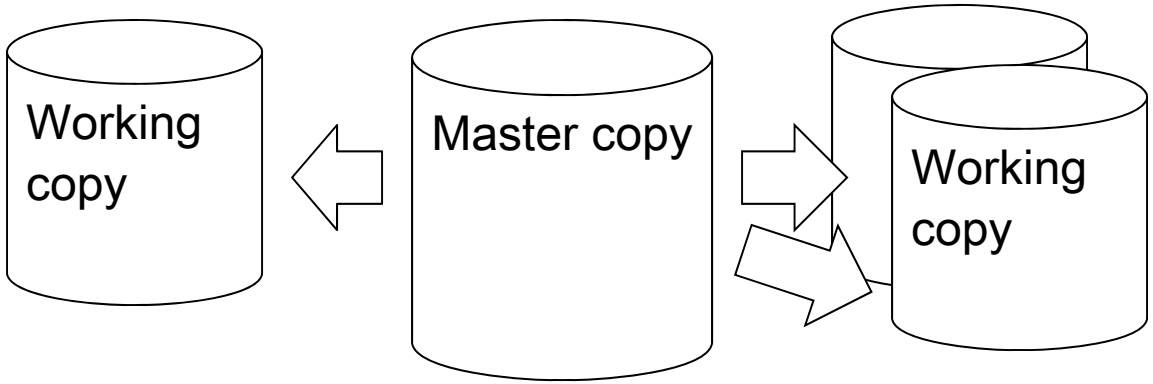
Multiversion Concurrency

- Motivation:
  - Replication
- Snapshots
- Write sets and read sets
- Conflict detection:
  - Snapshot Isolation
  - Optimistic strategies
- Phenomenon: Write skew

In-place Update vs. Multiversion

- So far we considered transaction management techniques where only a single copy of each data object exists.
- This approach gave rise to lock-based schedulers.
- There is an alternative approach that allows **multiple versions** of a data object to be active during **concurrent open transactions:** multiversion concurrency.
- This approach arises naturally in several designs for data intensive systems.
- We focus on one motivation in which the semantics of multiversion concurrency is easy to understand:
- A specific **database replication** protocol

Motivation: Replication with Master copy



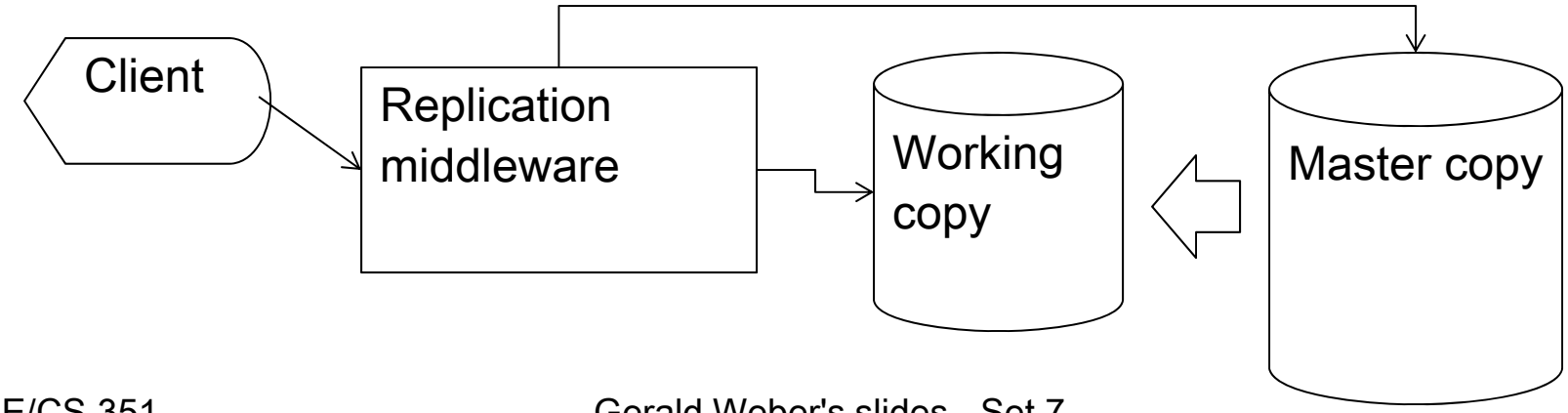
- A *master copy* of a database is replicated in several *working copies*. A.k.a. database caching.
- Clients work on working copies
- Motivation: faster read access
- Work (writes) shall be consolidated in the master copy.
- All databases together should behave as a single database.
- ACID properties for all clients.

In-place Update vs. Multiversion

- In-place update strategies motivate lock-based schedulers.
- In lock-based schedulers, transactions are delayed in case of conflict.
- Multiversion concurrency will not use locks, so transactions will not be delayed.
- Instead transactions compete.
- If they come into conflict, one transaction will be aborted.
- Several strategies:
  - First committer wins. Will fit well to our simple model of multiversion concurrency as replication protocol.
  - Other strategies are possible: first writer wins etc.

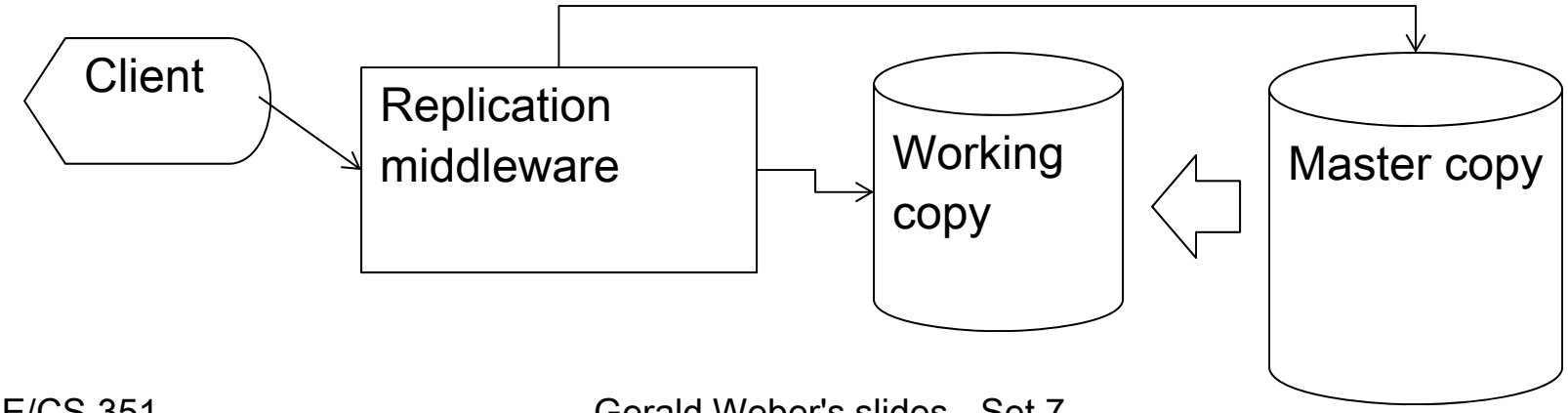
Replication

- Master copy is authoritative; transactions must be committed at master copy.
- Idea: coordination necessary only on commit.
- Clients do a *local transaction* on their working copy.
- They do that through a *replication middleware.*
- Replication middleware does the coordination with the master copy.



Replication Middleware

- The replication middleware logs all the commands of the client. In the basic transaction model, it will create *read sets* and *write sets*.
- During the open transaction, the replication middleware is just eavesdropping.
- at commit, the replication middleware interferes and communicates.

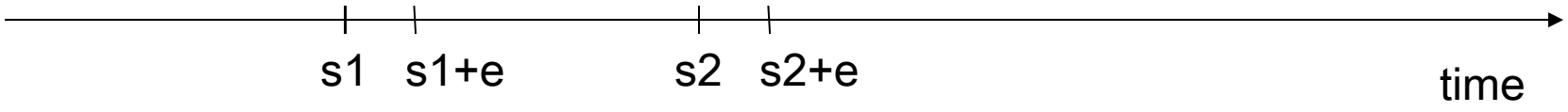


State of the working copy as a timestamp

- For the moment we can assume the following:
- Each working copy is only used for a single transaction, no concurrency at the working copy.
- At the start of a local transaction, a fresh copy of the master copy is made. (Later we can make this more efficient).
- This is a copy of a recent **clean** state of the master copy.
- This clean state is dated with the *commit timestamp* of the latest committed transaction.
- This commit timestamp is therefore also saying how up-to-date the working copy is.

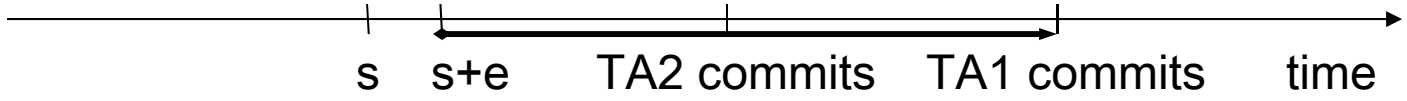
Local transactions work on a snapshot

- If the working copy is at timestamp *s* and local transaction TA1 is starting, then the working copy will not be updated with more data from the master copy, until TA1 commits.
- Hence the local transaction works on a single **snapshot** of the database: the state at timestamp *s*.
- We should consider the snapshot as a tiny moment *e* later than the commit timestamp *s*, but before any other commit.
- Therefore we call *s+e* the snapshot of TA1.
- This way it is clear that the snapshot *s+e* sees the clean state after *s*.



Commit as re-stamping

- The snapshot s+e is the state that transaction TA1 has seen.



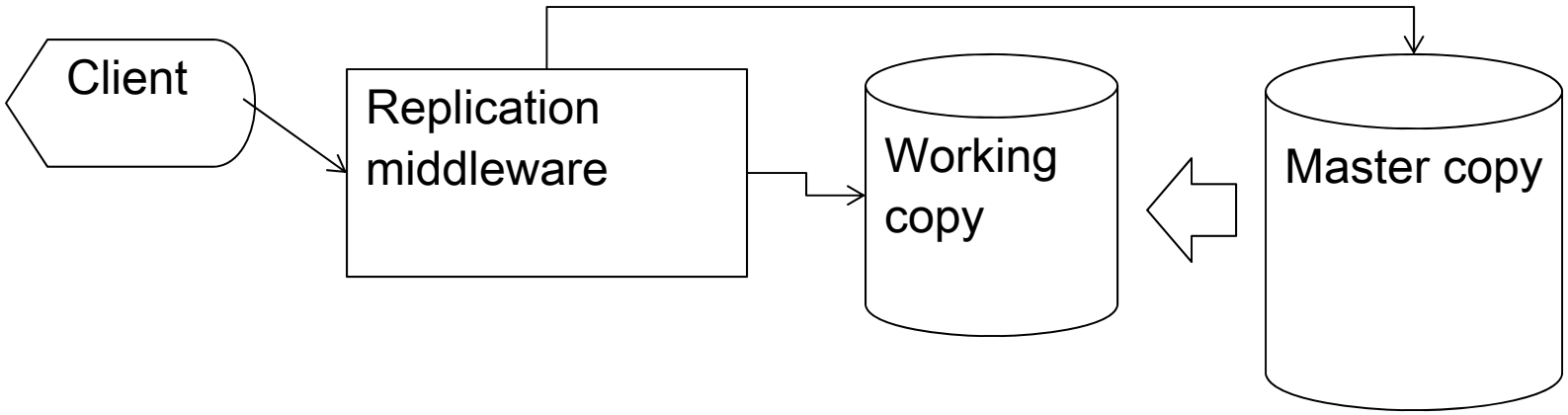
- Let's assume, TA1 requests commit at time *s+e+d*. Other transactions might have committed meanwhile on the master copy (*interlopers*). They are durable. What now?
- ACID Durability (for interlopers) requires:
- TA1 can only commit, if the master copy is still in the same state as it was at time s+e for everything concerning TA1.
- We can say the transaction TA1 must be **re-stamped** with timestamp *s+e+d*.
- The new timestamp is now the commit timestamp.

Conditions for re-stamping

- TA1 can only commit, if the database is still in the same state as it was at time s+e for everything concerning TA1.
- Simple cases: dejavu. If TA1 and each interloper TAx are:
  - data disjoint: no problem possible.
  - write disjoint: no problem possible.
- For other cases: two differently strict criteria will be needed.
- They all will guarantee ACID durability: no lost update.
- They will differ with respect to ACID isolation.
- The more generous criterion will lead to a new, interesting, relaxed isolation level: *snapshot isolation*.

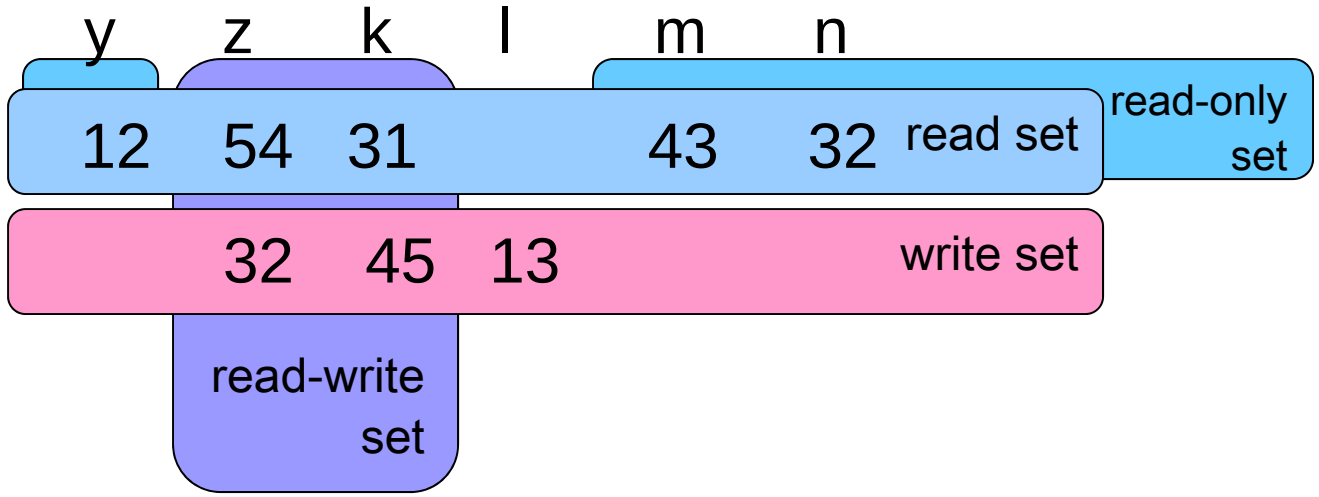
Commit through replication middleware

- at commit, the replication middleware interferes and communicates with the master copy:
- It will use the recorded information about the local transaction and work on the master copy.
- It will check if restamping is possible.
- If yes, then it will enact the changes of the local transaction at the master copy.



Read sets and write sets

- The commands executed by the local transaction are logged by the replication middleware in *read sets* and *write sets*. The sets are object sets, but values are attached.
- Since the transaction is alone on the snapshot, we have to record for each object at most one original read value (before-image) per object and at most one final write result (after-image).
- Object:
- read:
- final write:



Two strategies to check for changes

- At commit, the replication middleware will check on the master copy if the data affected by the transaction has been changed since the snapshot was taken.
- Based on the read sets and write sets we can see if the transaction and each interloper (pairwise) are write disjoint.
- Two ways to check:
  - Snapshot isolation: less strict, lower isolation
  - Optimistic locking: stricter, delivers full serializability

- Consequences of positive/negative outcome come later.

Check for write conflicts

- Snapshot isolation:
  - Check only read-write set
  - For each object: Is the read value still the current value in the master copy?
- Optimistic locking:
  - Check the whole read set (superset of read-write set)
  - For each object: Is the read value still the current value in the master copy?
- If yes: test passed, transaction is allowed to write,
- If not: test failed, transaction is aborted.

Replication middleware executes transaction

- If the test has passed:
- replication middleware executes the write set of the local transaction on the master copy.
- The master copy must be transactional: might be a conventional database, or might offer simpler mechanisms.
- The check and the writes happen in a single transaction on the master copy.
- This transaction on the master copy will be fast
  - since everything is prepared.
  - Transactions have all the same simple structure.

Expressing multiversion as linear schedules

- Concurrent multiversion transactions can be translated into the linear schedules we had so far by inserting all reads at their snapshot time and all writes at their commit timestamp.
- Example:

s:	$r_4[x], r_4[y],$	$w_3[x], c_3,$	$r_5[x], r_5[y],$	$w_4[x], c_4,$		$w_5[y], c_5$
Time:	s2+e	s3	s3+e	s4	s4+e	s5

- The schedule can violate lock-based scheduling rules, but *may* still be unproblematic.
- Problems should show as a (bad) phenomenon.

Write Skew

- Snapshot isolation allows a phenomenon: **write skew**
  - Two transactions “getting wires crossed”.
  - A “double almost-lost-update on different objects.”
- Can be expressed as a schedule:
- s: $r_1[x]$, $r_1[y]$, $r_2[x]$, $r_2[y]$, $w_1[x]$, c_1, $w_2[y]$, c_2
- We can see that the snapshot isolation test will succeed since the read/write sets are disjoint.
- Nevertheless violates serializability.
- Can be a problem
- Is rare and considered “mostly harmless”.

Write Skew business logic example

- An example where a write skew could appear:
- Bank grants overdraft based on general liquidity.
- Customer has two accounts, x and y.
- Bank allows any one account to go into negative if overall balance stays positive. e.g consider debiting account x:
- s: $r_1[x]$, $r_1[y]$, (compute overall bal.) $w_1[x]$, c_1
- The write brings account x into negative (withdrawal).
- Now the customer does two such transaction in parallel:
- s: $r_1[x]$, $r_1[y]$, $r_2[x]$, $r_2[y]$, $w_1[x]$, c_1, $w_2[y]$, c_2
- write skew! Both accounts are now negative.

Write skew in numbers

- Account x = $1000
- Account y = $800.
- Planned TA1: withdraw $1100 from x; ok since overall balance remains $700.
- Planned TA2: withdraw $900 from y; ok since overall balance remains $900.
- With snapshot isolation, if both transactions are executed concurrently, they might go through!
- New balance: ? Proposals please...
- Balance will be unwanted (bad),
  - but is less dramatic than lost update

Replication: keeping local copies up-to-date

- It is good to know that not a complete copy of the whole database has to be done at the start of a new transaction.
- Local copies can be kept up-to-date with incremental changes. The write sets can be used.
- There is an instance of the replication middleware for each client; these instances can communicate the write sets.