Forms of Serializability

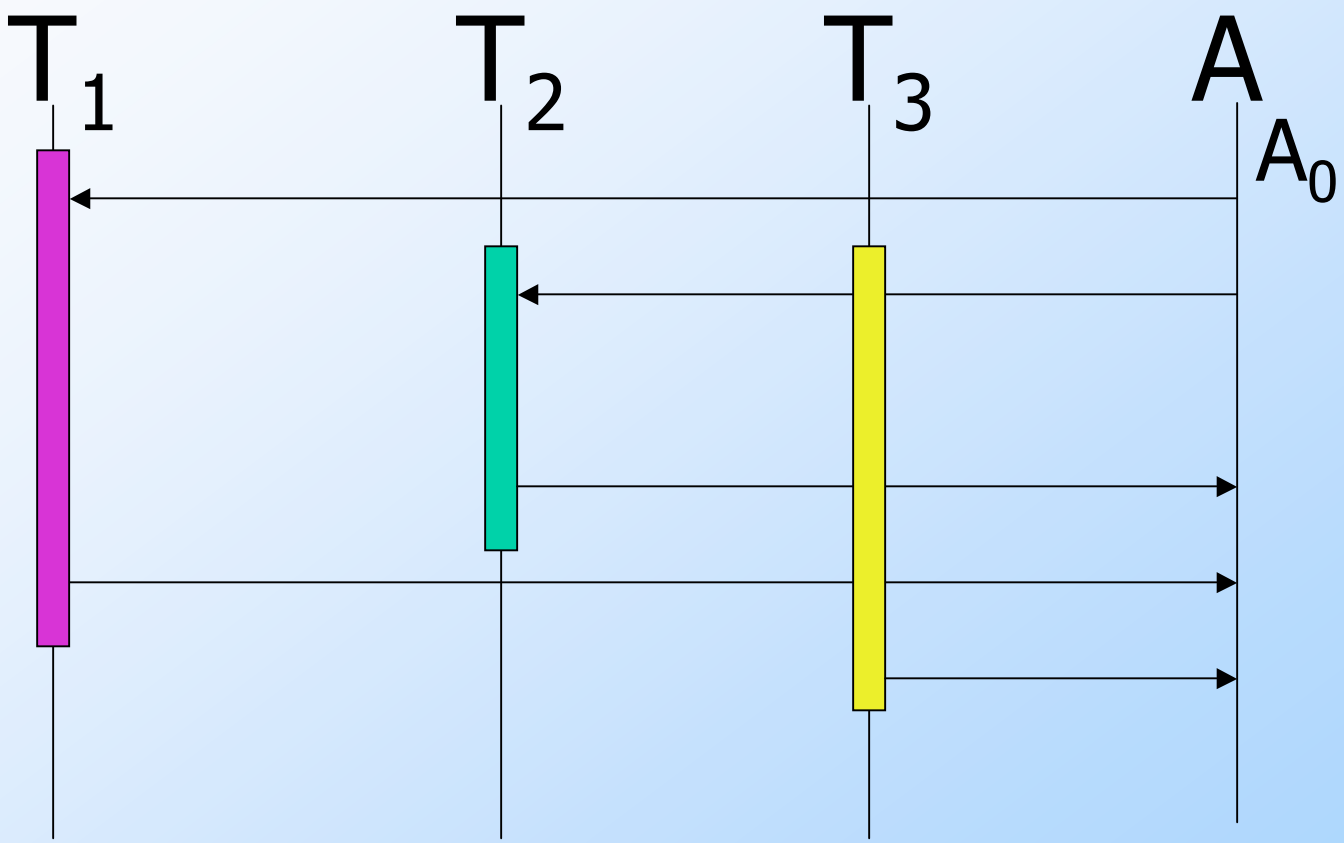
- Our previous version of serializability required that the ***sequence*** of values written to any data item be the same as some serial schedule.

- This form is sometimes called **conflict serializability**.

View Serializability

- A weaker form, known as **view serializability**, only demands that the **final** values of each data item be the same.
- conflict serializability $\Rightarrow$ view serializability but not conversely.

Not conflict serializable, but view serializable



Test for View Serializability

- A test for view serializability exists, but is significantly more complicated (e.g. "polygraph" construction).
- See "The Complete Book" for definition and examples.

Timestamp Concurrency Control

- This is an "optimistic" approach, requiring willingness to **rollback** (= abort, then restart) a transaction to resolve conflict.
- Each transaction has a unique **timestamp ts**.
- Transactions started later have a higher timestamp.
- The timestamp order is the same as the equivalent **serialized** order.

Timestamp Concurrency Control

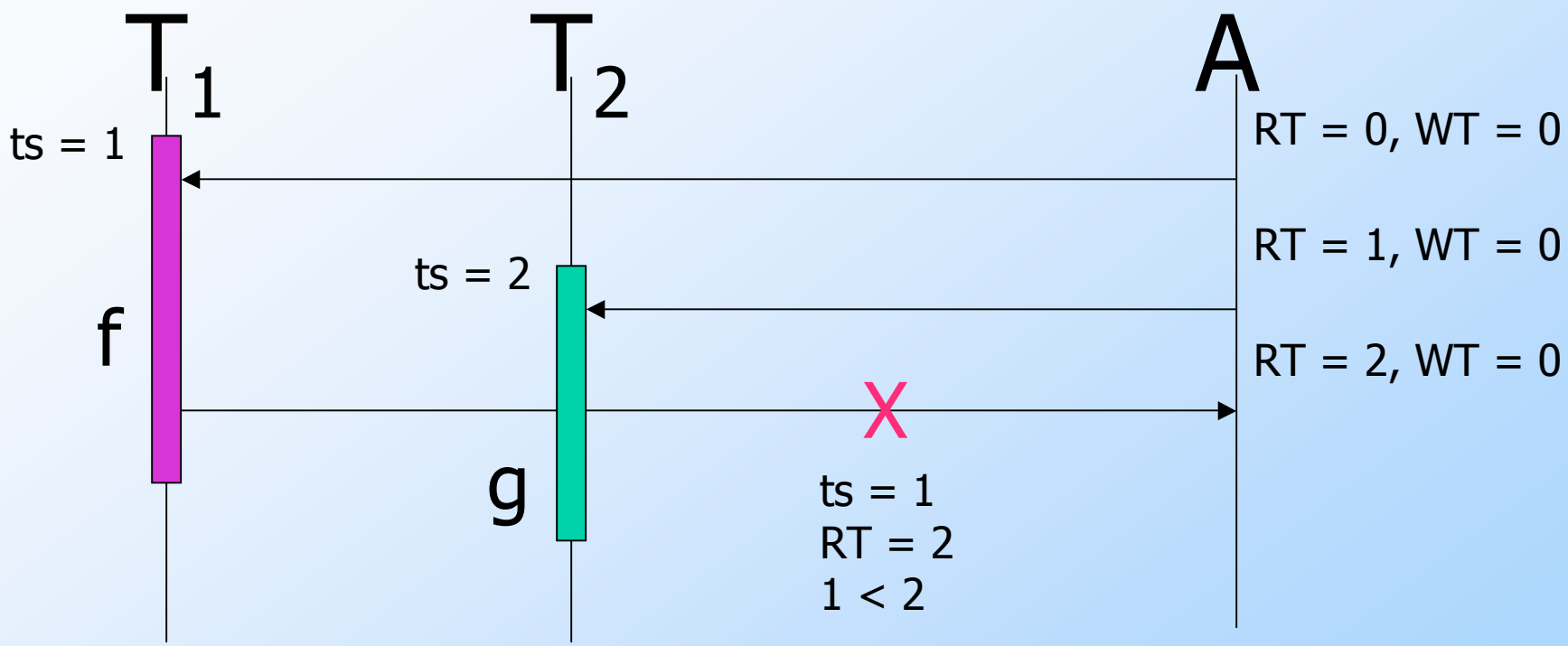
- With each **data item**, associate:
  - **RT**: The time stamp of the most recent transaction that **read** the item.
  - **WT**: The time stamp of the most recent transaction that **wrote** the item.
  - **C**: Whether the most transaction writing the item has committed.

Timestamp Constraints

- If transaction T wants to **read** a data item, its timestamp must **not** be **less than** than WT of the item. If not, T is aborted. (The attempted read is "too early".)

- If transaction T wants to **write** an item, ...

Timestamp Example



T_1 would be aborted.
It could be restarted and the equivalent serial order would be T_2 T_1 .

Use of the Commit Bit

- If the last transaction writing a data item has not committed, then the value of the item is **tentative** (also called "dirty").
- One policy would be to **wait** for the commit bit to be set before going ahead with the read.
- If we don't, and the writing transaction aborts, we'd have to abort the reading transaction as well (called "cascading rollback").

The Thomas Write-Rule

- If a **later** transaction wants to write to an item, and the **previous** writing transaction has **not committed**, it is ok to just do the **write anyway**.
- The problem with this rule is that if the later transaction aborts, we will have no way to **restore** the value that would have been over-written, so will be forced to abort the previous transaction as well.

Multi-Version Concurrency Control (MVCC)

- The number of rollbacks can be reduced by this scheme.
- Multiple versions of data items are kept, each with distinct values of WT.
- If a transaction's ts is less than that of the latest WT, an attempt is made to find a version with a WT less than the ts and use it. If found, this transaction does not have to rollback.

Related Ideas

- Multi-version time stamps are used in non-database areas, such as distributed simulation (e.g. the "**time warp**" simulation approach, invented by David Jefferson).
- A similar idea is found in processor architectures that use **register renaming**.

Discussion

- Under what conditions is time stamping better than locking?

Case Study: PostgreSQL

Unlike most other database systems which use locks for concurrency control, PostgreSQL maintains data consistency by using a multiversion model. This means that while querying a database each transaction sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing ***transaction isolation*** for each database session. The main difference between multiversion and lock models is that in MVCC locks acquired for querying (reading) data don't conflict with locks acquired for writing data and so reading never blocks writing and writing never blocks reading.

Serializable Isolation Level

When a transaction is on the **Serializable** level, a **SELECT** query sees only data committed before the transaction began and never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the **SELECT** does see the effects of previous updates executed within this same transaction, even though they are not yet committed.)

This is different from **Read Committed** in that the **SELECT** sees a **snapshot** as of the start of the transaction, not as of the start of the current query within the transaction. If a target row found by a query while executing an **UPDATE** statement (or **DELETE** or **SELECT FOR UPDATE**) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will **wait** for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row.

In the case of a concurrent transaction commit, a serializable transaction will be **rolled back** with the message`ERROR: Can't serialize access due to concurrent update` because a serializable transaction cannot modify rows changed by other transactions after the serializable transaction began. When the application receives this error message, it should abort the current transaction and then retry the whole transaction from the beginning. Note that only updating transactions may need to be retried --- **read-only transactions never have serialization conflicts**.

Application-Level Consistency Checks

Because **reading** transactions in PostgreSQL don't lock data, regardless of isolation level, data read by one transaction can be overwritten by another concurrent transaction. In other words, a row being returned by **SELECT** doesn't mean that the row **still exists** at the time it is returned (i.e., sometime after the current transaction began); the row might have been modified or deleted by a transaction that committed after this one started. Even if the row is still valid "now", it could be changed or deleted before the current transaction does a commit or rollback.

To ensure the current existence of a row and protect it against, concurrent updates one must use **SELECT FOR UPDATE** or an appropriate **LOCK TABLE** statement. (**SELECT FOR UPDATE** locks just the returned rows against concurrent updates, while **LOCK TABLE** protects the whole table.)

Database Recovery

Robert M. Keller
Harvey Mudd College
April 2004

Recovery from What?

- Hardware failure (e.g. disk head crash, communication failure)
- Software error (e.g. infinite loop in the middle of a transaction)
- Explicit abort during transaction (e.g. due to attempt to violate a constraint)
- O.S. error

Recovery in Hardware

- Redundancy
- High-Level:
  - Spare computer(s)
  - Backup database, archives
- Low-Level:
  - RAID
    - (Redundant Array of Inexpensive Disks)
    - Many levels of RAID

Main Issue: How to Reconstruct Information (1)

- Assume **transaction** basis
- Some data from completed transactions may be lost.
- We need to be able to **redo** those transactions from available data.
- A **checkpoint** is a representation of the state of the database at a certain point in time.
- Numerous checkpoints will be saved, on **alternative** (ideally, *less-volatile*) storage media. The most recent checkpoint available will be used for recovery.

Main Issue: How to Reconstruct Information (2)

- At the point of failure, some transactions may be **partially complete**:
  - Computation steps not complete, or
  - Steps complete, data not written
- We must be able to **undo** any latent effects of these transactions.
- **Ripple effects**: Undoing one transaction may make the effects of another meaningless, in which case the latter also needs to be undone, etc.

Which transactions are done?

- We need a **log** of completed transactions, to help with **redo**.
- We can also keep incomplete transactions in the log, e.g. to help with **undo**.
- The log is written sequentially (in a critical section).
- As with checkpoints, the log is ideally on separate (and less-volatile) media.

Checkpoints

- Checkpoint construction is not trivial.
- **Ideally**, a checkpoint represents **an image** of the database at a given point in time, with no transactions in progress.
- In a **high-availability** system, we **cannot** just stop everything, must devise a way to achieve the same effect, e.g. wait for current transactions to finish, treat new transactions *as if* not started.

Log Contents

- Various kinds of **log entries** indicating:
  - Transaction **start**.
  - Transaction **commit**, or Transaction **abort**
  - Page **update**: images of data
    - (before, as well as after write)
  - **Checkpoint start**
  - **Checkpoint complete**
  - **Failure** (in the event of recovering from a failure)
  - Other information: sequence number, transaction id, etc.
- The log entries are **write-only**. Best to use dedicated, non-volatile, media.
- The log does not record **reads** of data.

Log Sequence Number (LSN)

- Each log entry has a number
- Assigned while entry is waiting to be written to log
- Guaranteed to be monotone increasing with time.
- Therefore, can be used to determine if one event occurred before another.
- The LSN can also be "stamped" on a data page to show the last transaction that updated it. This can be useful in recovery.
- Log entries of distinct transactions can be intermingled in a range of LSN's.

Log Table as Relation (Jim Gray)

Log table is a sequential set (relation).
Log Records have standard part and then a log body.
Often want to query table via one attribute or another:

- **RMID, TRID, timestamp,**

```
create domain LSN   unsigned integer(64);          -- log sequence number (file #, rba)
create domain RMID unsigned integer;               -- resource manager identifier
create domain TRID  char(12);                      -- transaction identifier
create table  log_table (
                        lsn             LSN,       -- the record's log sequence number
                        prev_lsn        LSN,       -- the lsn of the previous record in log
                        timestamp TIMESTAMP,       --  time log record was created
                        resource_manager RMID,     -- resource mgr that wrote this record
                        trid            TRID,      -- id of transaction that wrote this record
                        tran_prev_lsn LSN,         -- prev log record of this transaction (or 0)
                        body            varchar,   --  log data:  rm understands it
                        primary key (lsn)          -- lsn is primary key
                        foreign key (prev_lsn)     -- previous log record  in this table
                                                      references a_log_table(lsn), --
                        foreign key (tran_prev_lsn)  -- transaction's prev  log rec  also in table
                                                      references a_log_table(lsn),
                        )               entry sequenced;   -- inserts go at end of file
```

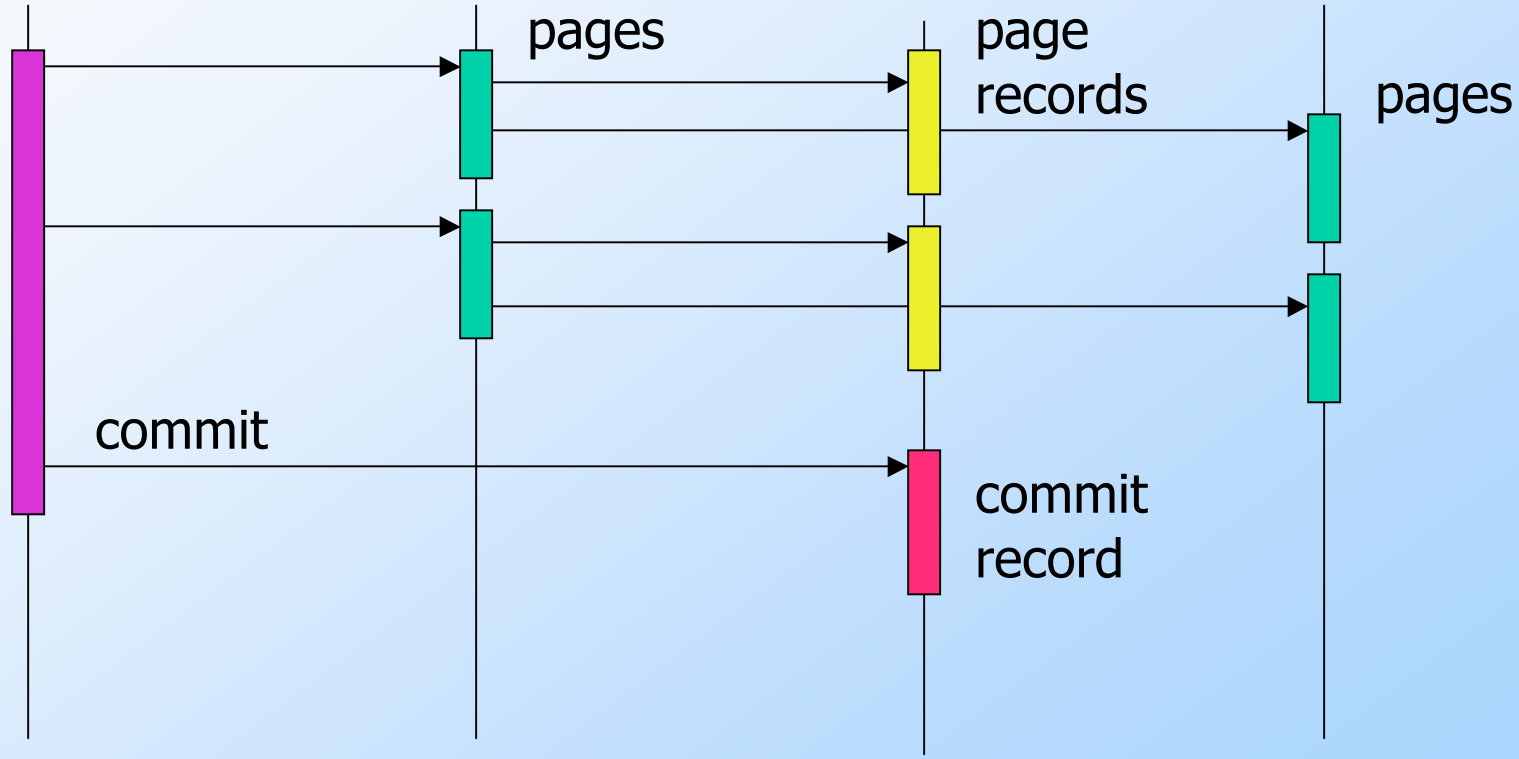
Transaction Parameters

- Is there any need to log transaction parameters?

Logging Order

- All pages must be successfully written to the database **before the commit** entry is written in the log.
  - So a failure in the midst of a transaction will show as a started, but uncommitted, transaction.
- **Before** each **page** is written, the log entry is written (called write-ahead log or **WAL**).

Logging Sequence Diagram



Simplification Alert

- We are avoiding the issue of how pages are **buffered** while waiting to be written.
- Normally, buffering is optimized to minimize page-transfer latency.
- This means pages will generally be written out in a different order from the one in which they are modified.

Committed Transactions

- Effects of committed transactions are supposed to be **Durable** (the "D" in ACID).
- When a **failure** occurs, the effects of those transactions, if lost, must be restored.
- Use the checkpoint and log records to restore.

Redoing

- At the point of failure, reconstruction begins with the **latest** complete/stable checkpoint.
- Using the page images in the transaction log, we can **redo** the effect of committed transactions, one page at a time.
- We can apply the page images in **log-order**, oldest first, ignoring changes made by uncommitted transactions.
- There is a **recoverability assumption** here: No non-committed transaction has written a page subsequently used by a committed transaction. This can be ensured the transaction protocol, e.g. **don't release locks until the transaction commits**. Without this, a transaction could commit based on data that itself was never committed. This assumption is called ***strict two-phase locking***.

Uncommitted Transactions

- **Uncommitted transactions** at the time of failure have to have any changes to the database undone. Force an **abort** entry into the log for these.
- How can uncommitted transactions be identified?
- Using the log, work **backwards** from the point of failure, replacing after-images with before- images, for the uncommitted transactions.
- Once the database is restored, the uncommitted transactions can be **retried**.

Undoing

- At the point of failure, reconstruction begins with the **latest** checkpoint.
- If a transaction was in progress at the checkpoint, any effects of it should be **undone**.
- The **youngest** transactions will be **undone first**, because earlier transactions may have written the data that they use.

Idempotency

- Restarting after a failure must be "Idempotent":
  - If a failure occurs during restart itself, still can recover.
  - For example, **redoing** an update should give the same result as no matter how many times it is done.
  - If a LSN is stamped on a page, then each transaction being redone can have its LSN compared to the page's. Only transactions with a higher LSN than the page should be applied.

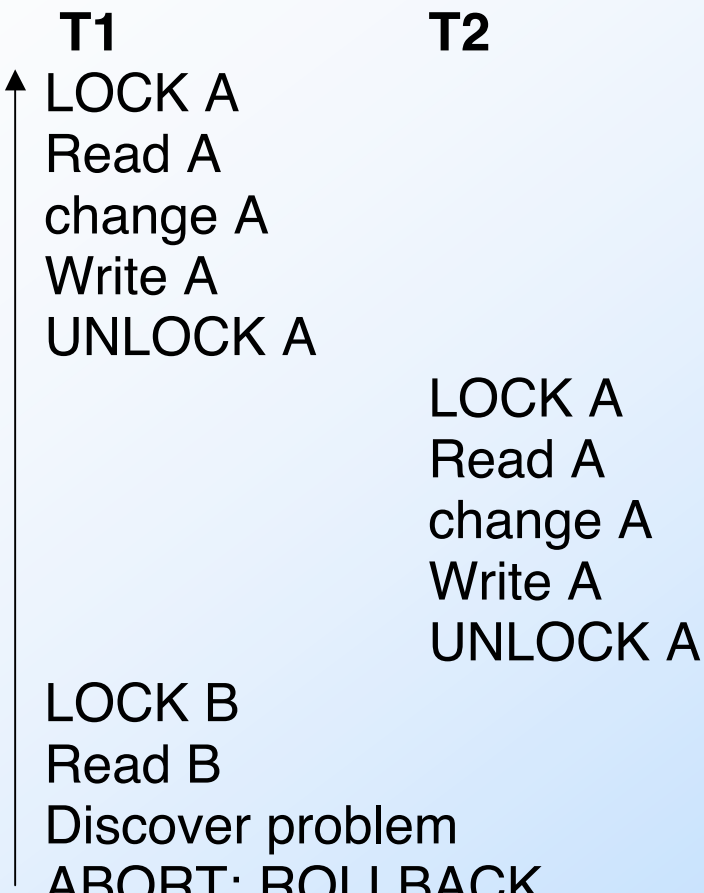
Recoverable Protocols

- A protocol is **recoverable** if each transaction ***commits*** only ***after*** all transactions from which it have read have committed.

- A protocol **avoids cascading rollback** if each transaction ***reads*** only ***after*** those values written by committed transactions (i.e. no tentative data is ever read).

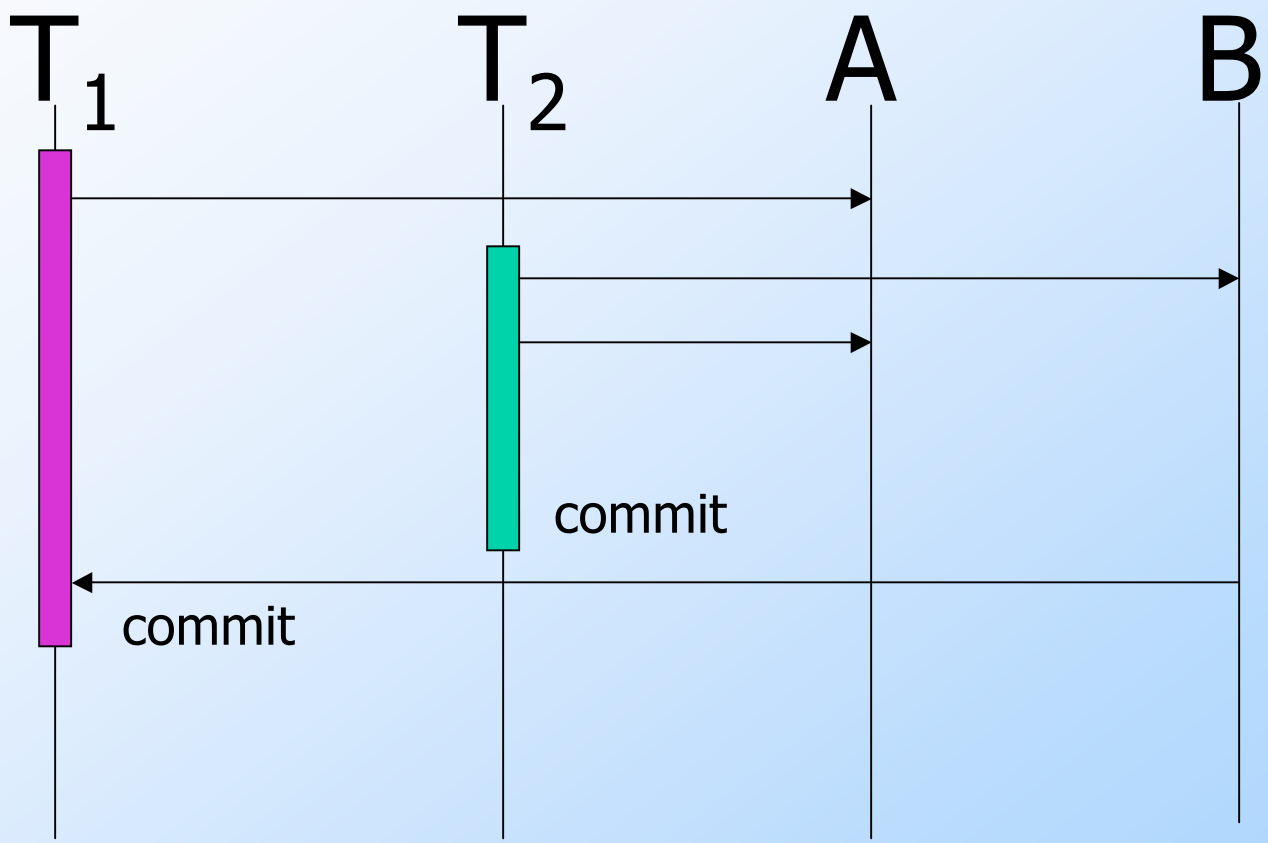
- Strict 2PL ⇒ ACR ⇒ recoverable.
- (recall 2PL ⇒ serializable)

Example of Cascading Rollback



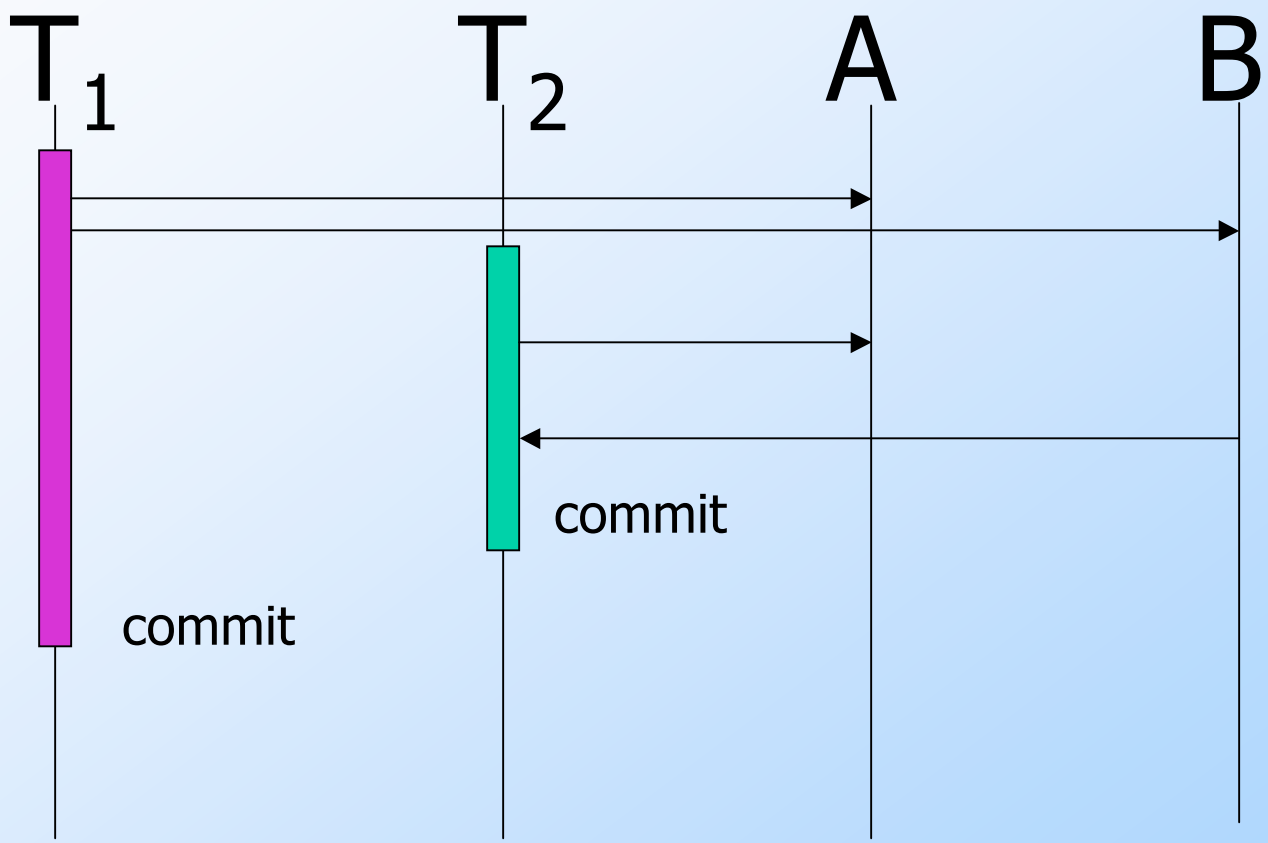
Need to undo the change to A by T2
that was made based on data written by T1.
(note: NOT 2PL)

Recoverable, not Serializable



ACR?

Serializable, not Recoverable



Case Study: PostgreSQL

- PostgreSQL maintains a WAL
- After a **checkpoint** has been made and the log flushed, the checkpoint's position is saved in the file `pg_control`. Therefore, when **recovery** is to be done, the backend first reads `pg_control` and then the checkpoint record; next it reads the **redo** record, whose position is saved in the checkpoint, and begins the REDO operation.
- Because the entire content of the pages is saved in the log on the first page modification after a checkpoint, the pages will be first restored to a consistent state.
- Using `pg_control` to get the checkpoint position speeds up the recovery process, but to handle possible corruption of `pg_control`, we should implement the reading of existing log segments in reverse order -- newest to oldest -- in order to find the last checkpoint.

Using page buffering in memory

- A simple way to avoid dirty pages on disk:
  - Only write to disk pages having a committed most-recent writer.
  - "Roll back" = just don't write

Other Related Topics

- Buffer management
- Lock granularity
- Distributed database concurrency control and recovery