Transactions and Concurrency Control

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

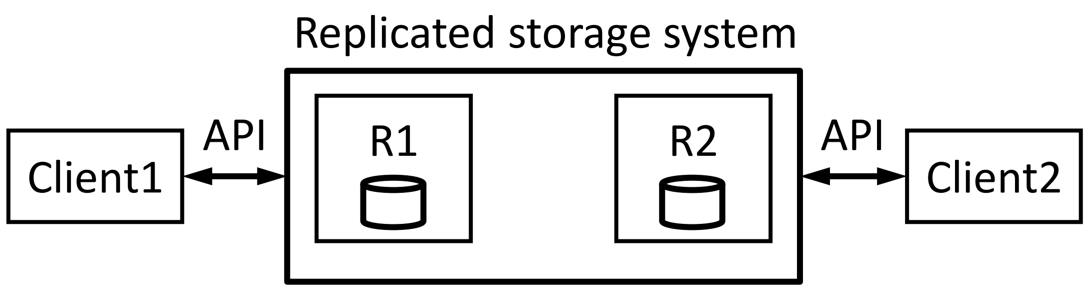
Distributed Systems
ECE419

Overview

- Introduction to transactions
- Concurrency control

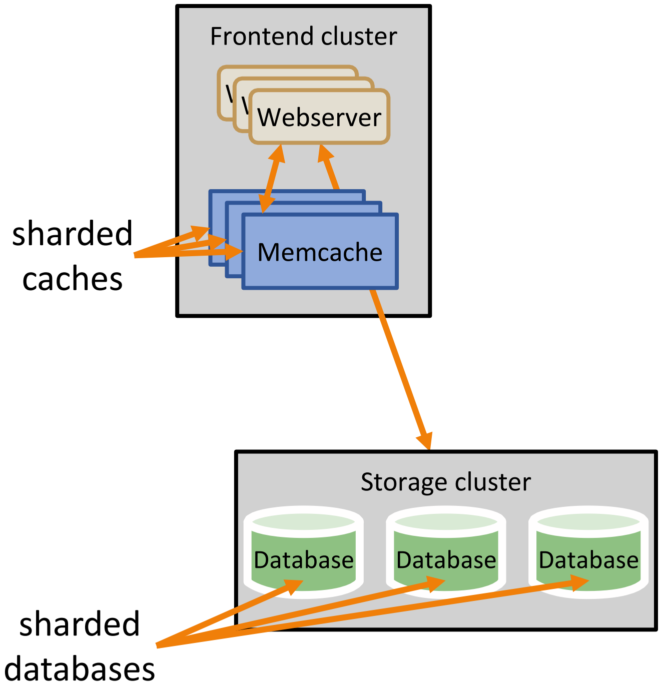
Fault-tolerant replicated systems

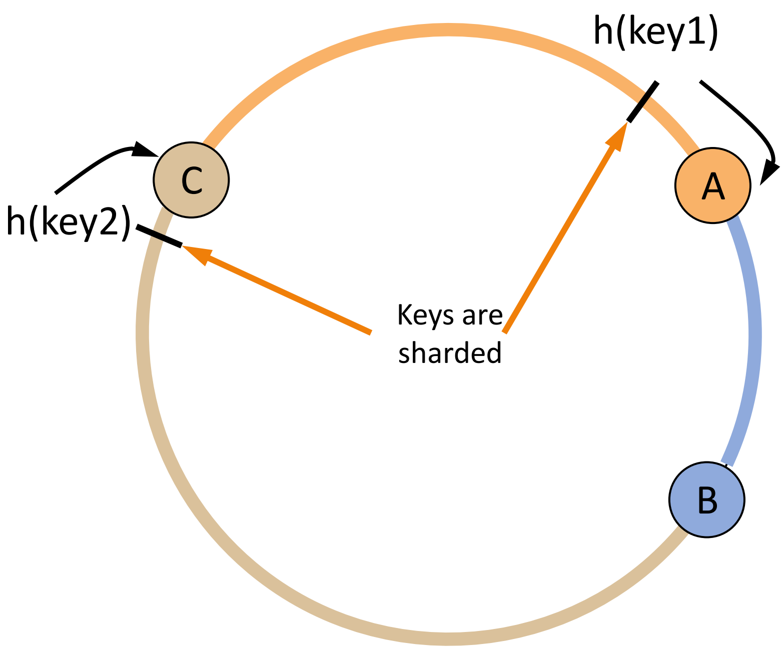
- We have seen systems that replicate data across nodes
  - E.g., Raft, ZooKeeper
- Replicated systems provide fault tolerance
  - Ideally, look like one reliable server



Scalable sharded systems

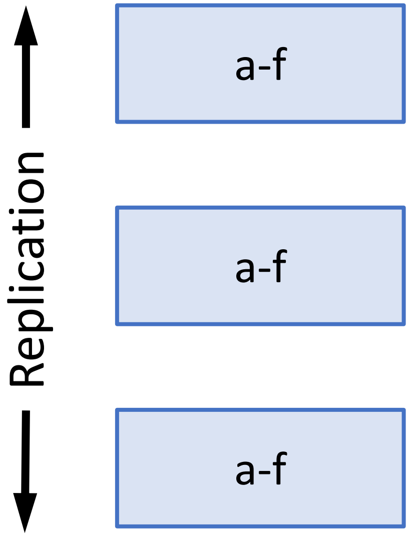
- We have also seen systems that shard data across nodes
  - E.g., Memcache, Dynamo
- Sharded systems enable scaling
  - Shards can be accessed in parallel





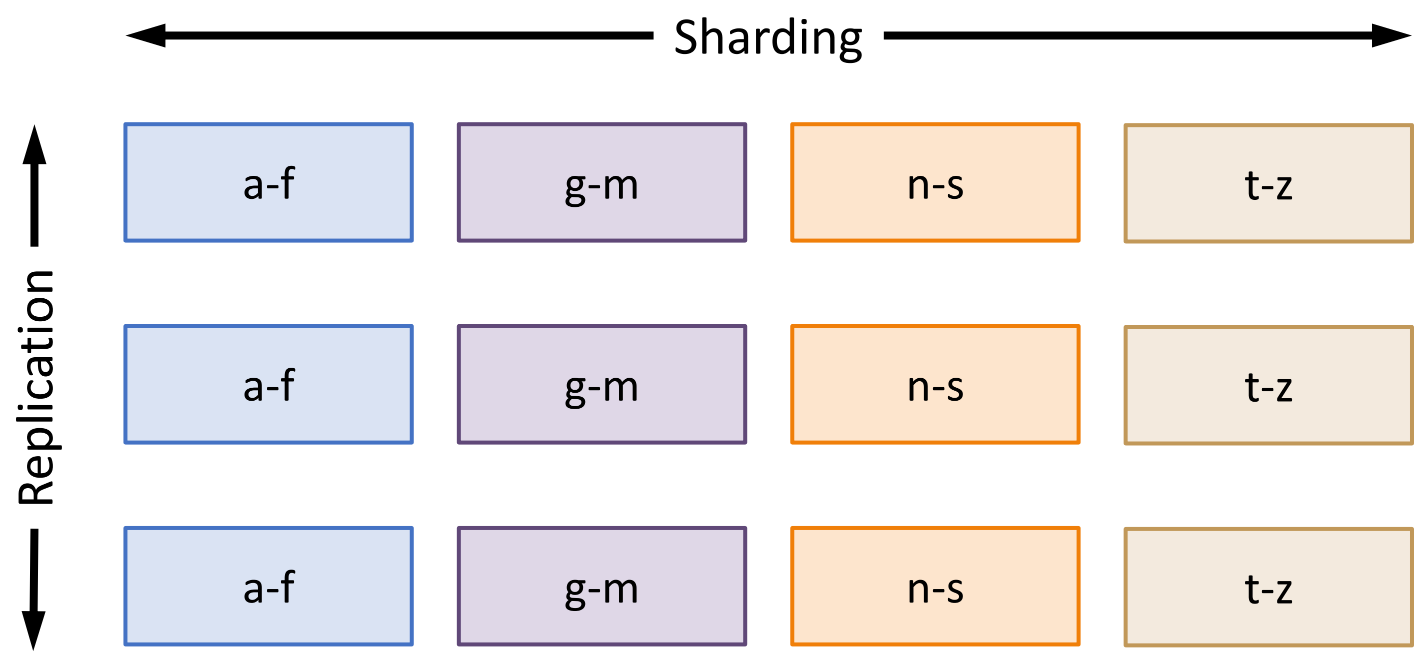
Combining replication and sharding

- Replication for fault tolerance



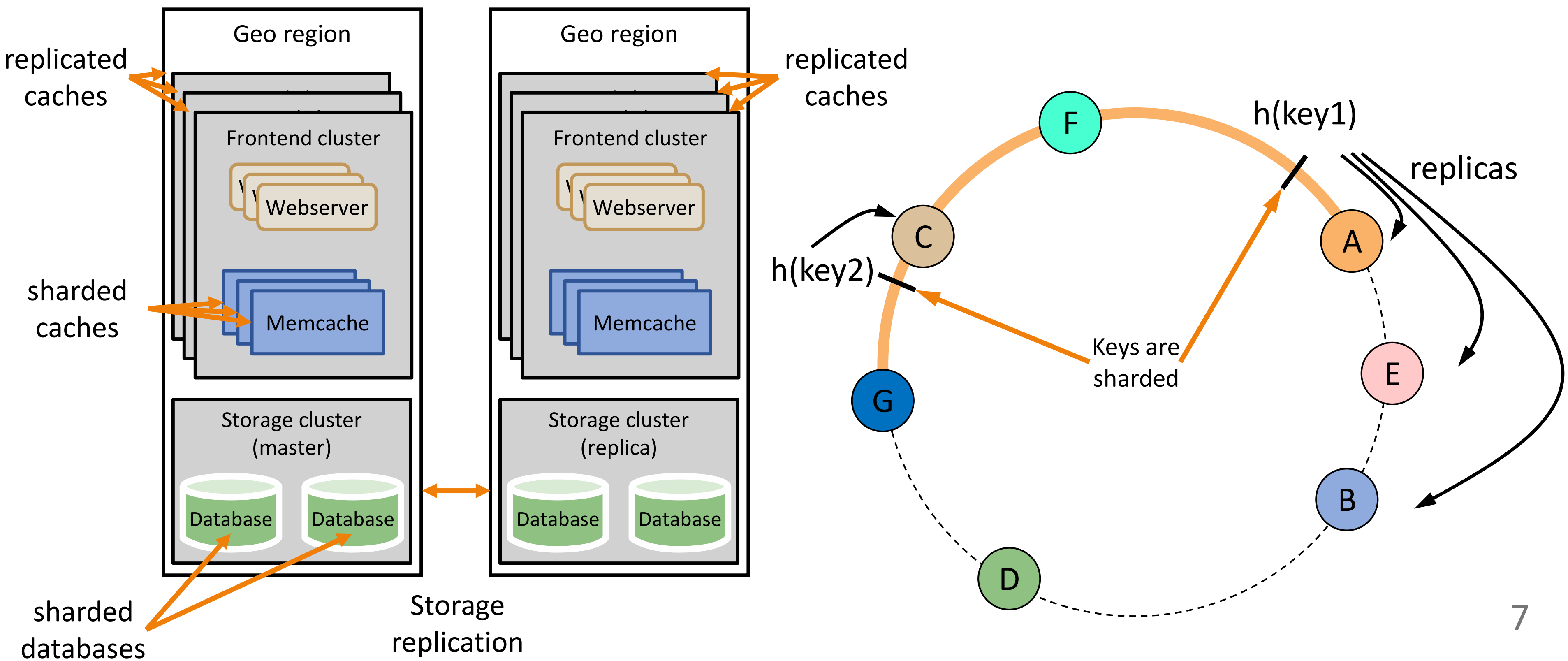
Combining replication and sharding

- Replication for fault tolerance
- Sharding for scalability



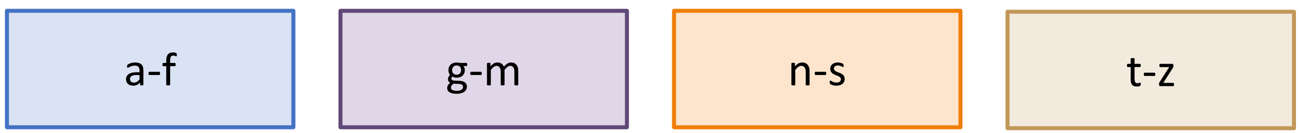
Scalable, fault-tolerant systems

- Real systems perform both
  - Replication for fault tolerance
  - Sharding for scalability



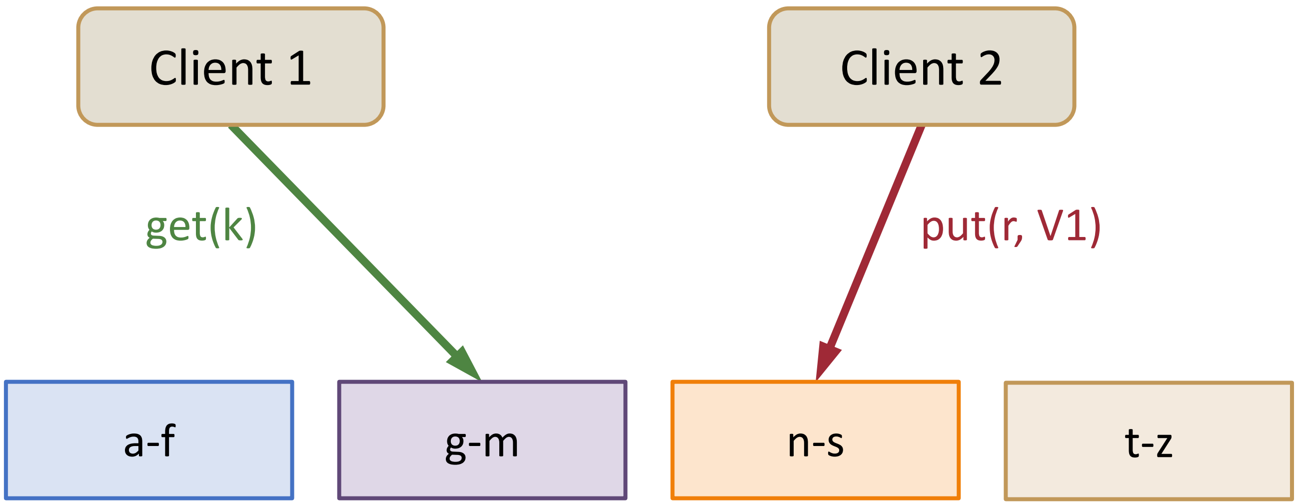
Sharding

- We will focus on sharded systems in this lecture



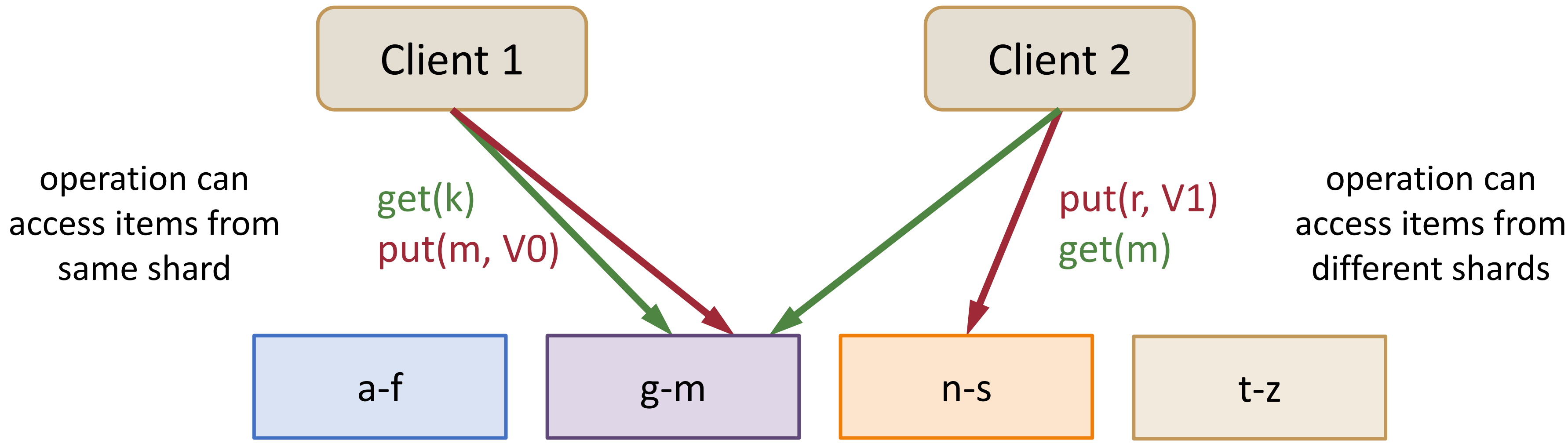
Operations access one item

- We have assumed operations access one item at a time

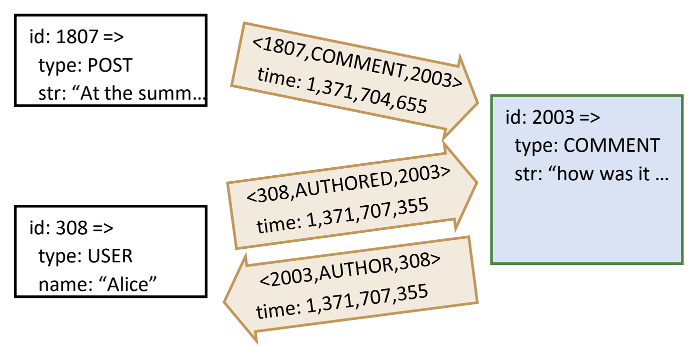


Operations access multiple items

- What if operations access multiple items at a time?

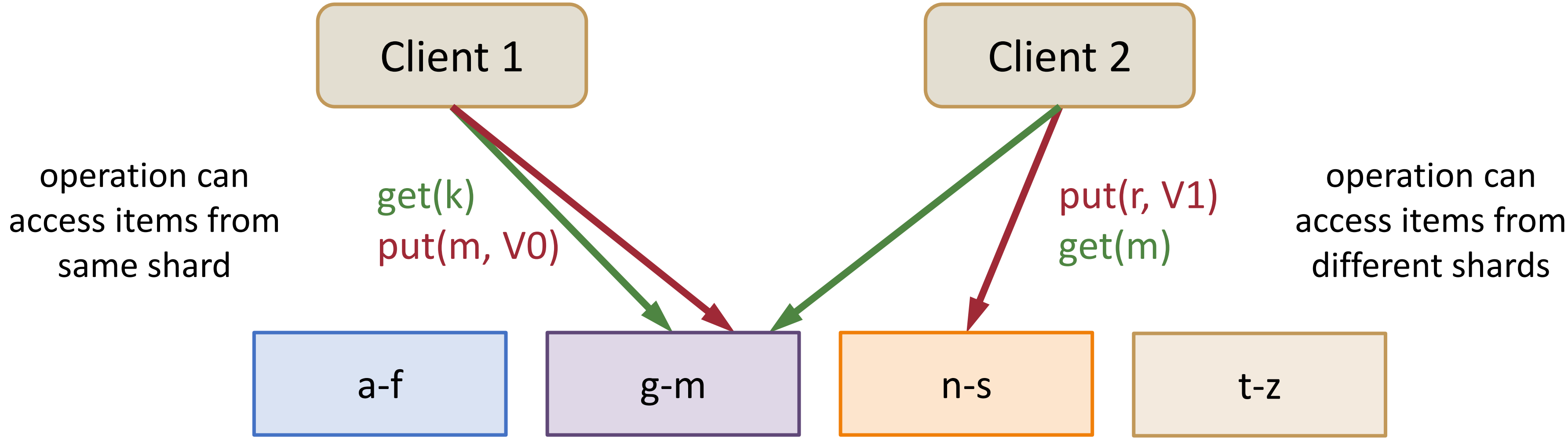


- Such operations are common
  - Create comment, add associations
  - Insert new record, add index entry for record



Operations access multiple items

- What if operations access multiple items at a time?



- We would like these operations to execute atomically
  - Appear to execute all accesses together (hide concurrency)
  - Appear to execute all accesses or none (hide failures)

Transactions

- We can use transactions, a well-known database solution
  - Programmer marks beginning/end of sequence of code
    - begin_tx: starts transaction
    - end_tx: transaction done
  - Code may access (e.g., read and write) multiple items (e.g., A, B)

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
    …
else
    write(A, a–10)
    b = read(B)
    write(B, b+10)
end_tx
```

```
sum(A, B):
begin_tx
a = read(A)
b = read(B)
print a + b
end_tx
```

Transaction commits

- When transaction is done, it is ready to commit
  - Commit may or may not succeed
  - If commit succeeds
    - All transaction modifications have been written to storage
    - Transaction results are sent to client

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
    …
else
    write(A, a–10)
    b = read(B)
    write(B, b+10)
end_tx
```

```
sum(A, B):
begin_tx
a = read(A)
b = read(B)
print a + b
end_tx
```

Transaction aborts

- When a transaction aborts (fails), all changes are undone
  - Aborts can occur for various reasons, at any time before commit
    - abort_tx: transaction code issues abort
    - System may force a transaction to abort, e.g., deadlock, out-of-memory
    - Server crashes, media failure

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
    abort_tx
else
    write(A, a-10)
    b = read(B)
    write(B, b+10)
end_tx
```

```
sum(A, B):
begin_tx
a = read(A)
b = read(B)
print a + b
end_tx
```

Transaction behavior

- System ensures transaction code runs atomically
  - System handles concurrent operations (e.g., via locking)
  - System adds failures (e.g., via crash recovery)
- Programmer is happy!

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
    abort_tx
else
    write(A, a-10)
    b = read(B)
    write(B, b+10)
end_tx
```

```
sum(A, B):
begin_tx
a = read(A)
b = read(B)
print a + b
end_tx
```

Transaction guarantees: ACID

- Atomic: transaction executes completely or not at all, despite failures
  - E.g., transfer(A, B) either commits or makes no changes
- Consistent: system ensures application-specific invariants
  - E.g., delete user and all user data together
- Isolated: no interference between concurrent transactions
  - E.g., sum(A, B) doesn’t read intermediate updates by transfer(A, B)
- Durable: committed transaction are not lost, despite failures

Transaction guarantees: ACID

- Atomic: transaction executes completely or not at all, despite failures
- Consistent: system ensures application-specific invariants
- Isolated: no interference between concurrent transactions
- Durable: committed transactions are not lost, despite failures

Guarantees about correctness under failures

Guarantee about correctness under concurrency

Concurrency Control

Isolation

- Goal: accesses in the transaction appear to happen together at a point in time
- Serial execution
  - Transactions are run in serial order, ensures isolation
  - Problem: poor performance, no concurrency possible
- Concurrent execution
  - Transactions are executed concurrently,
accesses are interleaved, provides good performance
  - Problem: certain interleaving of accesses may violate isolation, need to avoid them

Serializability

- A schedule is an ordering of the accesses (reads, writes) performed by a set of transactions
- A schedule is serializable if there exists some serial schedule that produces the same results
  - Results mean transaction outputs and database state
  - A serializable schedule provides isolation
    - Transactions appear to execute in some serial order (even if they don't)

Are schedules serializable?

r_A: read item A
w_A: write item A
©: commit txn

Assume A = 40, B = 20

transfer:	r_A	w_A	r_B	w_B	©			
sum:						r_A	r_B	©

Serializable

transfer:				r_A	w_A	r_B	w_B	©
sum:	r_A	r_B	©					

Serializable

transfer:	r_A	w_A				r_B	w_B	©
sum:			r_A	r_B	©			

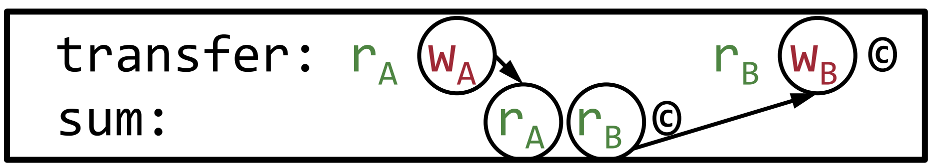
Non-serializable

transfer:		r_A	w_A			r_B	w_B	©
sum:	r_A			r_B	©			

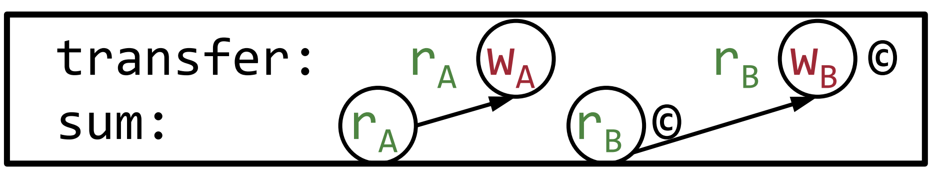
Serializable

Conflicts

- Two accesses from different transactions are conflicting if they operate on the same item and at least one is a write
  - Conflicting accesses (read-write, write-read, write-write) are non-commutative (cannot be reordered)
  - For serializability, conflicts must occur in same order



Non-serializable



Serializable

Implementing serializability

- Concurrent execution can violate serializability
  - We need to control concurrent execution to ensure serializability (i.e., so conflicts occur in same order), and so an implementation of isolation is also called concurrency control
- Two commonly used concurrency control schemes
  - Two-phase locking
  - Optimistic concurrency control

Two-phase locking (2PL)

- Every data item has an associated lock
  - Locks can be mutex or reader-writer locks
- Reader-writer locks
  - Shared: Acquire per-item lock before reading item
  - Exclusive: Acquire per-item lock before writing item

	Shared (S)	Exclusive (X)
Shared (S)	Yes	No
Exclusive (X)	No	No

2PL rule

- Once a transaction has released a lock,
  it is not allowed to acquire any other locks
  - Growing phase: transaction acquires locks on items it reads (read set) and writes (write set)
  - Shrinking phase: transaction releases locks
- In practice:
  - Growing phase is the entire transaction
  - Shrinking phase is after commit
  - Avoids the problem of transactions accessing data modified by a transaction that eventually aborts

2PL Example

- Database automatically
  - Acquires lock on first access to item
  - Releases lock on abort or commit

```
S(I):   acquire shared lock on item I
X(I):   acquire exclusive lock on item I
U(I):   release lock on item I
```

```
transfer(A, B):
begin_tx
a = read(A)              S(A)
if a < 10 then
    abort_tx             U(A)
else
    write(A, a-10)       X(A)
    b = read(B)          S(B)
    write(B, b+10)       X(B)
end_tx                   U(A),U(B)
```

```
sum(A, B):
begin_tx
a = read(A)     S(A)
b = read(B)     S(B)
print a + b
end_tx          U(A),U(B)
```

Are these schedules allowed under 2PL?

Assume A = 40, B = 20

transfer:	r_A	w_A	r_B	w_B	©			
sum:						r_A	r_B	©

Serializable, allowed

transfer:	r_A	w_A				r_B	w_B	©
sum:			r_A	r_B	©			

Non-serializable, not allowed

transfer:		r_A			w_A	r_B	w_B	©
sum:	r_A		r_B	©				

Serializable, allowed

transfer:	r_A	w_A		r_B	w_B	©		
sum:			r_A				r_B	©

Serializable, not allowed

Issues with 2PL

- What do we do if a lock is unavailable?
  - Wait: wait until lock becomes available
  - Die: give up immediately, i.e., abort
  - Wound: force the lock holder to abort to acquire lock
- Waiting for a lock can result in deadlock
  - Assuming order A and B are interchanged in the sum() code
    - Transfer has locked A, waits on B
    - Sum has locked B, waits on A
- Many ways to prevent, detect and handle deadlocks
  - Typically wait-die or wound-wait used for prevention

2PL is pessimistic

- Acquires locks before accesses
- Pros
  - Prevents all potential violations of serializability
- Cons
  - Conflicts lead to waiting on locks, which cause delays
  - Disallows certain concurrent accesses that are serializable

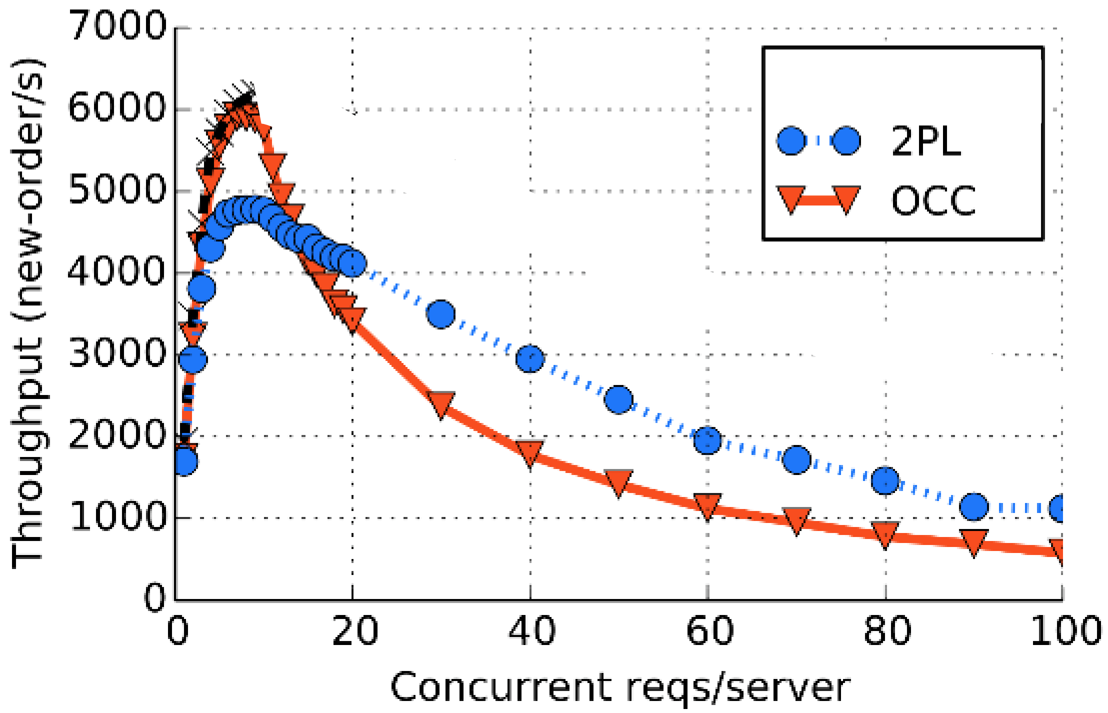
Optimistic Concurrency Control (OCC)

- Be optimistic, assume success!
  - Access items without locking, as if they will succeed
  - Only check whether reads/writes are serializable at commit time
  - If check fails, abort transaction
- Compared to locking, OCC has
  - Higher performance when transactions have few conflicts
  - Lower performance when transactions have many conflicts

OCC implementation

- Optimistic execution
  - Transaction executes initial reads from database (read set)
    - Caches reads locally, re-reads from cache
  - Buffers writes locally (write set)
- Validation and commit ⟵ Many ways to do validation
  1. Acquire shared locks on read set, exclusive locks on write set
  2. Validate (check) items in read set haven't changed
     - i.e., reading item in read set at commit would give the same result
  3. Apply buffered writes in write set to commit transaction
     - Abort if locks can't be acquired in Step 1 or validation fails in Step 2
  4. Release locks

2PL vs OCC: increasing conflict rate



From Rococo, OSDI 2014

Linearizability versus serializability

- Linearizability: a guarantee about single accesses on single items
  - Accesses (reads and write) have a total order
  - Once write completes, all reads that begin later (in real-time order) should reflect that write
- Serializability: a guarantee about multiple accesses on multiple items
  - Transactions appear to execute in some serial (total) order
  - Doesn’t impose any real-time constraints
- Strict serializability: intuitively, serializability + real-time constraints of linearizability

Conclusions

- Transactions enable executing operations atomically
  - All accesses appear to execute together (hide concurrency)
  - All accesses are executed or none or executed (hide failures)
- Concurrency control algorithms hide concurrency
  - Ensure serializability (equivalence to serial execution)
  - Two common methods: 2PL, OCC
  - 2PL better for high contention, OCC better for low contention
- Next, we will look at how transactions help hide failures