# Transactions

## What are transactions?

Transactions are an abstraction used by some databases that provide ACID guarantees about queries - each write in the transaction will either be committed or aborted entirely without any side effects

ACID: Atomicity, Consistency, Isolation, Durability

## ACID explained

Atomicity: If a client makes several writes, but a fault occurs after only some of the writes are completed, the existing completed writes will be rolled back (can be implemented with write ahead log for crash recovery)

Consistency: The application can rely on the properties of the database to ensure that invariants about the data will hold (in the face of faults)

Isolation: Concurrently executing transactions are isolated from one another (serializability), each transaction can pretend it is the only one running on the database

Durability: Once a transaction is completed, the data will never be forgotten, even in the face of faults

## Implementing Serializable Isolation

- Actual Serial Execution
- Two Phase Locking
- Serializable Snapshot Isolation

## Actual Serial Execution

Idea: implement all database queries on a single thread

This is possible under the following conditions:

- In memory database
- Use a stored procedure
- Attempt to keep all transactions limited to one partition

Pros: Very simple to implement, non analytics transactions are mostly short

Cons: Throughput limited to a single CPU core, have to use stored procedures (hard to version control and can be in weird languages)

## Two Phase Locking

Idea: Each object has a lock on it, which can be held in either shared mode or exclusive mode.

Multiple transactions can concurrently read from a row if they are holding the lock in shared mode, but if they want to write to it they must grab the lock in exclusive mode.  They can only do this if no other transactions are currently holding a shared or exclusive lock on the object.

## Two Phase Locking and Predicate Locks

Let’s imagine that we are running a transaction and want to book a meeting room. We can first query the database to see if anyone else has already booked the room, and if not we will book it ourself.  However, after we query for existing bookings, another transaction successfully adds a row to the database taking the booking that we wanted.  Since this row had yet to exist, we could not have put a lock on it, thus ruining our serializability.

To solve this phenomena, we need predicate locks!

## Predicate Locks

Apply to all objects matching a given search condition, perhaps even ones that don’t yet exist!

Issue: Performs very poorly, have to go through a bunch of rows checking to see if they match the conditions to apply a lock to.

Alternative: Index range locking

## Index Range Locking

| Id | Meeting Room | Day |
|---|---|---|
| 1 | Your mom's | Everyday |
| 2 | The Oval Office | Monday |
| 3 | The Oval Office | Wednesday |
| 4 | Google HQ | Friday |

| | |
|---|---|
| Your mom's | 1 |
| Oval Office | 2, 3 |
| Google HQ | 4 |

We want to book the Oval office for Tuesday, so we would write a query to put a predicate lock on all Oval Office meetings for Tuesday.  But this could take a while - instead we could just use our index to put a lock on all Oval Office meetings quickly, since it is a superset of those in the Oval Office and on Tuesday.

## Index Range Locking

| Id | Meeting Room | Day |
|---|---|---|
| 1 | Your mom’s | Everyday |
| 2 | The Oval Office | Monday |
| 3 | The Oval Office | Wednesday |
| 4 | Google HQ | Friday |

| | |
|---|---|
| Your mom’s | 1 |
| Oval Office | 2, 3 |
| Google HQ | 4 |

We want to book the Oval office for Tuesday, so we would write a query to put a predicate lock on all Oval Office meetings for Tuesday.  But this could take a while - instead we could just use our index to put a lock on all Oval Office meetings quickly, since it is a superset of those in the Oval Office and on Tuesday.

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

```
jordan_attractiveness: 8
shared_lock: []
exclusive_lock: []
```

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 8
shared_lock: []
exclusive_lock: []

T1: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks. The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 8
shared_lock: [T1]
exclusive_lock: []

T1: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 8
shared_lock: [T1]
exclusive_lock: []

T1: read jordan_attractiveness then add 1 to it

T2: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks. The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 8
shared_lock: [T1, T2]
exclusive_lock: []

T1: read jordan_attractiveness then add 1 to it

T2: read jordan_attractiveness then add 1 to it

# Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 8
shared_lock: [T1, T2]
exclusive_lock: []

T1: read jordan_attractiveness then add 1 to it

T2: read jordan_attractiveness then add 1 to it

Deadlock!  Both transactions want the exclusive lock but neither can get it because multiple transactions are holding the shared lock

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 8
shared_lock: [T1]
exclusive_lock: []

T1: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 8
shared_lock: []
exclusive_lock: [T1]

T1: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 9
shared_lock: []
exclusive_lock: []

T2: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

```
jordan_attractiveness: 9
shared_lock: [T2]
exclusive_lock: []
```

T2: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

jordan_attractiveness: 9
shared_lock: []
exclusive_lock: [T2]

T2: read jordan_attractiveness then add 1 to it

## Two Phase Locking Evaluated

Terrible performance because of frequent deadlocks.  The database must detect when transactions are having a deadlock, and abort one of them, let the other finish, and then retry the aborted one.

```
jordan_attractiveness: 10
shared_lock: []
exclusive_lock: []
```

## Serializable Snapshot Isolation

Idea: Let everything run concurrently as if there was no locking, and only revert a transaction if a concurrency bug has been detected.

All reads occur from a snapshot of the database:

- If an uncommitted write occurred before the read, need to check if that write has since been committed by the time we want to make our write (abort if so)
- Keep track of which transactions have read an item, and if another transaction writes to said item, abort all of the transactions that read it

## Serializable Snapshot Isolation Evaluated

Pros:

- If not many concurrency issues on the dataset, far faster than 2 phase locking, as it just allows the threads to run uninhibited by locks

Cons:

- If many transactions are resulting in concurrency bugs, there will be many retries, and as a result 2 phase locking may be better

## Transactions Conclusion

Transactions are a potentially very useful abstraction of multiple operations that needed to be bundled together in a database.  They either all succeed, or all fail (with no side effects).  However, they come at a great performance cost, and as a result many databases have chosen to implement weaker forms of isolation, which we will discuss in the next video.

## Isolation Implementations Conclusion

Actual Serial Execution:

- Very simple to implement and does not require any locks or reverting
- Throughput limited to single thread, have to keep data in memory, stored procedures

Two Phase Locking:

- Implementation used in most database, for a long time actual serial execution was not possible
- Poor performance due to unnecessary locking (not all transactions that touch the same objects result in concurrency bugs) and frequent deadlocks

Serializable Snapshot Isolation:

- Optimistic concurrency control leads to better performance than 2 phase locking in situations where concurrency bugs are infrequent, otherwise have to revert too many transactions