6. Transaction Management

Slides adapted from Pearson Ed.

Why transactions are important

- Sometimes a single task/operation/activity may require updating your database in multiple places.
  - If some but not all updates are carried out, the database may be in a bad / inconsistent state.
- Common example: banking transactions.
  - Withdrawing or transferring money should never allow money to just disappear or appear out of thin air.
  - Transferring money: requires debiting one account and crediting another. Don't want…
    - Double debit.
    - Debit without corresponding credit.
    - Credit without corresponding debit.

Transactions in SQL

- Can specify that a bundle of SQL statements should be committed or rolled back as a set (atomically).
- Optionally use ‘BEGIN TRANSACTION’.
- Or just use ‘COMMIT’ after a sequence of statements.
- Can specify beforehand how much the set of statements interact with each other.

Two scenarios that create challenges for transaction management

- System failure: system goes down in the middle of a transaction.
  - If transaction is not finished, database may be in an inconsistent state. Need to restore system back to consistent state.
  - Transaction may be finished, but have the changes made it to the file system?? System may be in the middle of transferring most up-to-date data from temporary storage (buffers) back to disk.
  - Always a relevant issue because database / system failures are not fully preventable.
- Concurrency: want to handle multiple transactions simultaneously.
  - Relevant for large, high-activity databases.

Example: issue caused by system failure

```
Begin transaction:
    **Read** payer's balance
    New balance = balance – 20
    **Write** new balance to payer
    **Read** payee's balance
    New balance = balance + 20
    **Write** new balance to payer
End transaction
```

System failure here renders the state inconsistent

Example: issue caused by concurrent transactions

Transaction #1: withdrawal	Transaction #2: deposit
Begin transaction:	Begin transaction:
Read user’s balance (100)	
New balance = balance – 20	
	Read user’s balance (100)
Write new balance to user (80)	New balance = balance + 20
End transaction	**Write** new balance to user (120)
	End transaction

- Lost update problem: effects of one update disappear.

Example: issue caused by concurrent transactions

Transaction #1: summation	Transaction #2: transfer
Begin transaction:	Begin transaction:
Read user 1’s balance (100)	
Total balance = balance1	**Read** user 1’s balance (100)
	Read user 2’s balance (30)
	Transfer £20 from user 1 to user 2
	Write new balances to user 1 (80) and user 2 (50)
Read user 2’s balance (50)	
Total balance += balance2	End transaction
Write total balance (150)	
End transaction	

- Inconsistent analysis problem or **dirty/nonrepeatable read**: data active in one transaction are altered during its operation.

ACID properties

- For these critical operations (transactions), it would be good for a DBMS to guarantee some properties always hold:
  - **Atomicity**: a transaction happens *in its entirety* or *not at all*.
  - **Consistency**: all transactions carry the database from one consistent state (cohering to the intended structure of system / intended effect of operations on system) to another.
  - **Isolation**: partial effects of incomplete transactions can't influence / be seen by other transactions.
  - **Durability**: completed transactions are permanently recorded and not lost.
    - Transaction can later be undone by a separate transaction (a **compensating transaction**), but if a transaction happens, it doesn't just disappear.

Three transaction states

- Ongoing / active but uncommitted: transaction started but not yet committed.
- Committed: transaction can no longer be undone / rolled back / aborted.
  - If the work of this transaction has been lost, the changes must be reapplied (durability property).
- Aborted: there are issues with this transaction that make completing it difficult or undesired. It must be undone.
  - Any changes to the database it made must be reversed.

Example: issue caused by concurrent transactions 3

Transaction #1: withdrawal
Begin transaction:
- **Read** user's balance (100)
- New balance = balance – 20
- **Write** new balance to user (80)
- Panic!
- **Rollback**
- …

Transaction #2: deposit
Begin transaction:
- **Read** user's balance (80)
- New balance = balance + 20
- **Write** new balance to user (100)

End transaction

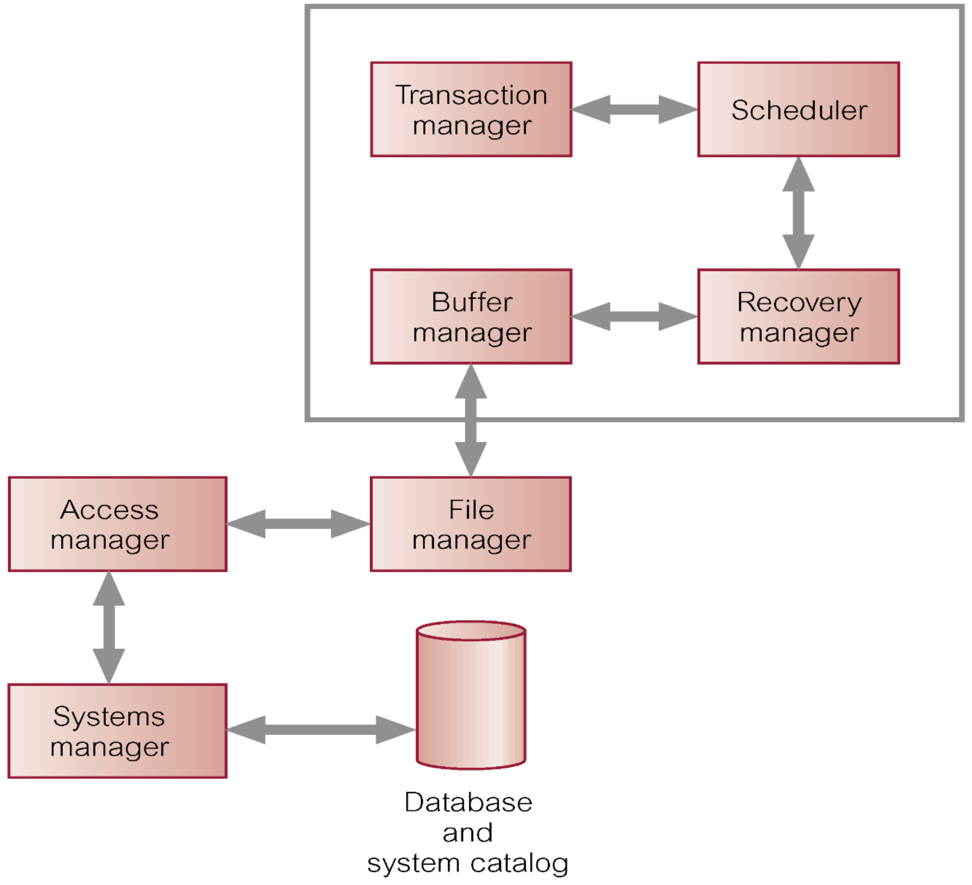
- Uncommitted dependency problem: aborted transaction's intermediate results used in another transaction even though only transaction #2 completed.

Components of transaction management

- Processing transactions,
- Concurrency (running multiple transactions simultaneously),
- Failure recovery.

Components of a transaction management subsystem

- Transaction system of a DBMS has several jobs and components for carrying them out:
  - Handle incoming requests from apps – transaction manager.
  - Schedule multiple transactions concurrently if possible – scheduler or sometimes lock manager.
  - Something that kicks in after a system failure – recovery manager.
  - Manage loading data to memory and saving results of transactions to disk for multiple concurrent transactions – buffer manager.



Overview of concurrency

Interleaved vs. serial schedules

- Interleaved schedules: multiple transactions run at the same time.
  - Their effects (operations) may be interleaved / occurring at the same time or almost the same time and can be summarized as a single- or multi-column sequence of events involving multiple transactions.
- Serial schedules: all of one transaction happens first, then all of another transaction, etc.

Serializability

- The property of an interleaved / non-serial schedule of having an equivalent effect on a database as SOME serial (single-tasking) schedule.
- Not all serial schedules will result in the database being in the same state.
  - Example: (T1) Reassign property P37 from staff S50 to staff S02. (T2) Delete staff S02.
    - Do (T1) then (T2): P37 is assigned to default staff member. Or (T2) is rejected entirely and P37 remains assigned to S02.
    - Do (T2) then (T1): Operation (T1) is rejected. P37 is still assigned to S50.
- But none of the serial schedules will introduce inconsistency.

Concurrency algorithms

- Different ways of concurrent scheduling (multitasking) that generate a subset of acceptable (serializable) interleaved schedules.
  - Simple serializability,
  - Conflict serializability,
  - View serializability,
  - Locking,
  - Timestamping.

Simple serializability

- Operations that ONLY read can be run concurrently as any other read-only op.
- Operations that only involve reading/writing sets of data with NO overlap can be run concurrently.
- Avoid concurrent scheduling of any two operations not meeting above criteria.
- Easy to check this sort of serializability: scan through the operations noting whether they ever write and what pieces of data they touch.
  - OK transactions to run simultaneously if they meet the criteria.
  - Delay a transaction that conflicts with an ongoing transaction until after said transaction commits.

Conflicting transactions

- With simple serializability, we can’t multi-task on **conflicting transactions**: transactions that touch the same data and that feature at least one write.
- Can we do better?

Conflict serializability

- Can verify if a particular schedule involving conflicting transactions is serializable by constructing a **precedence graph** for a particular interleaved schedule.
- Conflict serializability = interleaved schedule is equivalent to a serial schedule ordered by precedence relationships necessitated by conflicting operations.

Precedence graph

- Nodes are transactions.
- Directed edge from $T_i \rightarrow T_j$ means that, in an equivalent serial schedule, T_i must be executed before T_j.
- Use edges $T_i \rightarrow T_j$ to keep track of conflicts that arise: Create edge $T_i \rightarrow T_j$ if…
  - In interleaved schedule, T_j reads the value of an item previously written by T_i.
    - T_j dependent on T_i's earlier write for the value it read and later used.
  - In interleaved schedule, T_j writes a value into an item previously read by T_i.
    - T_i's earlier read was dependent on the fact that T_j's write had NOT happened yet.
  - In interleaved schedule, T_j writes the value of an item previously written by T_i.
    - Value of item in equivalent serial schedule depends on T_i occurring before T_j.
- If precedence graph has a cycle, it is not conflict serializable.

Exercise: conflict serializability

- Use a precedence graph to check if the following schedule is conflict serializable:

  T1: Read(A)
  T2: Read(A)
  T1: Write(A)
  T2: Write(A)

Handling a non-serializable interleaving

- Need to abort and rollback one transaction involved in each cycle discovered.
- Will cause the cycle to become disconnected.

Can we do better?

- Interleaved schedules with **blind writes**:

  T1: Read(x)
  T2: [beginning of transaction]
        Write(x)
  T1: Write(x)
  T3: [beginning of transaction]
        Write(x)

  - NOT conflict serializable: T1’s read relies on T2’s write NOT having happened yet (T1 → T2). But T1’s erasure of T2’s write relies on T1 executing after T2 (T2 → T1).
  - But if we knew T3 was going to render T1 and T2’s ordering inconsequential…

View serializability

- Offers less stringent definition of schedule equivalence than conflict serializability.
  - Involves enforcing that the data *viewed* by transactions in the interleaved schedule could exactly match that viewed by one in a serial schedule.
- Two schedules S_1 (interleaved) and S_2 (serial) are view equivalent if:
  - For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.
    - View of database data when transactions start concords with a serial schedule.
  - For each read on x by T_i in S_1, if value read by T_i was written there by T_j, T_i must also read value of x produced by T_j in S_2 (i.e. occur after T_j).
    - For a narrower set of conflicts, we must enforce a precedence relationship.
  - For each data item x, if last write on x is performed by T_i in S_1, T_i must also perform final write on x in S_2.
    - View of data at conclusion of transactions is same between schedules.

Notes on view serializability

- Every conflict serializable schedule is view serializable, although the converse is not true.
  - View serializable schedules that are not conflict serializable contain one or more blind writes.
- Can test whether an interleaved schedule is view serializable by constructing a different set of graphs and checking if any of them are acyclic.
  - If ANY are acyclic the schedule is fine.
- However: construction algorithm is complicated and known difficult (NP-complete) to solve.

Serializability is not the only thing we care about

- Serializability guarantees that the effect on the database remains consistent.
- However, we also need to be careful about when commits (are allowed to) happen.

Non-recoverable schedules

Transaction #1:
Begin transaction:
    Read x
    Write x
    …
    …
    …
    [system failure or detection of non-serializability]
   ABORT

Transaction #2:

Begin transaction:
    Read x
    Write x
    … other work …
    COMMIT

- Transaction #1 has to be rolled back / undone.
- Transaction #2 therefore must also be rolled back as it read #1's intermediate work.
  - But it's already been committed!

Recoverability

- Atomicity requires that, if a transaction fails, all effects of the transaction must be undone.
- Durability requires that, once transactions commit, its changes *cannot* be undone (without running another, compensating transaction).
- Recoverable schedules never generate this dilemma.
  - Guarantees that commit operations are ordered such that, if a transaction T_j relied on a value written by T_i in its read, it will not commit before T_i commits.
    - Ti.commit() < Tj.commit() if Ti.write(x) < Tj.read(x) for all x

Other concurrency control algorithms

- Two main strategies for avoiding concurrency issues:
  - Locking.
  - Timestamping.
- Both are conservative / pessimistic in that they pre-emptively choose to restrict to serial processing when worried about a conflict.
- Contrast with optimistic methods that allow concurrent processing by default and check for issues only when transactions are attempting to commit.

Locking

Locks

- A lock allows claiming some resource so that other people's attempts to access that resource are temporarily denied.
- Transactions claim one of two types of locks on data items:
  - Read only / shared: When the transaction only needs to read the item.
    - Does not prevent any number of other transactions claiming a read lock but does prevent all transactions from claiming a write lock / writing until relinquished.
  - Write / exclusive: When the transaction needs to write the item.
    - Prevents other transactions from reading OR writing.
- Some systems allow the type of lock to be changed mid-transaction.
  - Upgraded (read → write) or downgraded (write → read)

Some ways of locking don't guarantee serializability or recoverability

Transaction #1:	Transaction #2:
Lock_write(x)	
Read(x)	
Write(x)	
Unlock(x)	
	Lock_write(x)
	Read(x)
	Write(x)
	Unlock(x)
	Lock_write(y)
	Read(y)
	Write(y)
	Unlock(y)
Lock_write(y)	**COMMIT**
Read(y)	
Write(y)	
Unlock(y)	
COMMIT	

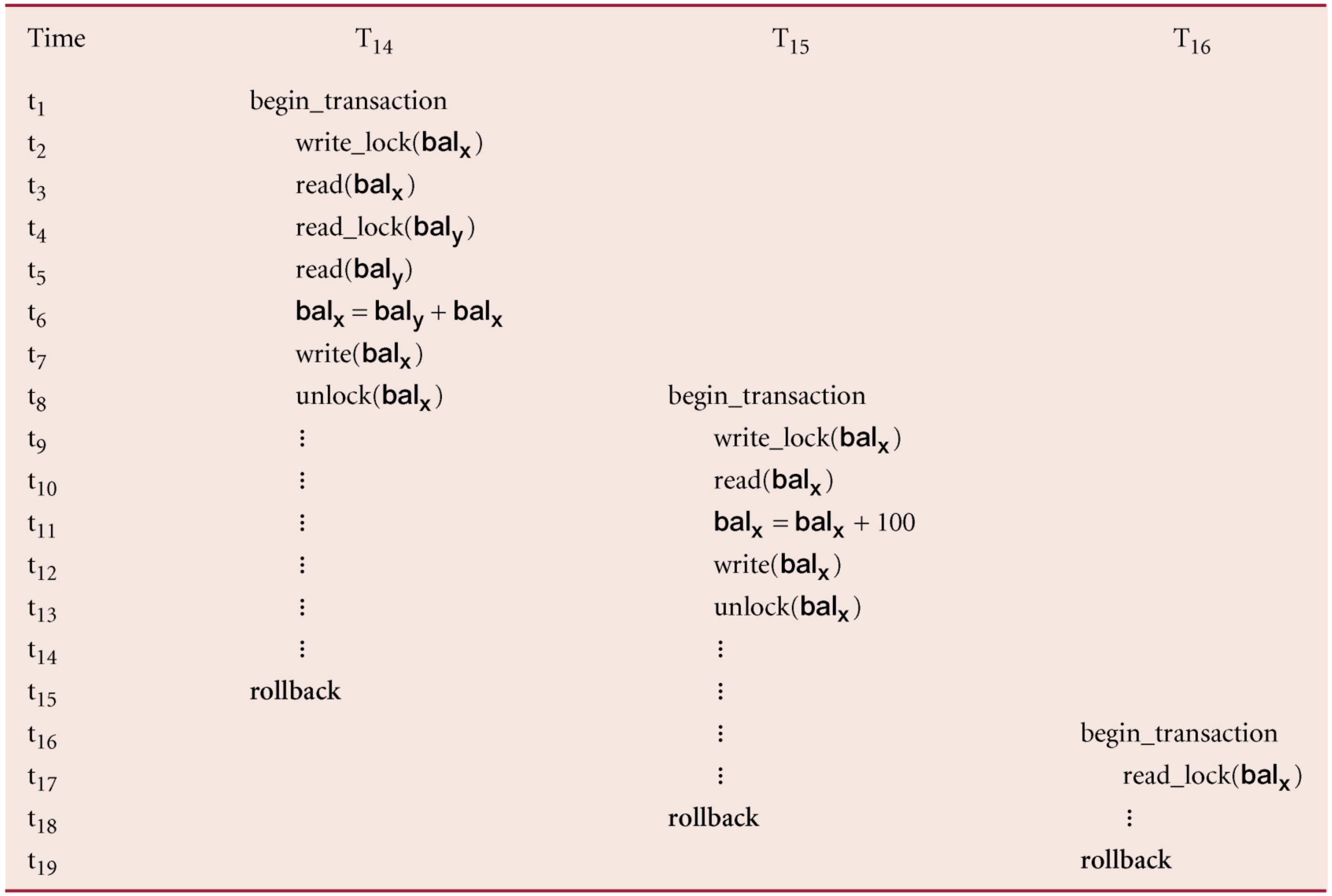
Locking protocols

- What went wrong?
  - One transaction should not have let another touch data that might potentially be rolled back.
  - That is, locks were released too early.
- Need to adhere to a safe/good locking practice:
  - **Two-phase locking (2PL)** protocol, which involves acquiring all locks in a transaction before releasing ANY.
    - Monotonically increasing locking (growing) phase,
    - Followed by a monotonically decreasing unlocking (shrinking) phase.
  - If all transactions use this protocol, conflict serializability is guaranteed.

Use stricter 2PL to prevent cascading rollbacks

- 2PL guarantees we can recover from failure by rolling back transactions.
- However, a rollback can trigger other transactions that saw the first transaction's changes to also need to be rolled back.
  - And the transactions that saw those intermediate results from those transactions…
- Avoid cascading rollbacks by waiting until the end of the transaction to release all locks at once (**rigorous 2PL**)

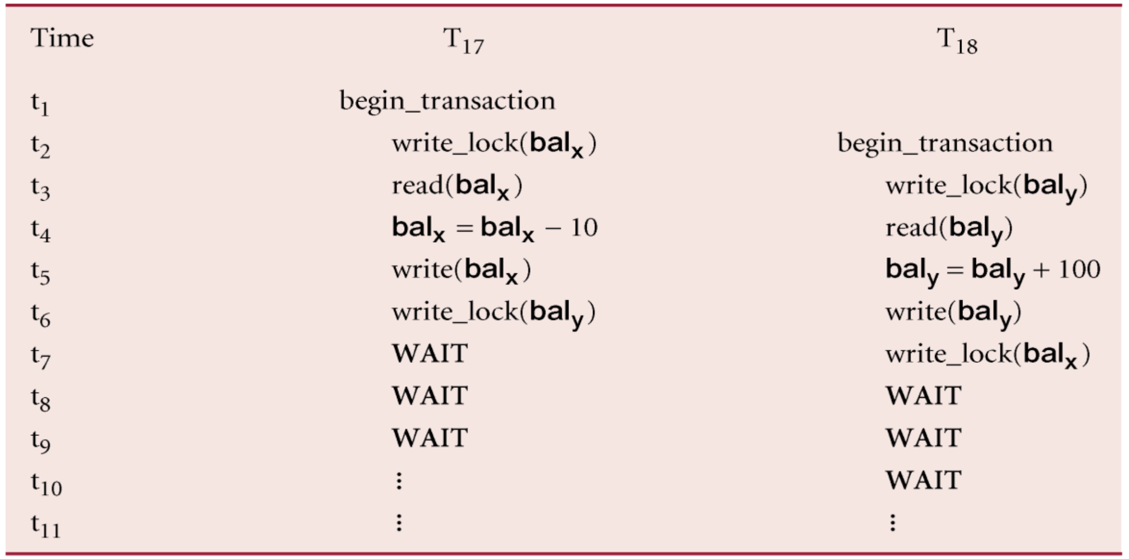
Cascading rollback example



Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock(**bal_x**)		
t_3	read(**bal_x**)		
t_4	read_lock(**bal_y**)		
t_5	read(**bal_y**)		
t_6	$bal_x = bal_y + bal_x$		
t_7	write(**bal_x**)		
t_8	unlock(**bal_x**)	begin_transaction	
t_9	⋮	write_lock(**bal_x**)	
t_{10}	⋮	read(**bal_x**)	
t_{11}	⋮	$bal_x = bal_x + 100$	
t_{12}	⋮	write(**bal_x**)	
t_{13}	⋮	unlock(**bal_x**)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock(**bal_x**)
t_{18}		rollback	⋮
t_{19}			rollback

Deadlock and livelock

- Problems unique to locking systems:
- Deadlock: 2+ transactions stuck because the locks required to continue are being held by the other(s).
- Livelock: transaction kept waiting indefinitely trying to acquire a lock (starvation)



Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(**bal**$_x$)	begin_transaction
t_3	read(**bal**$_x$)	write_lock(**bal**$_y$)
t_4	**bal**$_x$ = **bal**$_x$ − 10	read(**bal**$_y$)
t_5	write(**bal**$_x$)	**bal**$_y$ = **bal**$_y$ + 100
t_6	write_lock(**bal**$_y$)	write(**bal**$_y$)
t_7	WAIT	write_lock(**bal**$_x$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Dealing with deadlock

- To solve a deadlock, one or more transactions must be aborted.
- Three main techniques for handling deadlock:
  - Timeouts: all lock requests have a timeout after which the whole transaction is aborted and restarted.
  - Prevention: use a policy to avoid situations where deadlock may occur.
  - Detection & recovery: periodically generate graph to detect ongoing deadlock. Detector chooses which one(s) must abort.

Deadlock prevention

- Could use conservative 2PL (all locks needed in a transaction are obtained at transaction start),
  - But in practice, not practical and not used.
- Instead, use transaction timestamps to determine whether a transaction should wait on conflicting transactions or abort (die).
  - Wait-Die: an older transaction is allowed to wait for younger transactions to complete and relinquish locks. Younger transactions, if they require locks from an older transaction, must abort and try again (with same timestamp).
  - Wound-Wait: a younger transaction is allowed to wait for older transactions to complete and relinquish locks. However, if an older transaction wants a lock held by a younger transaction, younger transaction is forced to abort immediately (wounded).
- Prevents cyclical waiting patterns.

Deadlock detection and recovery

- At regular intervals, detector wakes up and does a deadlock check.
  - Constructs a wait-for graph with edges Ti → Tj if transaction Ti is waiting to lock an item locked by Tj.
  - Cycle in graph = deadlock.
- Recovery: detector chooses which transaction(s) to abort and restart, favoring younger, smaller, less finished transactions with fewer previous abortions.
  - Need to be careful to avoid starvation.

Timestamping

Timestamping concurrency control

- Assign each transaction a timestamp based on when it starts.
- Use timestamps to automatically abort and restart a transaction (the accessing transaction) when a possible conflict occurs.
- No locks so no deadlock.

Using timestamps to check for conflicts

- Timestamping assumes the equivalent serial schedule will involve executing each transaction one at a time in timestamp order.
- For each item in the database, track two timestamps (pointers to important transactions):
  - The timestamp of the last transaction to write the data.
  - The timestamp of the last transaction to read the data.
- Use data timestamps to detect outdated read/write attempts (assuming timestamp order) and restart this (accessing) transaction with a new timestamp.

Timestamp example

Transaction #1:
  Read user's balance
  New balance = balance – 20
  Write new balance to user

Transaction #2:
  Read user's balance
  New balance = balance + 20
  Write new balance to user

Interleaving:
 T1: **Read** user's balance (100)
 T2: **Read** user's balance (100)
 T1: **Write** new balance to user (80)
 T2: **Write** new balance to user (120)

User table

Read timestamp	Write timestamp	User ID	Balance	…
		…	…	…
		[this user]	100	…
		…	…	…

Detecting conflicts

Transaction T attempts to read data item x:

- Check if ts(T) < write_timestamp(x). If so,
  - The data T wants to read has been written by a newer transaction!
  - Transaction T needs to be aborted and undone, and reordered to occur after conflicting transaction.

Transaction T attempts to write data item x:

- Check if ts(T) < read_timestamp(x). If so,
  - The old value of the data T wants to write has already been seen by a newer transaction!
  - Transaction T is aborted and shifted to occur after conflicting transaction.

Other cases:

Transaction T attempts to write data item x:

- If ts(T) < write_timestamp(x)…
  - A newer transaction (later in the serial order) has updated x's value.
  - It would be a mistake for T to write to x (would not match serial schedule).
  - But T does not need to be restarted either! It suffices to skip past T's attempted write entirely – **ignore obsolete write rule**.
- If none of the previous hold, do the read/write and update the data item's read/write timestamp with ts(T) as appropriate.

Timestamp example (cont.)

Transaction #1:
  Read user's balance
  New balance = balance – 20
  Write new balance to user

Transaction #2:
  Read user's balance
  New balance = balance + 20
  Write new balance to user

Interleaving:
  T1: **Read** user's balance (100)
  T2: **Read** user's balance (100)
  T1: **Write** new balance to user (80)
  T2: **Write** new balance to user (120)

User table

Read timestamp	Write timestamp	User ID	Balance	…
		…	…	…
2		[this user]	100	…
		…	…	…

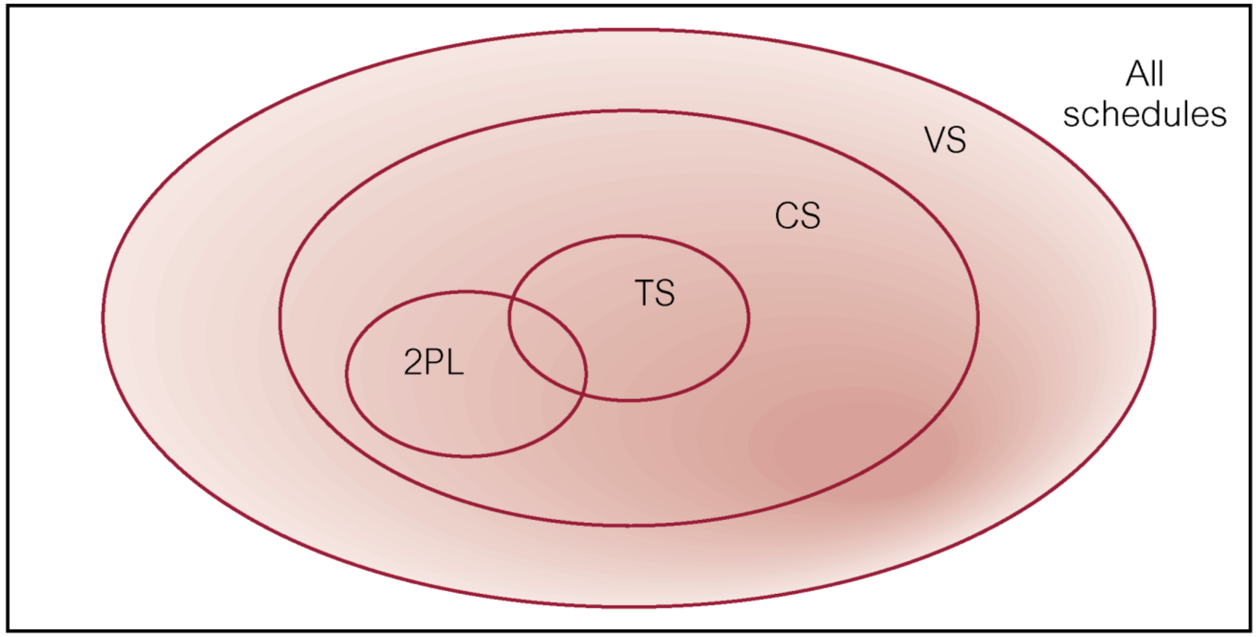
Timestamp of writing transaction (1)
<
read timestamp (2)

Versioning (Multiversion Timestamp Ordering)

- Versioning of data can be used to increase concurrency.
- Store multiple versions of a data item (complete with their own read/write timestamps) so transactions can select a consistent version and proceed without aborting in some cases.
  - In ALL cases for reads.
  - Use transaction's timestamp plus data items' read/write timestamps to select appropriate version.
- More concurrency in exchange for more storage required.

Comparison of concurrency methods

- Some interleaved schedules are possible under conflict serializable checking but not with locking or timestamping.
- Locking and timestamping always generate conflict serializable interleavings but not the same set.
- Some schedules possible with view serializable checking but not with conflict serializability checking.



Optimistic concurrency control

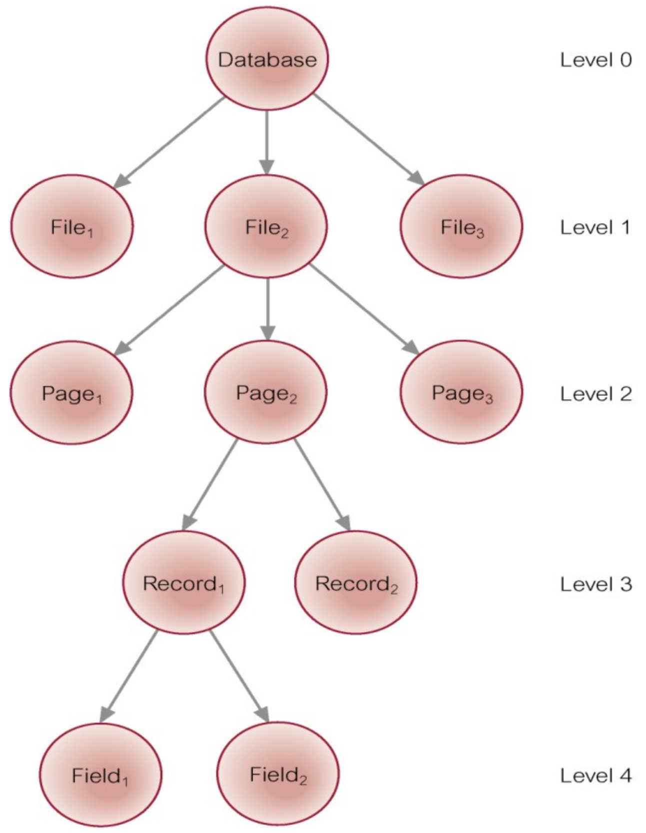
- Assume conflict is rare, so it is best to let operations run concurrently as a policy.
- How it works:
  - Track three timestamps per transaction: start time, start of validation, finish time.
  - Transactions alter *local* copies of data.
  - Before committing, check if a conflict occurred.
    - Earlier transaction committed after this transaction started, and updated data that this transaction read.
    - Earlier transaction hasn't successfully committed yet (list of changes not available).
  - If so, abort and restart.
    - Local changes mean no cascading rollbacks.
    - But abortion of long transactions can still be bad.

Granularity of data items

- Size of data items can range from one field of one record to entire database.
  - Granularity levels: database, file, page, record, field.
- Coarser granularity is low work/storage but also low concurrency.
- Fine granularity is high concurrency but storage-demanding.

Multiple-granularity locking

- For locking schemes, can represent database as a tree where parents contain children.
- Locking a node locks descendants and **intention locks** ancestors (flags to future transactions that a descendent is in use).
- Allows flexible usage of different lock granularities.



Example: multiple-granularity locking

T1: SELECT * FROM Staff

T2: INSERT (new property) INTO Property

- T1 read-locks Staff table. Intention locks whole database.
- T2 does not need whole database. Only needs a page (block) of the Property table where the new data will go. Can write-lock it.
- If a transaction intended to touch the whole database or parts of the Staff table, it would need to wait until T1 relinquishes its lock.

Summary

- Ensuring serializability can be done in many ways using various tools.
  - Often requires enforcing some ordering in the presence of potential conflicts.
  - Do this with timestamps, from examining conflicting ops, etc.
- Also need to incorporate care about when commits happen to avoid non-recoverable schedules.
- Such a system guarantees ACID properties for multi-operation transactions.

Recovery

Types of failures

- DBMSs have to deal with two types of issues:
  - Loss of main memory (temporary storage that is lost when computer shuts down).
    - Contents of buffers may be lost.
  - Loss of permanent storage of the database.
- Can happen for all sorts of reasons (software bug/crash, hardware failure, natural disaster, intentional sabotage, etc.)

Restoring database after a failure

- It’s possible for a transaction to commit before its changes make it to permanent storage, due to the time it takes to write (flush) buffers.
- In case of failure:
  - Uncommitted transactions have to be undone.
  - Committed transactions have to be checked if their changes were applied to the database and redone if not.
- How to do this?
  - Logs!

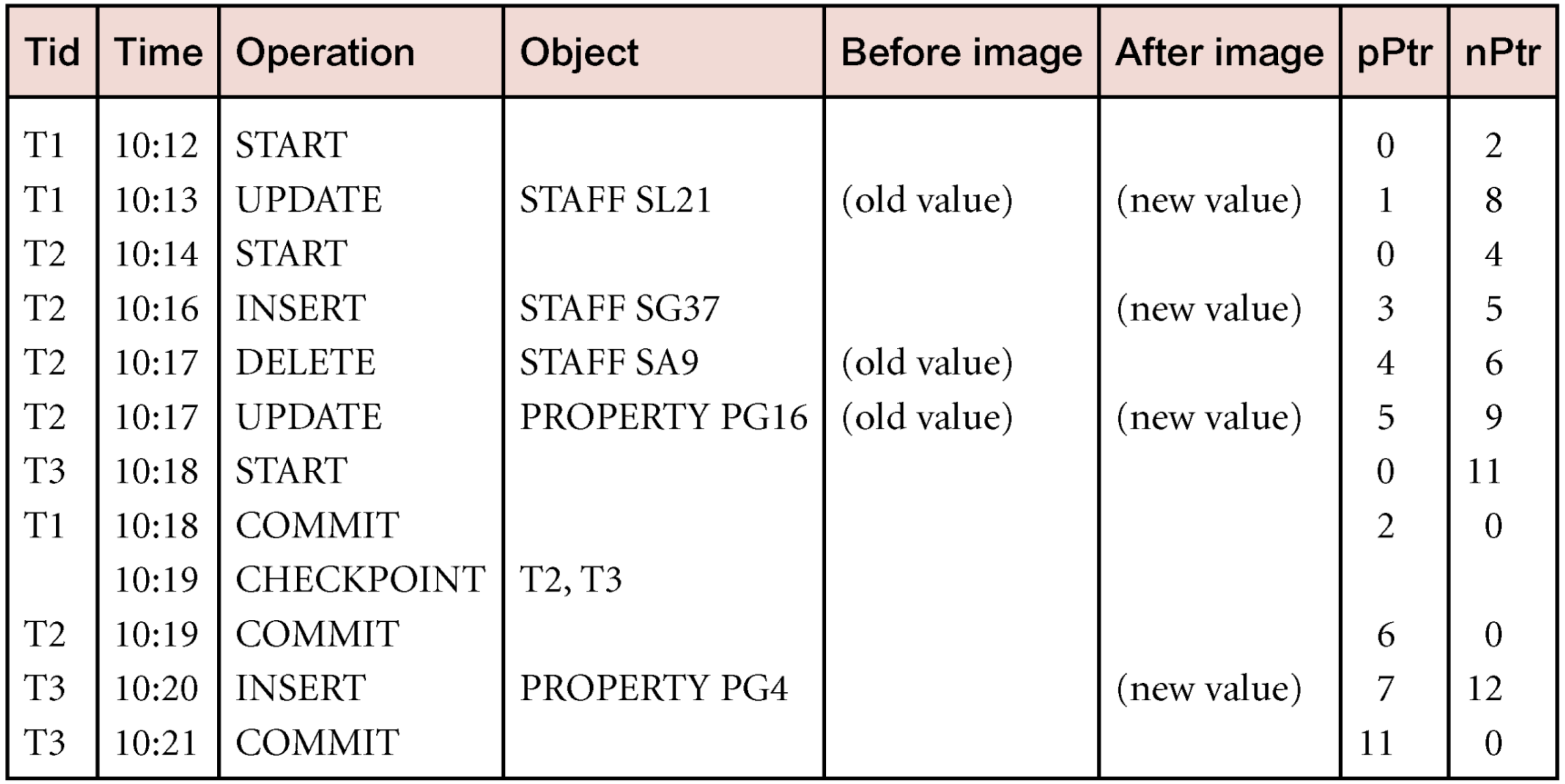
Recovery tools

- **Backups** (of both database and logs) should be made regularly.
  - Good for when permanent storage is damaged.
  - Starting place for restoring database state.
- **Log/journal** recording transaction and checkpoint operations.
  - Has other uses outside of recovery.
  - Note: may be a performance bottleneck.
- Regular **checkpoints** reduce the number of transactions to check when restoring the database.
  - Good for when database is intact but possibly inconsistent.

Log file: contents

- Transaction ID
- Timestamp
- Operation
- Data item affected
- Before-image / after-image (old/new values) of data item if applicable
- Helper info to locate next op associated with this transaction

Log file: example



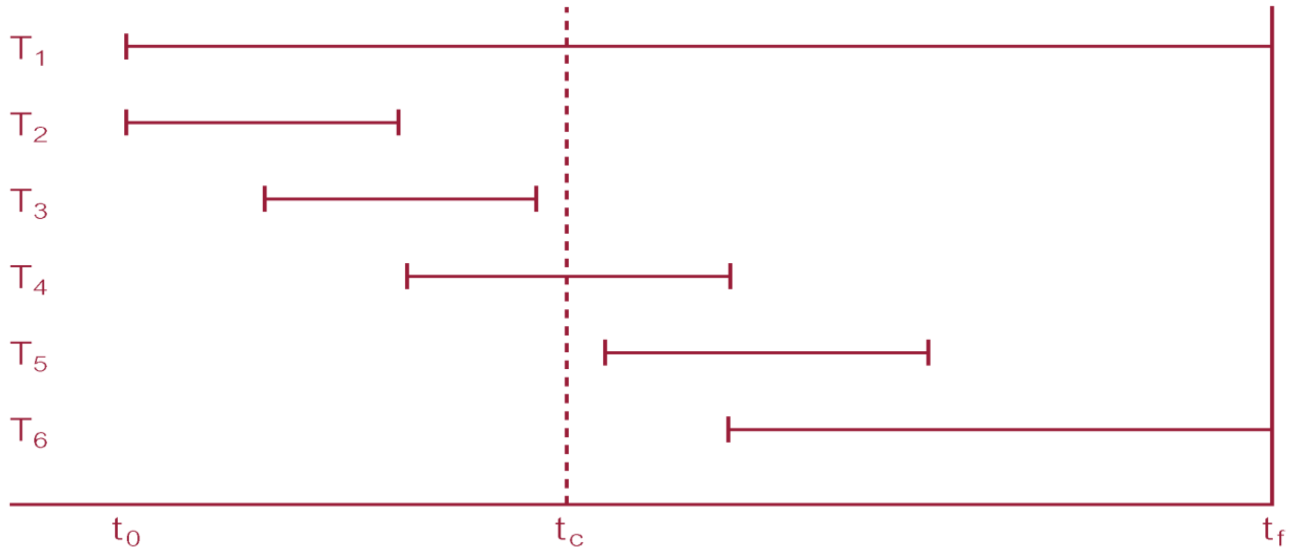
Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

Checkpoints

- All buffers are force-written to permanent storage.
- Record the checkpoint: make note of all transactions active at the time it was performed.
- After failure, undo all active transactions; redo all transactions committed after the checkpoint happened.

Checkpoint example

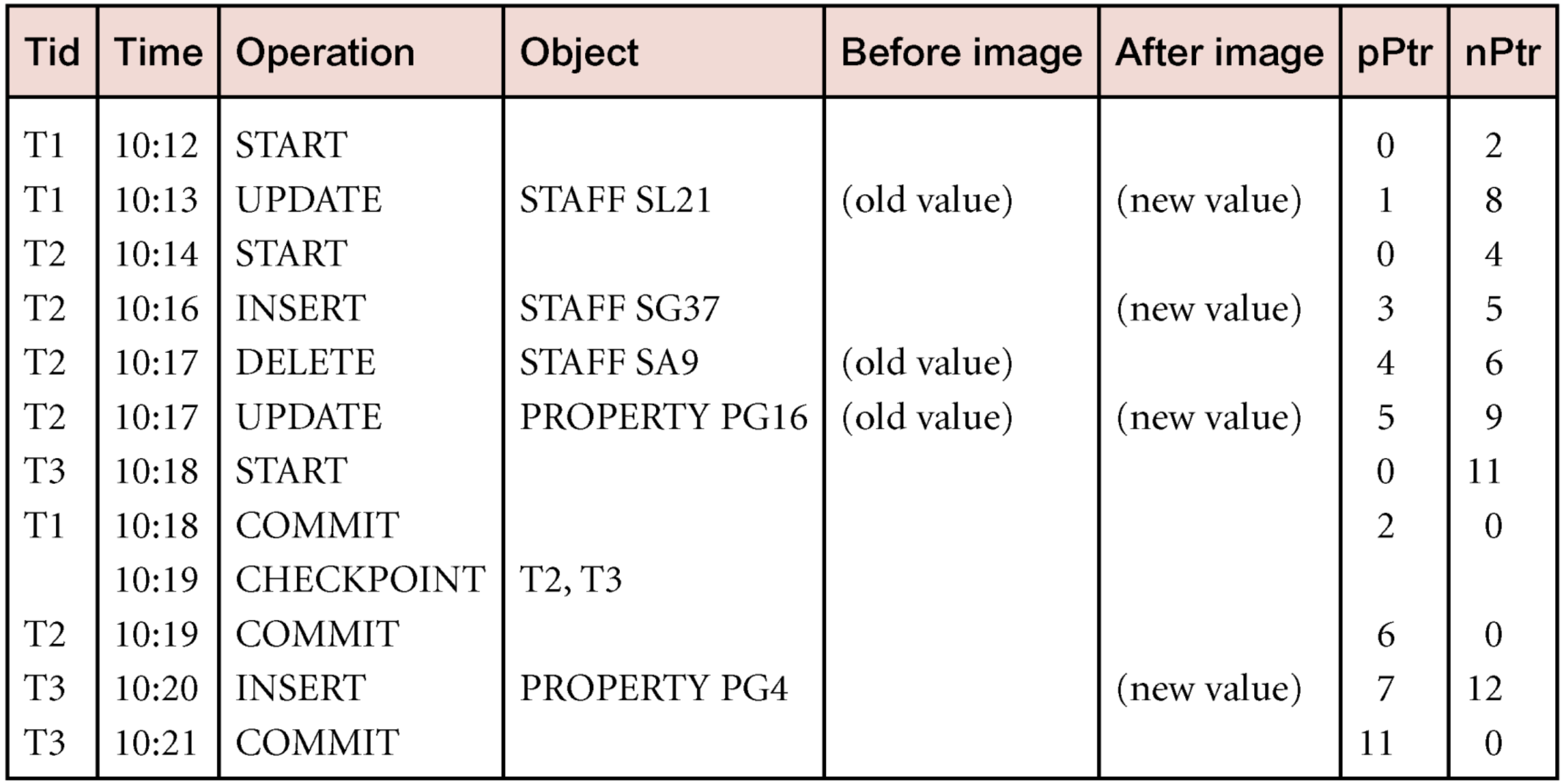
- Start of DBMS session at t_0.
- Take checkpoint at t_c.
- Fail at t_f.
- What transactions:
  - Are in secondary storage?
  - Must be redone?
  - Must be undone?



Recovery strategy

- If data has been lost:
  - Restore from backup and reapply committed transactions recorded in log.
    - Store backups, log, and database in separate locations.
- If data has only become inconsistent due to in-progress transactions:
  - Don’t need backup; apply changes to database.
  - Undo changes that caused inconsistency.
  - Redo committed transactions.

Recovery: example



Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

Main recovery techniques

- Three ways of handling storage and operations that change recovery strategy:
  - Deferred update
  - Immediate update
  - Shadow paging

Deferred update

- Updates are not written to the database until after a transaction has reached its commit point.
- If transaction fails before commit, it will not have modified database and so no undoing of changes required.
- May be necessary to redo updates of committed transactions as their effect may not have reached database.
- Simple.

Immediate update

- Updates written to database when buffers are flushed even if transaction that created them has not committed.
- Need to redo updates of committed transactions following a failure.
- Need to undo effects of transactions that had not committed at time of failure (no commit recorded in log).
  - Apply before-images recorded in log of aborted transactions in reverse order.
  - Need a **write-ahead log protocol** where write ops are always recorded before being performed.

Shadow paging

- Logless alternative featuring two page tables for each transaction.
  - Page tables are identical at start.
  - Shadow page table: doesn’t change and becomes the backup.
  - Current page table: records updates to values and becomes the shadow page on commit.
- Avoids overhead of log, but requires regular defragmentation and garbage recovery.

Advanced transaction models

- Models for advanced DBs may result in long or multi-agent transactions which make:
  - Concurrency or recovery difficult.
    - Longer transactions → touches many data items, higher likelihood of failure, less concurrent, more deadlock.
  - ACID properties (especially isolation) difficult or undesirable to maintain.
- Example: booking flights, hotels, and rental car, which may require making requests of multiple sites.

Techniques for complex transactions

- Break down large transactions into independent units.
- Run independent units simultaneously.
- Still need top-level transaction to ensure atomicity of the whole transaction depending on the success of the parts.
- Define ways of publishing desired effects / task requirements so that multi-agent systems can cooperate to perform or rollback a transaction.

Nested transactions

- Allow an atomic transaction to be broken down into parts that are fairly independent and can be run concurrently.
  - Example: booking flight, booking hotel, booking car rental.
    - Flight is further broken down into booking legs of the journey, which might involve multiple airlines.
- Parent transactions can specify their own recovery strategy.
- Whole transaction still has ACID properties.

Case study: transaction management in Oracle

- Oracle uses versioning to give transactions a stable/consistent view of data (multiversion read consistency)
  - Reads are never blocked.
- User can specify whether a set of statements is important enough to be guaranteed ACID properties as a transaction.
  - Otherwise, they are treated as separate operations, which are ACID individually.
- Oracle automatically handles the locking of data items needed to accomplish statement/transaction ACID, but user can manually override this behaviour.
  - Implements deadlock detection.
- Includes backup and recovery utilities.