Hyder: A Transactional Indexed Record Manager for Shared Flash Storage

Philip A. Bernstein
Joint work with Colin Reid, Sudipto Das,
Ming Wu, Xinhao Yuan

Microsoft Corporation

Presented at Amazon.com
December 5, 2011

What is Hyder?

It's a research project.

- We have two implementations

A software stack for transactional record management

- Stores [key, value] pairs, which are accessed within transactions
- It's a standard interface that underlies all database systems

Functionality

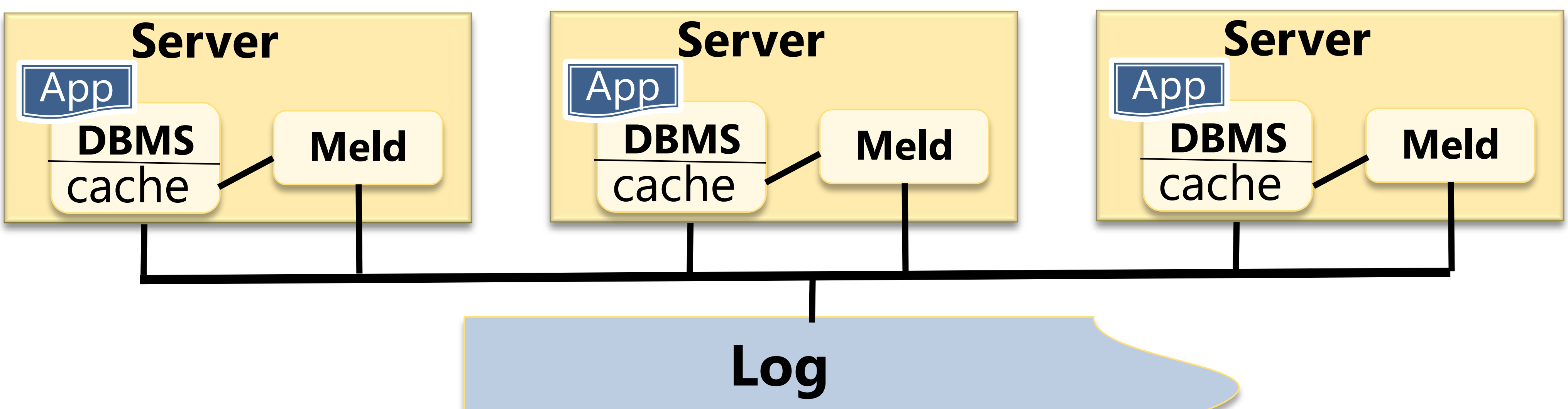
- Records: Stored [key, value] pairs
- Record operations: Insert, Delete, Update,
  Get record where field = X; Get next
- Transactions: Start, Commit, Abort

Why Build Another One?

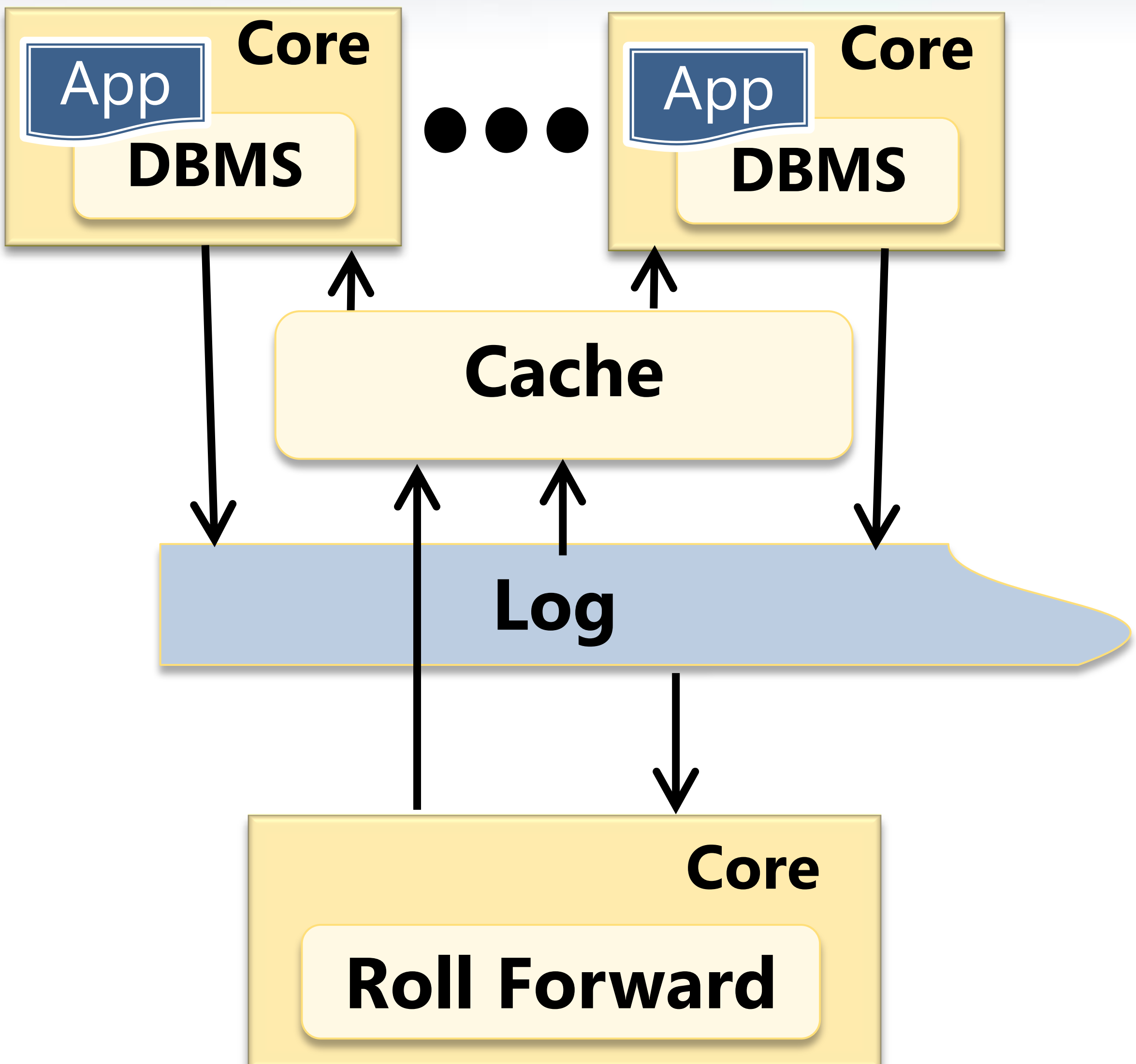
- Enables scaling-out large-scale web services without partitioning data or application
- Supports real-time data analytics
  - Uses multi-version data for high-speed transaction processing and queries on the same server
  - All isolation levels, including concurrency control over key-range operations.
- Exploits technology trends
  - flash memory, high-speed networks, multi-core

Scenario 1: A data-sharing System

- The log is the database. All servers can access it.
- Each transaction executes against its partial, cached, DB copy
- Then it appends its after-images to the log.
- Each server rolls forward the log on its partial, cached DB copy
- Roll forward (a.k.a. meld) does optimistic concurrency control
- N.B.: Log-append is the only server-to-server synchronization



Scenario 2

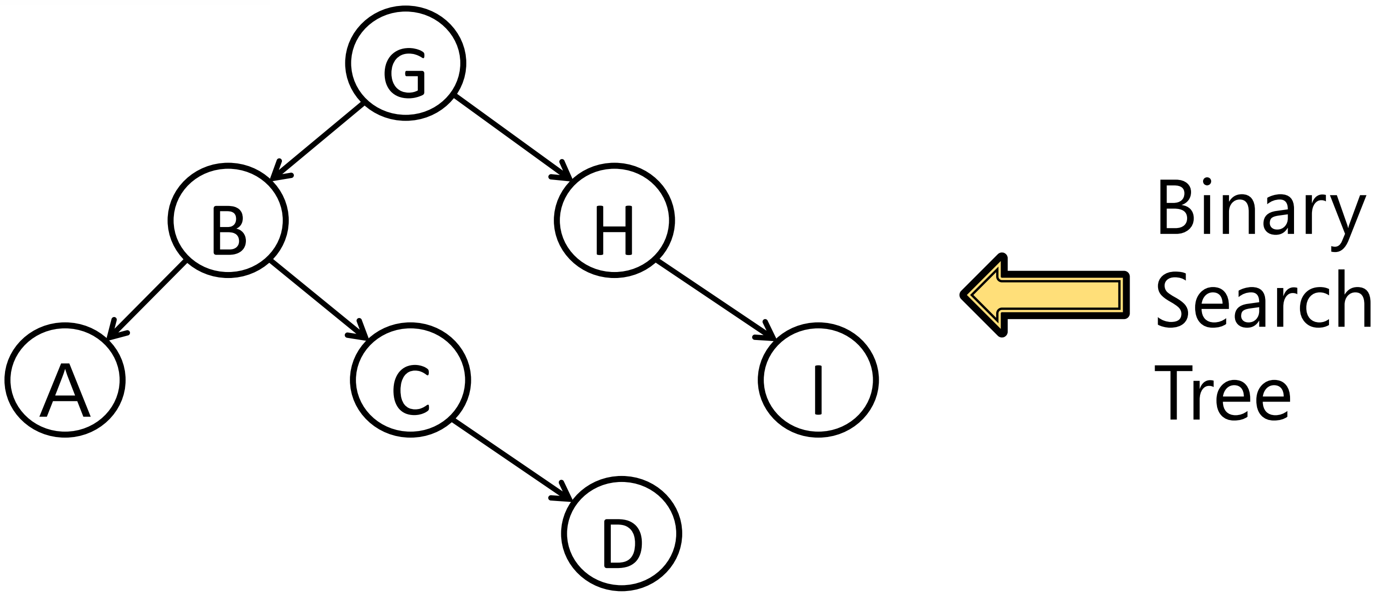


- The log is the database.
- All cores can access it.
- Each transaction appends its after-images to the log.
- One core runs meld to do OCC and roll forward the log

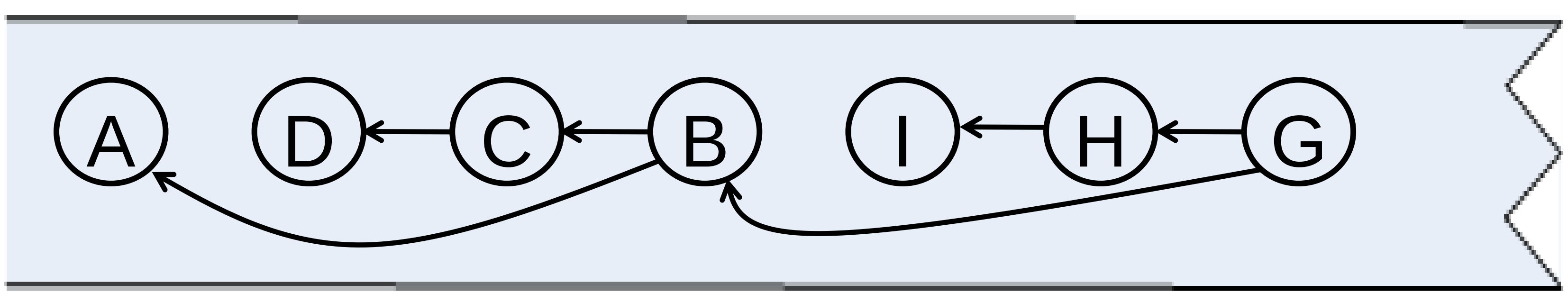
Outline

- ✓ Motivation
- System architecture
- Performance
- Related Work
- Conclusion

Database is a Binary Search Tree

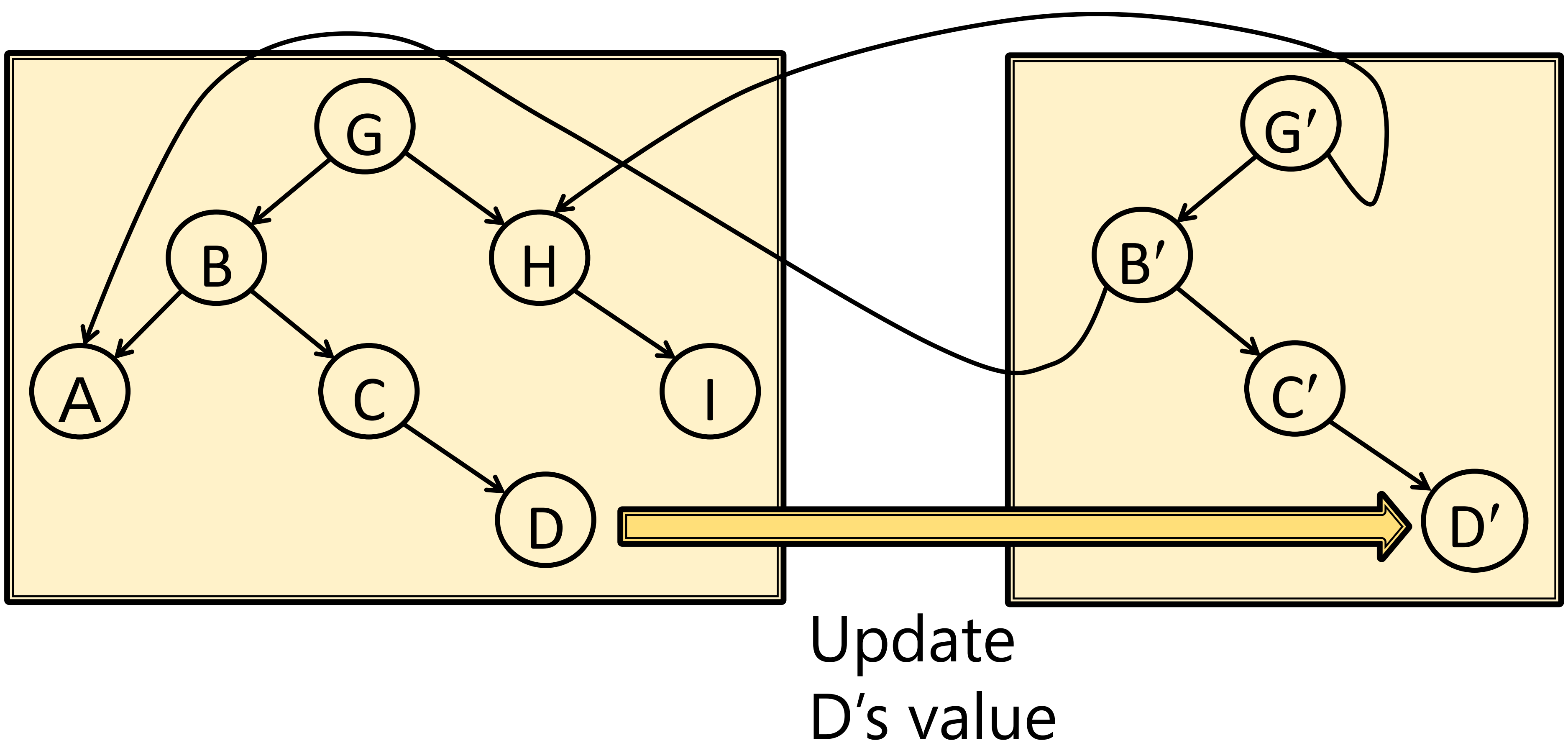


Tree is marshaled into the log



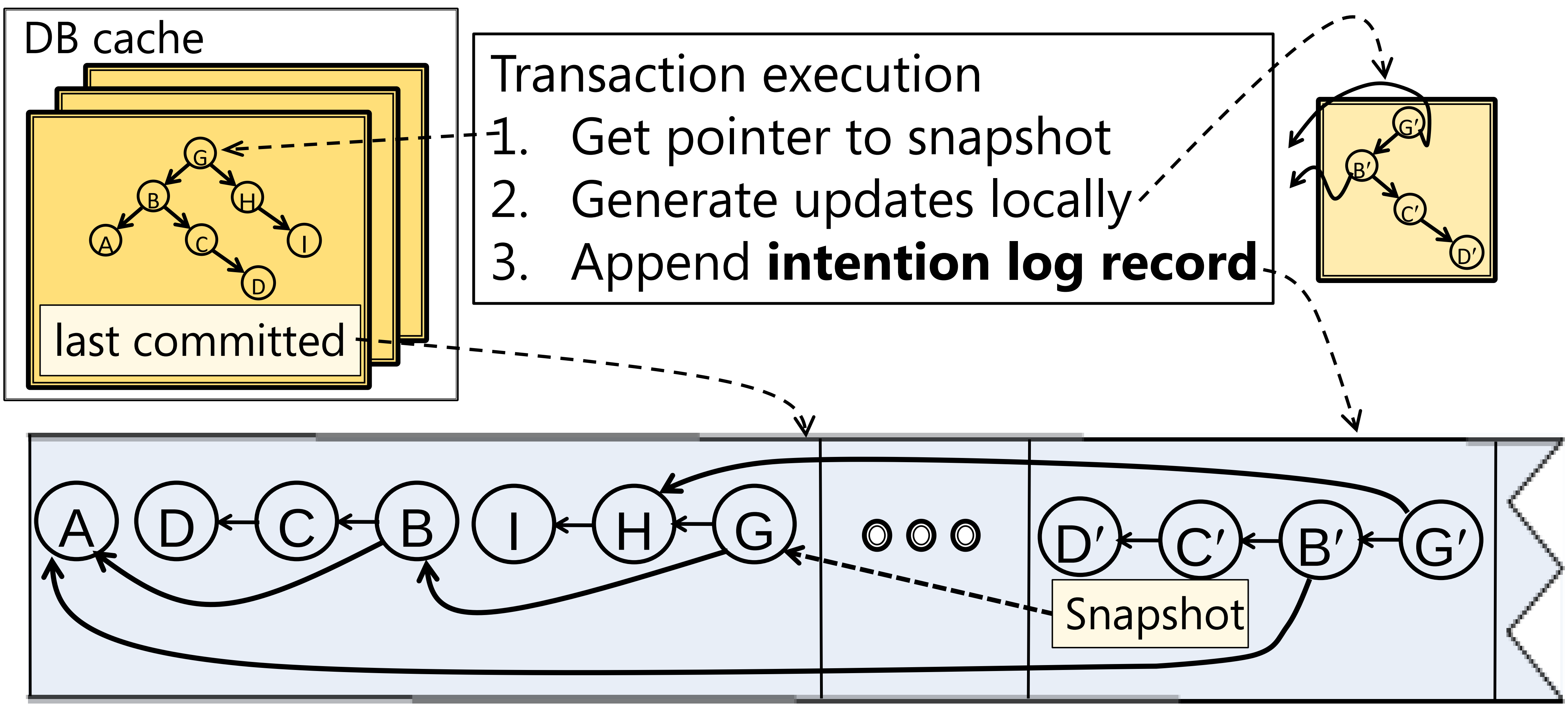
Binary Tree is Multi-versioned

- Copy on write
- To update a node, replace nodes up to the root



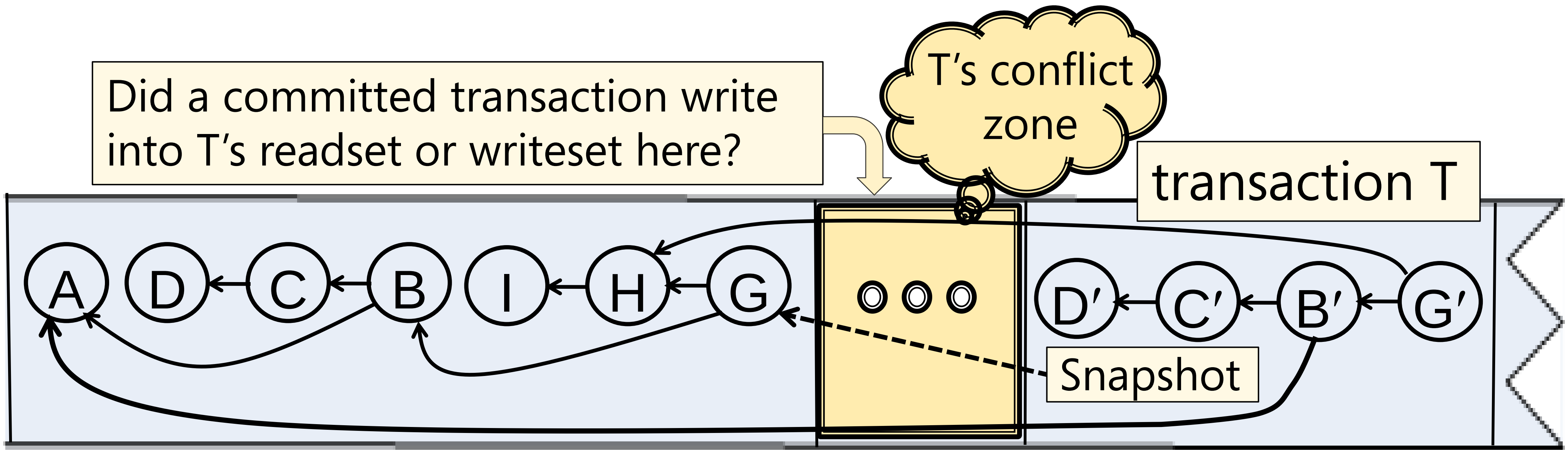
Transaction Execution

- Each server has a cache of the last committed DB state



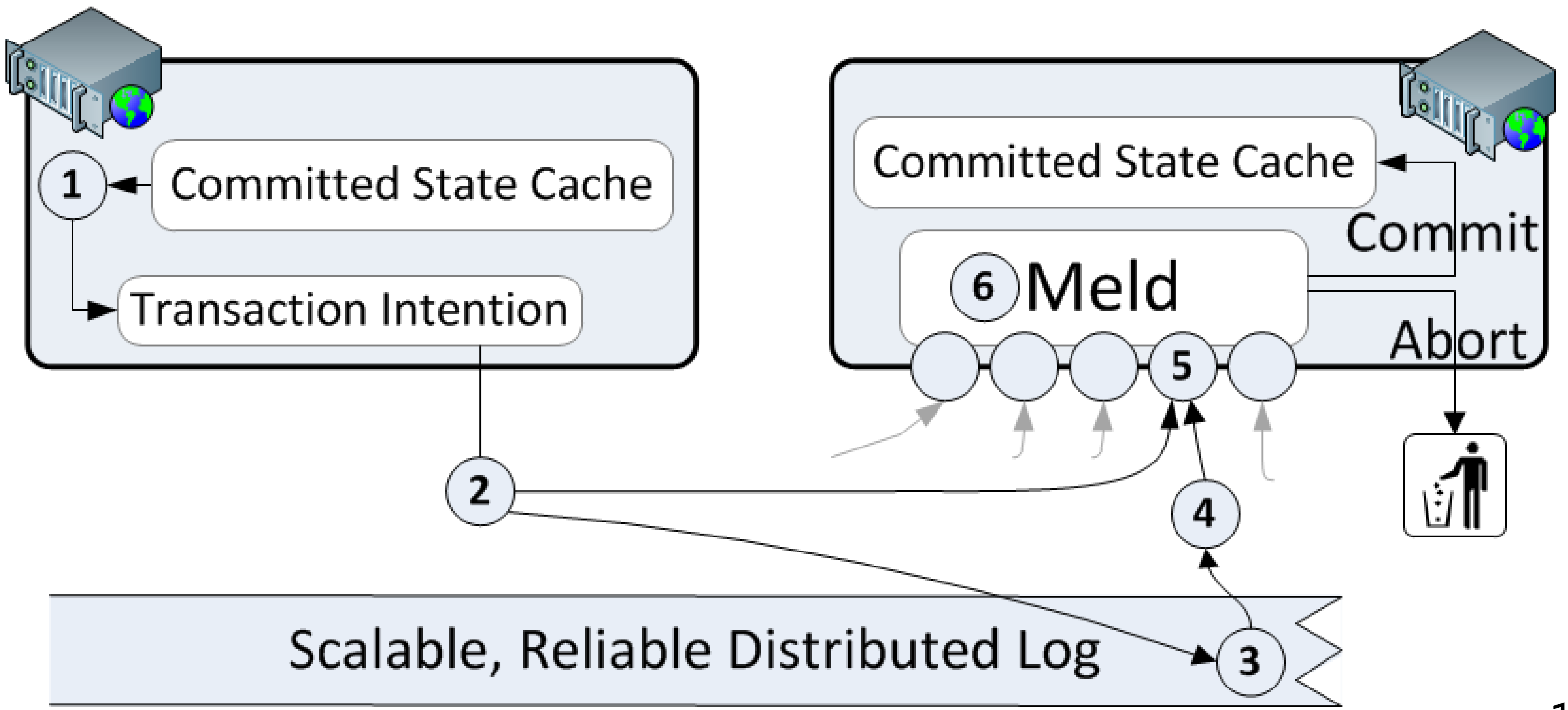
Meld: Log Roll-forward

- Each server processes intention records in sequence
- To process transaction T's intention record.
  - Check whether T experienced a conflict
  - If not, T committed, so the server merges the intention into its last committed state
- All servers make the same commit/abort decisions



Transaction Flow

1. Run transaction
2. Broadcast intention
3. Append intention to log
4. Send log location
5. De-serialize intention
6. Meld



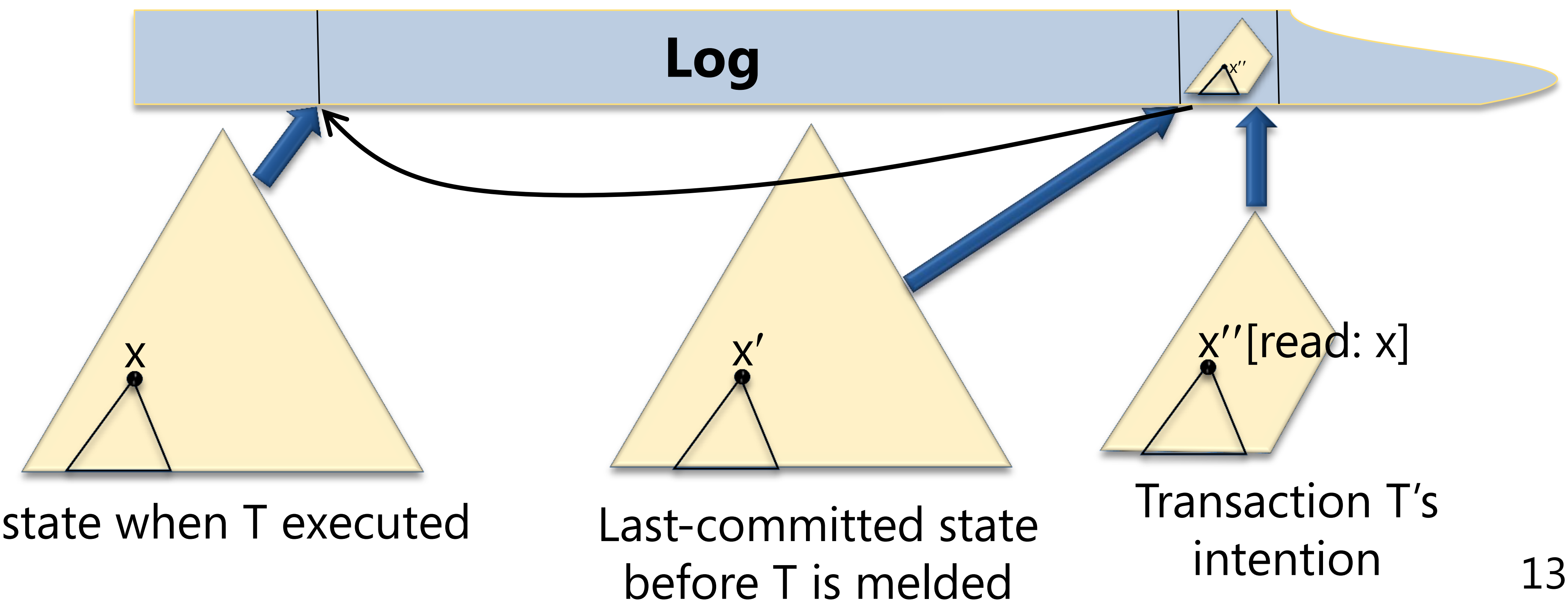
Bottlenecks

1. Broadcasting the intention
2. Appending intention to the log
3. Optimistic concurrency control (OCC)
4. Meld

- Technology will improve 1 & 2
- For 3, app behavior drives OCC performance
- But 4 depends on single-threaded processor performance, which isn't improving
- Hence, it's important to optimize Meld

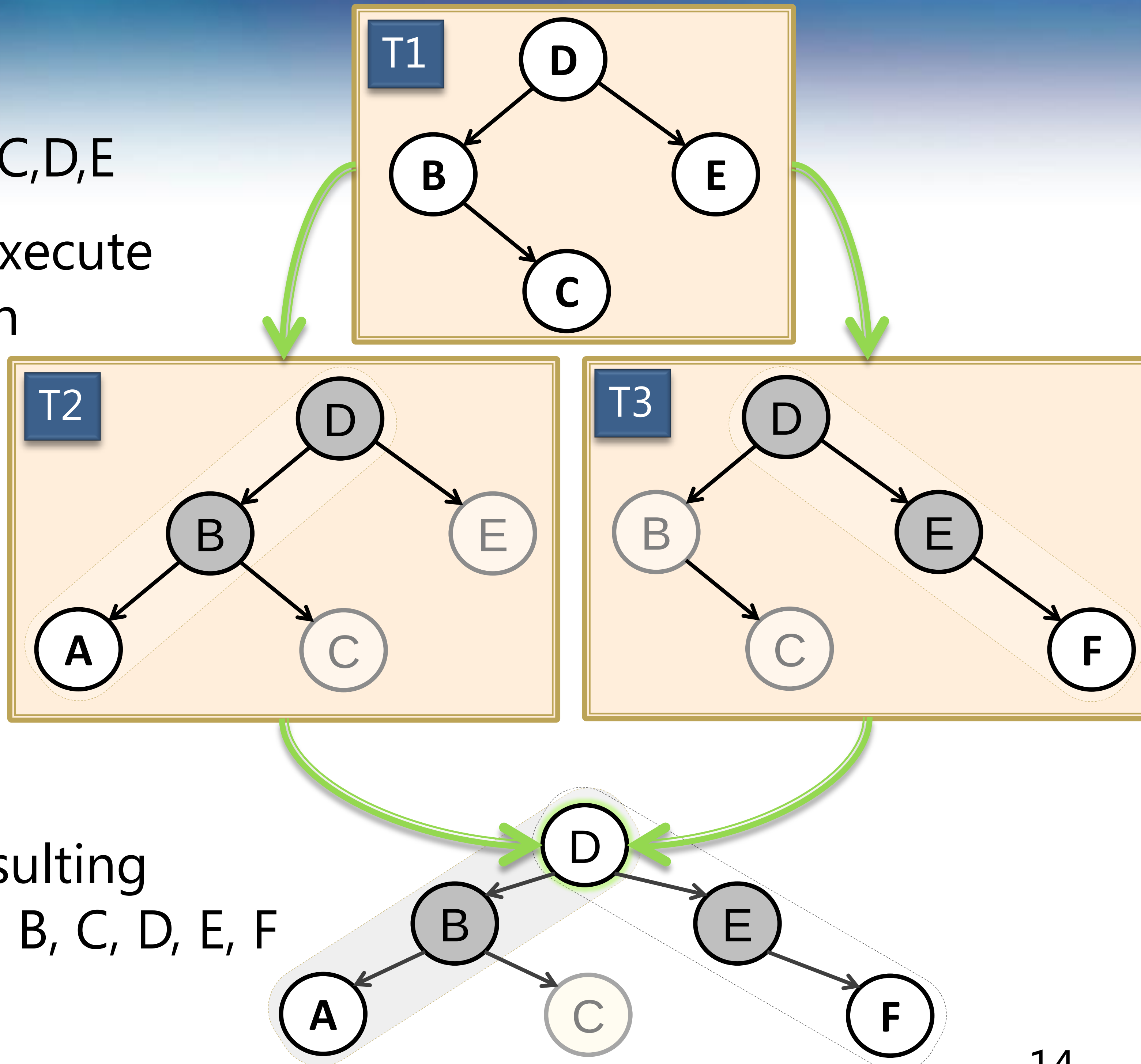
Main Idea: Fast Conflict Check

- Compare transaction T's after-image to the last committed state
  - which is annotated with version and dependency metadata
- Traverse T's intention, comparing versions to last-committed state
- Stop traversing when you reach an unchanged subtree
- If version(x)=version(x′) then simply replace x′ by x′′



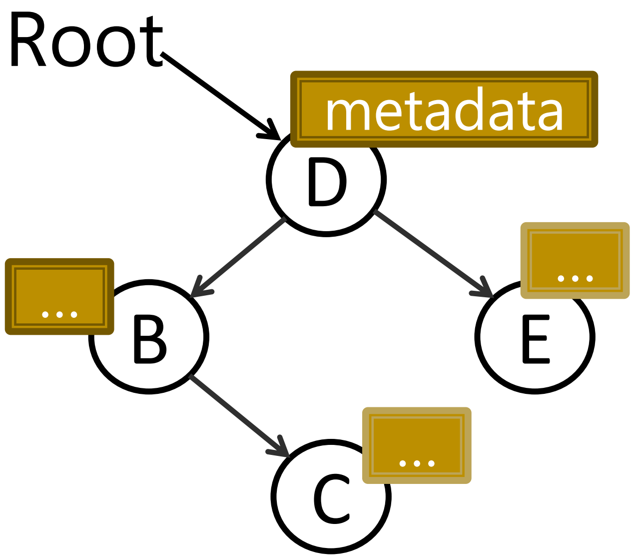
Running Example

- T1 creates keys B,C,D,E
- Then T2 and T3 execute concurrently, both based on the result of T1
- T2 inserts A
- T3 inserts F
- T2 and T3 do not conflict, so the resulting melded state is A, B, C, D, E, F



Intention Metadata

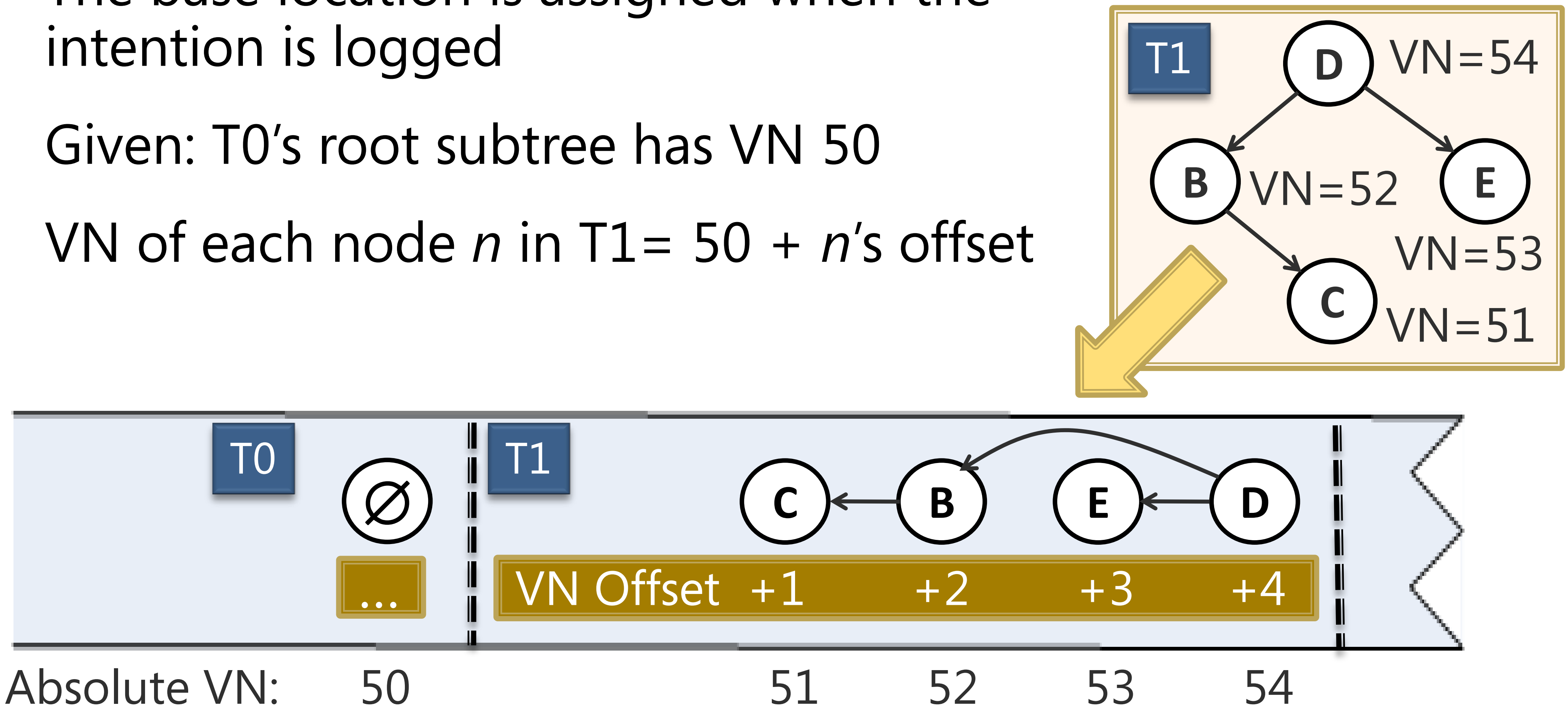
Node Metadata
- version of the subtree
- dependency info



- Every node *n* has a unique version number, VN(*n*), which identifies the exact content of *n*'s subtree
- Every node *n* in an intention T stores metadata about T's snapshot
  - Version of *n* in T's snapshot
  - Dependency information
  - metadata compresses to ~30 bytes

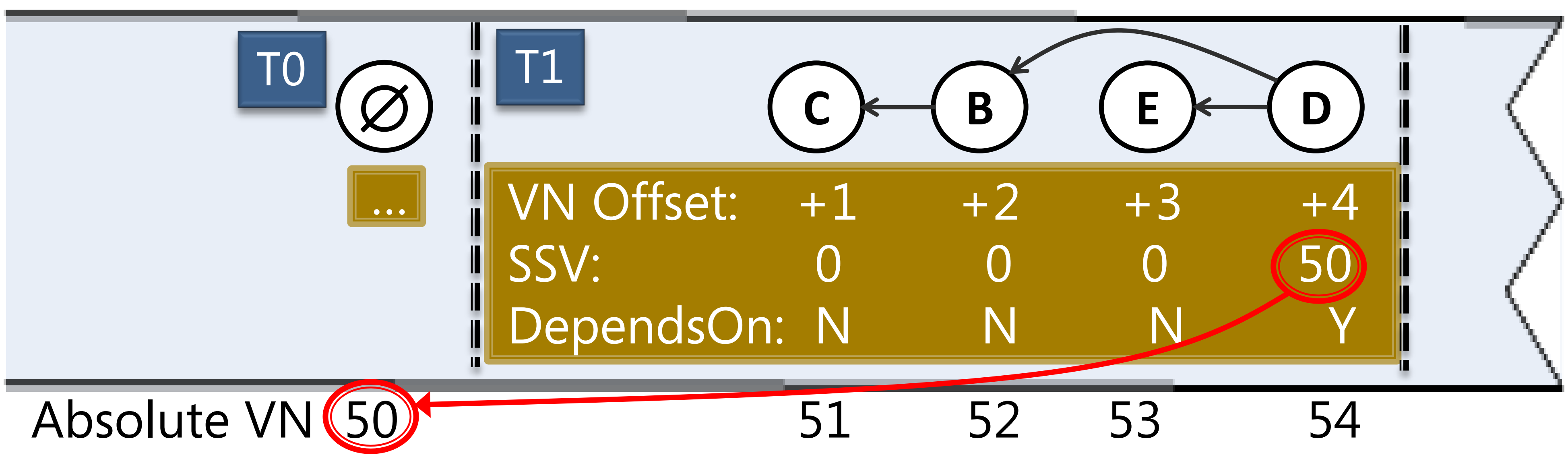
Lazy VN Assignment

- We need to avoid synchronization when assigning VNs
- VN(*n*) = intention base location + offset of *n* in its intention
- The base location is assigned when the intention is logged
- Given: T0's root subtree has VN 50
- VN of each node *n* in T1= 50 + *n*'s offset



Source Versions and Dependencies

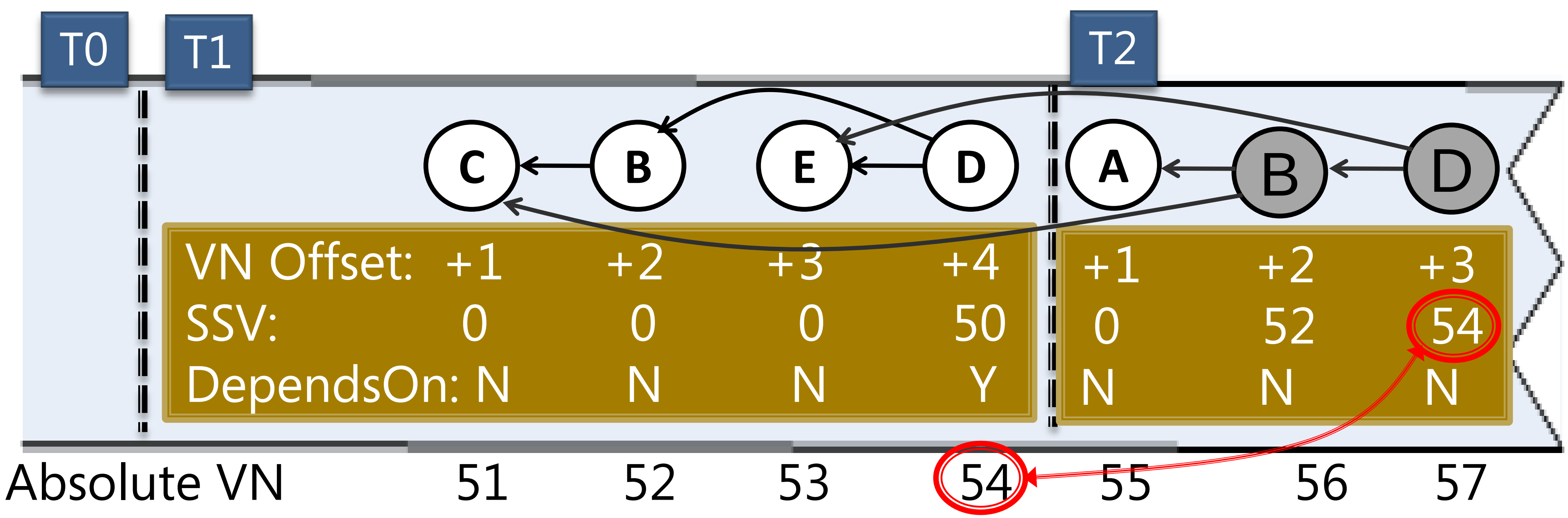
- Subtree metadata includes a **source structure version** (SSV).
- Intutively, SSV(*n*) = version of *n* in transaction T's snapshot
- DependsOn(*n*) = Yes if T depends on *n* not having changed while T executed



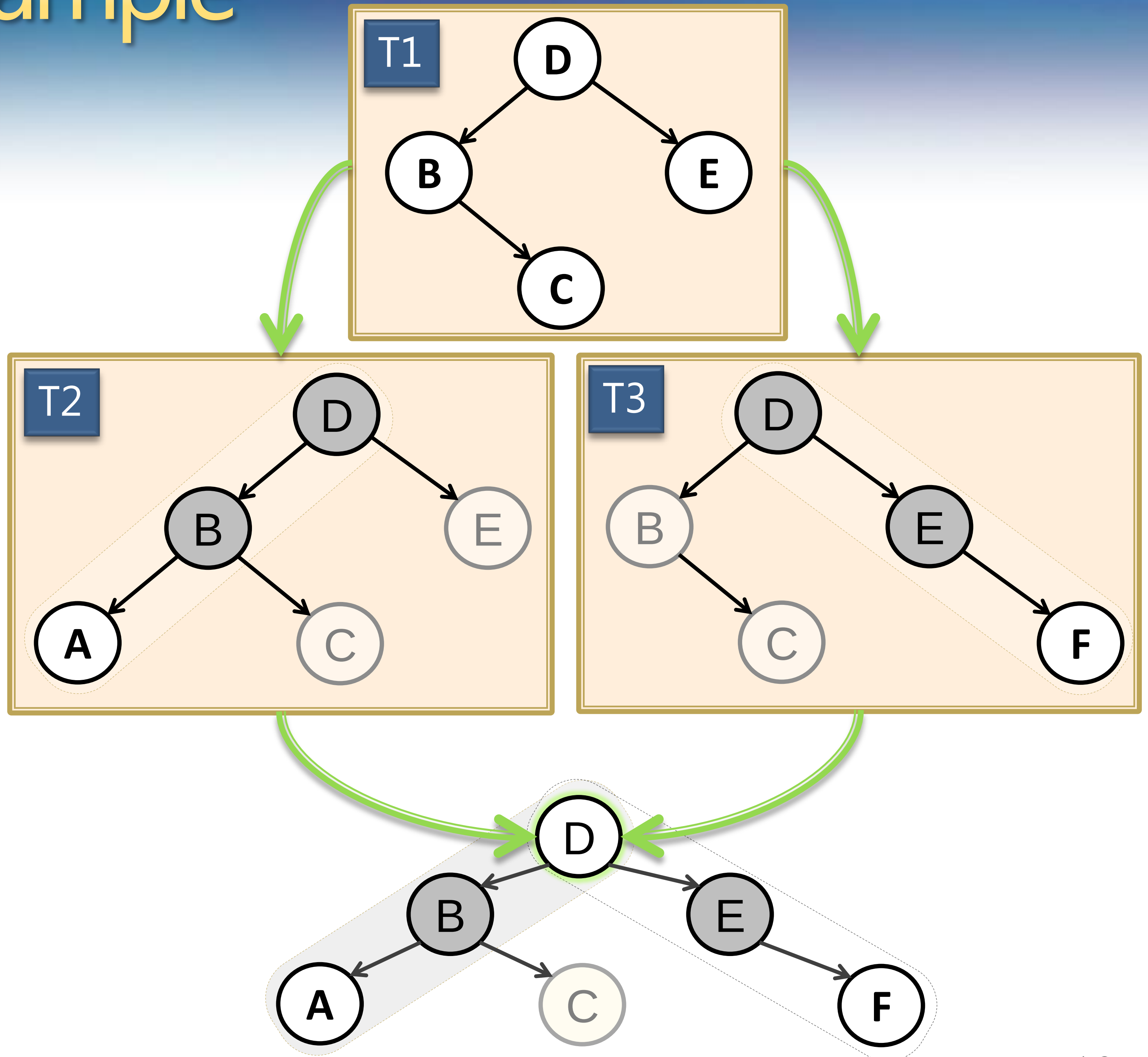
- T1's root subtree depends on the entire tree version 50.
- Since SSV(D) = VN(∅), T1 becomes the last-committed state.

Serial Intentions

- A **serial intention** is one whose source version is the last committed state.
- Meld is then trivial and needs to consider only the root node.
  - T1 was serial.
  - T2 is serial, so meld makes T2 the last committed state.
- Thus, a meld of a serial intention executes in constant time.

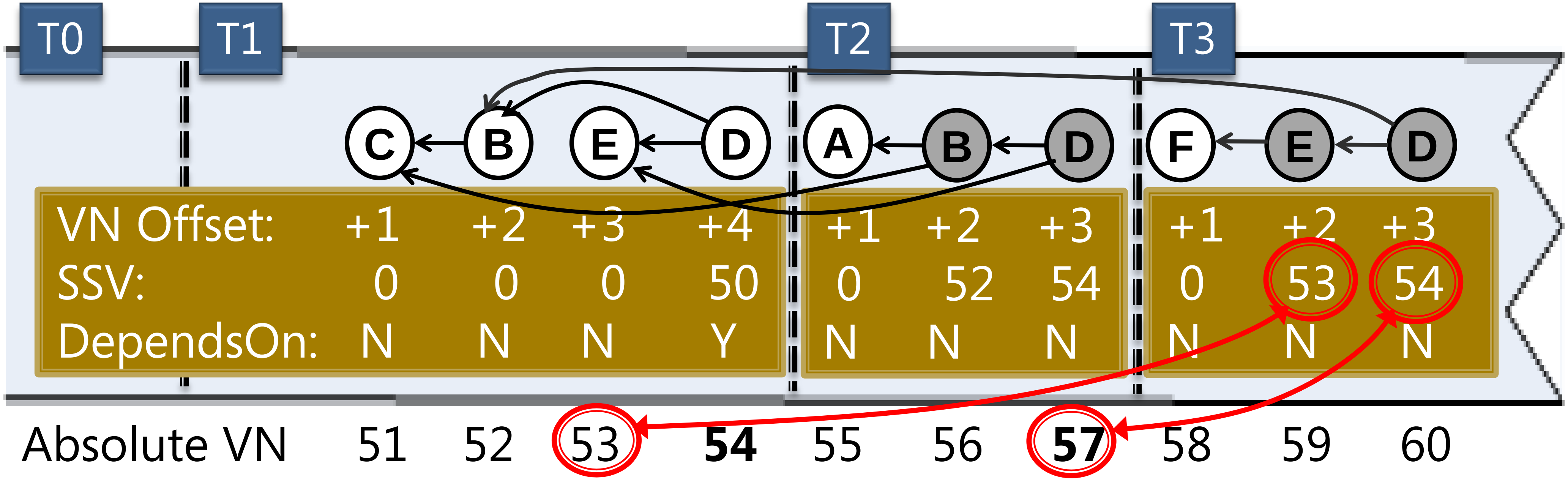


Running Example



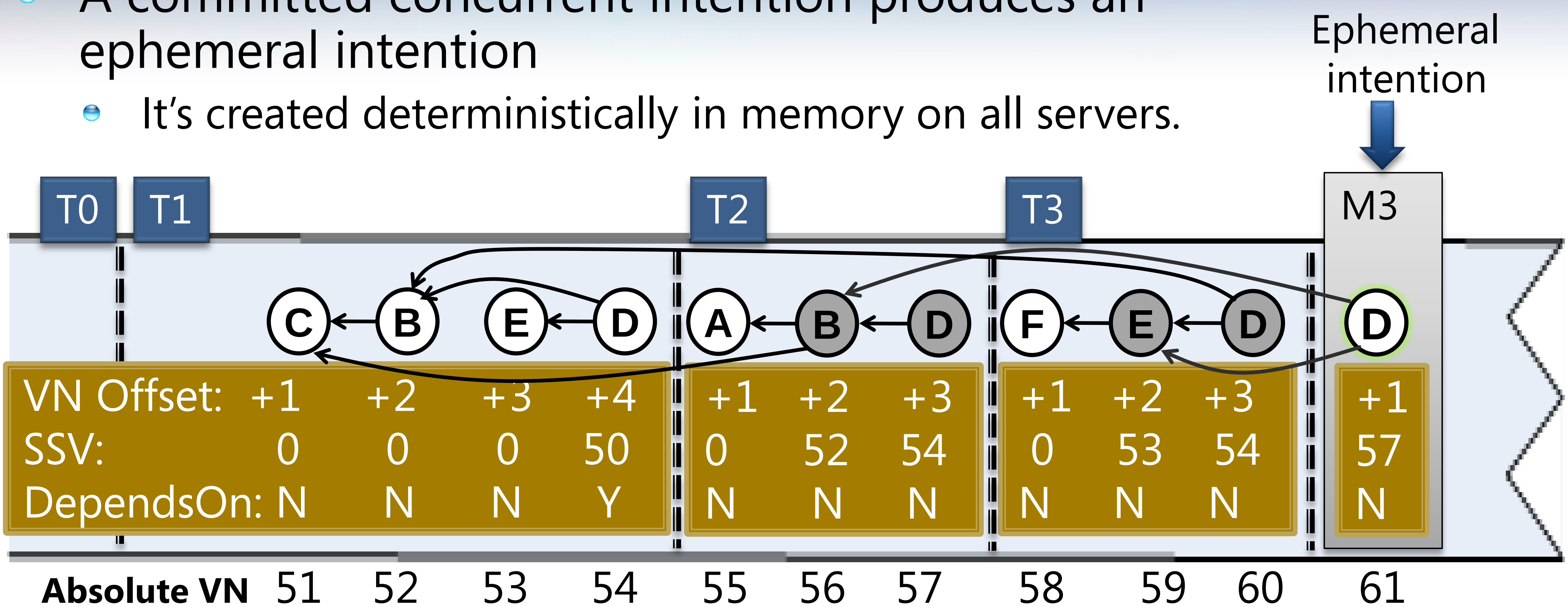
Concurrent (= non-serial) Intentions

- T3 is not serial because VN of D in T2 (= **57**) ≠ SSV(D) in T3 (= **54**).
- Meld checks if T3 conflicts with a transaction in its conflict zone
- Traverses T3, comparing T3's nodes to the last-committed state
- When a concurrent transaction (e.g. T3) experiences no conflicts, meld creates an **ephemeral intention** to merge its state



Ephemeral Intentions

- A committed concurrent intention produces an ephemeral intention
  - It's created deterministically in memory on all servers.



- It logically commits immediately after the intention it melds.

Other Important Details

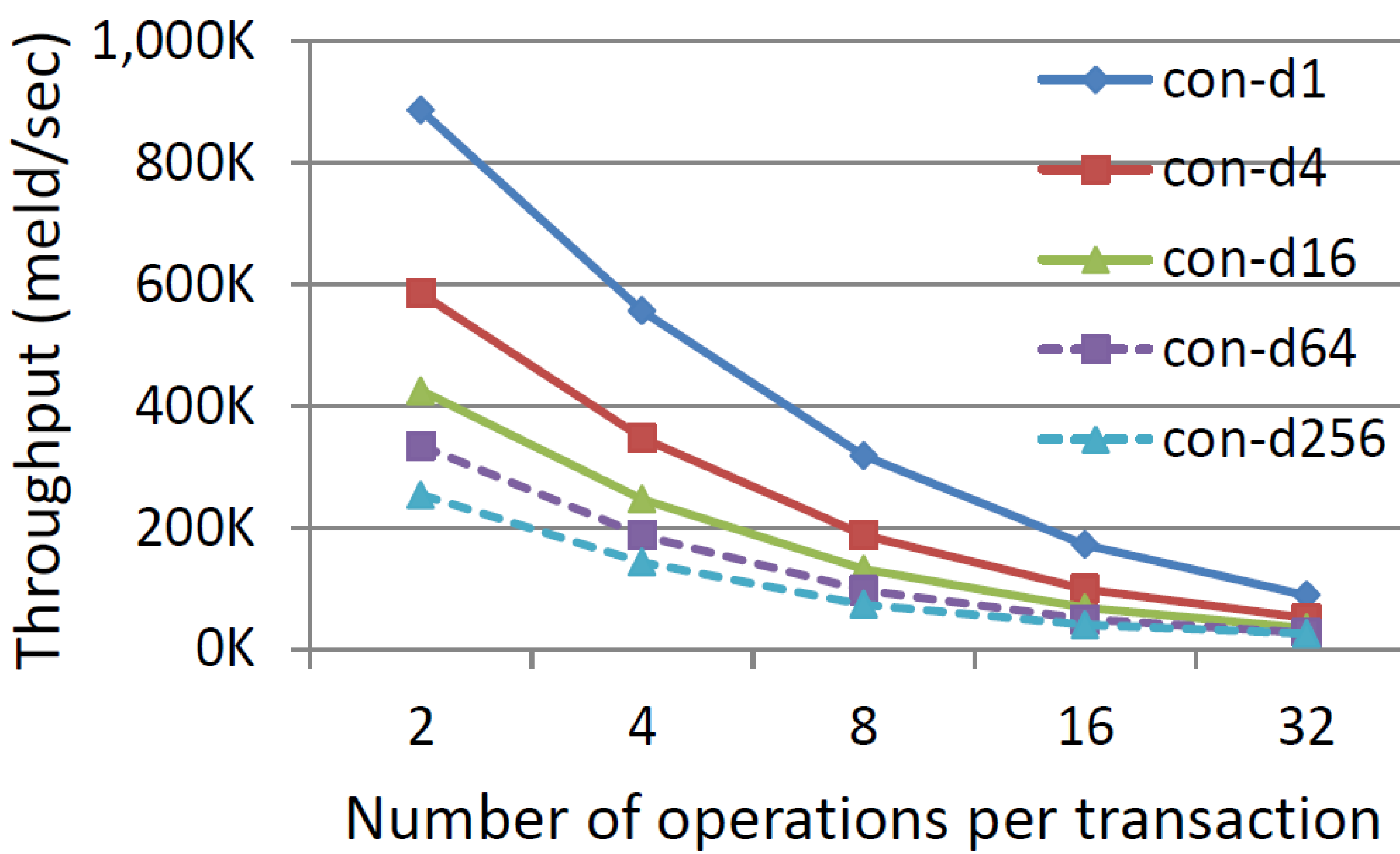
- Distinguishing payload updates from subtree updates
- Phantom detection
- Asymmetric meld operations
- Deletions, using tombstones in the intention header
- Garbage collection
- Checkpointing and recovery
- See [Bernstein et al., VLDB 2011]

Performance

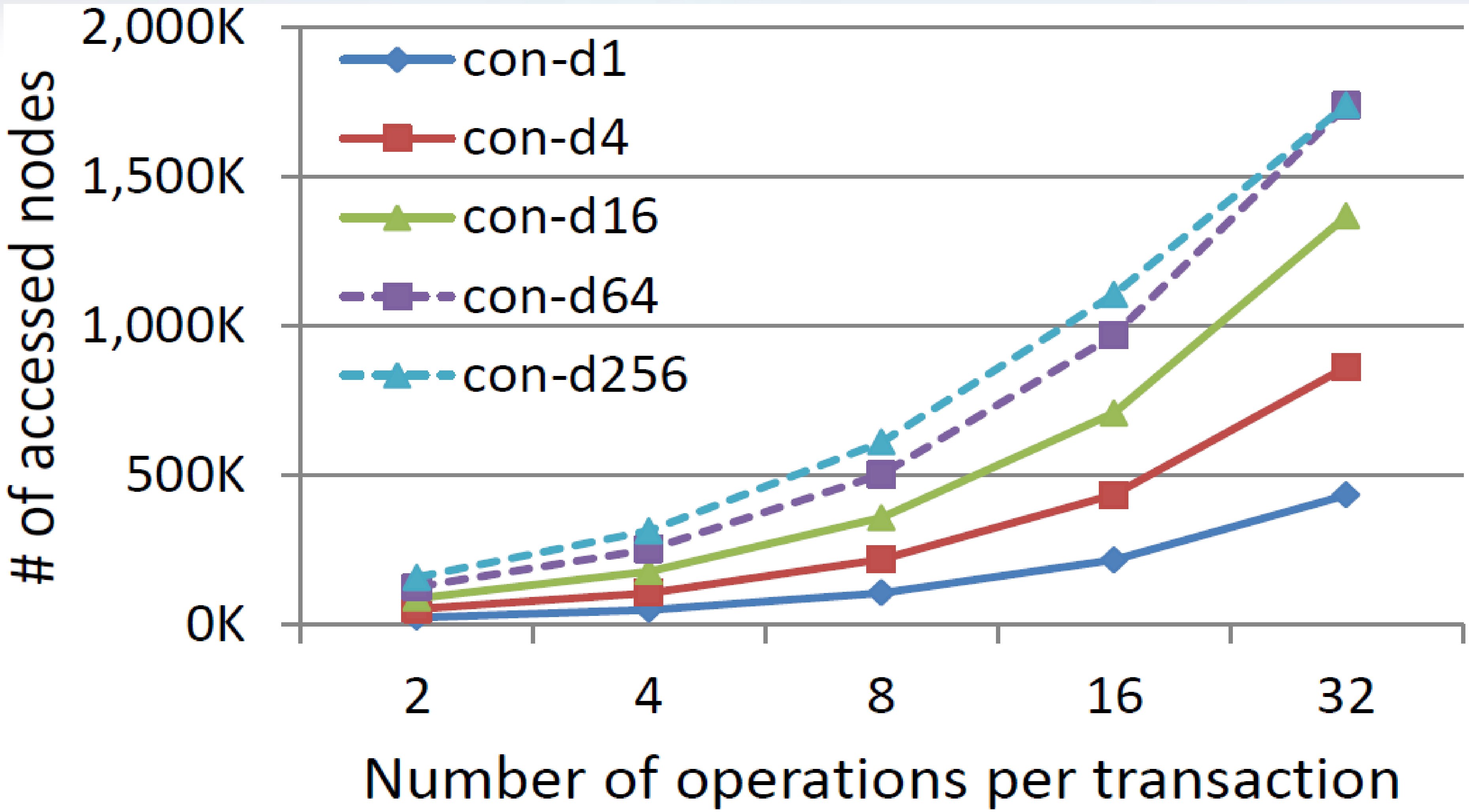
- Focus here is on meld throughput only
  - For latency, see our VLDB 2011 paper
  - We count committed and aborted transactions
- Experiment setup
  - 128K keys, all in main memory. Keys and payloads are 8 bytes.
  - Serializable isolation, so intentions contain readsets
  - De-serialize intentions on separate threads before meld
- Meld throughput depends on transaction size and conflict zone size (“concurrency degree”)
  - As transaction size or concurrency degree increase
    $\Rightarrow$ more concurrent transactions update keys with common ancestors
    $\Rightarrow$ meld has to traverse deeper in the tree

Throughput

- r:w ratio is 1:1
- con-d*i* = concurrency degree *i*

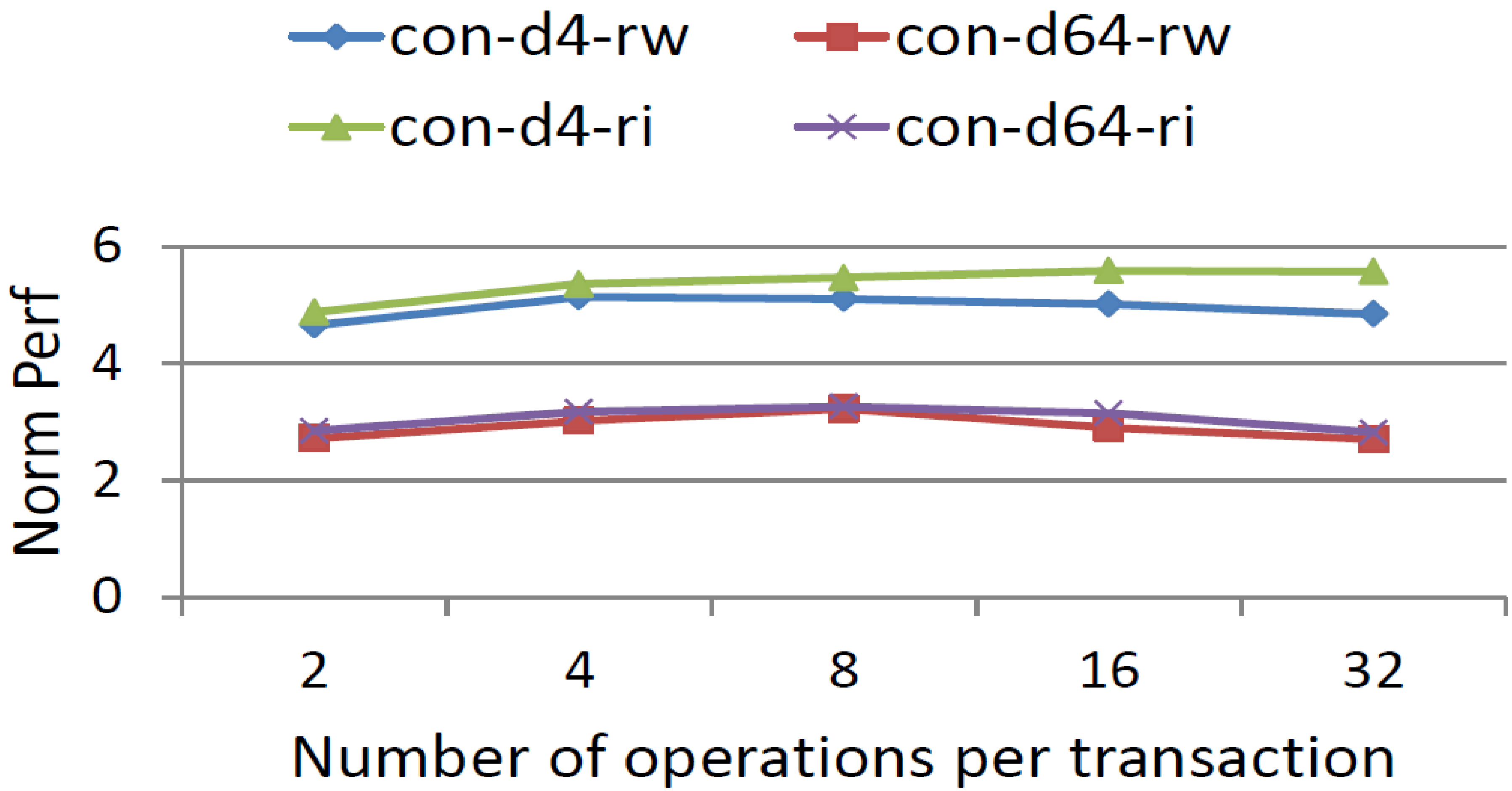


Number of Nodes Accessed



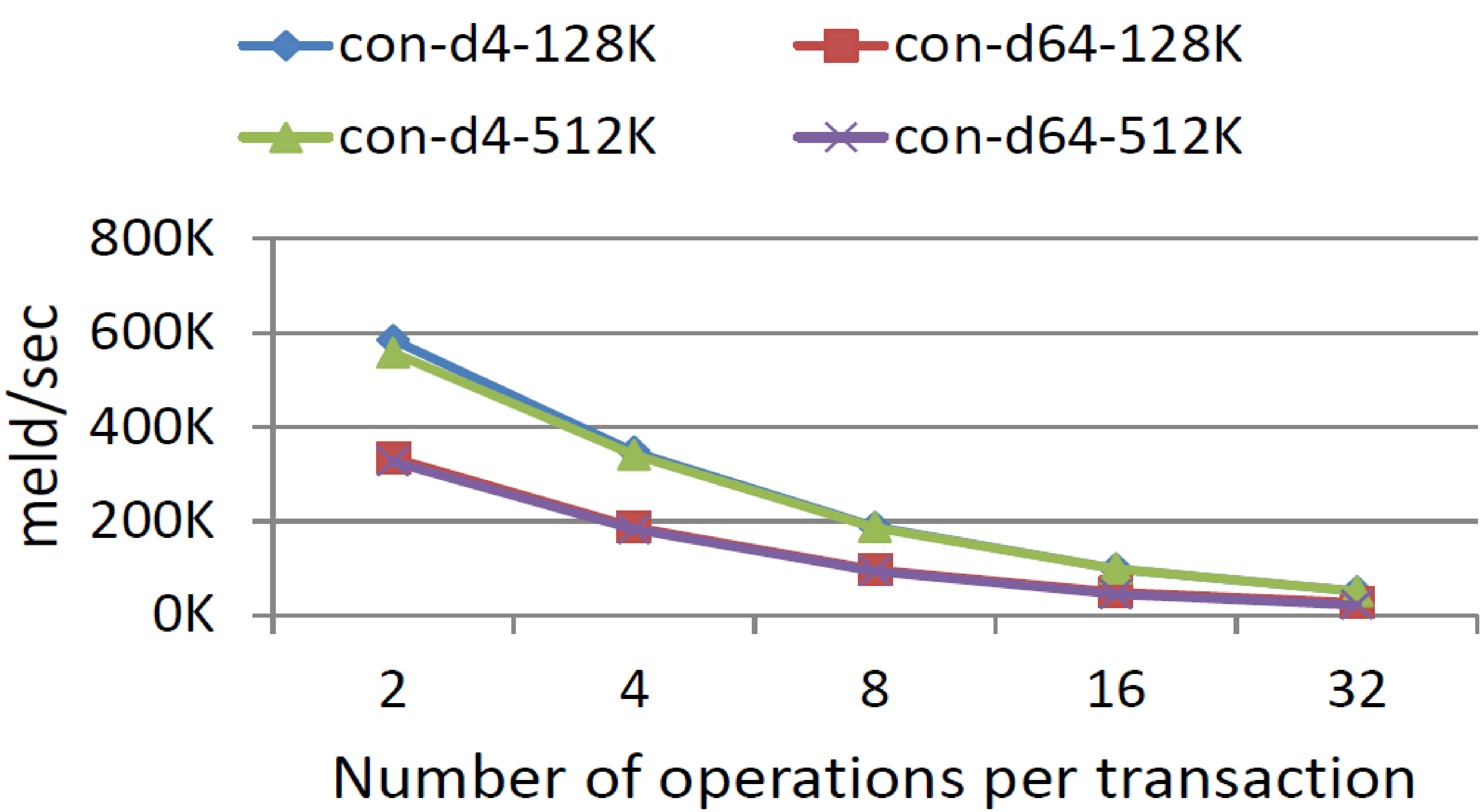
Meld Performance vs. Brute Force

- Brute force = traverse the whole tree



Effect of Tree Depth

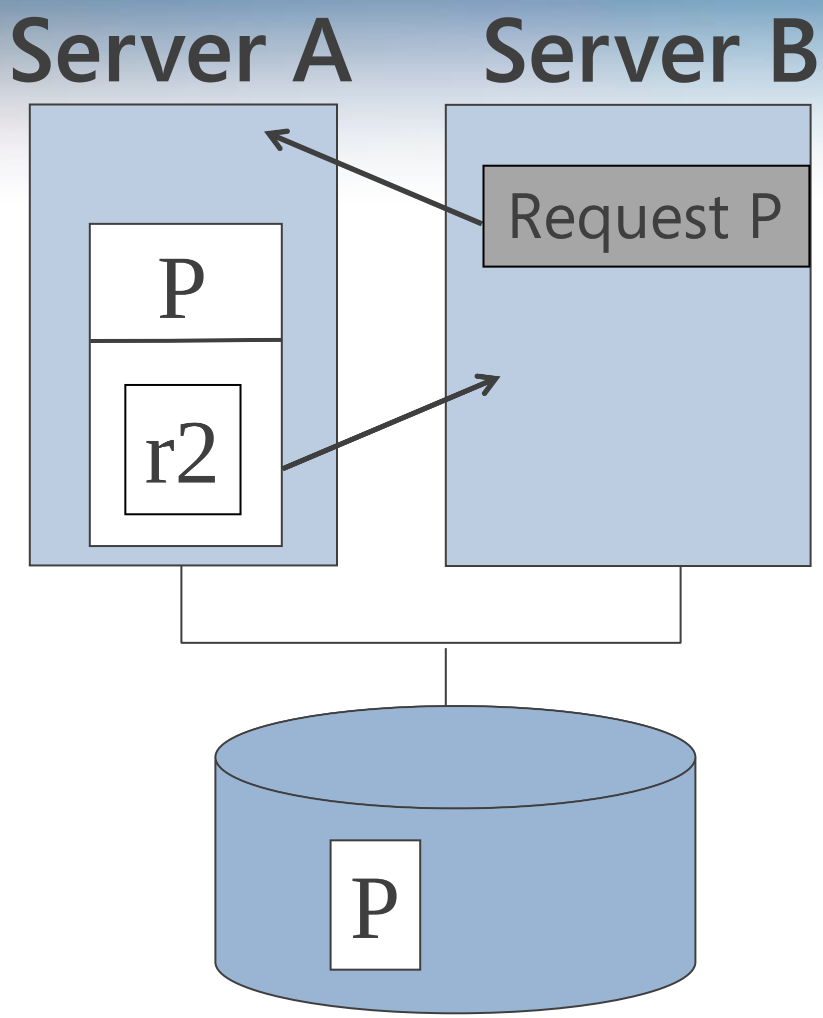
- Hardly any effect, indicating most traversals short-circuit high in the tree.



Related Work

- Hyder resembles a primary-copy replicated DB
  - Primary copy broadcasts only committed updates
  - Central transaction server is a bottleneck
  - In Hyder, only the log is centralized
- Hyder is a “data-sharing” DB system
  - Classical approach uses a distributed lock manager
  - Each server runs an ordinary single-server DBMS
  - But, before a server fetches a page, it locks the page

Data Sharing via Locking



- Server A gets a write-lock on page P and fetches P
- Server B requests a lock on P
- Lock manager forward request to A
- When A is able to unlock P, it releases the lock and sends P to B
- Need this synchronization even if B wants a different record than A

- Performance issues: remote lock requests; ping-pong pages
- Used in Oracle RAC & Exadata and IBM DB2 Data-Sharing
- Have not yet compared its performance to Hyder

Related Work on Meld

- Lots of OCC papers but none that give details of efficient conflict-testing
- By contrast, there's a huge literature on conflict-testing for locking
- Oxenstored [Gazagnairem & Hanquezis, ICFP 09]
  - Similar scenario: MV trees and OCC
  - However, very coarse-grain conflict-testing
  - Uses none of our optimizations

Summary

- New algorithm for OCC
- Developed many optimizations to truncate the conflict checking early in the tree traversal
- Implemented and measure it
- Future work:
  - Apply it to other tree structures
  - Measure it on various storage devices
  - Compare it with locking and other OCC methods on multiversion trees
  - Try to apply it to physiological logging

Publications

- C.W. Reid, P.A. Bernstein: Implementing an Append-Only Interface for Semiconductor Storage. IEEE Data Eng. Bull. 33(4): 14-20 (2010)
- P.A. Bernstein, C.W. Reid, S. Das: Hyder - A Transactional Record Manager for Shared Flash. CIDR 2011: 9-20
- P.A. Bernstein, C.W. Reid, M. Wu, X. Yuan: Optimistic Concurrency Control by Melding Trees. PVLDB 4(11): 944-955 (2011)

Microsoft Research

Microsoft®

Your potential. Our passion.™