DISTRIBUTED DB: MERGING STRATEGY

6/28/2022

CS 848 Presentation

Senyu Fu

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Outline

- Background
- Our System Design
- Challenges
- Merging Strategy
- Future Work

BACKGROUND

Motivation for Distributed DB

Background: Serverless Computing

- Serverless computing is becoming more and more popular
  - Function as a Service (FaaS)
- Advantages:
  - No explicit resource management
  - Scalability
- Challenges:
  - Functions are stateless
  - No direct communication between functions

Challenges

- Most applications are usually stateful
  - Store data in external storage system (S3, DynamoDB, and etc.)
- Application is divided into multiple functions
  - Share intermediate states through external storage system
- Problem: few fast databases for serverless computing

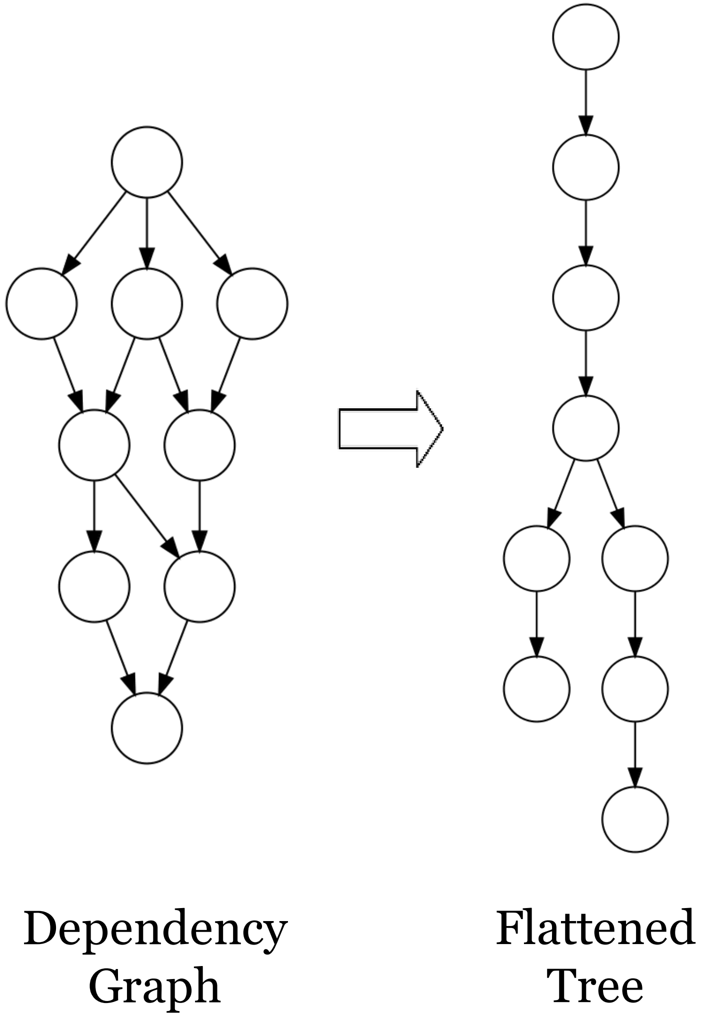
Database Requirements for Serverless Computing

- Scalable: serverless computing itself is scalable
- High concurrency: a lot of concurrent functions
- Low latency: share intermediate states

OUR SYSTEM DESIGN

System Model

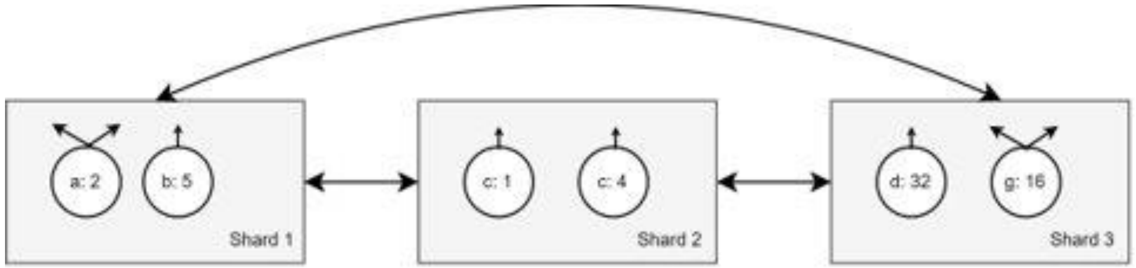
- Key-value store
- Updates stored in the form of dependency graph
  - Asynchronous write (low write latency)
- Flatten: process nodes in batch to form a serializable order
  - Non-serializable writes: branch on conflicts
  - Concurrency control (without locking)
  - High throughput
- At the end of each epoch
  - Keep the longest branch (prune and abort other branches)



Data Partitioning

- Hash-based sharding
  - Data only stored at designated shard

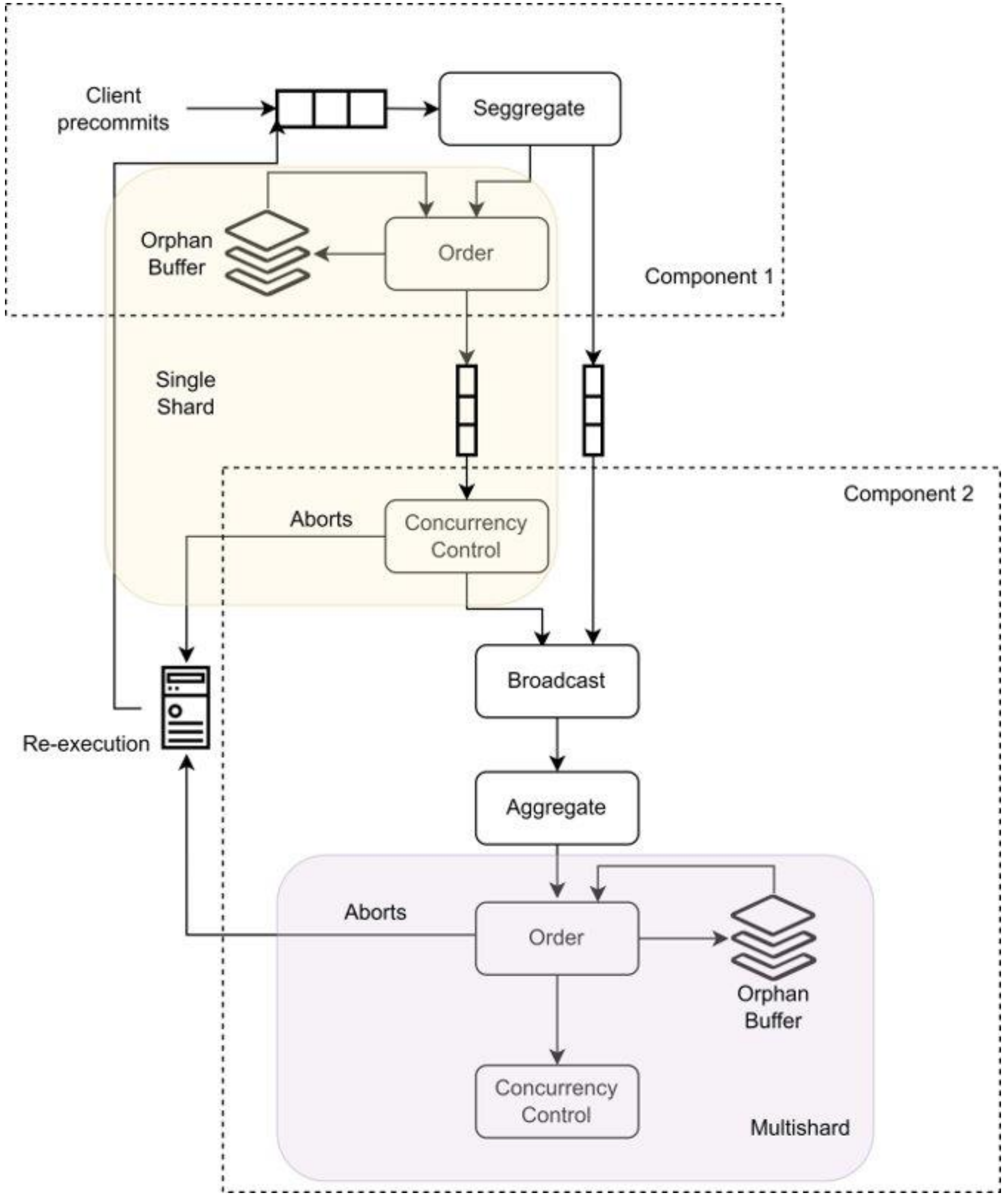
- Clients only send writes to designated shard
  - Each shard broadcasts dependency information in batch
  - Each shard combines the dependencies to construct the same dep graph



UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Architecture

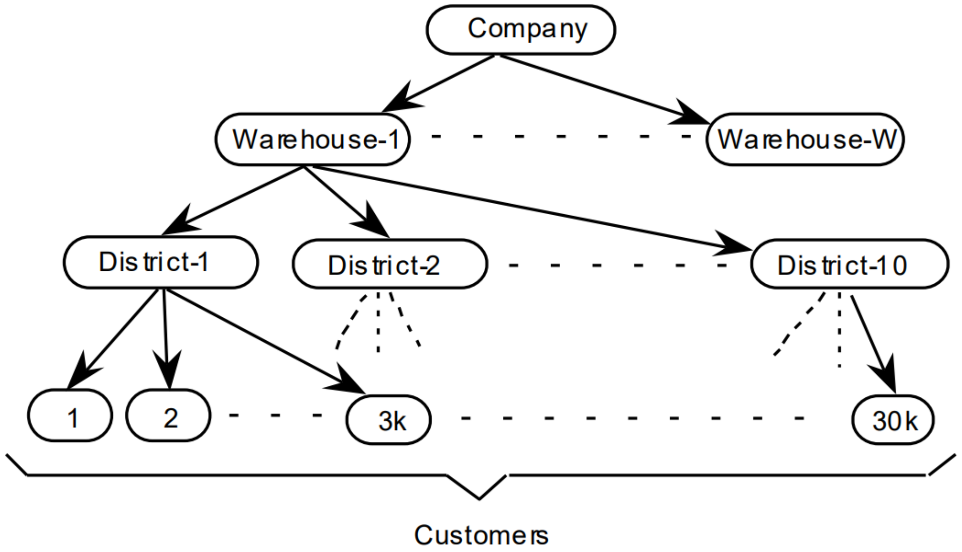
- Pipeline architecture: process multiple batches concurrently in different stages
- Single-shard transactions are processed independently at each shard
- Broadcast graph metadata: every shard can agree on the same order
- Multi-shard transactions are processed at every shard after receiving metadata
  - Dependencies are now available
  - Replicated work



CHALLENGES

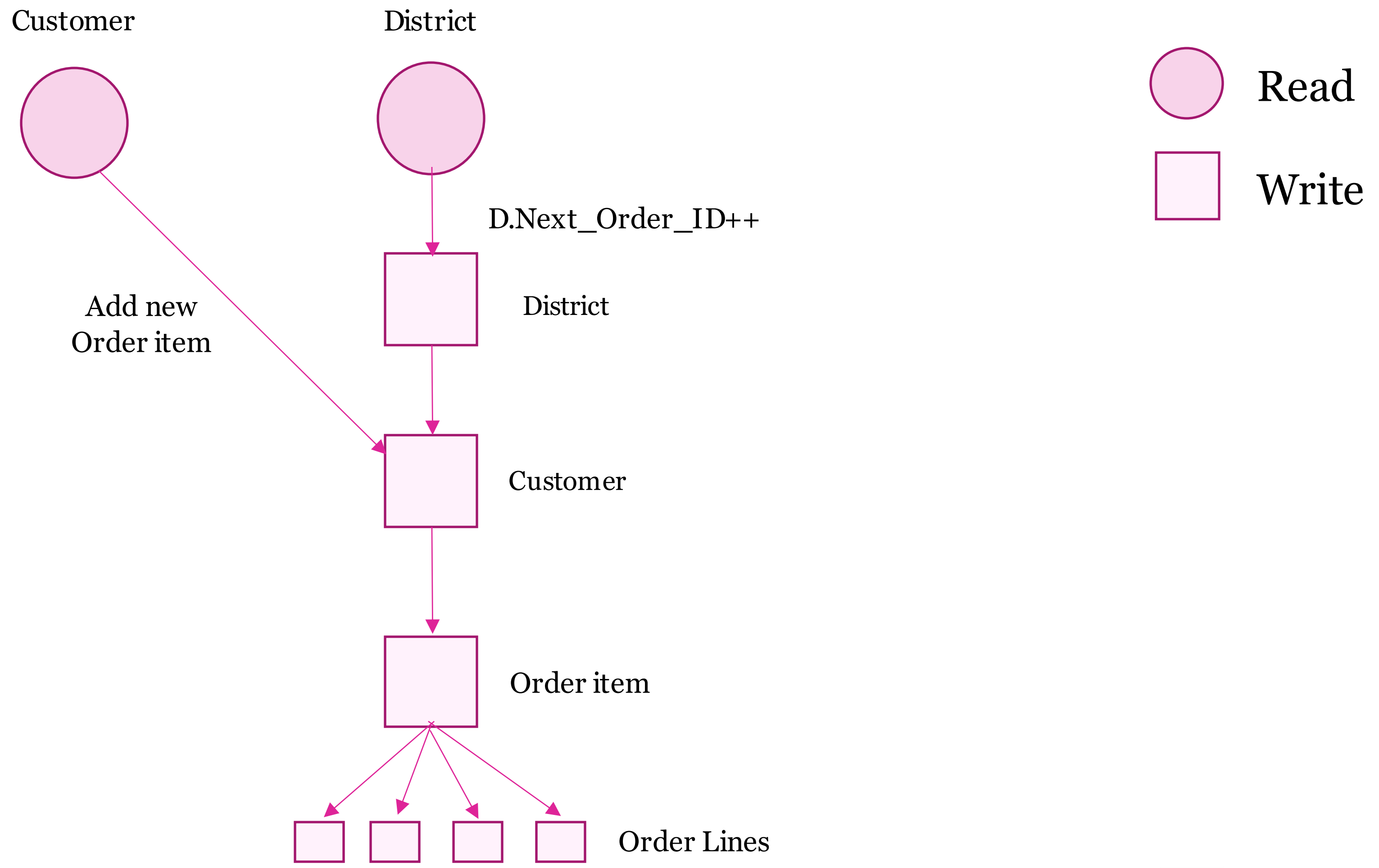
TPC-C Workload

- There is a 'Company' with W warehouses (10).
- Each warehouse has D (10) associated districts and 100K items.
- Totally, there are 30K customers which are spread out across the districts.
- W and D can be configured.

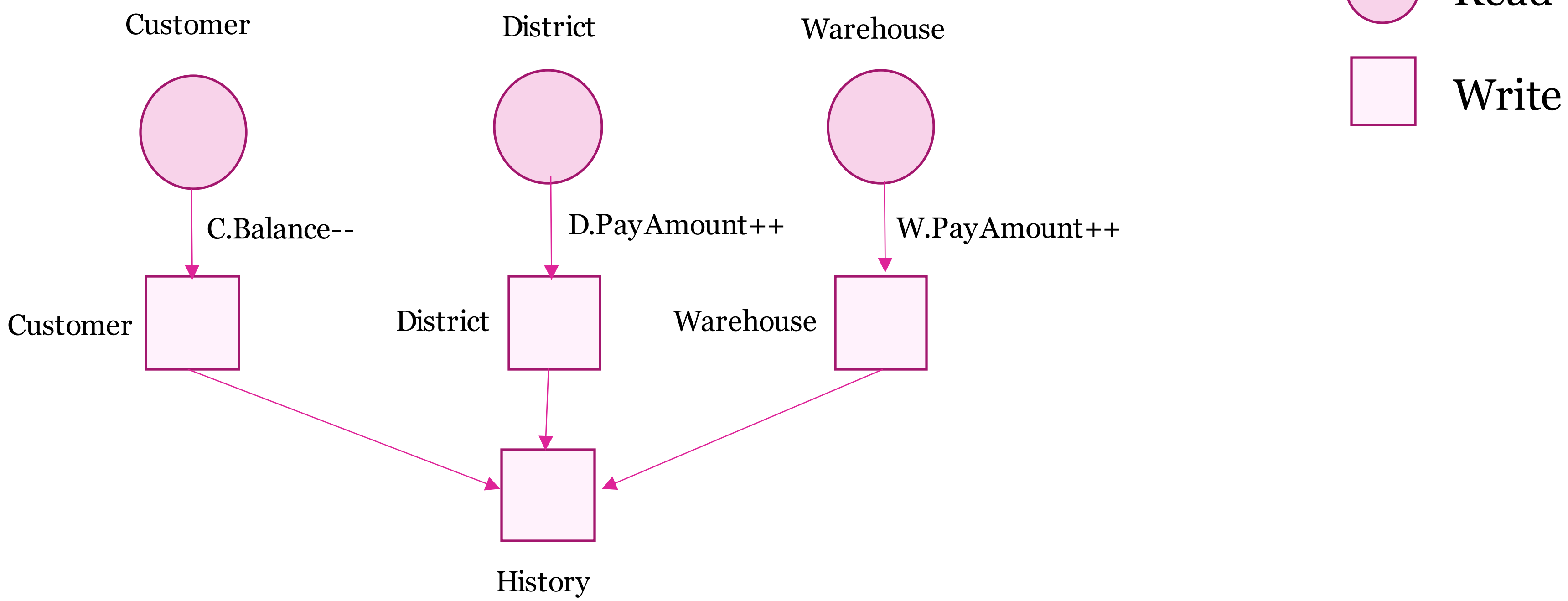


(From official TPC-C Specification)

TPC-C Workload: New Order Transaction



TPC-C Workload: Payment Transaction



TPC-C Workload

- Global states on warehouse and district
  - Next ID (District)
  - Payment Amount (District, Warehouse)
- Only 10 warehouses and 100 districts
  - Very high contention on counter values
- Without locking, a lot of conflicts
  - Very high abort rate
  - Low throughput

MERGING STRATEGY

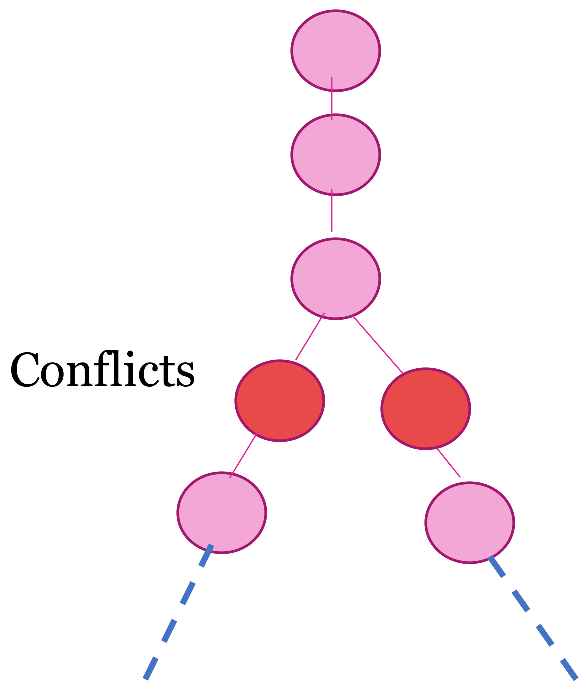
Observation

- Observation: contention mainly comes from counters
  - Next ID (District)
  - Payment Amount (District, Warehouse)

- Counters are easy to merge
  - Resolve conflicts without aborting them
  - Greatly reduce abort rate

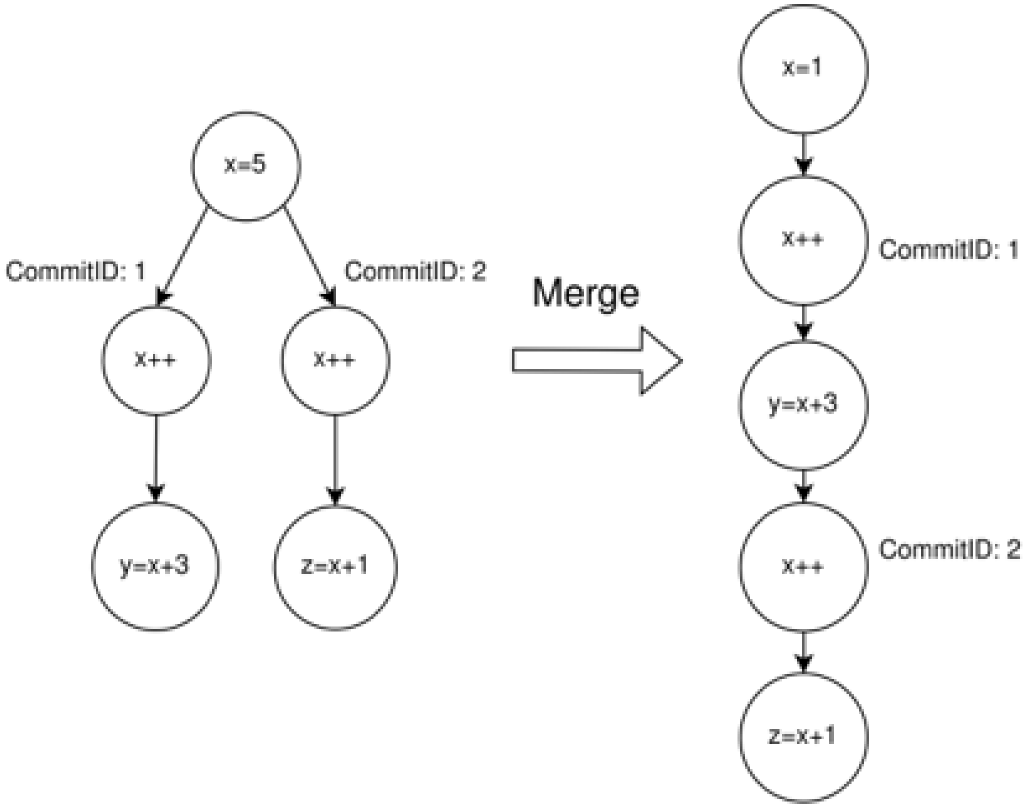
Merging

- To merge two conflicting nodes:
  - Re-compute this node
  - Re-compute nodes that depend on this node
- This is similar to aborting and re-execution
  - Less overhead
  - Avoid starvation (some txns may be aborted every time)
- Problem: each node in our graph stores computed value
  - Server doesn't know how to compute it



Merging: Solution

- Each node represents an update instead of concrete values
- To get the current value of a key: replay the updates
  - Use cache to accelerate it
- Benefit: when merging, the server can recompute the value.
  - Fast path



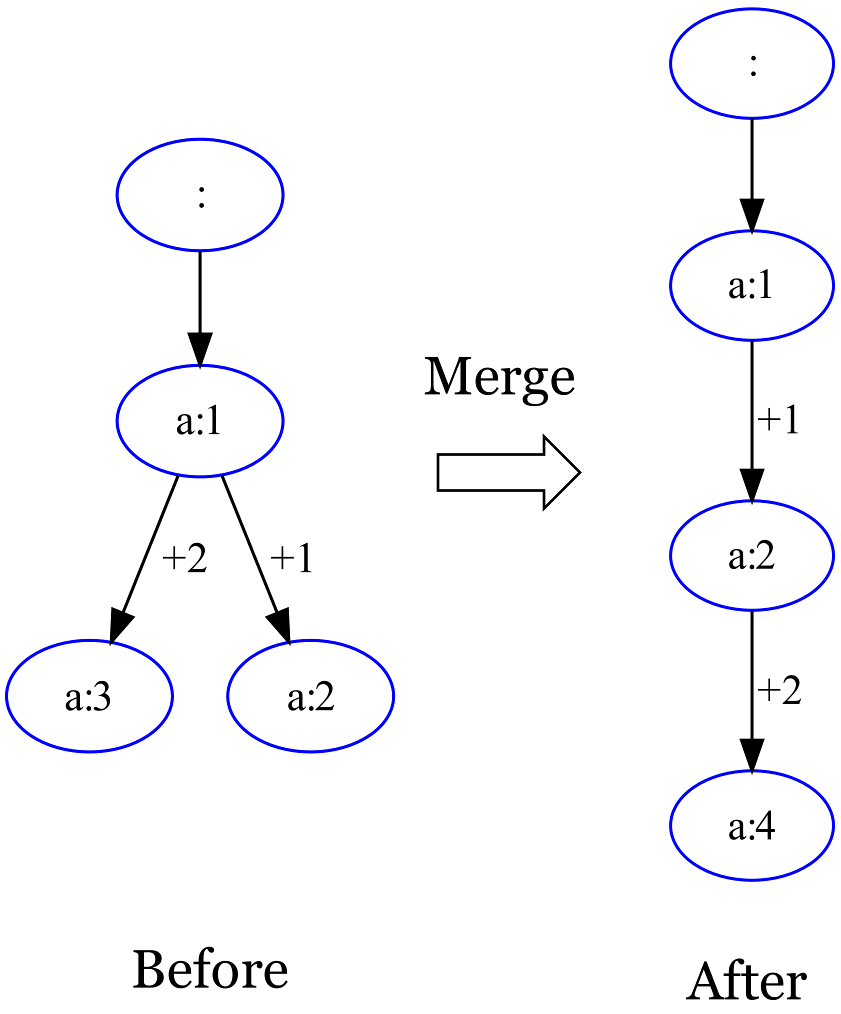
Data Types

- Basic mergeable types
  - Counter: increment and decrement
  - List: append and remove
  - …

- Complex data types:
  - Composition: decompose into different keys (O.counter, O.array)
  - Use transaction to provide atomicity

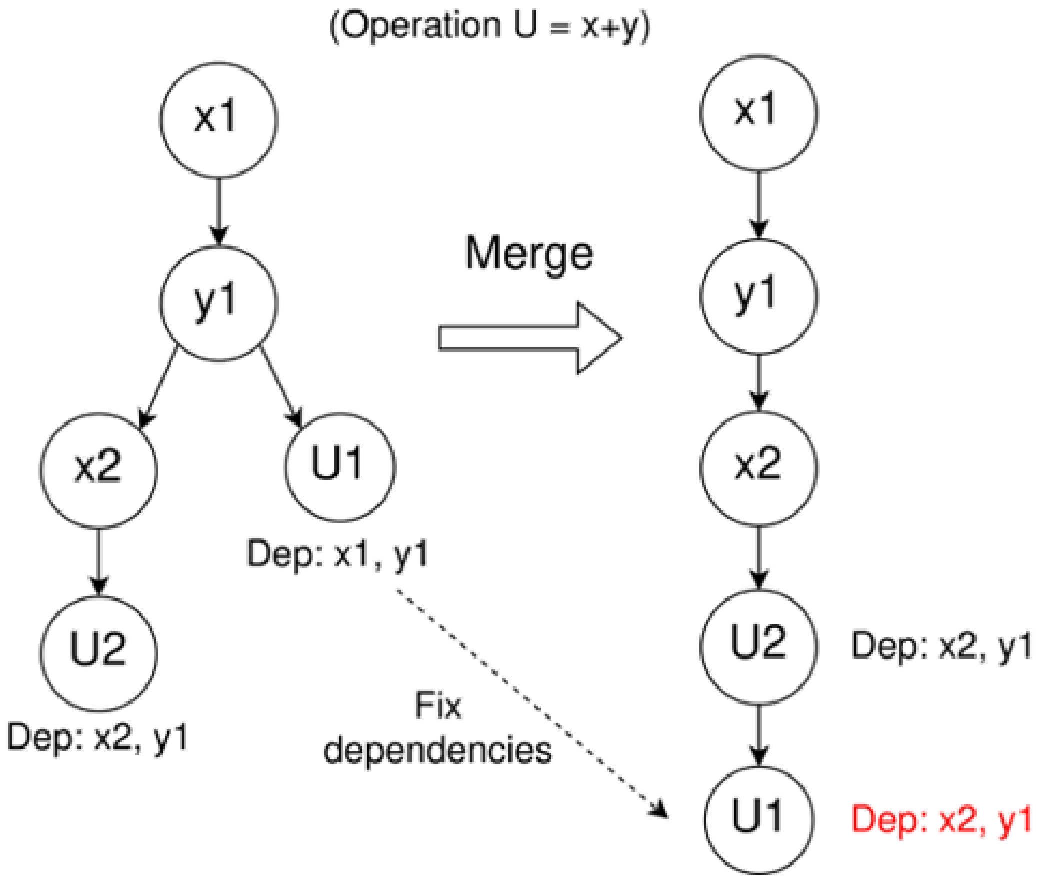
Counter

- Counter
  - Increment operation
  - Value is computed on demand
    - If cache not available, replay operations
    - Cache the value
- Dirty read
  - Client submits the value when committing
    - Used as cache for dirty read
  - Server invalidates cache after flattening



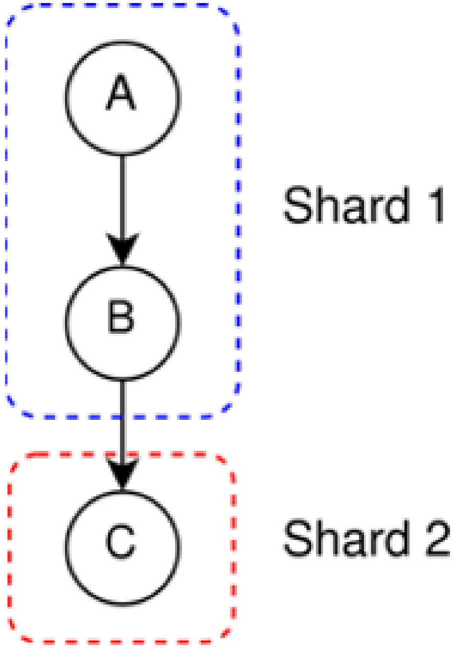
Merging: Dependencies

- Merge: re-order nodes
  - Merge into longer branch
- Dependencies may change
- Fix dependencies
  - Find nearest ancestor nodes with the same keys
  - Dependencies of U1
    - Before: x1, y1
    - After: x2, y1



Multi-shard Transactions

- After merging, recompute values for merged nodes
  - To compute values: replay operations
- Multi-shard transaction
  - The value may not be available locally
- Solution:
  - Use RPC to get the value (increased latency)
  - Batch requests



Merging: Verifier

- Merging may not be safe sometimes
  - e.g. Deduct the balance counter: verify the balance >= 0
- Verifier
  - Each transaction: multiple categories
  - Each category corresponds to a verifier
- When merging a transaction:
  - Run verifiers corresponded to its categories
  - If verification fails, abort

FUTURE WORK

Future Work

- Improve merging performance
  - Dependency fixing
- Read policy
  - Currently, read based on commit timestamp
  - After merging, read the nodes with greatest depth (latest)
- Support more mergeable data types
  - List
  - Set
  - ...

THANK YOU