

CS5412: TRANSACTIONS (I)

Lecture XVI

Ken Birman

Transactions

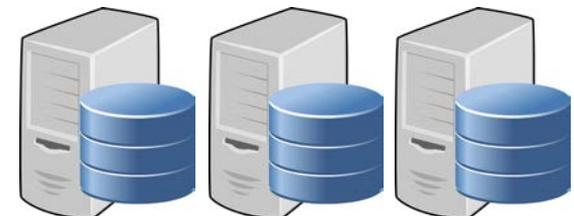
2

- A widely used reliability technology, despite the BASE methodology we use in the first tier
- Goal for this lecture and the next one: in-depth examination of topic
 - ▣ How transactional systems really work
 - ▣ Implementation considerations
 - ▣ Limitations and performance challenges
 - ▣ Scalability of transactional systems
- Transactions live “deeper” in the cloud, not tier 1 / 2.

3-Tier Architecture

3

- A question of terminology
 - **Client** application: Runs on a smart phone, etc.
 - **Tier1**: Your code that receives client system requests.
 - Those requests are customized by you: a procedure-call API
 - So you provide tier1 logic to carry out those requests
 - **Tier2**: Services tightly bound to tier1 code that often run as libraries right in the tier1 program's address space. Examples: Vsync, Dynamo, Cassandra
- **Tier3**: global file system and transactional services



Things to notice

4

- There could be a lot of clients, maybe millions
- There can be a lot of tier1 /tier2 instances
 - ▣ Lightweight, they run in the “soft state” layer of the cloud, it launches or shuts them down on demand
 - ▣ As casual as starting or stopping a normal program
- Tier3: relatively few servers, they run continuously and don't get launched or stopped so casually

How can t1 /t2 talk to t3?

5

- As you know, when a client application talks to t1, it looks like a procedure call
 - ▣ Under the surface request is encoded in a web page, sent via HTTP or HTTPS, then decoded. Same for reply.
- If t1 code talks to t3, similar mechanisms are used.
 - ▣ Requests don't use a web page encoding, but they are put in a message that leaves the computer and crosses the data center network, and the reply similarly comes back
 - ▣ This is somewhat costly and slower than calling a method in a library linked to your program

Transactions

6

- There are several perspectives on how to achieve reliability
 - ▣ We've talked at some length about non-transactional replication via multicast
 - ▣ Another approach focuses on reliability of communication channels and leaves application-oriented issues to the client or server – “stateless”
 - ▣ But many systems focus on the data managed by a system. This yields transactional applications

Transactions on a single database:

7

- In a client/server architecture,
- A transaction is an execution of a single program of the application(client) at the server.
 - ▣ Seen at the server as a series of reads and writes.
- We want this setup to work when
 - ▣ There are multiple simultaneous client transactions running at the server.
 - ▣ Client/Server could fail at any time.

The ACID Properties: Reminder

8

- Atomicity
 - ▣ All or nothing.
- Consistency:
 - ▣ Each transaction, if executed by itself, maintains the correctness of the database.
- Isolation (Serializability)
 - ▣ Transactions won't see partially completed results of other non-committed transactions
- Durability
 - ▣ Once a transaction commits, future transactions see its results

CAP conjecture



9

- Recall Brewer's CAP theorem: "you can't use transactions at large scale in the cloud".
- We saw that the real issue is mostly in t1 /t2: highly scalable and elastic outer tiers ("soft state tiers").
- In fact cloud systems use transactions all the time, but they do so in the "back end", and they shield that layer as much as they can to avoid overload

Transactions in the real world

10

- In cs5142 lectures, transactions are treated at the same level as other techniques
- But in the real world, transactions represent a huge chunk (in \$ value) of the existing market for distributed systems!
 - ▣ The web is gradually starting to shift the balance (not by reducing the size of the transaction market but by growing so fast that it is catching up)
 - ▣ On the web, we use transactions when we buy products
- So the real reason we don't emphasize them is this issue of them not working well in the first tier

The transactional model

11

- Applications are coded in a stylized way:
 - begin transaction
 - Perform a series of read, update operations
 - Terminate by commit or abort.
- Terminology
 - The application is the transaction manager
 - The data manager is presented with operations from concurrently active transactions
 - It schedules them in an interleaved but serializable order

A side remark

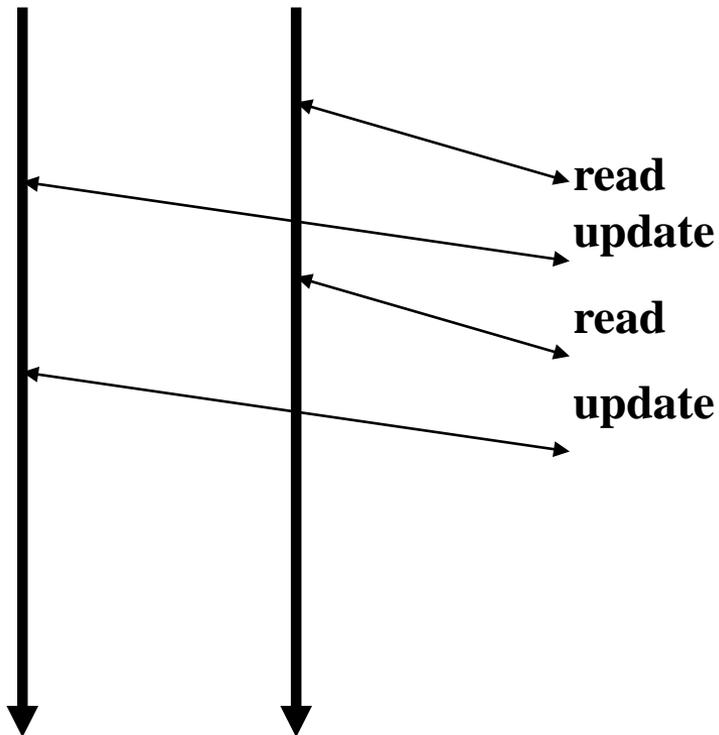
12

- Each transaction is built up incrementally
 - ▣ Application runs
 - ▣ And as it runs, it issues operations
 - ▣ The data manager sees them one by one
- But often we talk as if we knew the whole thing at one time
 - ▣ We're careful to do this in ways that make sense
 - ▣ In any case, we usually don't need to say anything until a "commit" is issued

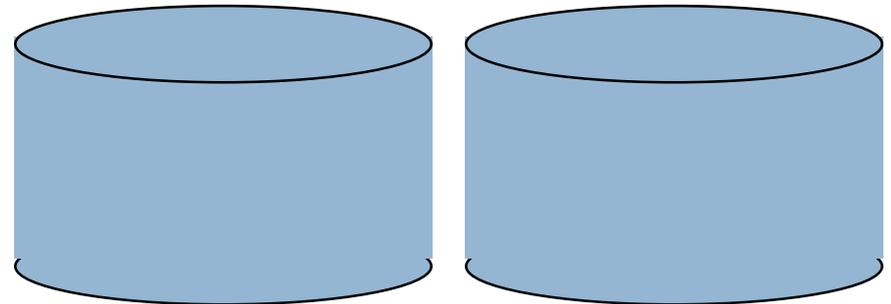
Transaction and Data Managers

13

Transactions



Data (and Lock) Managers



transactions are stateful: transaction “knows” about database contents and updates

Typical transactional program

14

```
begin transaction;
```

```
  x = read("x-values", ....);
```

```
  y = read("y-values", ....);
```

```
  z = x+y;
```

```
  write("z-values", z, ....);
```

```
commit transaction;
```

What about locks?

15

- Unlike some other kinds of distributed systems, transactional systems typically lock the data they access
- They obtain these locks as they run:
 - ▣ Before accessing “x” get a lock on “x”
 - ▣ Usually we assume that the application knows enough to get the right kind of lock. It is not good to get a read lock if you’ll later need to update the object
- In clever applications, one lock will often cover many objects

Locking rule

16

- Suppose that transaction T will access object x .
 - ▣ We need to know that first, T gets a lock that “covers” x
- What does coverage entail?
 - ▣ We need to know that if any other transaction T' tries to access x it will attempt to get the same lock

Examples of lock coverage

17

- We could have one lock per object
- ... or one lock for the whole database
- ... or one lock for a category of objects
 - ▣ In a tree, we could have one lock for the whole tree associated with the root
 - ▣ In a table we could have one lock for row, or one for each column, or one for the whole table
- All transactions must use the same rules!
- And if you will update the object, the lock must be a “write” lock, not a “read” lock

Transactional Execution Log

- As the transaction runs, it creates a history of its actions. Suppose we were to write down the sequence of operations it performs.
- Data manager does this, one by one
- This yields a “schedule”
 - ▣ Operations and order they executed
 - ▣ Can infer order in which transactions ran
- Scheduling is called “concurrency control”

Observations

- Program runs “by itself”, doesn’t talk to others
- All the work is done in one program, in straight-line fashion. If an application requires running several programs, like a C compilation, it would run as several separate transactions!
- The persistent data is maintained in files or database relations external to the application

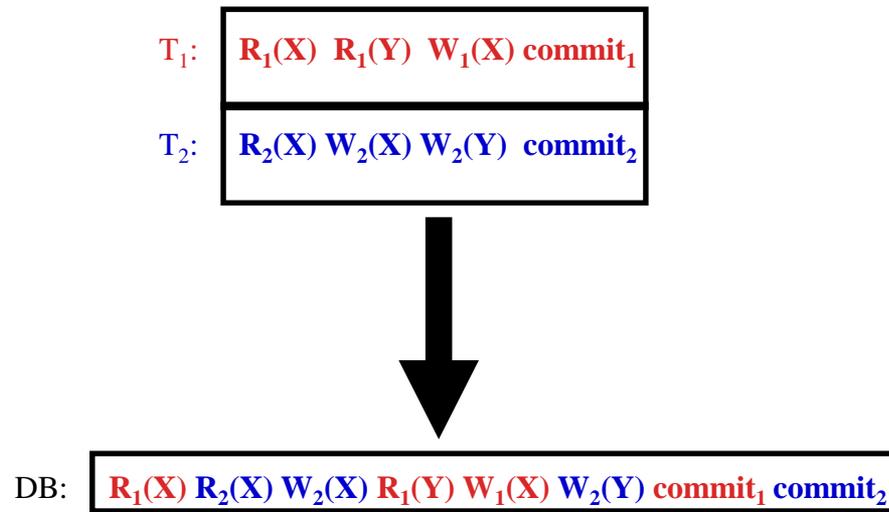
Serializability

20

- Means that effect of the interleaved execution is indistinguishable from some possible serial execution of the committed transactions
- For example: T1 and T2 are interleaved but it “looks like” T2 ran before T1
- Idea is that transactions can be coded to be correct if run in isolation, and yet will run correctly when executed concurrently (and hence gain a speedup)

Need for serializable execution

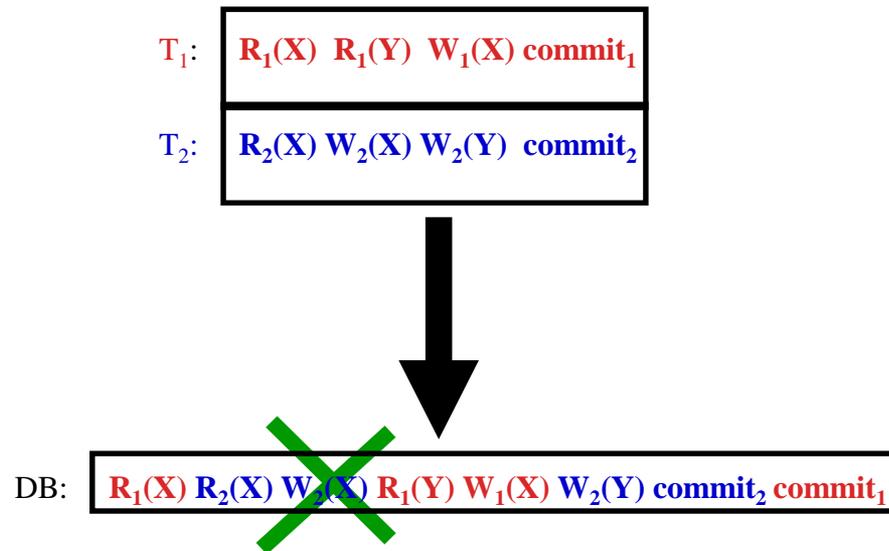
21



Data manager interleaves operations to improve concurrency

Non serializable execution

22

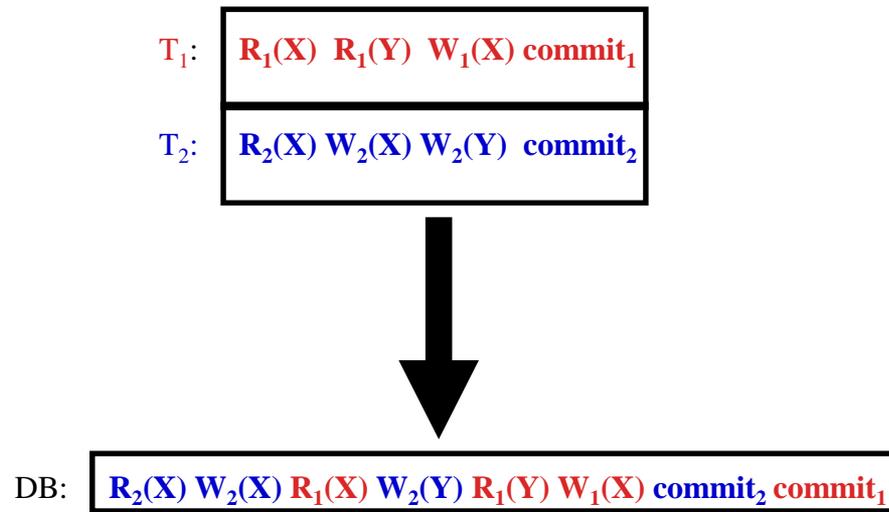


Unsafe! Not serializable

Problem: transactions may “interfere”. Here, T₂ changes x, hence T₁ should have either run first (read and write) or after (reading the changed value).

Serializable execution

23



Data manager interleaves operations to improve concurrency but schedules them so that it looks as if one transaction ran at a time. This schedule “looks” like T₂ ran first.

But our serializable example has issues

24

- Our serializable example shows T_1 reading from T_2 before T_2 commits.
 - In practice, if T_2 had aborted, T_1 would also have to abort, so T_1 can't commit until we learn the outcome for T_2
 - We say that T_2 has become “dependent” on T_1
 - In general, database systems that are aggressive about allowing interleaving can make their commit decisions complex.
- Are there simple ways to allow interleaved executions but also be sure it will be easy to commit?

Atomicity considerations

25

- If application (“transaction manager”) crashes, treat as an abort
- If data manager crashes, abort any non-committed transactions, but committed state is persistent
 - ▣ Aborted transactions leave no effect, either in database itself or in terms of indirect side-effects
 - ▣ Only need to consider committed operations in determining serializability

Components of transactional system

26

- Runtime environment: responsible for assigning transaction id's and labeling each operation with the correct id.
- Concurrency control subsystem: responsible for scheduling operations so that outcome will be serializable
- Data manager: responsible for implementing the database storage and retrieval functions

Transactions at a “single” database

27

- Normally use 2-phase locking or timestamps for concurrency control
- Intentions list tracks “intended updates” for each active transaction
- Write-ahead log used to ensure all-or-nothing aspect of commit operations
- Can achieve thousands of transactions per second

Strict two-phase locking: how it works

28

- Transaction must have a lock on each data item it will access.
 - ▣ Gets a “write lock” if it will (ever) update the item
 - ▣ Use “read lock” if it will (only) read the item. Can’t change its mind!
- Obtains all the locks it needs while it runs and hold onto them even if no longer needed
- Releases locks only after making commit/abort decision and only after updates are persistent

Why do we call it “Strict” two phase?

29

- 2-phase locking: Locks only acquired during the ‘growing’ phase, only released during the ‘shrinking’ phase.
- Strict: Locks are only released after the commit decision
 - ▣ Read locks don’t conflict with each other (hence T’ can read x even if T holds a read lock on x)
 - ▣ Update locks conflict with everything (are “exclusive”)

Strict Two-phase Locking

30

T₁: begin read(x) read(y) write(x) commit

T₂: begin read(x) write(x) write(y) commit

Acquires locks

Releases locks

- A very simple rule: no lock can be released by a transaction until all locks have been acquired
- Limits concurrency but easy to implement, so popular

Notes

31

- Notice that locks must be kept even if you are certain that the locked objects won't be revisited
 - ▣ This can be a problem in long-running applications!
 - ▣ Also becomes an issue in systems that crash and then recover
 - Often, they “forget” locks when this happens
 - Called “broken locks”. We say that a crash may “break” current locks...
 - ▣ The policy is conservative: the “serializable” interleaved execution from a few slides back wouldn't be allowed because for T_1 to run while T_2 was still active, T_1 needed a lock that T_2 would still have been holding.

Why does strict 2PL imply serializability?

32

- Suppose that T' will perform an operation that conflicts with an operation that T has done:
 - T' will update data item X that T read or updated
 - T updated item Y and T' will read or update it
- T must have had a lock on X/Y that conflicts with the lock that T' wants
- T won't release it until it commits or aborts
- So T' will wait until T commits or aborts

Acyclic conflict graph implies serializability

33

- Can represent conflicts between operations and between locks by a graph (e.g. first T1 reads x and then T2 writes x)
- If this graph is acyclic, can easily show that transactions are serializable
- Two-phase locking produces acyclic conflict graphs

Two-phase locking is “pessimistic”

34

- Acts to prevent non-serializable schedules from arising: pessimistically assumes conflicts are fairly likely
- Can deadlock, e.g. T1 reads x then writes y; T2 reads y then writes x. This doesn't always deadlock but it is capable of deadlocking
 - ▣ Overcome by aborting if we wait for too long,
 - ▣ Or by designing transactions to obtain locks in a known and agreed upon ordering

Contrast: Timestamped approach

35

- Using a fine-grained clock, assign a “time” to each transaction, uniquely. E.g. T1 is at time 1, T2 is at time 2
- Now data manager tracks temporal history of each data item, responds to requests as if they had occurred at time given by timestamp
- At commit stage, make sure that commit is consistent with serializability and, if not, abort

Example of when we abort

36

- T1 runs, updates x , setting to 3
- T2 runs concurrently but has a larger timestamp. It reads $x=3$
- T1 eventually aborts
- ... T2 must abort too, since it read a value of x that is no longer a committed value
 - ▣ Called a cascaded abort since abort of T1 triggers abort of T2

Pros and cons of approaches

37

- Locking scheme works best when conflicts between transactions are common and transactions are short-running
- Timestamped scheme works best when conflicts are rare and transactions are relatively long-running
- Weihl has suggested hybrid approaches but these are not common in real systems

Intentions list concept

38

- Idea is to separate persistent state of database from the updates that have yet to commit
 - ▣ Many systems update in place, roll back on abort. For these, a log of prior versions is needed.
 - ▣ A few systems flip this and keep a list of what changes they intend to make. Intentions list may simply be the in-memory cached database state (e.g. change a cached copy, but temporarily leave the disk copy).
- Either way, as a transaction runs it builds a set of updates that it intends to commit, if it commits

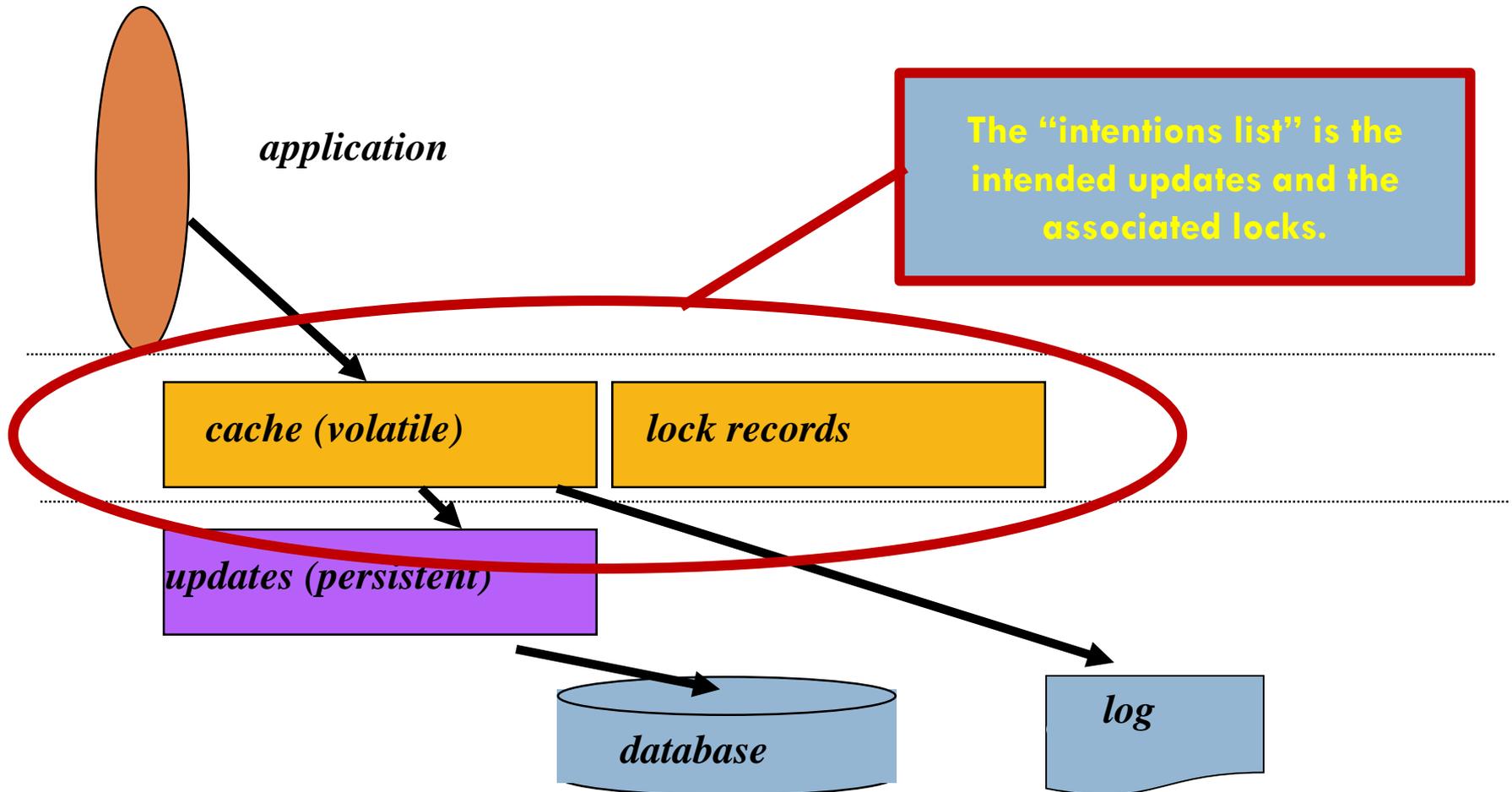
Role of write-ahead log

39

- Used to save either old or new state of database to either permit abort by rollback (need old state) or to ensure that commit is all-or-nothing (by being able to repeat updates until all are completed)
- Rule is that log must be written before database is modified
- After commit record is persistently stored and all updates are done, can erase log contents

Structure of a transactional system

40



Recovery?

41

- Transactional data manager reboots
- It rescans the log
 - ▣ Ignores non-committed transactions
 - ▣ Reapplies any updates
 - ▣ These must be “idempotent”
 - Can be repeated many times with exactly the same effect as a single time
 - E.g. $x := 3$, but not $x := x.\text{prev} + 1$
- Then clears log records
- (In normal use, log records are deleted once transaction commits)

Transactions in distributed systems

42

- Notice that client and data manager might not run on same computer
 - ▣ Both may not fail at same time
 - ▣ Also, either could timeout waiting for the other in normal situations
- When this happens, we normally abort the transaction
 - ▣ Exception is a timeout that occurs while commit is being processed
 - ▣ If server fails, one effect of crash is to break locks even for read-only access

Transactions in distributed systems

43

- What if data is on multiple servers?
 - ▣ In a non-distributed system, transactions run against a single database system
 - Indeed, many systems structured to use just a single operation – a “one shot” transaction!
 - ▣ In distributed systems may want one application to talk to multiple databases

Transactions in distributed systems

44

- Main issue that arises is that now we can have multiple database servers that are touched by one transaction
- Reasons?
 - ▣ Data spread around: each owns subset
 - ▣ Could have replicated some data object on multiple servers, e.g. to load-balance read access for large client set
 - ▣ Might do this for high availability
- Solve using 2-phase commit protocol!

Unilateral abort

45

- Any data manager can unilaterally abort a transaction until it has said “prepared”
- Useful if transaction manager seems to have failed
- Also arises if data manager crashes and restarts (hence will have lost any non-persistent intended updates and locks)
- Implication: even a data manager where only reads were done must participate in 2PC protocol!

Transactions on distributed objects

46

- Idea was proposed by Liskov's Argus group and then became popular again recently
- Each object translates an abstract set of operations into the concrete operations that implement it
- Result is that object invocations may “nest”:
 - ▣ Library “update” operations, do
 - ▣ A series of file read and write operations that do
 - ▣ A series of accesses to the disk device

Nested transactions

47

- Call the traditional style of flat transaction a “top level” transaction
 - ▣ Argus short hand: “actions”
- The main program becomes the top level action
- Within it objects run as nested actions

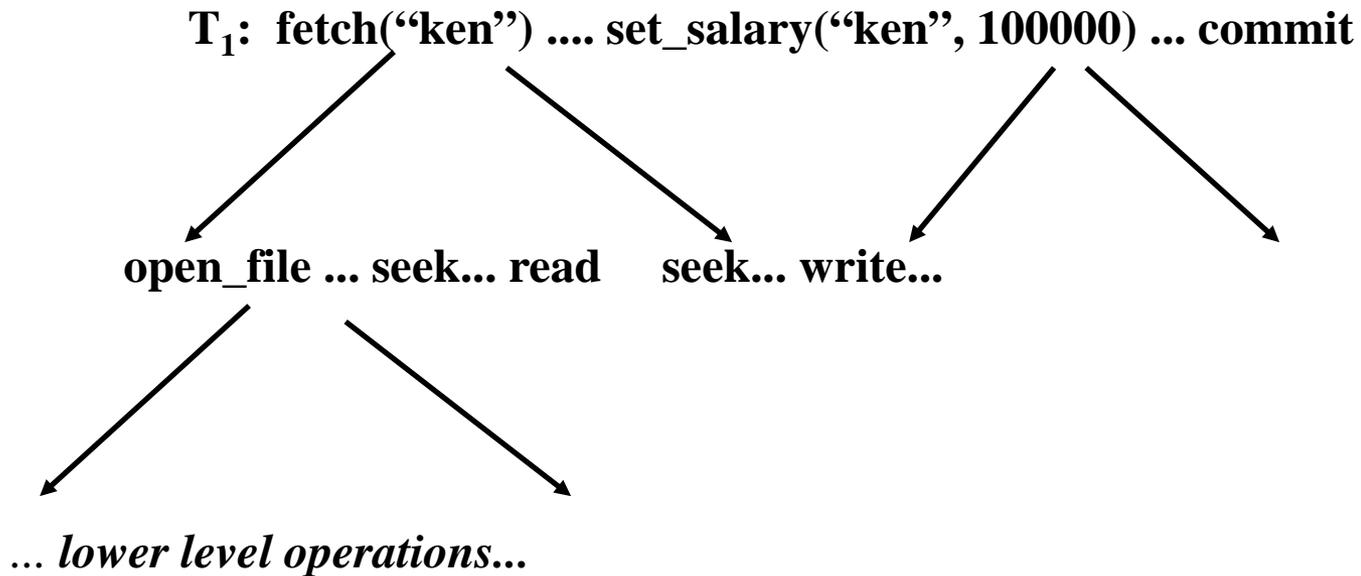
Arguments for nested transactions

48

- It makes sense to treat each object invocation as a small transaction: begin when the invocation is done, and commit or abort when result is returned
 - ▣ Can use abort as a “tool”: try something; if it doesn’t work just do an abort to back out of it.
 - ▣ Turns out we can easily extend transactional model to accommodate nested transactions
- Liskov argues that in this approach we have a simple conceptual framework for distributed computing

Nested transactions: picture

49



Observations

50

- Can number operations using the obvious notation
 - ▣ T1, T1.2.1.....
- Subtransaction commit should make results visible to the parent transaction
- Subtransaction abort should return to state when subtransaction (not parent) was initiated
- Data managers maintain a stack of data versions

Stacking rule

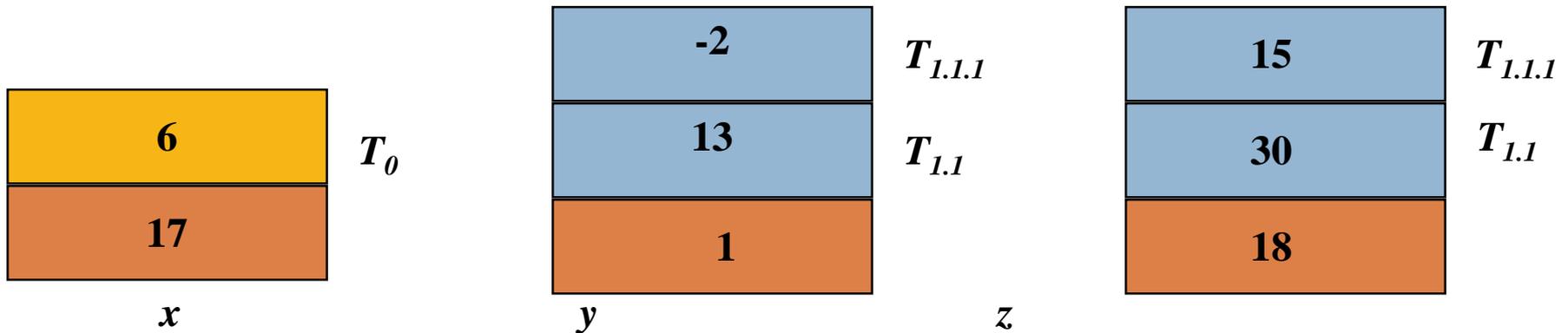
51

- Abstractly, when subtransaction starts, we push a new copy of each data item on top of the stack for that item
- When subtransaction aborts we pop the stack
- When subtransaction commits we pop two items and push top one back on again
- In practice, can implement this much more efficiently!!!

Data objects viewed as “stacks”

52

- Transaction T_0 wrote 6 into x
- Transaction T_1 spawned subtransactions that wrote new values for y and z



Locking rules?

53

- When subtransaction requests lock, it should be able to obtain locks held by its parent
- Subtransaction aborts, locks return to “prior state”
- Subtransaction commits, locks retained by parent
- ... Moss has shown that this extended version of 2-phase locking guarantees serializability of nested transactions

Relatively recent developments

54

- Many cloud-computing solutions favor non-transactional tables to reduce delays even if consistency is much weaker
 - ▣ Called the NoSQL movement: “Not SQL”
 - ▣ Application must somehow cope with inconsistencies and failure issues. E.g. your problem, not the platform’s.
- Also widely used: a model called “Snapshot isolation”. Gives a form of consistency for reads and for updates, but not full serializability

How does NoSQL work?

55

- We saw this a few lectures back
- You take a transactional program but then try and break it down into a non-transactional one that can run against data in a DHT without locks or consistency guarantees. Everything is a get/put, no joins or fancy database queries supported.
- Not always feasible, but if you pull it off, your code is much more scalable and faster.

What about snapshot isolation

56

- This is for real database transactions
- Basically the idea is to have the database do the updates, but handle the read-only transactions from cached data in a clever way that avoids locking.
- The problem is that it can sometimes violate the ACID guarantees. So it works pretty well but isn't perfect and those mistakes could be costly!

Summary

57

- Transactional model lets us deal with large databases or other large data stores
- Provides a model for achieving high concurrency
- Concurrent transactions won't stumble over one-another because ACID model offers efficient ways to achieve required guarantees