Distributed Systems in the Cloud

NewSQL

Sejal Lohiya

- Why NEWSQL(Motivation)
- What is NEWSQL ?
- Implementation
- Summary

Everyone is talking about Big Data

- Huge data is being amassed

  -Web-based data

  -OLTP

  -Real time analytics

- Big Question: How do we store this data and generate value from it?
- Enterprises want to store and query this data

Old SQL

- Relational DBMS Model
- Parallel, shared nothing architectures
- Underlying Idea: Partiton data and parallelize computation

OldSQL Problems

- Closed architecture
- Pay to scale
- Still cant scale much

  - reasons like 2Phase Commit, structure of the data, etc
- Buffer pool
- Row-level locking- reads / writes / deadlock detection
- Recovery - writing logs

NOSQL

- GFS- Big byte stream files and replication
- MapReduce
- Give up SQL
- Give up ACID
- Concentrate on Scalability and Performance
- Eventual consistency: In absence of updates, all replicas will converge towards identical copies

Give Up SQL? Problems

- SQL is not overhead
- High level languages are good
- Hard to beat the compiler
- Features : Data independence, less code, etc

Give Up ACID? Problems

- ACID is not overhead
- Implementing ACID in user code is difficult
- Can you guarantee you won’t need ACID tomorrow?

Who needs ACID

- Everybody with Integrity constraints
- Huge number of transactions
- Order sensitive transactions
- When eventual consistency gives incorrect results

NewSQL

- Preserve SQL
- Preserve ACID
- Improve performance and scalability with innovative architecture
- Eliminate Locking: MVCC, etc
- Support Built in-replication: PAXOS, etc
- Reduce logging overhead - failover

What is NEWSQL

- SQL/high programming language as the primary mechanism for application interaction
- ACID support for transactions
- Non-locking concurrency control mechanism so real-time reads will not conflict with writes, and thereby cause them to stall.
- An architecture providing much higher per-node performance
- A scale-out, shared-nothing architecture, capable of running on a large number of nodes without bottlenecking

Implementations

- Spanner: Google's Globally-Distributed Database, Google Inc.

- Storage Management in AsterixDB, UCI et al.

Spanner

- Globally distributed database
- Synchronous Replication
- Externally consistent
- Non- blocking reads
- Lock-free read-only transactions
- Application-controlled replication configurations

Logical Data Layout

Albums

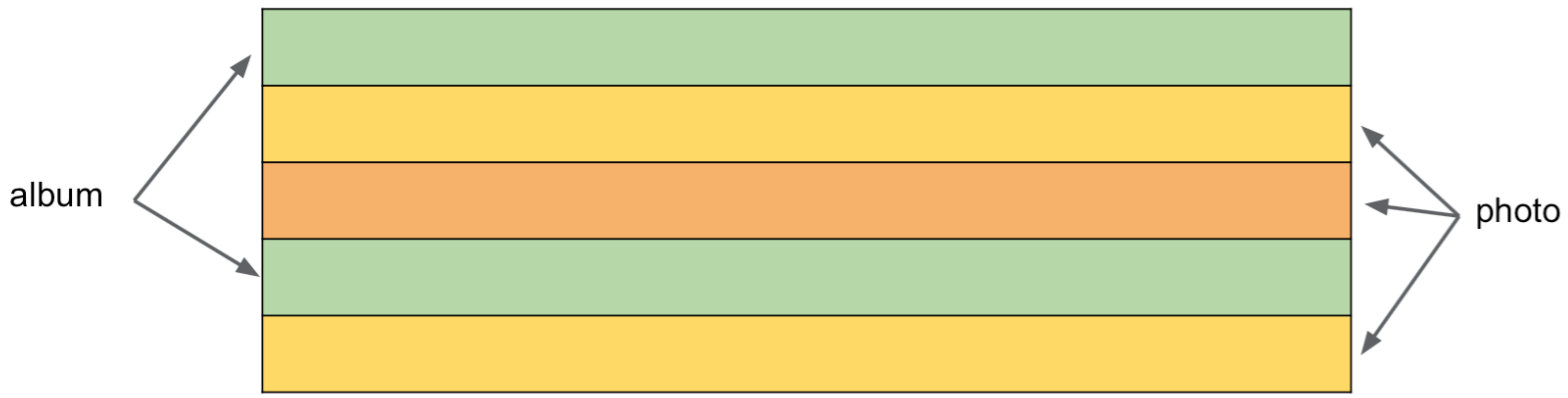
user_id	album_id	name
1	1	Maui
1	2	St. Louis

Photos

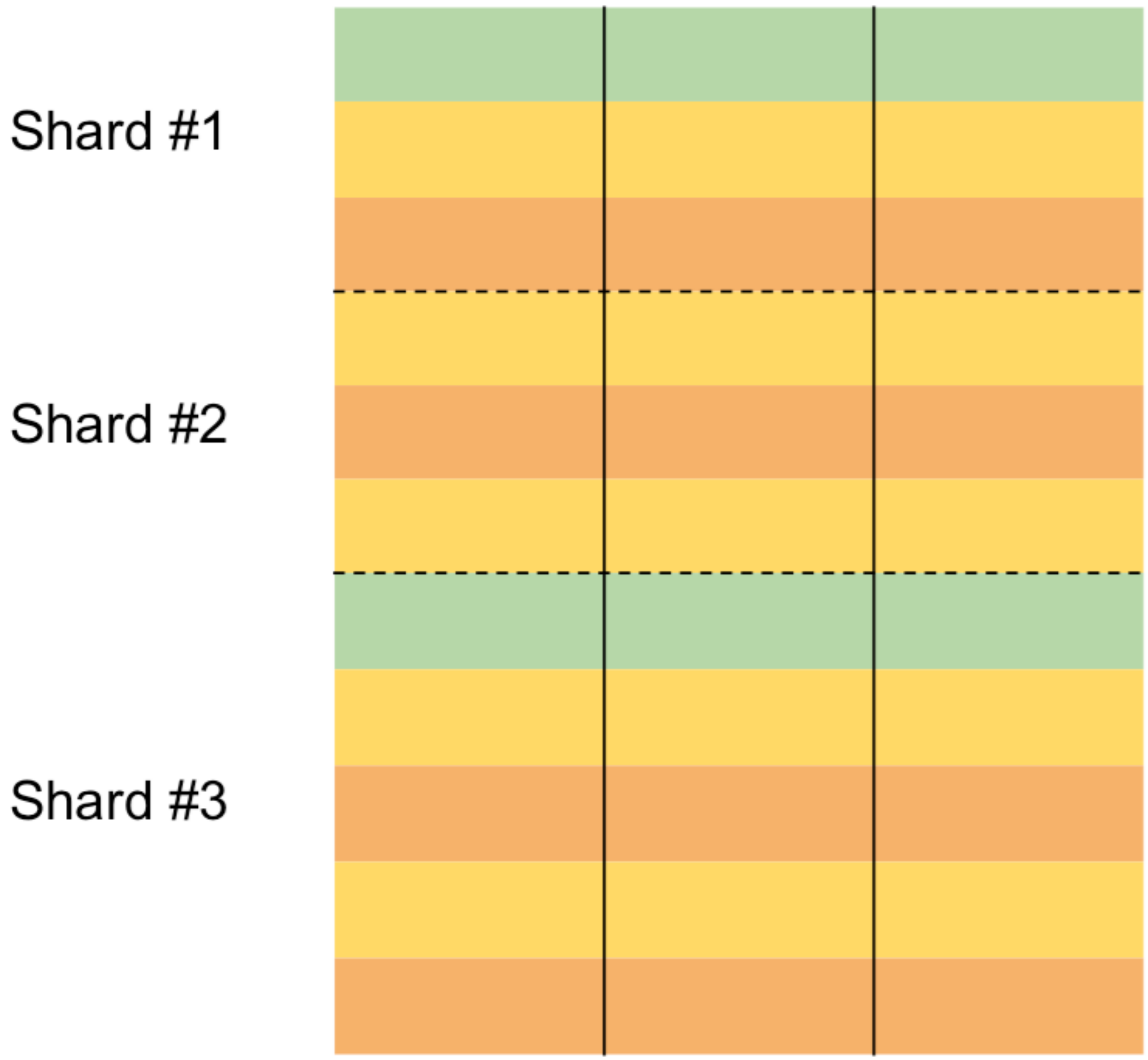
user_id	album_id	photo_id	title
1	1	2	Beach
1	1	5	Snorkeling
1	2	3	Gateway Arch

Physical Data Layout

- Interleaved tables



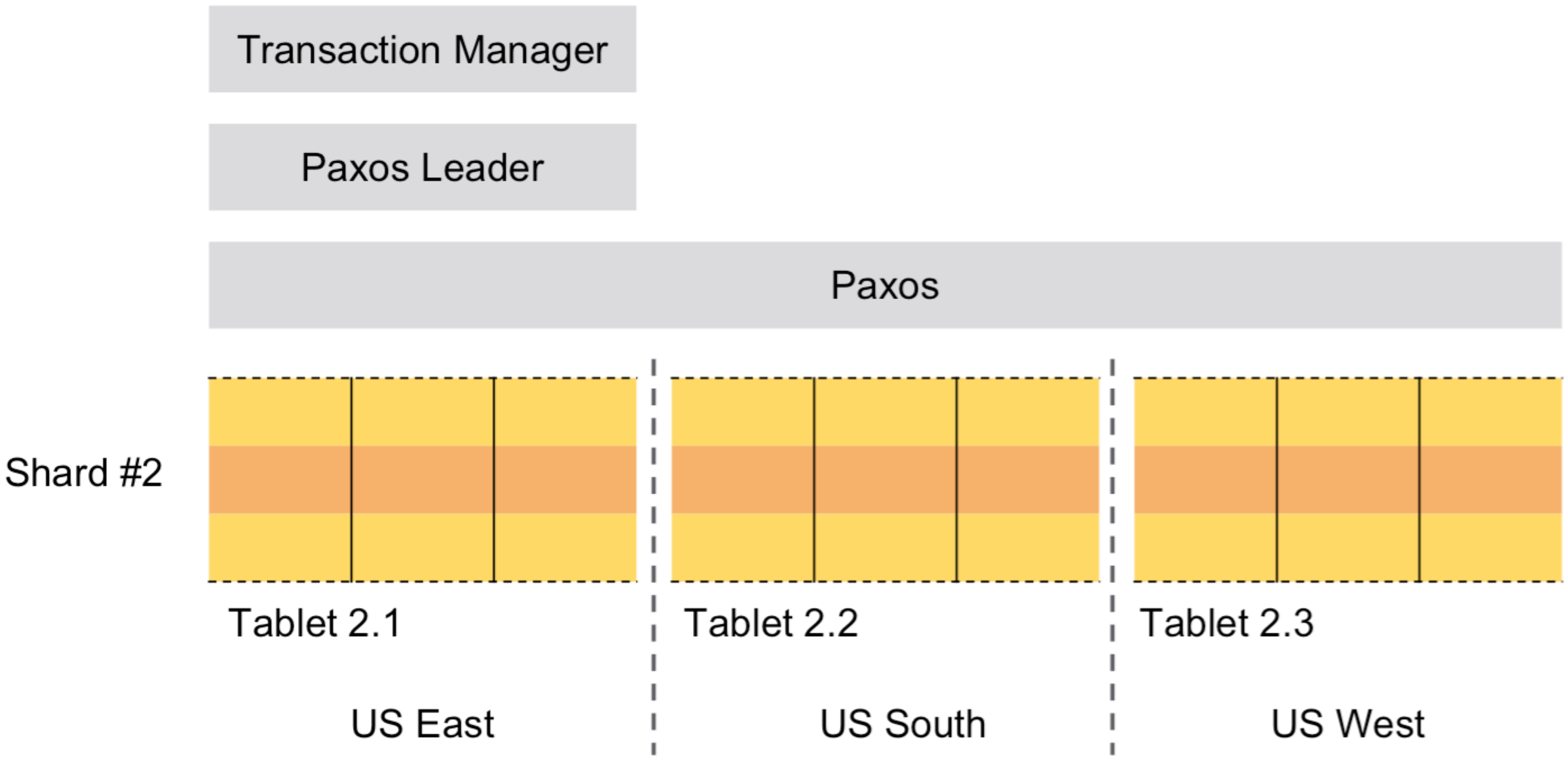
Sharding



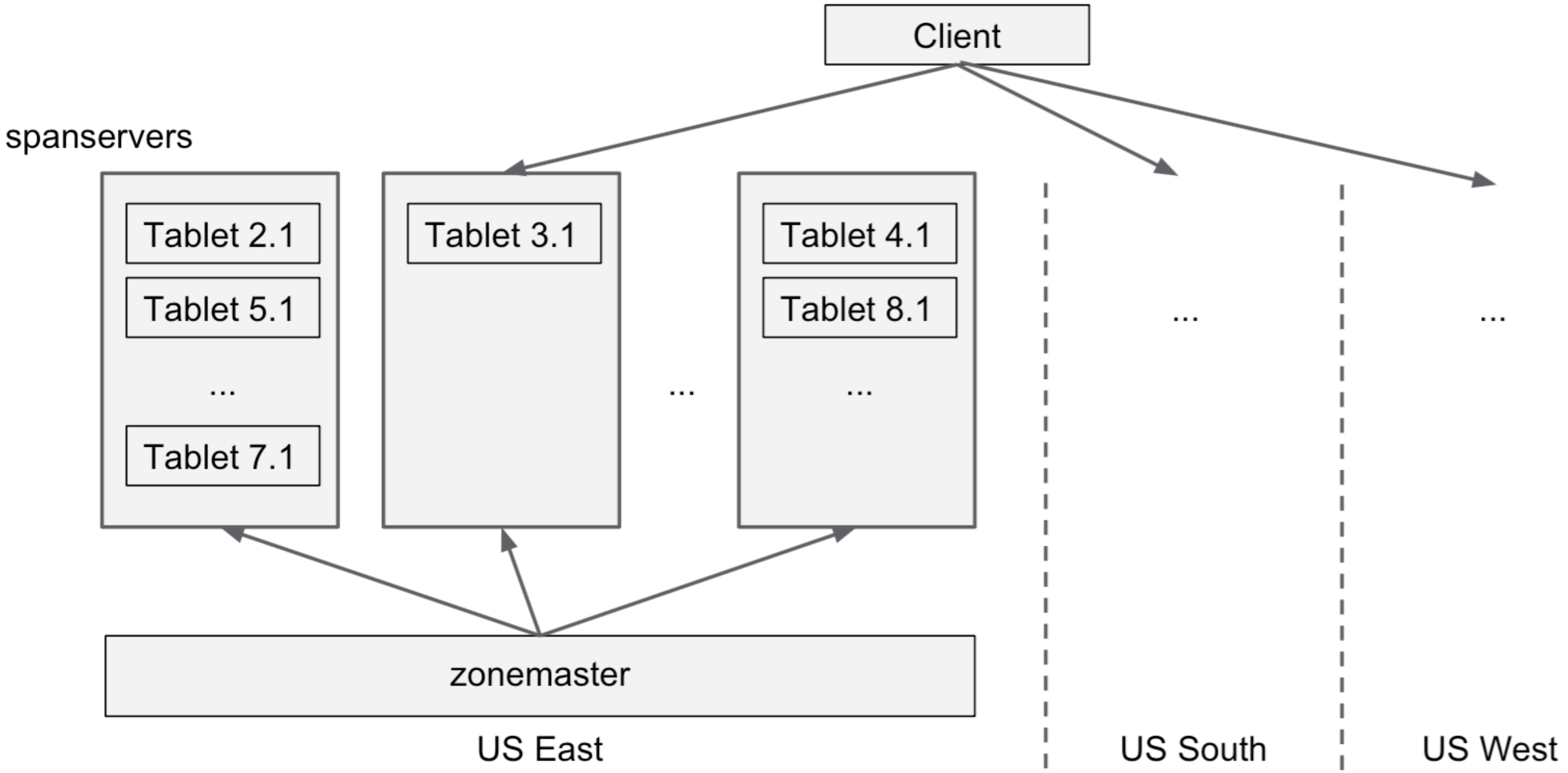
But still support:

- Transactions across shards
- Consistent snapshot reads (range scans) across shards

Replication



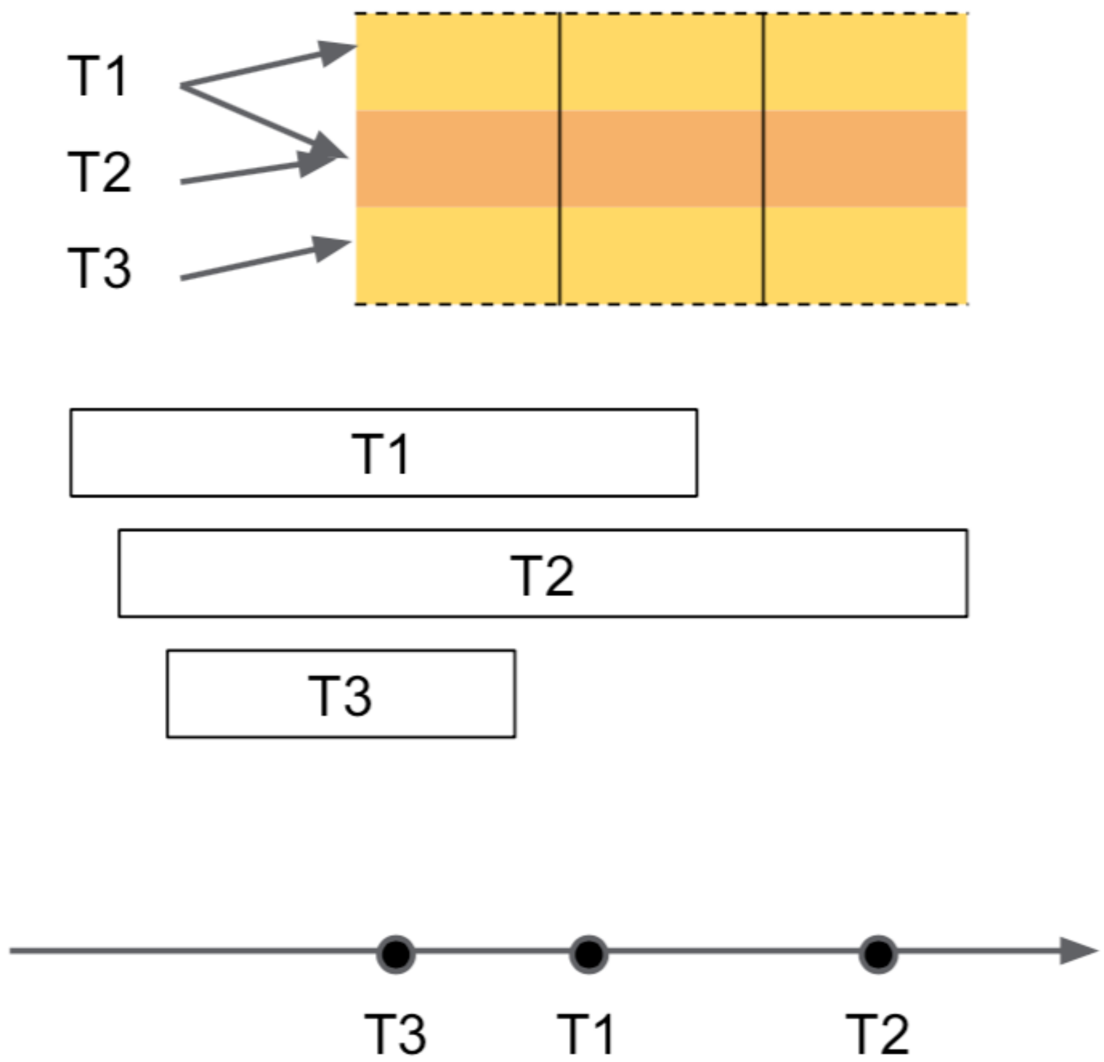
Serving Structure



Strict Two Phase Locking

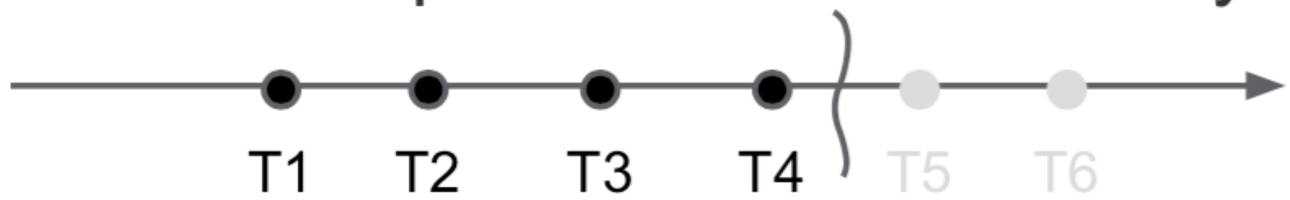
Life of a transaction:

1. Acquire locks
2. Execute reads
3. Pick commit timestamp
4. Replicate writes (through paxos)
5. Ack commit
6. Apply writes
7. Release locks



Snapshot Reads

Choose a prefix of commit history



Properties of snapshots:

- immutable
- consistent

Can be used for:

- long-running batch operations (e.g. map reduce)
- stale reads (e.g. 10s old)
- strong (current) reads: lock-free, don't block writers

Picking Commit Timestamps

Attempt #1: Assign from local (monotonic) clock

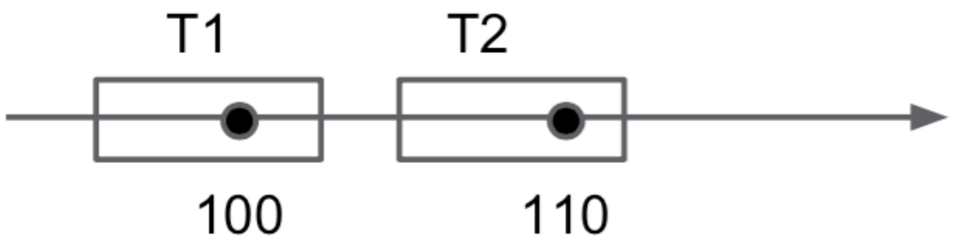
1. Acquire locks
2. Execute reads
3. **Pick commit timestamp = now()**
4. Replicate writes (through paxos)
5. Ack commit
6. Apply writes
7. Release locks

External Consistency

Definition:

If T1 commits before T2 starts, T1 should be serialized before T2. In other words, T2's commit timestamp should be greater than T1's commit timestamp.

Note: Applies even if T1 and T2 don't conflict.



True Time

Idea: There is a global “true” time t

TT.now() = [earliest, latest] ∍ t.

- TT.now().earliest definitely in the past
- TT.now().latest definitely in the future

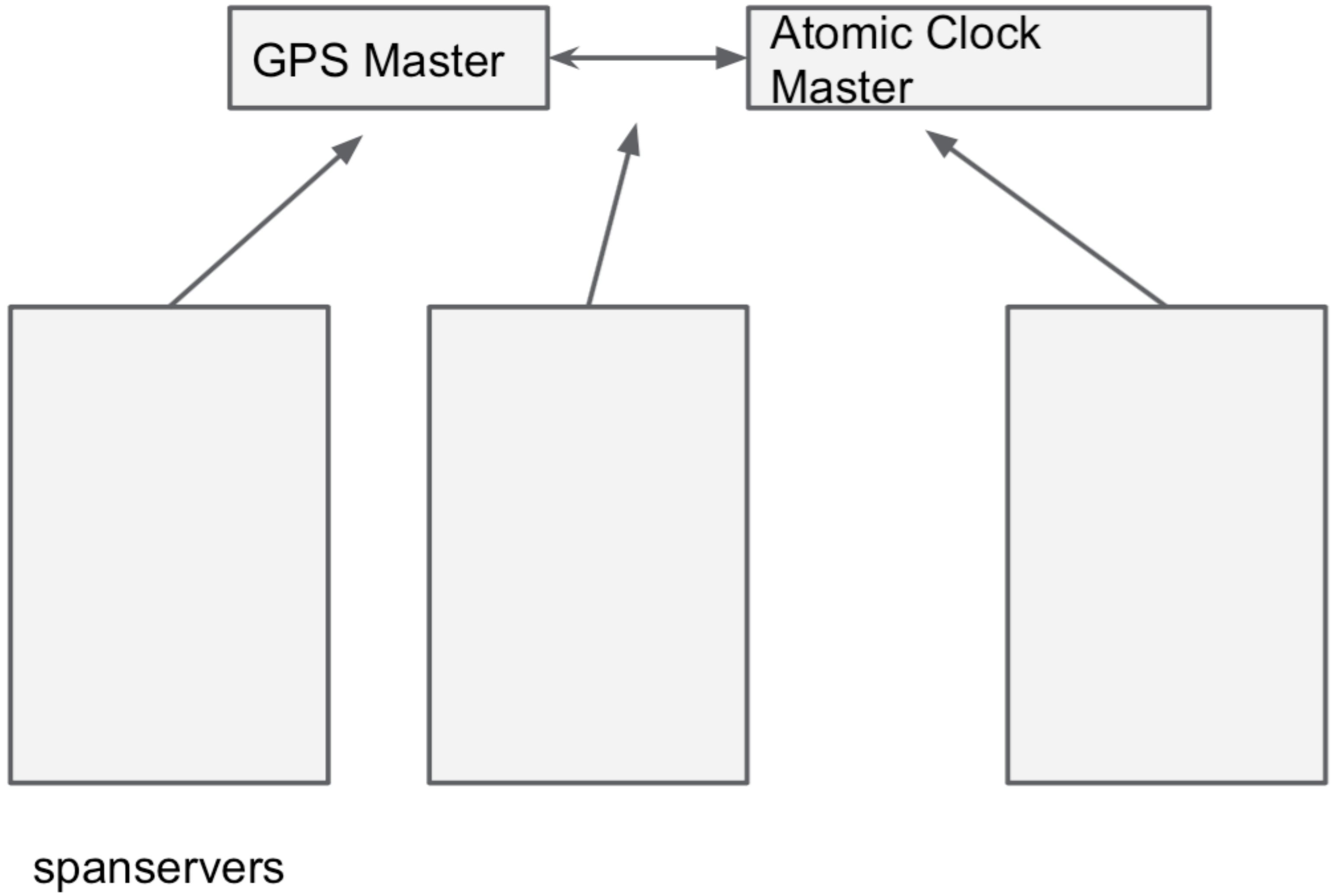
Timestamp Assignment: True Time

Transaction protocol becomes:

1. Acquire locks
2. Execute reads
3. **Pick commit timestamp T = TT.now().latest**
4. Replicate writes (through paxos)
5. **Wait until TT.now().earliest > T**
6. Ack transaction commit
7. Apply write
8. Release locks

Strong reads: T = TT.now().latest

True Time Architecture



periodic poll: [earliest, latest]

In-between polls, uncertainty radius grows based on worst-case clock drift (200 usec / sec)

Storage Management in AsterixDB

- Unstructured web application data
- Write intensive data
- Problem to ingest, store, analyze and index data
- Solution: AsterixDB

Storage Management

- Log structured Merge Design
- ACID

LSM-ification

- Data writes are buffered in memory
- Flushed to disk in batched append only manner
- Writes are sequential disk access(fast)
- Reads are multiple random access(slow)
- Merge disk components
- Efficient reconciliation

Summary

- One size does not fit all!
- Don’t compromise semantics
- Use the right tool for the job.

Thank You!