Lecture 15: Optimistic Concurrency Control

Recap

Basic T/O

- Txns read and write objects without locks.
- Every object X is tagged with timestamp of the last txn that successfully did read/write:
  - $W-TS(X)$ – Write timestamp on X
  - $R-TS(X)$ – Read timestamp on X
- Check timestamps for every operation:
  - If txn tries to access an object **from the future**, it aborts and restarts.

Partition-based T/O

- Split the database up in disjoint subsets called **horizontal partitions** (aka shards).
- Use timestamps to order txns for serial execution at each partition.
  - Only check for conflicts between txns that are running in the same partition.

Observation

- If you assume that conflicts between txns are **rare** and that most txns are **short-lived**, then forcing txns to wait to acquire locks adds a lot of overhead.
- A better approach is to optimize for the **no-conflict** case.

Today's Agenda

- Optimistic Concurrency Control
- Phantoms
- Isolation Levels

Optimistic Concurrency Control

Optimistic Concurrency Control

- The DBMS creates a private workspace for each txn.
  - Any object read is copied into workspace.
  - Modifications are applied to workspace.
- When a txn commits, the DBMS compares workspace **write set** to see whether it conflicts with other txns.
- If there are no conflicts, the write set is installed into the **global database**.

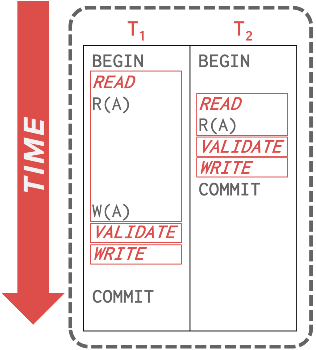
OCC Phases

- **Phase 1 – Read**:
  - Track the read/write sets of txns and store their writes in a private workspace.
- **Phase 2 – Validation**:
  - When a txn commits, check whether it conflicts with other txns.
- **Phase 3 – Write**:
  - If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.

 Optimistic Concurrency Control

OCC – Example

Schedule

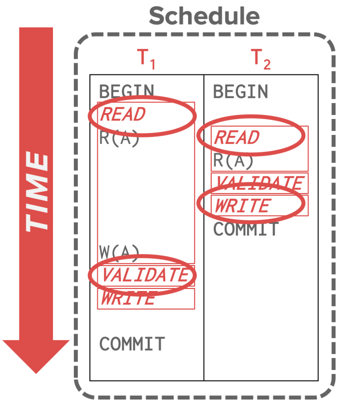


Database

Object	Value	W-TS
A	123	0
-	-	-

 Optimistic Concurrency Control

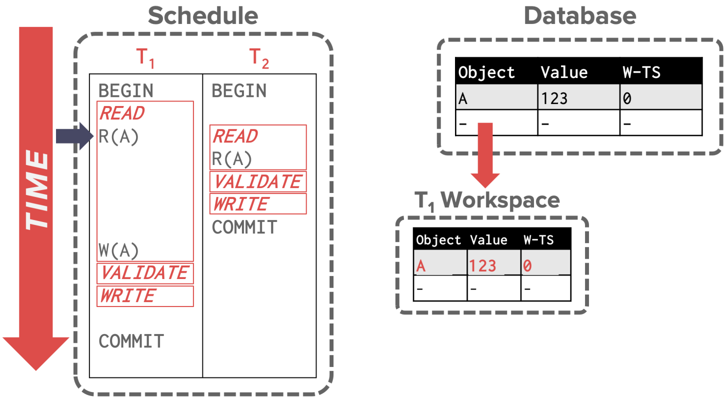
OCC – Example



Database

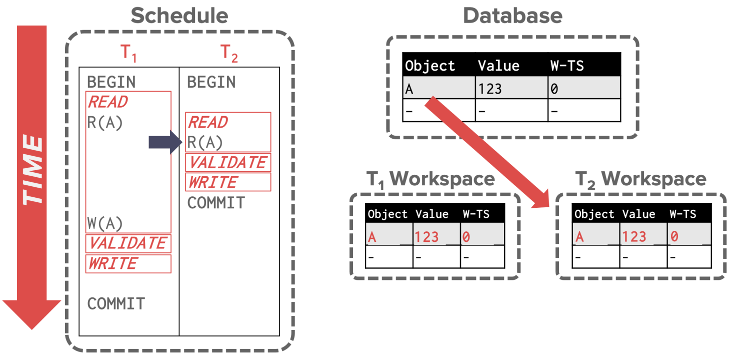
Object	Value	W-TS
A	123	0
–	–	–

OCC – Example

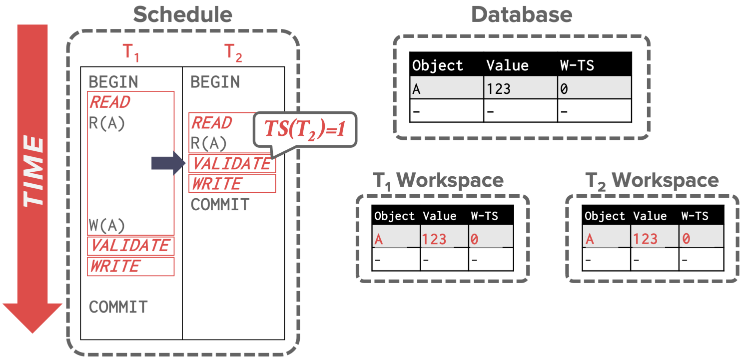


Schedule
TIME
T_1
T_2
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT
BEGIN
READ
R(A)
VALIDATE
WRITE
COMMIT
Database
Object Value W-TS
A 123 0
- - -
T_1 Workspace
Object Value W-TS
A 123 0
- - -

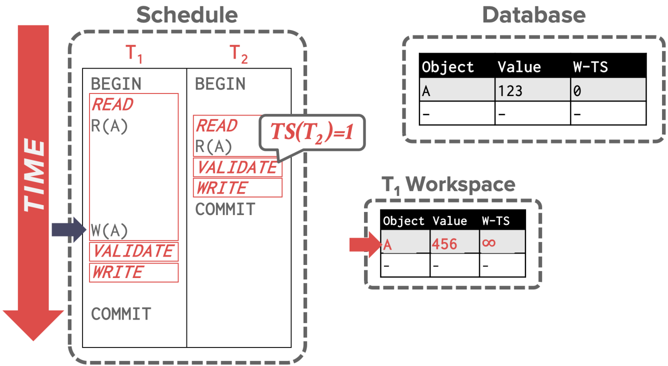
OCC – Example


Schedule
TIME
T1
T2
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT
BEGIN
READ
R(A)
VALIDATE
WRITE
COMMIT
Database
Object Value W-TS
A 123 0
– – –
T1 Workspace
Object Value W-TS
A 123 0
– – –
T2 Workspace
Object Value W-TS
A 123 0
– – –

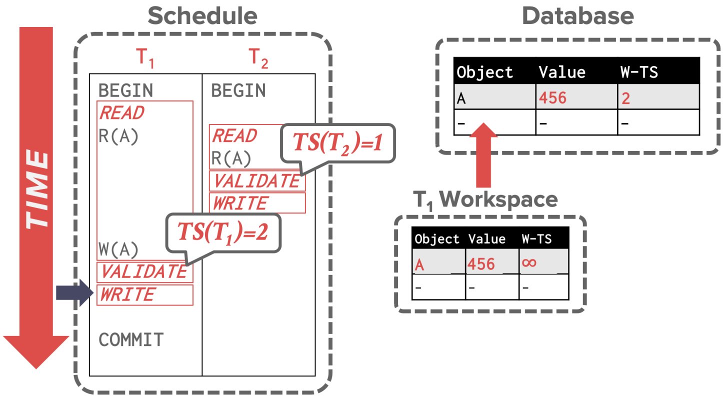
OCC – Example


Schedule
TIME
T1
T2
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT
BEGIN
READ
R(A)
VALIDATE
WRITE
COMMIT
TS(T2)=1
Database
Object Value W-TS
A 123 0
– – –
T1 Workspace
Object Value W-TS
A 123 0
– – –
T2 Workspace
Object Value W-TS
A 123 0
– – –

OCC – Example


Schedule
TIME
T1
T2
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT
BEGIN
READ
R(A)
VALIDATE
WRITE
COMMIT
TS(T2)=1
Database
Object Value W-TS
A 123 0
- - -
T1 Workspace
Object Value W-TS
A 456 ∞
- - -

OCC – Example


Schedule
Database
TIME
T1
T2
BEGIN
READ
R(A)
W(A)
VALIDATE
WRITE
COMMIT
BEGIN
READ
R(A)
VALIDATE
WRITE
TS(T2)=1
TS(T1)=2
Object Value W-TS
A 456 2
- - -
T1 Workspace
Object Value W-TS
A 456 ∞
- - -

OCC – Validation Phase

- The DBMS needs to guarantee only serializable schedules are permitted.
- T_i checks other txns for RW and WW conflicts and makes sure that all conflicts go one way (from older txns to younger txns).

OCC – Serial Validation

- Maintain global view of all active txns.
- Record read set and write set while txns are running and write into private workspace.
- Execute **Validation** and **Write** phase inside a protected **critical section**.

OCC – Read Phase

- Track the read/write sets of txns and store their writes in a private workspace.
- The DBMS copies every tuple that the txn accesses from the shared database to its workspace ensure repeatable reads.

OCC – Validation Phase

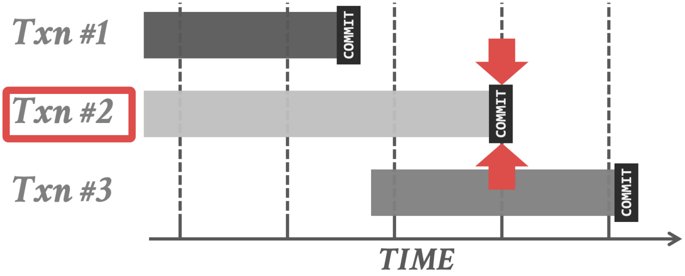
- Each txn's timestamp is assigned at the **beginning of the validation phase** (different from 2PL).
- Check the timestamp ordering of the committing txn with all other running txns.
- If $TS(T_i) < TS(T_j)$, then one of the following three scenarios must hold. . .

OCC – Validation Phase

- When the txn invokes COMMIT, the DBMS checks if it conflicts with other txns.
- Two methods for this phase:
  - Backward Validation
  - Forward Validation

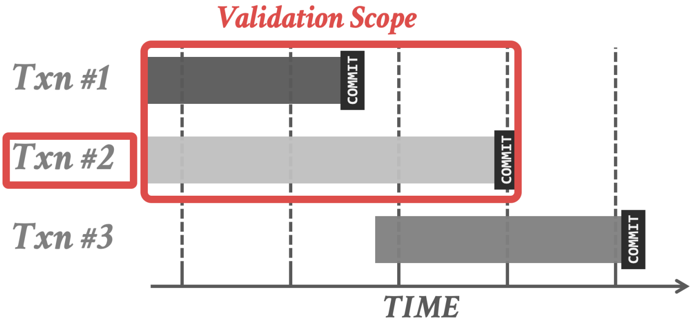
OCC – Backward Validation

- Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.



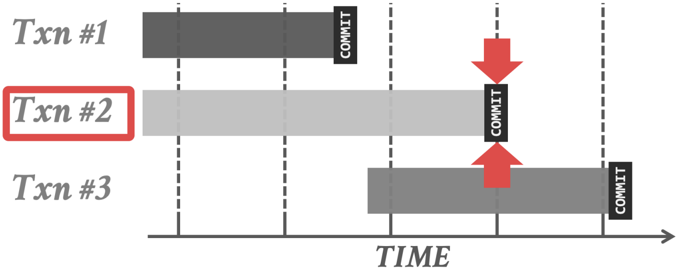
OCC – Backward Validation

- Check whether the committing txn intersects its read/write sets with those of any txns that have already committed.



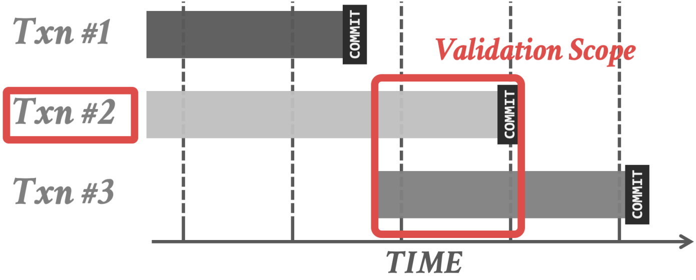
OCC – Forward Validation

- Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.



OCC – Forward Validation

- Check whether the committing txn intersects its read/write sets with any active txns that have not yet committed.

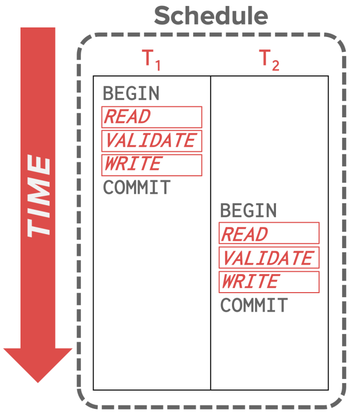


Optimistic Concurrency Control

OCC – Validation Step 1

- **Scenario 1:**
- T_i completes all three phases before T_j begins.

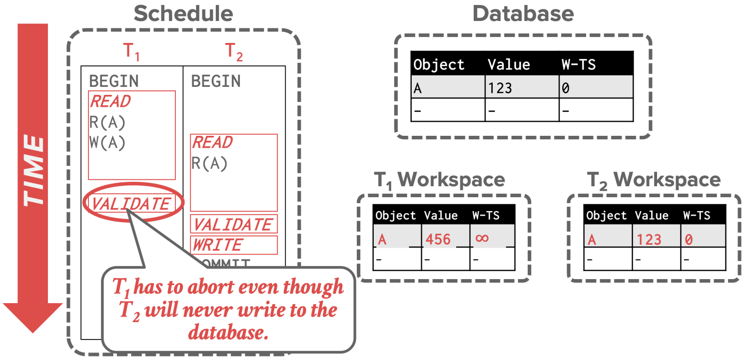
OCC – Validation Step 1



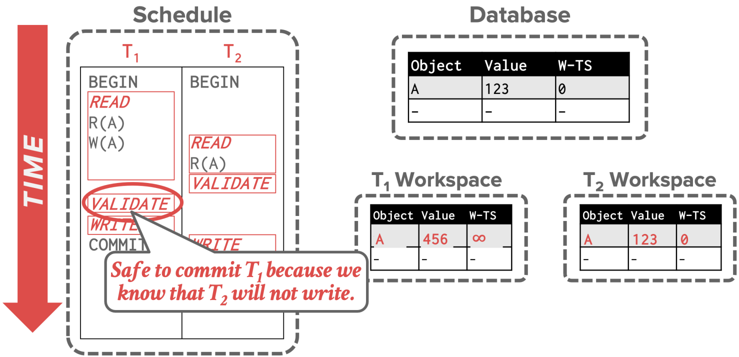
OCC – Validation Step 2

- **Scenario 2:**
- T_i completes before T_j starts its **Write** phase, and T_i does not write to any object read by T_j.
  - $WriteSet(T_i) \cap ReadSet(T_j) =$

OCC – Validation Step 2



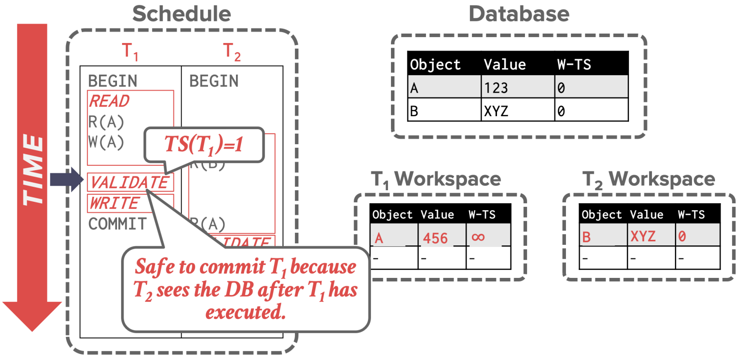
OCC – Validation Step 2



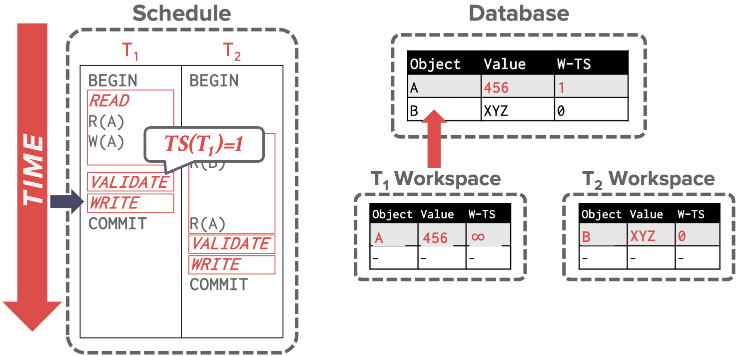
OCC – Validation Step 3

- **Scenario 3:**
- Ti completes its Read phase before Tj completes its Read phase
- And Ti does not write to any object that is either read or written by Tj:
  - WriteSet(Ti) $\cap$ ReadSet(Tj) =
  - WriteSet(Ti) $\cap$ WriteSet(Tj) =

OCC – Validation Step 3

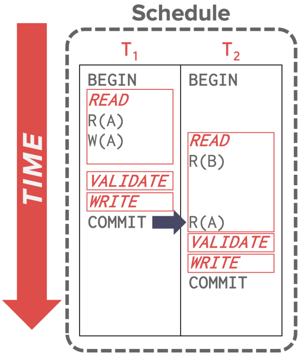


OCC – Validation Step 3



 Optimistic Concurrency Control

OCC – Validation Step 3



Database

Object	Value	W-TS
A	456	1
B	XYZ	0

T_2 Workspace

Object	Value	W-TS
B	XYZ	0
A	456	1

OCC – Observation

- OCC works well when the number of conflicts is low:
  - All txns are read-only (ideal).
  - Txns access disjoint subsets of data.
- If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.

OCC – Performance Issues

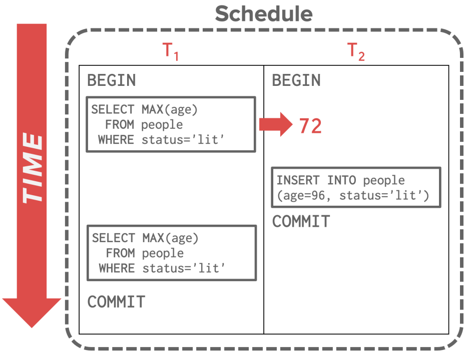
- High overhead for copying data locally.
- Validation/Write phase bottlenecks.
- Aborts are more wasteful than in 2PL because they only occur **after** a txn has already executed.

Observation

- Recall that so far we have only dealing with transactions that read and update data.
- But now if we have insertions, updates, and deletions, we have new problems. . .

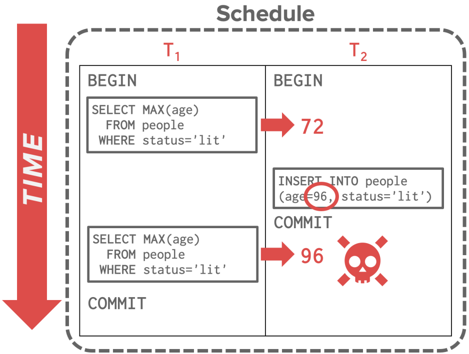
Phantoms

The Phantom Problem




```
CREATE TABLE people (
  id SERIAL,
  name VARCHAR,
  age INT,
  status VARCHAR
);
```

The Phantom Problem




```
CREATE TABLE people (
  id SERIAL,
  name VARCHAR,
  age INT,
  status VARCHAR
);
```

The Phantom Problem

- How did this happen?
  - Because T_1 locked only existing records and not ones under way!
- Conflict serializability on reads and writes of individual items guarantees serializability **only** if the set of objects is fixed.

Predicate Locking

- Lock records that satisfy a logical predicate:
  - Example: $status =' lit'$
- In general, predicate locking has a lot of locking overhead.
- Index locking is a special case of predicate locking that is potentially more efficient.

Index Locking

- If there is a dense index on the status field then the txn can lock index page containing the data with $\mathtt{status} =' \mathtt{lit}'$.
- If there are no records with $\mathtt{status} =' \mathtt{lit}'$, the txn must lock the index page where such a data entry would be, if it existed.

Locking without an Index

- If there is no suitable index, then the txn must obtain:
  - A lock on every page in the table to prevent a record's $\mathtt{status} =' \mathtt{lit}'$ from being changed to lit.
  - The lock for the table itself to prevent records with $\mathtt{status} =' \mathtt{lit}'$ from being added or deleted.

Repeating Scans

- An alternative is to just re-execute every scan again when the txn commits and check whether it gets the same result.
  - Have to retain the **scan set** for every range query in a txn.

Weaker Levels of Isolation

- Serializability is useful because it allows programmers to ignore concurrency issues.
- But enforcing it may allow too little concurrency and limit performance.
- We may want to use a weaker level of consistency to improve scalability.

Isolation Levels

Isolation Levels

- Controls the extent that a txn is exposed to the actions of other concurrent txns.
- Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
  - Dirty Reads
  - Unrepeatable Reads
  - Phantom Reads

Isolation Levels

- Isolation (High→Low)
- **SERIALIZABLE:** No phantoms, all reads repeatable, no dirty reads.
- **REPEATABLE READS:** Phantoms may happen.
- **READ COMMITTED:** Phantoms and unrepeatable reads may happen.
- **READ UNCOMMITTED:** All of them may happen.

Isolation Levels

Level	**Dirty Read**	**Unrepeatable Read**	**Phantom**
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe

Isolation Levels

- **SERIALIZABLE:** Obtain all locks first; plus index locks, plus strict 2PL.
- **REPEATABLE READS:** Same as above, but no index locks.
- **READ COMMITTED:** Same as above, but S locks are released immediately.
- **READ UNCOMMITTED:** Same as above, but allows dirty reads (no S locks).

SQL-92 Isolation Levels

- You set a txn's isolation level before you execute any queries in that txn.
- Not all DBMSs support all isolation levels in all execution scenarios
  - Replicated Environments
- The default depends on implementation. . .

```
SET TRANSACTION Isolation LEVEL  <isolation-level>;
```

```
BEGIN TRANSACTION ISOLATION LEVEL  <isolation-level>;
```

Isolation Levels (2013)

DBMS	Default	Maximum
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITTED	READ COMMITTED
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

- Source

SQL-92 Access Modes

- You can provide hints to the DBMS about whether a txn will modify the database during its lifetime.
- Only two possible modes:
  - **READ WRITE** (Default)
  - **READ ONLY**
- Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.

```
SET TRANSACTION <access-mode>;
```

```
BEGIN TRANSACTION <access-mode>;
```

Conclusion

Parting Thoughts

- Every concurrency control can be broken down into the basic concepts that I have described in the last two lectures.
  - Two-Phase Locking (2PL): Assumption that collisions are commonplace
  - Timestamp Ordering (T/O): Assumption that collisions are rare.
- Optimistic protocols defer the validation phase to the end of the txn
- I am not showing benchmark results because I don't want you to get the wrong idea.

Next Class

- Multi-Version Concurrency Control