Cockroach Labs

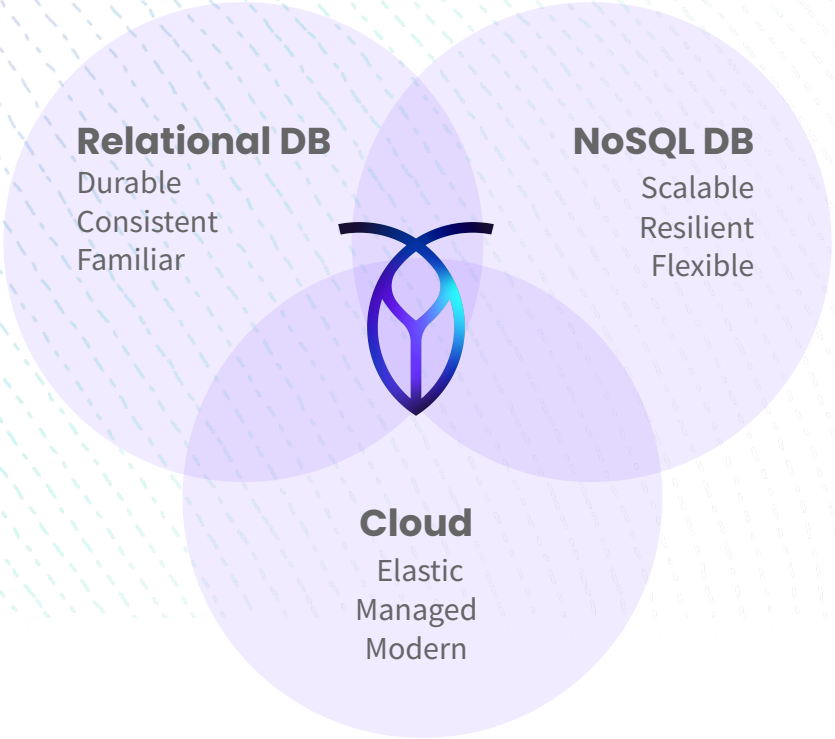
Isolation levels without the anomaly table

Ben Darnell, Chief Architect, Cockroach Labs

PGConf India, 2024-03-01

About CockroachDB

Built from ground up to meet the demands of today's data-driven world in the cloud



CockroachDB

An agile, distributed database architected and built for the cloud

---

Fully Managed Service

Guaranteed Transactions

Inherent Resilience & Scale

Familiar, Consistent SQL

..in a **truly globally-distributed database**

Agenda

- Isolation levels and anomalies
- What does isolation mean for my code?
- Differences between databases
- How to choose an isolation level

Isolation levels and anomalies

SQL Isolation Levels

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

The Anomaly Table

The SQL standard and PostgreSQL-implemented transaction isolation levels are described in Table 13.1.

Table 13.1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

It’s incomplete

- Researchers have identified more anomalies
  - Write skew (Berenson et al, 1995)
  - Anti-dependency cycle and more (Adya, 1999)
- And more isolation levels
  - Snapshot isolation (Berenson)
  - Monotonic view and Consistent View (Adya)
- And implemented old levels in new ways
  - Serializable Snapshot Isolation
  - SQL Server’s READ_COMMITTED_SNAPSHOT option

No one thinks this way

“My application can tolerate phantom reads but not write skew, so REPEATABLE READ is the best isolation level for it.”

What does isolation mean for my code?

Low isolation needs explicit locks

- In levels below `SERIALIZABLE`, must sometimes use `SELECT FOR UPDATE`
  - Or `FOR SHARE`
- Missing locks can allow transactions to incorrectly overwrite each other's data
- Too many locks hurts performance

High isolation causes aborts and retries

- Sometimes two transactions conflict and one must be aborted
  - Deadlocks can happen in any isolation level
  - More common as isolation level increases
- Application must catch error and retry to avoid user-visible failure

Example

- Balance is read in several places
- What if it changes between SELECT and UPDATEs?

```
def transfer(db_conn, from_account, to_account, amount):
    with db_conn.transaction() as txn:
        balance = txn.execute("SELECT balance FROM accounts WHERE id=?",
                              from_account).fetchone()[0]
        if balance < amount:
            return
        txn.execute("UPDATE accounts SET balance=balance-? WHERE id=?",
                    amount, from_account)
        txn.execute("UPDATE accounts SET balance=balance+? WHERE id=?",
                    amount, to_account)
```

Example: READ COMMITTED

- Each statement sees different balance values
- Balance could become negative
- Must add `FOR UPDATE` to `SELECT` statement to fix

```
def transfer(db_conn, from_account, to_account, amount):
    with db_conn.transaction() as txn:
        balance = txn.execute("SELECT balance FROM accounts WHERE id=?",
                              from_account).fetchone()[0]
        if balance < amount:
            return
        txn.execute("UPDATE accounts SET balance=balance-? WHERE id=?",
                    amount, from_account)
        txn.execute("UPDATE accounts SET balance=balance+? WHERE id=?",
                    amount, to_account)
```

Example: SERIALIZABLE

- First update statement raises error
  - `could not serialize access due to concurrent update`
- Can catch error and retry
- Error is necessary because database doesn't know what happened in the Python `if` statement

```
def transfer(db_conn, from_account, to_account, amount):
    with db_conn.transaction() as txn:
        balance = txn.execute("SELECT balance FROM accounts WHERE id=?",
                              from_account).fetchone()[0]
        if balance < amount:
            return
        txn.execute("UPDATE accounts SET balance=balance-? WHERE id=?",
                    amount, from_account)
        txn.execute("UPDATE accounts SET balance=balance+? WHERE id=?",
                    amount, to_account)
```

Example: REPEATABLE READ

- Databases differ
- This example usually works like SERIALIZABLE
  - PostgreSQL
  - CockroachDB
  - SQL Server
  - Oracle
- Sometimes it's like READ COMMITTED
  - MySQL

```
def transfer(db_conn, from_account, to_account, amount):
    with db_conn.transaction() as txn:
        balance = txn.execute("SELECT balance FROM accounts WHERE id=?",
                              from_account).fetchone()[0]
        if balance < amount:
            return
        txn.execute("UPDATE accounts SET balance=balance-? WHERE id=?",
                    amount, from_account)
        txn.execute("UPDATE accounts SET balance=balance+? WHERE id=?",
                    amount, to_account)
```

Differences between databases

“Allowed, but not in PG”

The SQL standard and PostgreSQL-implemented transaction isolation levels are described in Table 13.1.

Table 13.1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Write skew and snapshot isolation

- SQL standard says that `REPEATABLE READ` permits **phantom reads**
- PostgreSQL's `REPEATABLE READ` doesn't permit phantom reads
  - But it does permit **write skew**
- This means it's actually `SNAPSHOT ISOLATION`

Locking vs multi-versioning (MVCC)

- Two approaches to isolating reads
  - Shared lock on all records accessed until end of transaction
  - Store multiple versions, overwritten records are not immediately deleted
- Older DBs mostly used locking, newer ones mostly MVCC
- SQL standard tried to be implementation-independent
  - But “phantom reads” are mainly relevant to locking implementations

`REPEATABLE READ` is poorly defined…

- …and that's fine
- Don't look to closely at phantom reads and write skew
- Reads repeat, and this is the only universal guarantee

How to choose an isolation level

Performance? It's complicated

- High isolation has high *variance*
  - A small fraction of transactions take twice as long as usual
- Low isolation is often faster, but not always
  - In CockroachDB, TPC-C is slightly faster in `SERIALIZABLE` than `READ COMMITTED`
  - In other benchmarks, `READ COMMITTED` can be much faster
- Defaults matter
  - Default isolation level gets more optimization effort
  - CockroachDB defaults to `SERIALIZABLE` so we've optimized it to be competitive with RC
  - Other DB's `SERIALIZABLE` implementations usually have worse performance

Use `READ UNCOMMITTED` if…

- Consistently low latency is more important than the right answer
- And your database implements `READ UNCOMMITTED`
  - Most don't today
  - SQL Server does, but consider `read_committed_snapshot` instead

Use `READ COMMITTED` if…

- Consistently low latency (at 99+ percentile) is important
- It is difficult to add retry loops to the application
  - But it's easier to add `FOR UPDATE` where needed
- It's the default for your database
  - Probably the most optimized

Use `REPEATABLE READ` if…

- The transaction is read-only
  - Read-only `READ COMMITTED` “transactions” don’t really do anything
  - `SERIALIZABLE` and `REPEATABLE READ` are equivalent for read-only transactions
- Portability to other databases is not important
  - Implementations of `REPEATABLE READ` differ more than RC or `SERIALIZABLE`

Use SERIALIZABLE if…

- Data accuracy is paramount
- You are able to use abstractions to manage retry loops
- Another system is already handling retries
    - Mobile apps often retry on network errors, so they can retry on DB errors too

Conclusion

Conclusion

- Anomalies aren’t the most important thing
- Instead, pick isolation level based on
  - Blocking behavior
  - Client retries

CockroachDB

The most highly evolved database on the planet

Scale Fast | **Survive Anything** | **Thrive Everywhere**