

1 Introduction to Transactions

A transaction is a logical unit of database work that consists of one or more operations. A transaction might transfer money between bank accounts, register a customer for a course, reserve a rental car, or update inventory after a purchase. Although these operations may involve multiple SQL statements and multiple database records, they should often be treated as one indivisible operation.

Consider a bank transfer of \$100 from account A to account B. The database must subtract \$100 from A and add \$100 to B. If the first operation succeeds but the second fails, the database will contain an incorrect result: the money will have disappeared from the accounts covered by the transfer. A transaction allows the database to treat both operations as one unit: either both changes become permanent, or neither does.

Transactions are particularly important in multiuser databases. Multiple applications and users may read and modify the same records at the same time. Without appropriate concurrency control, one transaction might observe another transaction's incomplete changes, overwrite an update, or calculate a result using mutually inconsistent data. Transaction management provides mechanisms to control these situations while allowing multiple users to work concurrently.

2 Starting, Committing, and Rolling Back Transactions

2.1 Basic Transaction Commands

The fundamental transaction commands are `BEGIN`, `COMMIT`, and `ROLLBACK`. Depending on the database system, `BEGIN TRANSACTION` or `START TRANSACTION` may also be used to start a transaction.

```
BEGIN;

UPDATE accounts
SET balance = balance - 100
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 100
WHERE account_id = 2;

COMMIT;
```

The `BEGIN` statement starts a transaction. The two `UPDATE` statements modify the accounts within that transaction. The `COMMIT` statement makes the transaction's changes permanent, subject to the database's durability guarantees.

If an error occurs before the transaction commits, the application can issue `ROLLBACK` to undo its uncommitted changes.

```
BEGIN;

UPDATE accounts
SET balance = balance - 100
WHERE account_id = 1;

-- An error is detected before the transfer is completed.

ROLLBACK;
```

After a successful rollback, the balance change made by this transaction is undone. The database returns the affected data to its state before the transaction began, except for effects that are outside the transaction's rollback mechanism.

2.2 Autocommit and Transaction Boundaries

Many database clients use autocommit mode by default. In this mode, each successful SQL statement is committed automatically unless an explicit transaction is started or the client changes the transaction configuration. For example, an individual `UPDATE` may become permanent as soon as it succeeds.

This behavior is convenient for independent statements, but it is dangerous when several statements must succeed together. If two updates in a transfer are committed independently, rolling back the second statement cannot undo the first committed update. The transaction boundary must include every operation that belongs to the same logical unit of work.

Transaction syntax and autocommit behavior vary across database systems and client libraries. Some systems automatically begin transactions, while others use explicit transaction blocks. Certain DDL statements, such as table creation or schema changes, may cause implicit commits or may not be permitted inside ordinary transactions in some systems. Applications should understand the behavior of their particular database and driver.

2.3 What Happens When a Statement Fails?

A failed statement does not necessarily have the same effect in every database. Some systems mark the entire transaction as failed and require a rollback before further work can continue. Other systems allow the application to handle certain statement errors and continue the transaction. Savepoints can provide an intermediate recovery point.

```
BEGIN;

UPDATE accounts
SET balance = balance - 100
```

```
WHERE account_id = 1;

SAVEPOINT before_second_update;

UPDATE accounts
SET balance = balance + 100
WHERE account_id = 999;

-- If the second update was logically incorrect:
ROLLBACK TO SAVEPOINT before_second_update;

ROLLBACK;
```

A savepoint is a named position inside a transaction. Rolling back to a savepoint undoes changes made after that savepoint without necessarily undoing earlier work. A savepoint does not commit any changes. In this example, the final `ROLLBACK` undoes the remaining transaction, including the first update.

A particularly important practical issue is that a successful SQL statement does not guarantee that the intended business operation succeeded. The second update above might affect zero rows rather than raise an error. The application should check affected-row counts and validate business conditions. Database constraints can also prevent invalid states.

3 The ACID Properties

ACID is a set of four properties used to describe the expected behavior of database transactions: atomicity, consistency, isolation, and durability.

3.1 Atomicity

Atomicity means that a transaction's changes are treated as one unit. Either the transaction commits its changes, or its uncommitted changes are rolled back. A transaction should not leave behind only half of a multi-step operation.

For example, a reservation transaction might create a booking, assign a vehicle, and mark the vehicle unavailable. If one step fails, the database should not retain a booking without a corresponding vehicle assignment. Atomicity is implemented through mechanisms such as transaction logs, undo information, and recovery procedures.

Atomicity does not automatically cover external systems. If a transaction sends an email or calls a payment service, rolling back the database transaction does not undo that external action. Such operations require additional application-level design.

3.2 Consistency

Consistency means that a committed transaction preserves the database invariants that the system is designed to enforce. Examples include a foreign key referencing an existing customer, a nonnegative inventory quantity, or a rule that an account balance cannot be negative.

Database constraints such as `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `CHECK`, and `NOT NULL` help enforce consistency. For example:

```
CREATE TABLE accounts (  
    account_id INTEGER PRIMARY KEY,  
    balance DECIMAL(12,2) NOT NULL,  
    CHECK (balance >= 0)  
);
```

Consistency does not mean that the database automatically understands every business rule. A database may satisfy all its declared constraints while containing a logically incorrect result. For example, a transaction could debit an account and forget to credit the destination account if the application implements the transfer incorrectly. Correct transaction logic and correctly defined constraints are both necessary.

3.3 Isolation

Isolation concerns the interaction between concurrent transactions. A transaction should not be affected by other transactions in ways that violate the guarantees of its selected isolation level.

For example, one transaction may be calculating a report while another is updating several related records. Depending on the isolation level, the report may see the old committed data, the new committed data, or a snapshot of the data from a particular point in time. Isolation levels determine which concurrency anomalies are permitted.

Isolation does not necessarily mean that transactions execute one after another. Database systems use locking, multiversion concurrency control, and other techniques to allow concurrent execution while enforcing the required guarantees.

3.4 Durability

Durability means that once a transaction successfully commits, its changes are preserved even if the database subsequently experiences a failure covered by its durability guarantees.

Database systems commonly use write-ahead logging and recovery procedures to provide durability. In a typical write-ahead logging system, the necessary log records are made durable before the database acknowledges a commit. The database can then recover committed

changes after a crash, even if some modified data pages had not yet been written to their final locations.

The precise guarantee depends on configuration. For example, a system configured to acknowledge commits before synchronously flushing required log records to durable storage may lose some recently acknowledged transactions after a power failure. Replication and backup policies also affect disaster recovery, but replication alone is not a substitute for backups.

4 Why Concurrent Transactions Are Difficult

A database may have many concurrent transactions operating on the same data. Concurrency improves throughput and resource utilization, but the order in which operations interleave can affect their results. A concurrency anomaly occurs when an interleaving produces behavior that violates the guarantees expected by the application.

Consider two transactions that read the same account balance and then update it. If both read the old value and each writes a result based on that value, one update may overwrite the other. This is called a lost update. Other anomalies involve reading uncommitted data, obtaining different answers from repeated queries, or making decisions based on incomplete information about other transactions.

Concurrency anomalies are not necessarily caused by programming mistakes in individual SQL statements. Each statement may be correct in isolation, yet their combined execution may be incorrect. Correct transaction design therefore requires reasoning about the behavior of multiple transactions executing at the same time.

5 Common Concurrency Anomalies

5.1 Dirty Reads

A dirty read occurs when one transaction reads a change made by another transaction that has not yet committed.

Suppose account A initially contains \$500.

```
-- Transaction T1
BEGIN;
UPDATE accounts
SET balance = 100
WHERE account_id = 1;
-- T1 has not committed.

-- Transaction T2, if dirty reads are permitted
```

```
SELECT balance
FROM accounts
WHERE account_id = 1;
-- T2 reads 100.

-- Transaction T1
ROLLBACK;
```

Transaction T2 observed a balance of \$100, even though T1 ultimately rolled back and the balance returned to \$500. If T2 used the observed value to approve a payment or produce a report, its decision would be based on a value that was never committed.

Dirty reads are generally prevented by the standard **READ COMMITTED** isolation level and stronger levels. Some database systems do not allow dirty reads even when a client requests a weaker isolation level.

5.2 Nonrepeatable Reads

A nonrepeatable read occurs when a transaction reads the same row twice and obtains different values because another transaction committed an update between the two reads.

Suppose a transaction checks an employee's salary and then checks it again later.

```
-- Transaction T1
BEGIN;
SELECT salary
FROM employees
WHERE employee_id = 10;
-- Returns 80000.

-- Transaction T2
UPDATE employees
SET salary = 90000
WHERE employee_id = 10;
COMMIT;

-- Transaction T1
SELECT salary
FROM employees
WHERE employee_id = 10;
-- Under READ COMMITTED, may return 90000.
```

Both queries in T1 are valid. Nevertheless, the transaction sees two different committed values for the same row. This can be problematic if the transaction assumes that a value remains stable throughout its work.

The usual **READ COMMITTED** guarantee does not prevent this anomaly. Repeatable-read and

serializable implementations provide stronger protections, although the exact behavior of repeatable read depends on the database system.

5.3 Phantom Reads

A phantom read occurs when a transaction repeats a query that identifies a set of rows and obtains a different set because another transaction inserted, deleted, or modified rows so that they now satisfy the query predicate.

For example:

```
-- Transaction T1
BEGIN;
SELECT COUNT(*)
FROM reservations
WHERE vehicle_id = 5
      AND status = 'ACTIVE';
-- Returns 2.

-- Transaction T2
INSERT INTO reservations
      (reservation_id, vehicle_id, status)
VALUES
      (101, 5, 'ACTIVE');
COMMIT;

-- Transaction T1
SELECT COUNT(*)
FROM reservations
WHERE vehicle_id = 5
      AND status = 'ACTIVE';
-- May return 3 under READ COMMITTED.
```

The second query finds a row that was not in the result of the first query. The row is called a phantom relative to the original query.

Preventing phantom reads may require predicate or range locking, or a multiversion snapshot that keeps the visible set of rows stable. The implementation depends on the isolation level and database engine.

5.4 Lost Updates

A lost update occurs when concurrent transactions modify the same logical value and one transaction's change overwrites the other's change.

Suppose an inventory row contains 10 units. Two transactions each sell one unit. If each

transaction reads 10, calculates 9, and then writes 9, the final quantity is 9 rather than 8.

```
-- T1 reads quantity = 10.  
-- T2 reads quantity = 10.  
-- T1 writes quantity = 9 and commits.  
-- T2 writes quantity = 9 and commits.
```

The database has lost one of the two decrements. This problem is especially common when an application reads a value, performs arithmetic in application code, and later writes the computed value.

A safer design uses an atomic update:

```
UPDATE inventory  
SET quantity = quantity - 1  
WHERE product_id = 50  
    AND quantity >= 1;
```

The application should verify that exactly one row was updated. The database performs the arithmetic against the row's current value, subject to the engine's concurrency rules. If the quantity is insufficient, the update affects zero rows.

Atomic updates are often preferable to a separate read followed by a write. They do not, however, solve every concurrency problem, such as a business rule involving several different rows.

5.5 Write Skew

Write skew is a subtler anomaly that can occur when concurrent transactions read overlapping data but update different rows. Each transaction may preserve a business rule according to the data it read, while their combined changes violate that rule.

Suppose a hospital requires at least one doctor to remain on call. Doctors A and B are both on call. Each transaction checks that the other doctor is on call and then turns off its own doctor's status.

```
-- T1 reads: A is on call, B is on call.  
-- T2 reads: A is on call, B is on call.  
  
-- T1 updates A to off call.  
-- T2 updates B to off call.  
  
-- Both transactions commit under an isolation  
-- level that permits this write skew.
```

The final state has no doctor on call. Neither transaction overwrote the other's row, so simply locking the rows being updated may not prevent the anomaly.

Possible solutions include using serializable isolation, explicitly locking a shared coordination row, or redesigning the schema so that the business rule can be enforced through a suitable database constraint. This example illustrates why concurrency correctness cannot always be achieved by examining each updated row independently.

6 Transaction Isolation Levels

SQL defines four commonly discussed isolation levels: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`. They represent different trade-offs between concurrency and protection from anomalies. Real database implementations may offer additional options, and the behavior of a named level can differ across engines.

The isolation level is not simply a switch between correct and incorrect execution. Weaker levels may be appropriate for certain approximate reports or workloads, while stronger levels may be necessary for financial transfers, reservations, and other operations that enforce business invariants. The application must understand which anomalies are acceptable.

6.1 `READ UNCOMMITTED`

`READ UNCOMMITTED` is the weakest standard isolation level. It permits dirty reads in the SQL isolation model, in addition to other anomalies allowed by weaker isolation levels. A transaction may observe changes that another transaction subsequently rolls back.

This level may appear attractive when performance is important and perfectly consistent results are unnecessary. However, it is inappropriate for decisions that depend on trustworthy committed data. Even a reporting query can produce misleading results if it observes temporary values during an update.

Some database engines treat `READ UNCOMMITTED` as `READ COMMITTED`, or otherwise restrict its behavior. Therefore, the actual guarantees must be checked for the chosen database system.

6.2 `READ COMMITTED`

`READ COMMITTED` prevents dirty reads: a transaction reads only data that has been committed. It is a common default isolation level and offers a useful balance between concurrency and consistency.

However, each statement may observe a different committed state. Another transaction may update a row between two reads, or insert new rows that satisfy a query predicate. Consequently, repeated queries within one transaction may produce different results. Depending on the database implementation, concurrent updates can also require explicit protection against lost updates or other application-level anomalies.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN;

SELECT balance
FROM accounts
WHERE account_id = 1;

-- Other transactions may commit changes here.

SELECT balance
FROM accounts
WHERE account_id = 1;

COMMIT;
```

The exact placement of the isolation-level command and the transaction-start command varies by SQL dialect. The example illustrates the intended configuration rather than a universally portable sequence.

6.3 REPEATABLE READ

REPEATABLE READ provides stronger protection than **READ COMMITTED**. Once a transaction has read a row, its subsequent reads of that row should not observe an intervening change that violates the isolation guarantee. This prevents nonrepeatable reads under the standard model.

Phantom-read protection and other concurrency guarantees depend on the implementation. Some database systems use snapshot-based execution, while others use locks or a combination of techniques. A snapshot-based implementation may prevent many anomalies without holding ordinary read locks for the entire transaction, but it may still permit write skew unless it also detects the relevant conflicts.

Do not assume that a database's **REPEATABLE READ** level provides the same guarantees as another database's level with the same name. When business correctness depends on a particular concurrency guarantee, consult the engine's documentation and test the relevant transaction pattern.

6.4 SERIALIZABLE

SERIALIZABLE is the strongest standard isolation level. Its purpose is to ensure that committed transactions behave as if they had executed in some serial order, one transaction at a time. The database may still execute operations concurrently, provided the outcome is equivalent to a valid serial execution.

Serializable execution prevents the concurrency anomalies that would violate this guarantee, including problematic write skew. The database might enforce serializability using strict

locking, predicate or range locks, multiversion concurrency control, conflict detection, or other techniques.

Serializable transactions can be more expensive. They may block other transactions, encounter deadlocks, or fail with serialization errors that require the application to retry the transaction. Thus, an application must be prepared for a transaction to fail even when its SQL statements are individually valid.

6.5 Comparison of Isolation Levels

The following table summarizes the traditional SQL isolation model. The entries describe the standard guarantees; actual behavior can vary with the database implementation.

Isolation level	Dirty reads	Nonrepeatable reads	Phantom reads
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	No	Possible	Possible
REPEATABLE READ	No	No	Possible in the standard model
SERIALIZABLE	No	No	No

This table does not completely characterize all concurrency anomalies. In particular, write skew and lost updates require additional attention. A database's implementation of repeatable read may prevent more anomalies than the minimum standard guarantee, and some engines use mechanisms that do not fit neatly into a simple locking-versus-isolation distinction.

7 An Example: Incorrect Results Under READ COMMITTED

Suppose a university allows a student to register for a course only if fewer than 30 students are currently enrolled. The application checks the enrollment count and inserts a registration if the count is below the limit.

Assume there are initially 29 students enrolled. Two transactions execute concurrently:

```
-- T1
BEGIN;
SELECT COUNT(*)
FROM registrations
WHERE course_id = 10;
-- Returns 29.

-- T2
BEGIN;
```

```
SELECT COUNT(*)
FROM registrations
WHERE course_id = 10;
-- Returns 29.

-- Both transactions conclude that a seat is available.

-- T1 inserts a registration and commits.
-- T2 inserts a registration and commits.
```

The final count is 31, violating the capacity limit. The individual count queries were correct when they executed, and both transactions read committed data. The problem is that the business decision depended on a condition over a set of rows, and both transactions acted on the same earlier result.

Using **SERIALIZABLE** can prevent both transactions from committing if their combined outcome cannot be represented by a valid serial execution. One transaction may be blocked or aborted, depending on the database implementation.

Another solution is to maintain a course row containing the remaining capacity and update it atomically:

```
UPDATE courses
SET remaining_seats = remaining_seats - 1
WHERE course_id = 10
    AND remaining_seats > 0;
```

The application verifies that one row was updated before inserting the registration, and performs both operations in the same transaction. The course row becomes a coordination point: concurrent attempts to claim the final seat cannot both successfully decrement the same remaining-seat value from zero or one. A suitable constraint can additionally enforce that remaining seats never become negative.

This design may serialize attempts to reserve seats for the same course, but it avoids relying on a separate count followed by an unprotected insert. For systems where registration correctness is critical, this explicit design is often easier to reason about than an application-level check alone.

8 Locking and Its Impact on Database Usage

8.1 What Is a Lock?

A lock is a mechanism that restricts access to data while a transaction is using or modifying it. Database systems commonly use shared locks for certain reads and exclusive locks for writes. Shared locks can generally coexist with other shared locks, while an exclusive lock

conflicts with other incompatible locks.

For example, a transaction updating an account may obtain an exclusive lock on the affected row. Another transaction attempting to update the same row may have to wait until the first transaction commits or rolls back. This prevents the two updates from simultaneously overwriting each other without coordination.

Locks can be applied at different granularities, such as rows, pages, tables, or ranges of index keys. Fine-grained locks permit more concurrent activity but require additional bookkeeping. Coarse locks are simpler but can prevent otherwise independent operations from proceeding. Some systems can escalate many fine-grained locks into a larger lock.

8.2 Blocking and Waiting

Consider two transactions operating on the same account.

```
-- T1
BEGIN;
UPDATE accounts
SET balance = balance - 100
WHERE account_id = 1;

-- T1 keeps the transaction open.

-- T2
UPDATE accounts
SET balance = balance + 50
WHERE account_id = 1;
-- T2 may wait for T1 to finish.
```

If T1 holds an incompatible lock, T2 may block until T1 commits or rolls back. Blocking is normal and is often necessary to preserve correctness. However, a long-running transaction can delay many other users, reduce throughput, and increase application response times.

For example, an application might begin a transaction, update a row, and then wait for a user to fill out a form before committing. During that time, the transaction may retain locks even though it is not actively executing SQL. This is a poor design. Transactions should generally be kept short, and user interaction should take place outside the period in which database locks are held.

Not every read blocks a write. Some database engines use multiversion concurrency control (MVCC), which allows readers to observe an older committed version while a writer updates the current version. Nevertheless, writers may still block other writers, and some operations require stronger locks.

8.3 Deadlocks

A deadlock occurs when transactions wait for one another in a cycle. For example, T1 locks account A and then requests account B, while T2 locks account B and then requests account A. Neither can proceed until one releases its lock.

```
-- T1
BEGIN;
UPDATE accounts
SET balance = balance - 10
WHERE account_id = 1;
-- T1 holds a lock on account 1.

-- T2
BEGIN;
UPDATE accounts
SET balance = balance - 20
WHERE account_id = 2;
-- T2 holds a lock on account 2.

-- T1 now tries to update account 2.
-- T2 now tries to update account 1.
```

The database typically detects the deadlock and aborts one transaction, allowing the other to continue. The aborted transaction must be rolled back, and the application may need to retry it.

A common way to reduce deadlocks is to acquire locks in a consistent order. For example, a transfer routine could always lock the account with the smaller identifier first, regardless of the direction of the transfer. This reduces the chance that transactions acquire the same locks in opposite orders. Keeping transactions short and accessing only necessary rows also helps.

Deadlocks are not the same as ordinary blocking. In blocking, a transaction waits for another transaction to finish. In a deadlock, the waiting relationships form a cycle, so the database generally must break that cycle by aborting a transaction or using another deadlock-resolution mechanism.

8.4 Explicit Locking

Applications can sometimes request locks explicitly when they need to read data and then make a decision based on that data. A common SQL construct is `SELECT ... FOR UPDATE`, supported by many relational database systems.

```
BEGIN;

SELECT balance
```

```
FROM accounts
WHERE account_id = 1
FOR UPDATE;

-- Application checks the balance.

UPDATE accounts
SET balance = balance - 100
WHERE account_id = 1;

COMMIT;
```

The locking read requests protection against conflicting modifications while the transaction performs its work. Other transactions that need incompatible access may have to wait. The exact lock behavior and syntax are database-specific, and some engines use different mechanisms under snapshot-based isolation.

Explicit locking can be appropriate when an application must examine a row and then conditionally modify it. However, it is not a substitute for understanding the transaction's isolation level. Locking one row does not necessarily protect a condition involving many rows or the absence of a matching row. Predicate locks, range locks, a coordination row, or serializable isolation may be required.

9 Multiversion Concurrency Control (MVCC)

Multiversion concurrency control is a technique in which the database maintains multiple versions of data so that transactions can read a version appropriate to their visibility rules. Instead of requiring every reader to wait for a writer, an MVCC-based system can often allow a reader to access a previously committed version while a concurrent transaction modifies the current version.

For example, a report may read the committed balance of an account while another transaction is updating that account. Depending on the isolation level, the report can continue reading the appropriate visible version without blocking on the write. This is one reason MVCC can provide good read performance in workloads with many concurrent readers and writers.

MVCC does not eliminate all locks or all waiting. Updates to the same row may still conflict, and transactions may require locks for coordination, schema changes, or other operations. Old versions also need to be cleaned up after they are no longer visible to active transactions. Long-running transactions can delay this cleanup and cause storage growth or additional maintenance work.

Snapshot-based isolation is closely related to MVCC. A transaction may see a consistent snapshot of committed data as of a particular point in time. This can make reports more consistent than executing each query against a newly committed state. However, snapshot

consistency alone is not always equivalent to serializability: concurrent transactions can sometimes commit individually valid changes that jointly violate a business rule. The database's exact isolation guarantees still matter.

10 Alternative Designs That Reduce Lock Contention

10.1 Use Atomic SQL Statements

A common source of unnecessary contention and incorrect results is the read-modify-write pattern implemented as separate application operations. An application reads a value, computes a replacement, and later writes it. This creates an opportunity for another transaction to modify the value in between.

When possible, express the operation directly in SQL:

```
UPDATE inventory
SET quantity = quantity - 1
WHERE product_id = 50
  AND quantity > 0;
```

The database performs the conditional modification as one statement. The application checks the affected-row count to determine whether the operation succeeded. This design is usually simpler and safer than fetching the quantity, decrementing it in application code, and writing the result back.

Atomic updates can reduce the amount of application-side coordination, but they do not guarantee that every multistep business operation is correct. If a sale must also create an order and a payment record, those changes should generally be part of the same transaction.

10.2 Use Optimistic Concurrency Control

Optimistic concurrency control allows an application to perform work without holding a long-lived exclusive lock, then checks whether the data changed before the update is accepted. One common implementation uses a version column.

```
UPDATE inventory
SET quantity = 9,
  version = version + 1
WHERE product_id = 50
  AND version = 7;
```

The application must have read version 7 before issuing this update. If another transaction has already changed the row's version, the update affects zero rows. The application detects the conflict and can reload the data, recompute the operation, or report a conflict to the user.

Optimistic concurrency is useful when conflicts are uncommon and transactions do not need to hold locks during lengthy application processing. It does not mean that the final database write requires no locking; the database still has to coordinate concurrent writes. It also requires care when a business rule depends on multiple rows. A version check on one row does not automatically protect a condition involving other rows.

10.3 Keep User Interaction Outside Transactions

A transaction should not remain open while an application waits for a person to approve a payment, complete a form, or respond to a notification. Instead, the application should gather input first, then open a transaction, validate the current database state, perform the required updates, and commit promptly.

For example, an online booking application should not lock a vehicle while a customer spends five minutes entering personal information. It can collect the information first and then attempt to reserve the vehicle in a short transaction. If another customer has already reserved it, the transaction should detect the conflict and return an appropriate response.

Sometimes the application needs to preserve a reservation across a longer business process. In that case, it can use an explicit reservation or lease with an expiration time rather than holding a database transaction open. The reservation itself is committed data, not a long-running database lock. The application must then correctly manage expiration, cancellation, and competing claims.

10.4 Use Constraints and Coordination Records

When a business rule can be represented as a database constraint, the constraint is often a valuable final line of defense. A unique constraint can prevent duplicate registrations for the same student and course, for example.

```
ALTER TABLE registrations
ADD CONSTRAINT unique_student_course
UNIQUE (student_id, course_id);
```

Even if two concurrent transactions both check that a registration does not exist, the unique constraint prevents both from committing duplicate registrations. One transaction may fail and should handle that outcome appropriately.

For business rules that cannot be expressed as a simple constraint, a dedicated coordination row can provide a point at which competing transactions synchronize. The course-capacity example can use a single course record with a remaining-seat counter. This approach may serialize updates for the same course, but transactions for different courses can proceed independently. A well-designed coordination record limits the scope of contention rather than eliminating all contention.

10.5 Split Long Operations Into Short Transactions

Some tasks process thousands or millions of records. Performing the entire task in one transaction may hold locks for too long, consume substantial resources, delay cleanup of old row versions, and make recovery expensive.

If the operation can be safely divided, the application can process smaller batches and commit each batch separately. For example, a data-cleanup job might update a few thousand rows per transaction rather than locking a large portion of a table for a long period.

Batching changes the atomicity guarantee. If the process fails after five batches, those five batches may already be committed. Therefore, the operation should be designed to resume safely, record progress, or compensate for partial completion where necessary. Batching is not appropriate when the entire operation must be all-or-nothing.

11 Retries and Idempotency

Concurrency control sometimes causes transactions to fail rather than allowing them to commit with a potentially incorrect result. For example, a serializable transaction may encounter a serialization failure, or a database may abort a transaction to resolve a deadlock. These are expected possibilities in concurrent applications.

Applications should distinguish retryable transaction failures from permanent errors such as invalid input or a violated business rule. When a failure is retryable, the application generally needs to restart the entire transaction from the beginning, including its reads and business-rule checks. Retrying only the final SQL statement may reuse assumptions that are no longer valid.

An idempotent operation can be safely repeated without producing unintended additional effects. For example, an API might assign each payment request a unique identifier and store that identifier with the resulting transaction. If the client times out and retries the request, the application can detect that the payment was already processed rather than charging the customer twice.

Idempotency is particularly important when the application cannot tell whether a transaction committed before a network connection failed. A client that loses its connection after sending **COMMIT** cannot always conclude that the transaction rolled back. It should verify the outcome or retry through a mechanism designed to recognize duplicate requests.

12 Transactions Across Multiple Systems

A database transaction ordinarily protects operations within one transactional resource. If an application also calls an external payment service, sends a message to a queue, or updates another database, the local database transaction does not automatically make all those

operations atomic.

Consider an application that commits an order to its database and then crashes before publishing an order-created event. The order exists, but downstream systems never receive the event. Reversing the order of the operations introduces the opposite problem: the event might be published even though the database transaction rolls back.

One common solution is the transactional outbox pattern. The application inserts both the order and an outbox message into the same database transaction. A separate process later publishes the message and records its delivery status. The publisher may send a message more than once, so consumers should be designed to handle duplicates. This approach does not provide instantaneous global atomicity, but it reliably connects a committed database change to eventual message delivery.

Distributed transactions and two-phase commit are another option when multiple transactional resources must participate in one coordinated commit. However, distributed transaction protocols add operational complexity and may keep resources unavailable while participants wait for the outcome. For many applications, an outbox combined with idempotent consumers is a more practical design.

13 Practical Transaction Design Guidelines

When writing transactional application code, begin by identifying the complete logical unit of work. Every database change that must succeed or fail together should be included in the same transaction. Validate inputs, use constraints, and check the results of conditional updates rather than assuming that a statement affected the intended rows.

Keep transactions as short as possible. Perform expensive calculations, network calls, and user interaction outside the transaction whenever doing so preserves correctness. Access records in a consistent order when practical, and avoid locking rows or tables that the operation does not need. These practices reduce blocking and deadlock risk.

Choose an isolation level based on the actual business invariant. `READ COMMITTED` is often sufficient when individual statements are independent and atomic updates or constraints enforce the important rules. Snapshot-based repeatable reads can be useful for consistent multi-query reads. Serializable isolation is appropriate when concurrent transactions must preserve more complex invariants and the application can handle blocking or retryable failures.

Finally, test concurrency explicitly. A test that runs a single transaction at a time will not reveal many concurrency bugs. Use multiple database sessions to reproduce overlapping reads and writes, and test the behavior when transactions commit, roll back, block, or fail. Concurrency correctness is a property of the combined execution of transactions, not just the correctness of individual SQL statements.

14 Summary

A transaction groups database operations into a logical unit and provides a controlled way to commit or undo changes. The ACID properties describe atomicity, consistency, isolation, and durability. These properties are related but distinct: atomicity protects the transaction's changes, consistency preserves required invariants, isolation controls interactions with concurrent work, and durability preserves committed changes.

Isolation levels determine which concurrent executions are permitted. **READ COMMITTED** prevents dirty reads but allows repeated queries to observe different committed states. Stronger isolation levels provide additional protection, and **SERIALIZABLE** requires committed outcomes to be equivalent to some serial execution. No isolation-level name should be assumed to have identical implementation details in every database engine.

Locks and MVCC are mechanisms for coordinating concurrent work. Locks can cause blocking and deadlocks, while MVCC can let readers observe suitable versions without waiting for writers. Good database design combines the right isolation guarantees with atomic SQL statements, constraints, short transactions, optimistic concurrency where appropriate, and carefully designed retries.

The central lesson is that a correct transaction must do more than execute valid SQL statements. It must preserve the application's business rules even when other transactions execute concurrently, fail unexpectedly, or retry after an error.