Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo

Lecture#23: Concurrency Control – Part 2

(R&G ch. 17)

Concurrency Control Approaches

- **Two-Phase Locking (2PL)**
  - Determine serializability order of conflicting operations at runtime while txns execute.
- **Timestamp Ordering (T/O)**
  - Determine serializability order of txns before they execute.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O

- The Phantom Problem
- Weaker Isolation Levels

Timestamp Allocation

- Each txn Ti is assigned a unique fixed timestamp that is monotonically increasing.
  - Let **TS**(Ti) be the timestamp allocated to txn Ti
  - Different schemes assign timestamps at different times during the txn.
- Multiple implementation strategies:
  - System Clock.
  - Logical Counter.
  - Hybrid.

T/O Concurrency Control

- Use these timestamps to determine the serializability order.
- If **TS**(Ti) < **TS**(Tj), then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where Ti appears before Tj.

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.

Basic T/O – Reads

- If **TS**(Ti) < **W-TS**(X), this violates timestamp order of Ti w.r.t. writer of X.
  - Abort Ti and restart it (with same TS? why?)
- Else:
  - Allow Ti to read X.
  - Update **R-TS**(X) to **max(R-TS**(X), **TS**(Ti))
  - Have to make a local copy of X to ensure repeatable reads for Ti.

Basic T/O – Writes

- If **TS**(Ti) < **R-TS**(X) or **TS**(Ti) < **W-TS**(X)
  - Abort and restart Ti.
- Else:
  - Allow Ti to write X and update **W-TS**(X)
  - Also have to make a local copy of X to ensure repeatable reads for Ti.

Basic T/O – Example #1

TS(T1)=1 TS(T2)=2

Schedule

T1	T2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

TIME

Database

Object	R-TS	W-TS
A	2	2
B	2	2
-	-	

No violations so both txns are safe to commit.

Basic T/O – Example #2

Schedule

T1	T2
BEGIN	
R(A)	
	BEGIN
	W(A)
	COMMIT
W(A)	
COMMIT	

TIME

Database

Object	R-TS	W-TS
A	1	2
-	-	-
-		

Violation: **TS**(T1) < **W-TS**(A)

T1 cannot overwrite update by T2, so it has to abort+restart.

Basic T/O – Thomas Write Rule

- If **TS**(Ti) < **R-TS**(X):
  - Abort and restart Ti.
- If **TS**(Ti) < **W-TS**(X):
  - **Thomas Write Rule:** Ignore the write and allow the txn to continue.
  - This violates timestamp order of Ti
- Else:
  - Allow Ti to write X and update **W-TS**(X)

Basic T/O – Thomas Write Rule

Schedule

T1	T2
BEGIN	
R(A)	
	BEGIN
	W(A)
	COMMIT
W(A)	
COMMIT	

TIME

Database

Object	R-TS	W-TS
A	1	2
-		
-		

We do not update **W-TS**(A)

Ignore the write and allow T1 to commit.

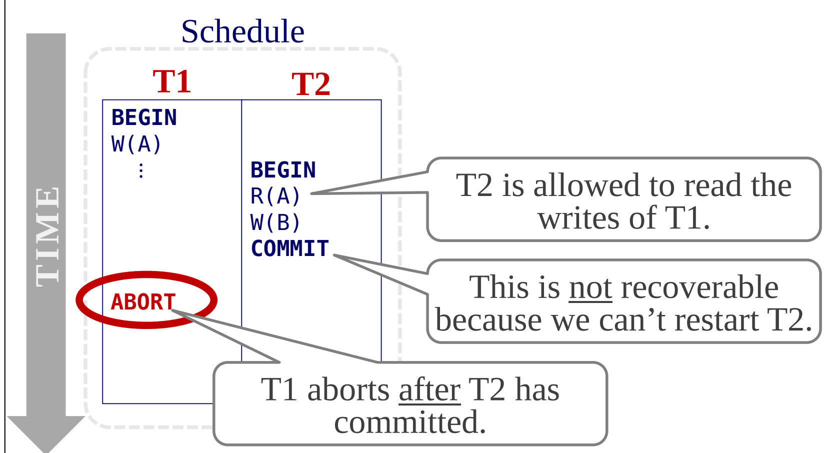
Basic T/O

- Ensures conflict serializability if you don't use the Thomas Write Rule.
- No deadlocks because no txn ever waits.
- Possibility of starvation for long txns if short txns keep causing conflicts.
- Permits schedules that are not ***recoverable***.

Recoverable Schedules

- Transactions commit only after all transactions whose changes they read, commit.

Recoverability



Basic T/O – Performance Issues

- High overhead from copying data to txn's workspace and from updating timestamps.
- Long running txns can get starved.
- Suffers from timestamp bottleneck.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O

- The Phantom Problem
- Weaker Isolation Levels

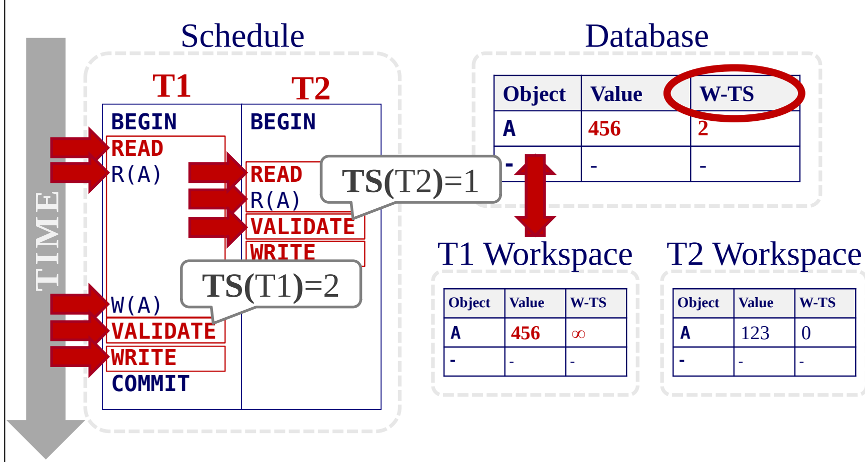
Optimistic Concurrency Control

- Assumption: Conflicts are rare
- Forcing txns to wait to acquire locks adds a lot of overhead.
- Optimize for the no-conflict case.

OCC Phases

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

OCC – Example



OCC – Validation Phase

- Need to guarantee only serializable schedules are permitted.
- At validation, Ti 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.

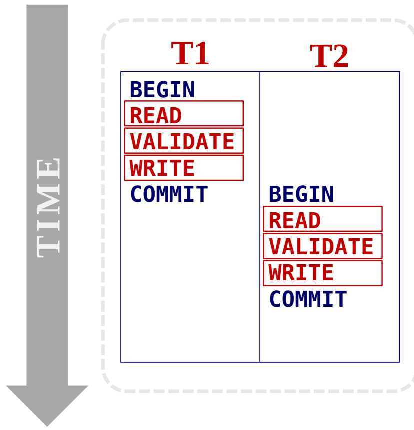
OCC – Validation Phase

- Each txn's timestamp is assigned at the beginning of the validation phase.
- Check the timestamp ordering of the committing txn with all other running txns.
- If **TS**(Ti**)** < **TS**(Tj**)**, then <u>one</u> of the following three conditions must hold…

OCC – Validation #1

- Ti completes all three phases before Tj begins.

OCC – Validation #1

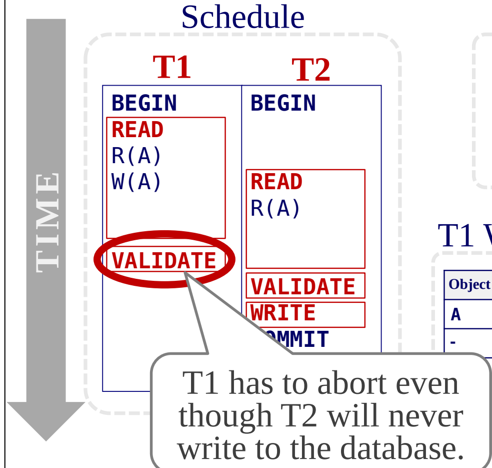


OCC – Validation #2

- Ti completes before Tj starts its **Write** phase, and Ti does not write to any object read by Tj.
  - WriteSet(Ti) ∩ ReadSet(Tj) = Ø

OCC – Validation #2

Schedule



Database

Object	Value	W-TS
A	123	0
-	-	-

T1 Workspace

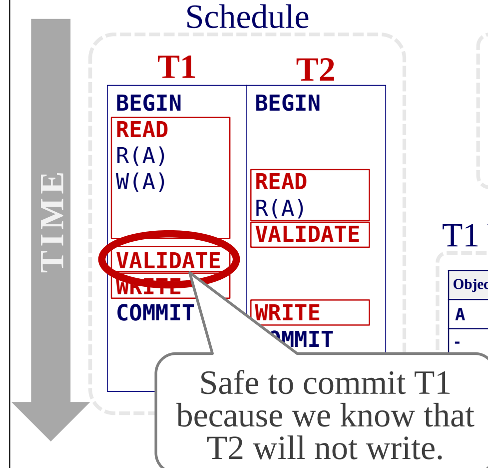
Object	Value	W-TS
A	456	∞
-	-	-

T2 Workspace

Object	Value	W-TS
A	123	0
-	-	-

OCC – Validation #2

Schedule



Database

Object	Value	W-TS
A	123	0
-	-	-

T1 Workspace

Object	Value	W-TS
A	456	∞
-	-	-

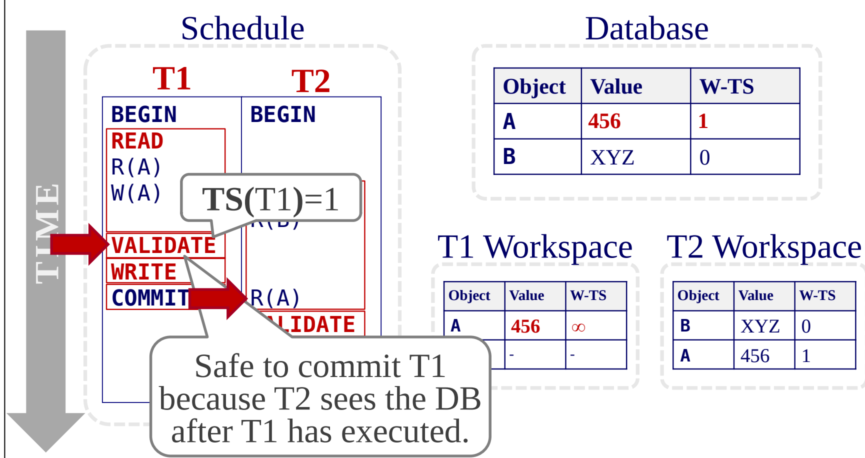
T2 Workspace

Object	Value	W-TS
A	123	0
-	-	-

OCC – Validation #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) ∩ ReadSet(Tj) = Ø
  - WriteSet(Ti) ∩ WriteSet(Tj) = Ø

OCC – Validation #3



OCC – Observations

- **Q:** When does OCC work well?
- **A:** When # 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 because they only occur *after* a txn has already executed.
- Suffers from timestamp allocation bottleneck.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O

- The Phantom Problem
- Weaker Isolation Levels

Multi-Version Concurrency Control

- Writes create new versions of objects instead of in-place updates:
  - Each successful write results in the creation of a new version of the data item written.
- Use write timestamps to label versions.
  - Let X_k denote the version of X where for a given txn Ti: **W-TS**(X_k) ≤ **TS**(Ti)

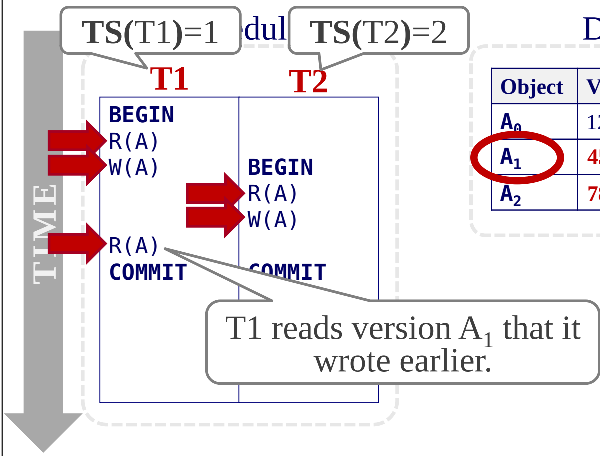
MVCC – Reads

- Any read operation sees the latest version of an object from right before that txn started.
- Every read request can be satisfied without blocking the txn.
- If **TS**(Ti) > **R-TS**(X_k):
  - Set **R-TS**(X_k) = **TS**(Ti)

MVCC – Writes

- If **TS**(Ti) < **R-TS**(X_k):
  - Abort and restart Ti.
- If **TS**(Ti) = **W-TS**(X_k):
  - Overwrite the contents of X_k.
- Else:
  - Create a new version of X_{k+1} and set its write timestamp to **TS**(Ti).

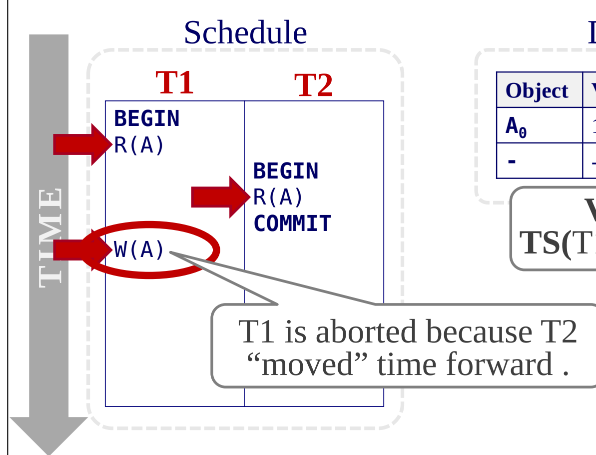
MVCC – Example #1



Database

Object	Value	R-TS	W-TS
A_0	123	1	0
A_1	456	2	1
A_2	789	2	2

MVCC – Example #2



Database

Object	Value	R-TS	W-TS
A_0	123	2	0
-	-	-	

Violation:
TS(T1) < R-TS(A_0)

MVCC

- Can still incur cascading aborts because a txn sees uncommitted versions from txns that started before it did.
- Old versions of tuples accumulate.
- The DBMS needs a way to remove old versions to reclaim storage space.

MVCC Implementations

- Store versions directly in main tables:
  - Postgres, Firebird/Interbase
- Store versions in separate temp tables:
  - MSFT SQL Server
- Only store a single master version:
  - Oracle, MySQL

Garbage Collection – Postgres

- Never overwrites older versions.
- New tuples are appended to table.
- Deleted tuples are marked with a tombstone and then left in place.
- Separate background threads (**VACUUM**) has to scan tables to find tuples to remove.

Garbage Collection – MySQL

- Only one “master” version for each tuple.
- Information about older versions are put in temp rollback segment and then pruned over time with a single thread (**PURGE**).
- Deleted tuples are left in place and the space is reused.

MVCC – Performance Issues

- High abort overhead cost.
- Suffers from timestamp allocation bottleneck.
- Garbage collection overhead.
- Requires stalls to ensure recoverability.

MVCC+2PL

- Combine the advantages of MVCC and 2PL together in a single scheme.
- Use different concurrency control scheme for read-only txns than for update txns.

MVCC+2PL – Reads

- Use MVCC for read-only txns so that they never block on a writer
- Read-only txns are assigned a timestamp when they enter the system.
- Any read operations see the latest version of an object from right before that txn started.

MVCC+2PL – Writes

- Use strict 2PL to schedule the operations of update txns:
  – Read-only txns are essentially ignored.
- Txns never overwrite objects:
  – Create a new copy for each write and set its timestamp to ∞.
  – Set the correct timestamp when txn commits.
  – Only one txn can commit at a time.

MVCC+2PL – Performance Issues

- All the lock contention of 2PL.
- Suffers from timestamp allocation bottleneck.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O

- The Phantom Problem
- Weaker Isolation Levels

Observation

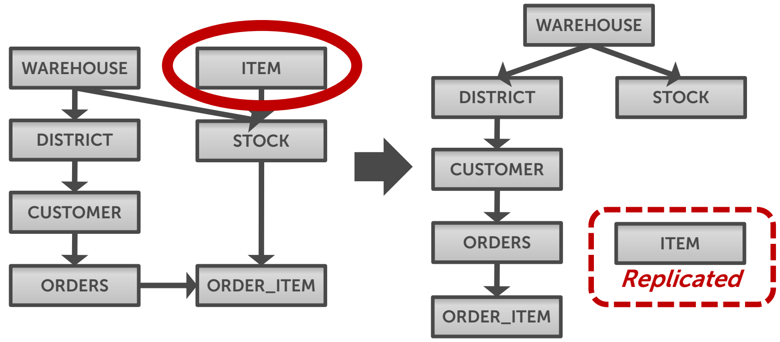
- When a txn commits, all previous T/O schemes check to see whether there is a conflict with concurrent txns.
- This requires locks/latches/mutexes.
- If you have a lot of concurrent txns, then this is slow even if the conflict rate is low.

Partition-based T/O

- Split the database up in disjoint subsets called *partitions* (aka *shards*).
- Only check for conflicts between txns that are running in the same partition.

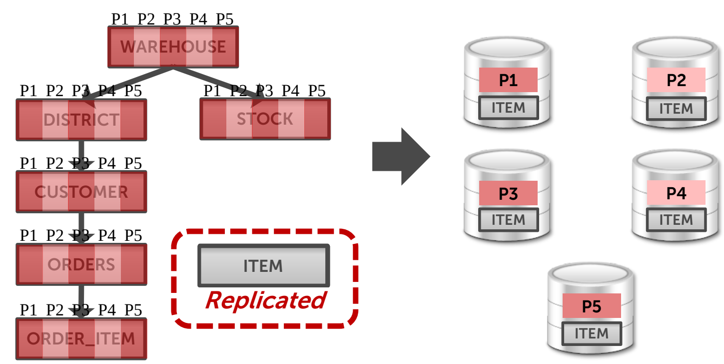
Database Partitioning

Schema Schema Tree



Database Partitioning

Schema Tree Partitions



Partition-based T/O

- Txns are assigned timestamps based on when they arrive at the DBMS.
- Partitions are protected by a single lock:
  - Each txn is queued at the partitions it needs.
  - The txn acquires a partition's lock if it has the lowest timestamp in that partition's queue.
  - The txn starts when it has all of the locks for all the partitions that it will read/write.

Partition-based T/O – Reads

- Do not need to maintain multiple versions.
- Txns can read anything that they want at the partitions that they have locked.
- If a txn tries to access a partition that it does not have the lock, it is aborted + restarted.

Partition-based T/O – Writes

- All updates occur in place.
  - Maintain a separate in-memory buffer to undo changes if the txn aborts.
- If a txn tries to access a partition that it does not have the lock, it is aborted + restarted.

Partition-based T/O – Performance Issues

- Partition-based T/O protocol is very fast if:
  - The DBMS knows what partitions the txn needs before it starts.
  - Most (if not all) txns only need to access a single partition.
- Multi-partition txns causes partitions to be idle while txn executes.

Today's Class

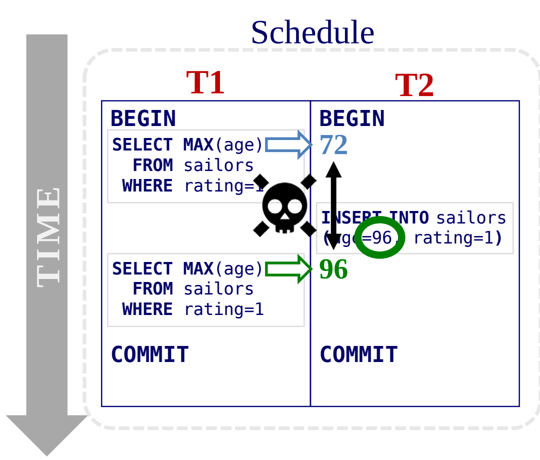
- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O

- The Phantom Problem
- Weaker Isolation Levels

Dynamic Databases

- 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…

The Phantom Problem



How did this happen?

- Because T1 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.
- Solution?

Predicate Locking

- Lock records that satisfy a logical predicate:
  - Example: **`rating=1`**.
- 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 **`rating`** field then the txn can lock index page containing the data with **`rating=1`**.
- If there are no records with **`rating=1`**, 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 **`rating`** from being changed to 1.
  - The lock for the table itself to prevent records with **`rating=1`** from being added or deleted.

Today's Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control
- Partition-based T/O

- The Phantom Problem
- Weaker Isolation Levels

Weaker Levels of Consistency

- 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

- 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

	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

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

- Default: Depends…
- Not all DBMS support all isolation levels in all execution scenarios (e.g., replication).

Isolation Levels

	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: Peter Bailis, *When is "ACID" ACID? Rarely.* January 2013

Access Modes

- You can also provide hints to the DBMS about whether a txn will modify the database.
- Only two possible modes:
  - **READ WRITE**
  - **READ ONLY**

SQL-92 Access Modes

SQL-92

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

Postgres + MySQL 5.6

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

- Default: **READ WRITE**
- Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.

Which CC Scheme is Best?

- Like many things in life, it depends…
  - How skewed is the workload?
  - Are the txns short or long?
  - Is the workload mostly read-only?

Real Systems

	Scheme	Released
Ingres	Strict 2PL	1975
Informix	Strict 2PL	1980
IBM DB2	Strict 2PL	1983
Oracle	MVCC	1984*
Postgres	MVCC	1985
MS SQL Server	Strict 2PL or MVCC	1992*
MySQL (InnoDB)	MVCC+2PL	2001
Aerospike	OCC	2009
SAP HANA	MVCC	2010
VoltDB	Partition T/O	2010
MemSQL	MVCC	2011
MS Hekaton	MVCC+OCC	2013

Summary

- Concurrency control is hard.