one stock row can reject a request whose quantity exceeds available stock; an aggregate reservation across warehouses needs a defined lock/serialization boundary. Store the client idempotency key under a uniqueness rule so retries converge on one order. The provider call is external: use the durable outbox and a webhook event ID to deduplicate it.

Worked trace — competing last-unit requests

1. T1 conditionally reserves the last unit and commits.
2. T2 executes the same conditional update after T1; its affected-row count is zero.
3. T2 records or returns a rejection without creating a successful order state.
4. A committed outbox worker can retry provider communication with the same idempotency key.

REMARK **Essential evidence retained in concurrency verification**

Keep initial fixture state, transaction timestamps/order, statement outcomes, isolation setting, locks or serialization errors, final rows, and emitted outbox IDs. A final stock value alone cannot distinguish a correct rejection from a lost update.

STRUCTURING THE CHECKOUT AND PAYMENT BOUNDARY

During flash-sale contention, a checkout service begins a database transaction, locks the `warehouse_stock` row with `SELECT ... FOR UPDATE`, makes a synchronous 400 ms HTTPS call to an external payment gateway, and then commits the order.

Evaluate the architectural trade-offs of holding stock locks across external payment gateway invocations:

Reference decision: Synchronous external HTTPS calls inflate lock holding durations, exhausting connection pools and causing throughput collapse under Little's Law.

Holding database locks during remote HTTP calls exhausts connection pools and creates severe lock convoys. Decouple local reservations from external calls using transactional outboxes and idempotent workers.

Measure the workload and physical design

WARNING

An index, partition, or replica is a workload claim. Do not add one because it is familiar; name the query, cardinality, write cost, and changed-workload counterexample.

Workload	Candidate evidence	Required counterexample
tenant order history	plan for tenant/time range; index on tenant then time	tenant with 90% of all orders
stock lookup/reservation	primary key or narrow tenant/warehouse/product path	concurrent requests for same SKU
daily revenue	grain-first aggregate and calendar left join	day with zero sales, duplicate lines
webhook processing	uniqueness lookup on provider event ID	duplicate and out-of-order event
retention	time partition/pruning plan and drop policy	tenant-only query that cannot prune

Seed enough data to expose selectivity, skew, and cache behavior. Record database version, schema/index definition, data generator distribution, concurrency level, warm-up, query text, plan, estimated and actual rows, and latency distribution. A single low elapsed time is not a benchmark and a single plan screenshot is not a physical-design argument.

Transfer — reject a seductive index

A proposal adds a wide covering index on every order column to speed a nightly export. The table receives continuous inserts and status updates. What evidence would justify or reject it?

Compare the export plan and buffer/I/O behavior with and without the index, then measure insert/update throughput, index size, vacuum/replication cost, and cache pressure under representative concurrency. If the export reads most rows once nightly, a sequential or partition-aware plan may dominate. The index earns its cost only when measured read benefit exceeds the continuing maintenance cost for the stated workload.

Rehearse recovery, replication, and security

DEFINITION

A production guarantee is incomplete without an operational test: a backup must restore, a failover must fence stale writers, and a least-privilege policy must reject a real adversarial request.

Drill	Pass evidence
bad migration	isolated point-in-time restore, invariant checks, measured RPO/RTO
primary loss	promotion path, client fencing/epoch behavior, acknowledged-write loss statement
replica lag	read-after-write behavior and stale-result contract
leaked runtime credential	revocation/rotation time, narrowed replacement role, audit review
tenant attack	altered tenant ID, direct runtime-role query, composite-FK and policy rejection
negative inventory alert	freeze affected workflow, trace first invariant break, reconcile from order/audit facts

Write an incident runbook for each drill: detection, containment, evidence preservation, integrity check, recovery action, client communication boundary, and regression test. A replica is not a backup after a bad delete; a backup is not security control if its archive role is broadly shared.

Synthesis analysis prompt

Choose one intentional relaxation: asynchronous search indexing, a delayed analytics aggregate, time partitioning, a cross-service saga, or a readable replica. State the invariant it does **not** preserve immediately, the visible state it permits, the repair/reconciliation process, and the test that detects a broken implementation.

A strong answer names one bounded fact that may lag and a separate fact that still cannot be violated.

A satisfactory analysis distinguishes the relaxed projection from the authoritative transaction. For example, search may lag after an order commit, but the order row and tenant authorization remain authoritative; an idempotent event consumer replays missed work and a watermark/lag alert detects delay. Do not call the relaxation safe without an observable bound, source of truth, and repair path.

System specification and verification checklist

> **REMARK**
>
> Implementation artifacts are evidence containers: a diagram alone, a fast query alone, or a backup policy alone cannot establish the system contract.

A complete implementation package runs from an empty database and includes:

1. domain contract, ER/domain diagram, DDL, migration plan, and seed fixture;
2. query portfolio with declared result grain, expected outputs, adversarial NULL/duplicate/tenant tests, and representative plans;
3. transaction/schedule dossier for inventory, idempotency, webhook order, and one aggregate invariant;
4. physical-design and benchmark report with workload assumptions and a rejected alternative;
5. recovery, replication, security, and incident-runbook evidence; and
6. a concise design analysis that identifies one limit the system intentionally accepts.

Criterion	Invariant standard	Observable verification evidence
semantic and tenant correctness	strict tenant isolation across all tables	executable constraints, counterex-amples, and tested ownership
query/physical reasoning	plans reflect intended index usage and grain	measured plans plus write-cost and skew evidence
concurrency and recovery	ACID isolation and crash recovery	traces, failure injection, safe retries, restore drill
security and operations	least-privilege roles and audit trail	least privilege, attack tests, audit/rotation/runbooks
clarity of trade-offs	explicit failure boundaries and non-goals	rejected alternatives and explicit assumptions

CAPSTONE INVARIANT VERIFICATION

Which combination of mechanisms prevents both cross-tenant order leakage and duplicate payment state transitions in the marketplace capstone?

Answer: Composite tenant-aware foreign keys combined with Row-Level Security policies on least-privilege roles enforce strict cross-tenant data isolation.; A unique constraint on provider_event_id combined with monotonic order state machine checks prevents duplicate webhook processing.

Defense-in-depth tenant isolation combines structural composite foreign keys with non-bypassable RLS policies on least-privilege runtime roles, while idempotent webhook processing requires uniqueness constraints and monotonic state machines.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.4.1: Multi-Tenant Foreign Key and RLS Defense-in-Depth

Verification application

In the marketplace schema, a malicious tenant attempts to attach an order line referencing product ID `prod-99` belonging to a rival merchant. Verify how the combination of composite foreign keys `FOREIGN KEY (tenant_id, product_id) REFERENCES Product(tenant_id, product_id)` and PostgreSQL Row-Level Security prevents unauthorized reference even if application-level tenant filters are bypassed.

Analyze relational constraint checks during foreign key evaluation under active RLS policies.

Even if an attacker manipulates the application API to insert an order line with a foreign `product_id`, the composite foreign key forces the database engine to locate a parent tuple matching `(attacker_tenant_id, prod-99)` in `Product`. Because `prod-99` belongs to a different tenant, no such tuple exists within the attacker's tenant key domain, causing immediate referential integrity rejection (`violates foreign key constraint`). Furthermore, active RLS policies on `Product` render the rival tenant's row completely invisible to the attacker's database session.

Exercise 5.4.2: Idempotent Payment Webhook Processing and Inventory Invariants

Description recall

Describe the database mechanisms required to guarantee that duplicate delivery of a payment gateway webhook (arriving concurrently across two webhook worker processes) cannot result in double-crediting an order or releasing reserved inventory twice.

Combine database unique constraints with atomic conditional status transitions.

1. Unique constraint: Webhooks record their arrival in `processed_webhooks` keyed on `UNIQUE (provider_event_id)`. The first worker commits the insert; the concurrent duplicate worker fails immediately with a unique key violation (`23505`) and safely halts execution.
2. Monotonic state transition: Order status is advanced using guarded conditional updates: `UPDATE orders SET status = 'paid' WHERE id = $1 AND status = 'pending';`. If the status is already `paid`, 0 rows are affected, preventing re-execution of inventory fulfillment triggers.

Exercise 5.4.3: High-Volume Order Partitioning and Pruning Strategy

Construction application

An enterprise marketplace accumulates 20,000,000 order records monthly. Construct a PostgreSQL table partitioning scheme for `orders` partitioned by month, and specify the indexing strategy that enables fast customer point lookups while retaining single-partition query pruning for monthly reports.

Range partition by created_at; include created_at in the composite primary key.

```sql
CREATE TABLE orders (
  tenant_id UUID NOT NULL,
  order_id BIGINT NOT NULL,
  created_at TIMESTAMPTZ NOT NULL,
  customer_id BIGINT NOT NULL,
  total_cents BIGINT NOT NULL,
  PRIMARY KEY (tenant_id, order_id, created_at)
) PARTITION BY RANGE (created_at);

-- Monthly partition
CREATE TABLE orders_2026_03 PARTITION OF orders
  FOR VALUES FROM ('2026-03-01 00:00:00+00') TO ('2026-04-01
00:00:00+00');

-- Local index for customer point lookups
CREATE INDEX idx_orders_customer ON orders (tenant_id,
customer_id, created_at DESC);
```

Queries filtering by `created_at` prune away 11 out of 12 monthly partitions. Lookups by customer leverage the local B-tree index across partitions.

Exercise 5.4.4: End-to-End Architectural Defense: RPO/RTO Failure Runbook

Generalisation transfer

An automated failover promoter initiates promotion of a replica after network partition, but the old primary resumes accepting writes. Specify the runbook protocol and fencing mechanisms to guarantee zero split-brain data corruption and satisfy an RTO under 15 minutes.

Use STONITH/fencing tokens and verify client connection pool draining before acknowledging failover.

1. Fencing and STONITH: Before promoting the standby, the cluster consensus coordinator issues a shoot-the-other-node-in-the-head (STONITH) or revokes the storage controller lease of the old primary. The old primary's storage access is immediately terminated, ensuring it cannot commit in-flight writes.
2. Promotion with Epoch: The standby promotes with generation epoch $E + 1$.
3. DNS / Virtual IP Re-route: Connection routers update endpoint targets to the new primary at epoch $E + 1$.
4. Pool validation: Connection pools close stale idle sockets; client drivers attempting to connect with epoch E receive connection-refused errors.
5. Post-failover audit: Compare the promoted primary's WAL position against archive storage to verify zero lost committed transactions ($\mathrm{RPO} = 0$).

References & Further Reading

- PostgreSQL Documentation
- PostgreSQL: Constraints
- PostgreSQL: Backup and Restore
- PostgreSQL: Row Security Policies
- CMU 15-445/645 Database Systems

Course Practice

SECTION DETAIL

Database Systems Practice Workshop

Problem Workshop

This workshop generates algorithmic practice problems spanning database systems: relational algebra operations, functional dependency closures and normal forms, query optimization cost models, transaction conflict serializability, and ARIES crash recovery analysis.

Recommended Workflow

1. Derive complete query plans, attribute closures, and recovery states independently before inspecting hints.
2. Consult progressive hints when reaching a modeling or algorithmic impasse.
3. Verify derivations against the complete worked solution, checking invariant conditions and state transitions.

INTERACTIVE PRACTICE WORKSHOP

Randomized parameter generation, progressive hints, and worked derivations are available in the interactive web edition. Covered topics:

- Relational Algebra
- Functional Dependencies & Normalization
- Query Optimization & Cost Models
- Transactions & Concurrency Control
- Crash Recovery & Logging

References & Further Reading

- Raghu Ramakrishnan and Johannes Gehrke (2003). *Database Management Systems* (3rd ed.). McGraw-Hill.
- Abraham Silberschatz, Henry F. Korth, and S. Sudarshan (2019). *Database System Concepts* (7th ed.). McGraw-Hill.
- C. Mohan, Don Haderle, et al. (1992). *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*. ACM TODS.