Optimistic Concurrency Control by Melding Trees

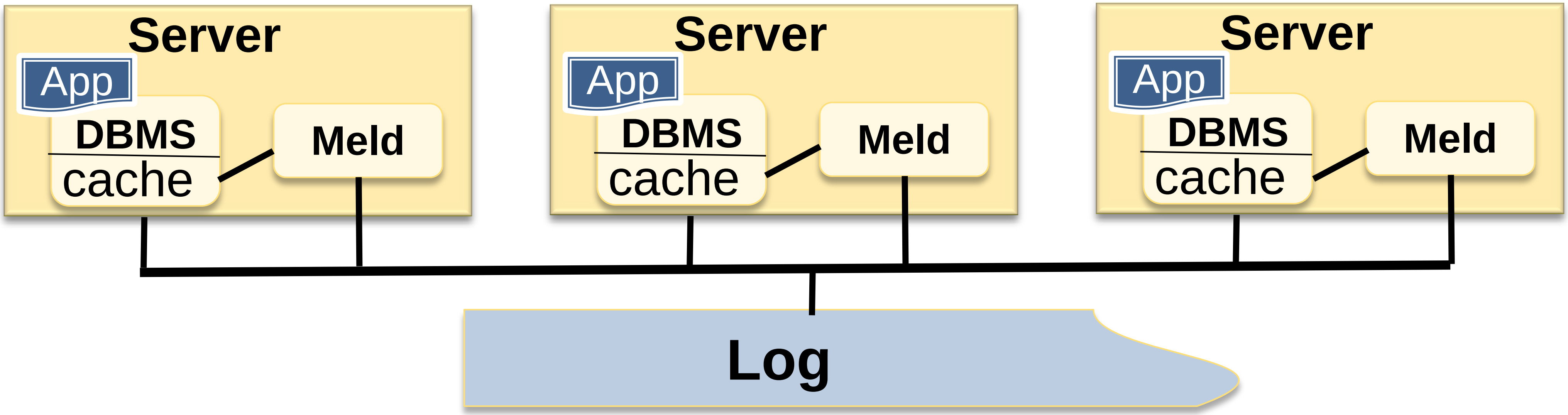
Philip A. Bernstein
Colin Reid
Ming Wu
Xinhao Yuan

Microsoft Corporation
March 7, 2012

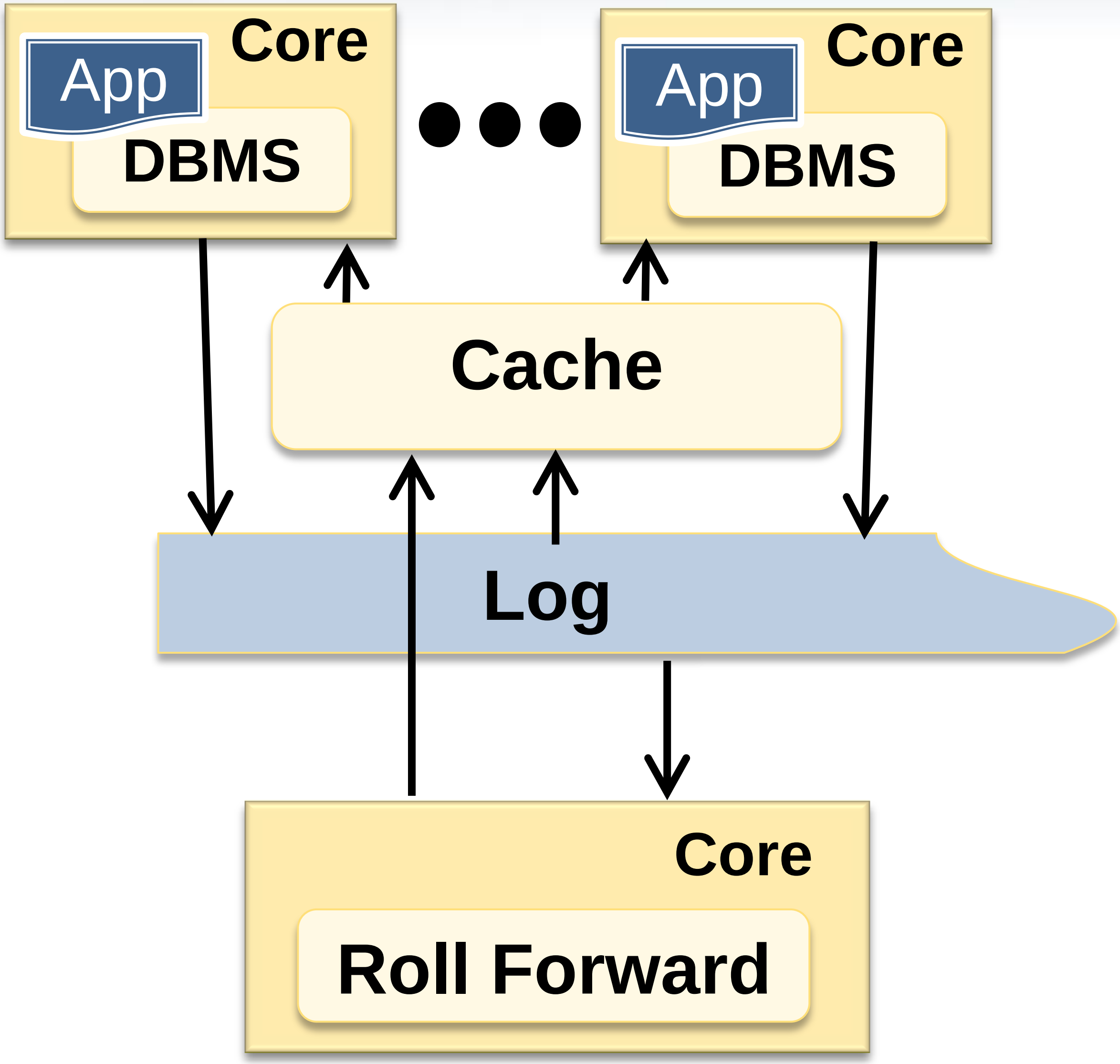
Published at VLDB 2011: http://www.vldb.org/pvldb/vol4/p944-bernstein.pdf

Introduction

- A new algorithm for optimistic concurrency control (OCC) on tree-structured indices, called **meld**.
- Scenario 1: A data-sharing system
  - The log is the database. All servers can access it.
  - Each transaction appends its after-images to the log.
  - Each server runs meld to do OCC and roll forward the log



Scenario 2

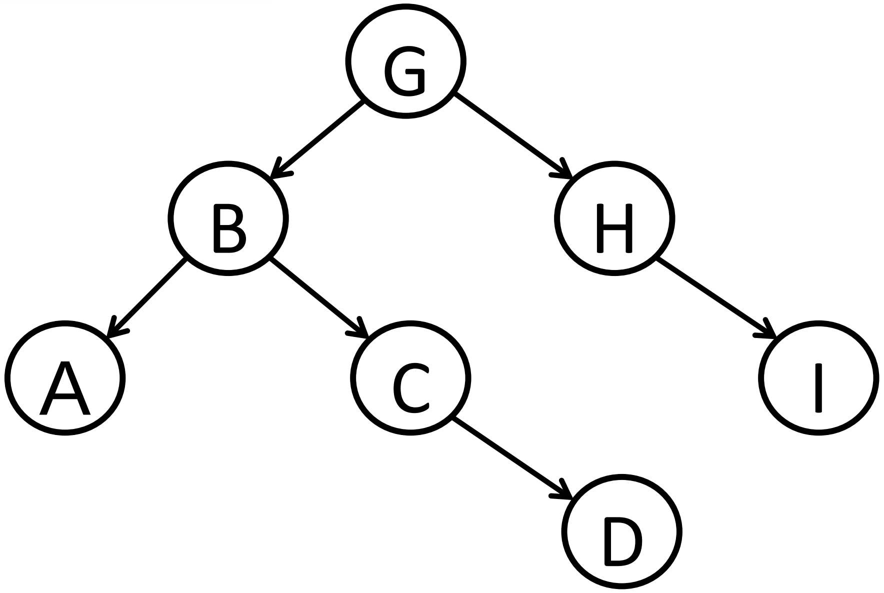


- The log is the database.
- All cores can access it.
- Each transaction appends its after-images to the log.
- One core runs meld to do OCC and roll forward the log

Outline

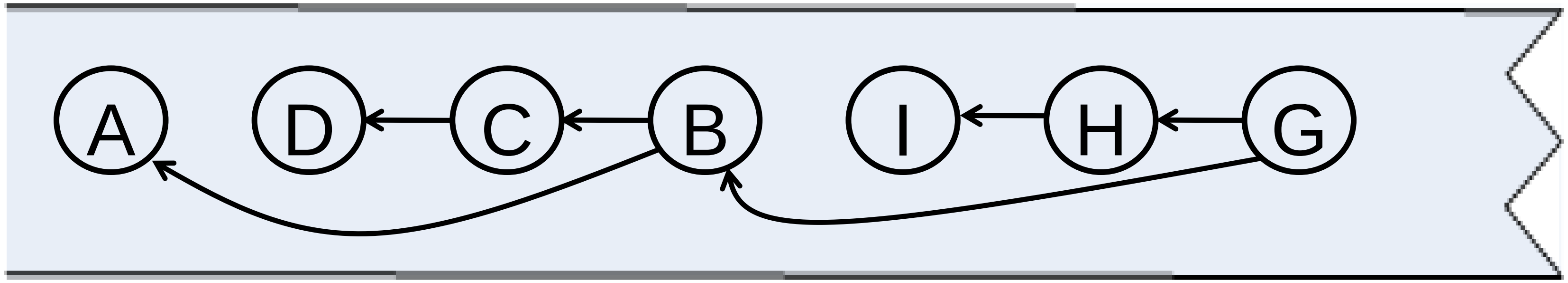
- ✓ Motivation
- System architecture
- Meld Algorithm
- Performance
- Conclusion

Database is a Binary Search Tree



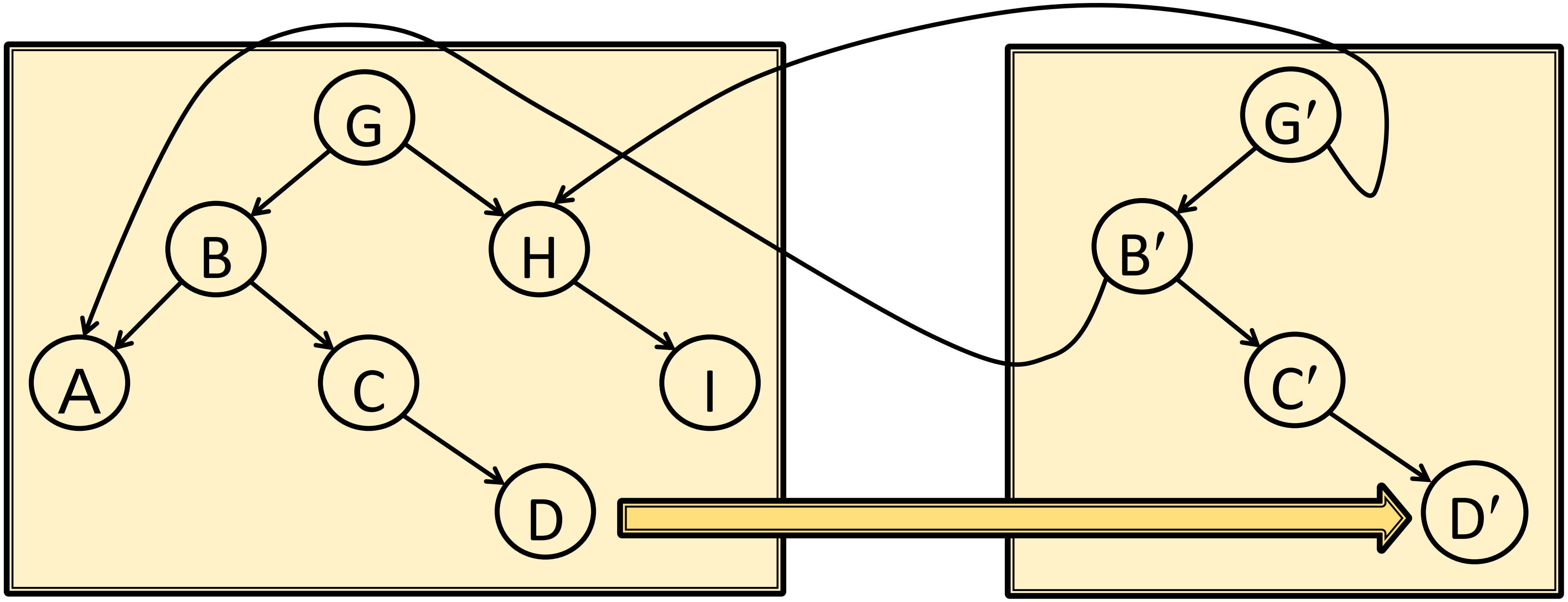
Binary
Search
Tree

Tree is marshaled into the log



Binary Tree is Multi-versioned

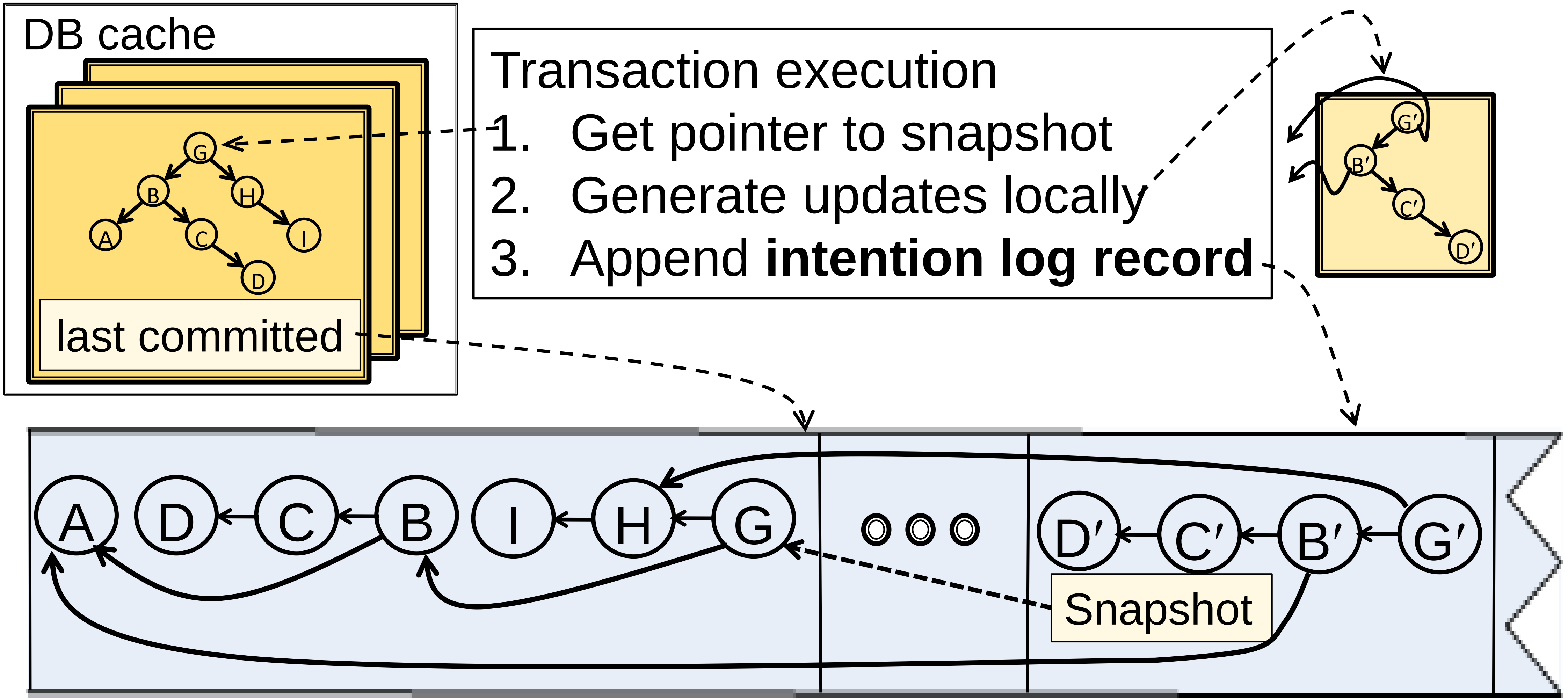
- Copy on write
- To update a node, replace nodes up to the root



Update
D’s value

Transaction Execution

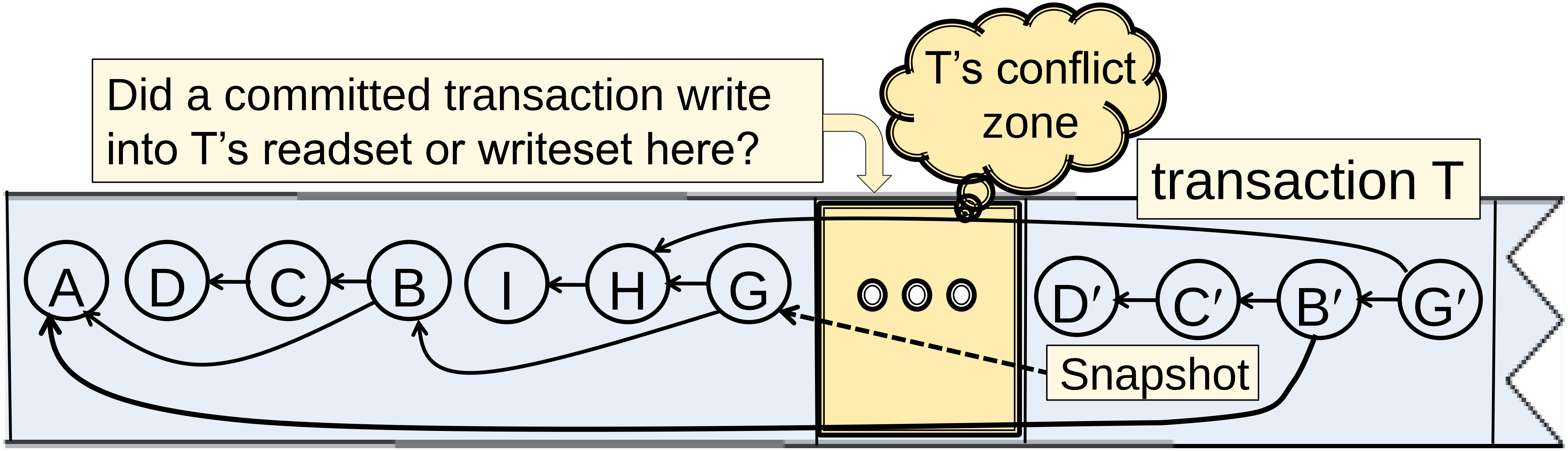
- Each server has a cache of the last committed DB state



Transaction execution
1. Get pointer to snapshot
2. Generate updates locally
3. Append **intention log record**

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



Did a committed transaction write into T's readset or writeset here?

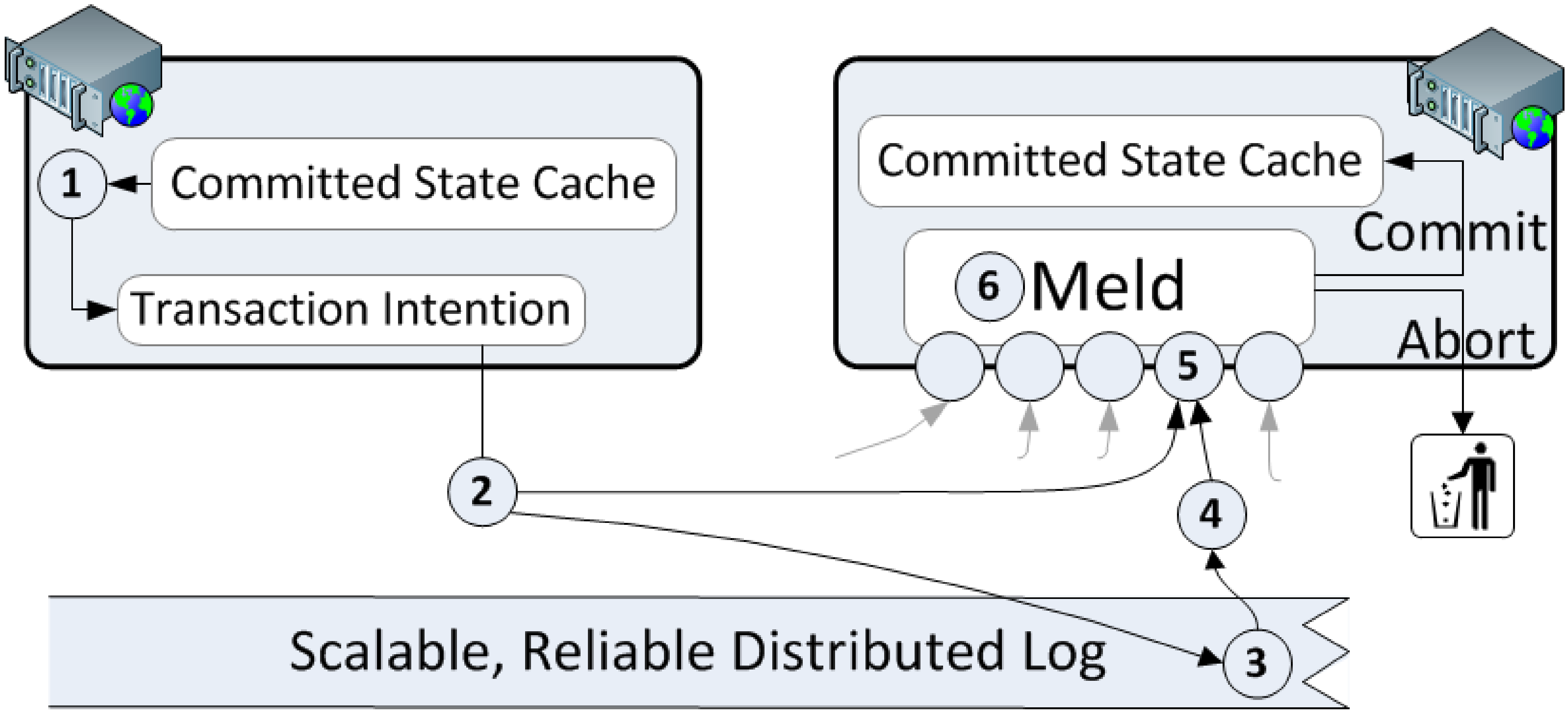
T's conflict zone

transaction T

Snapshot

Transaction Flow

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



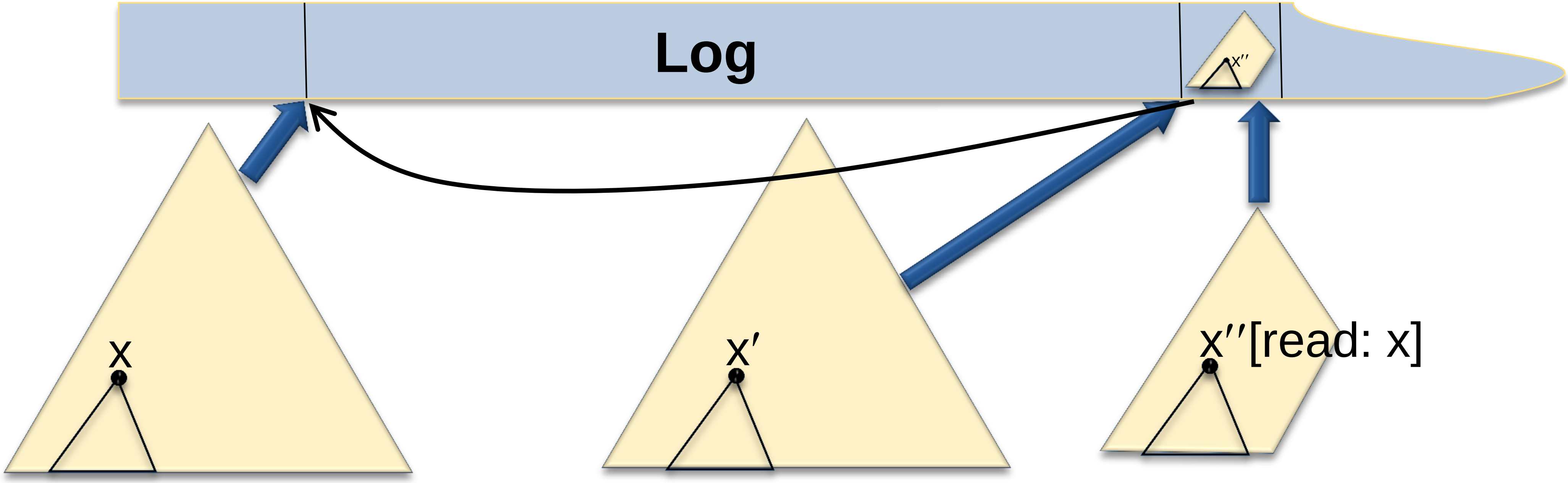
Bottlenecks

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″



state when T executed

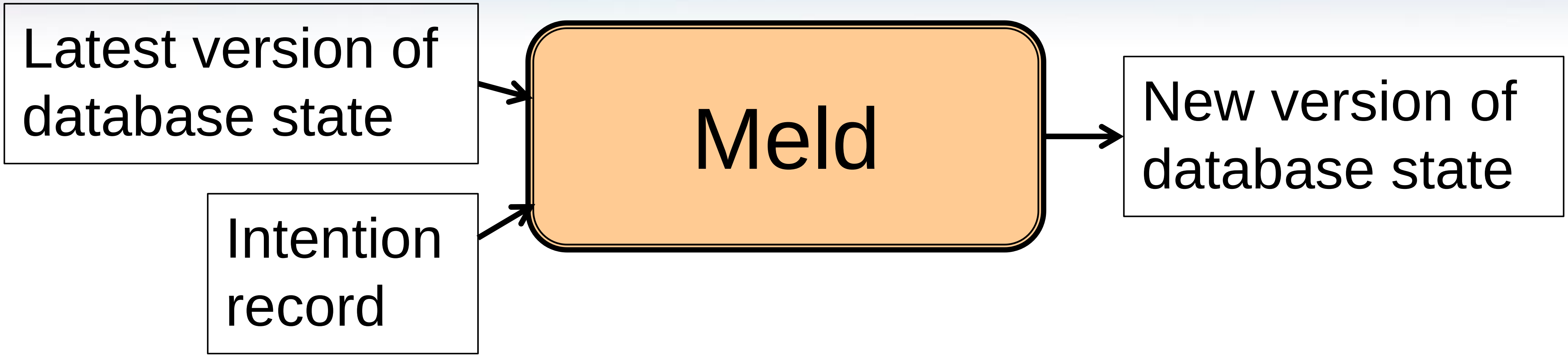
Last-committed state before T

Transaction T's intention

Outline

- ✓ Motivation
- ✓ System architecture
- Meld Algorithm
- Performance
- Conclusion

Meld



Latest version of database state

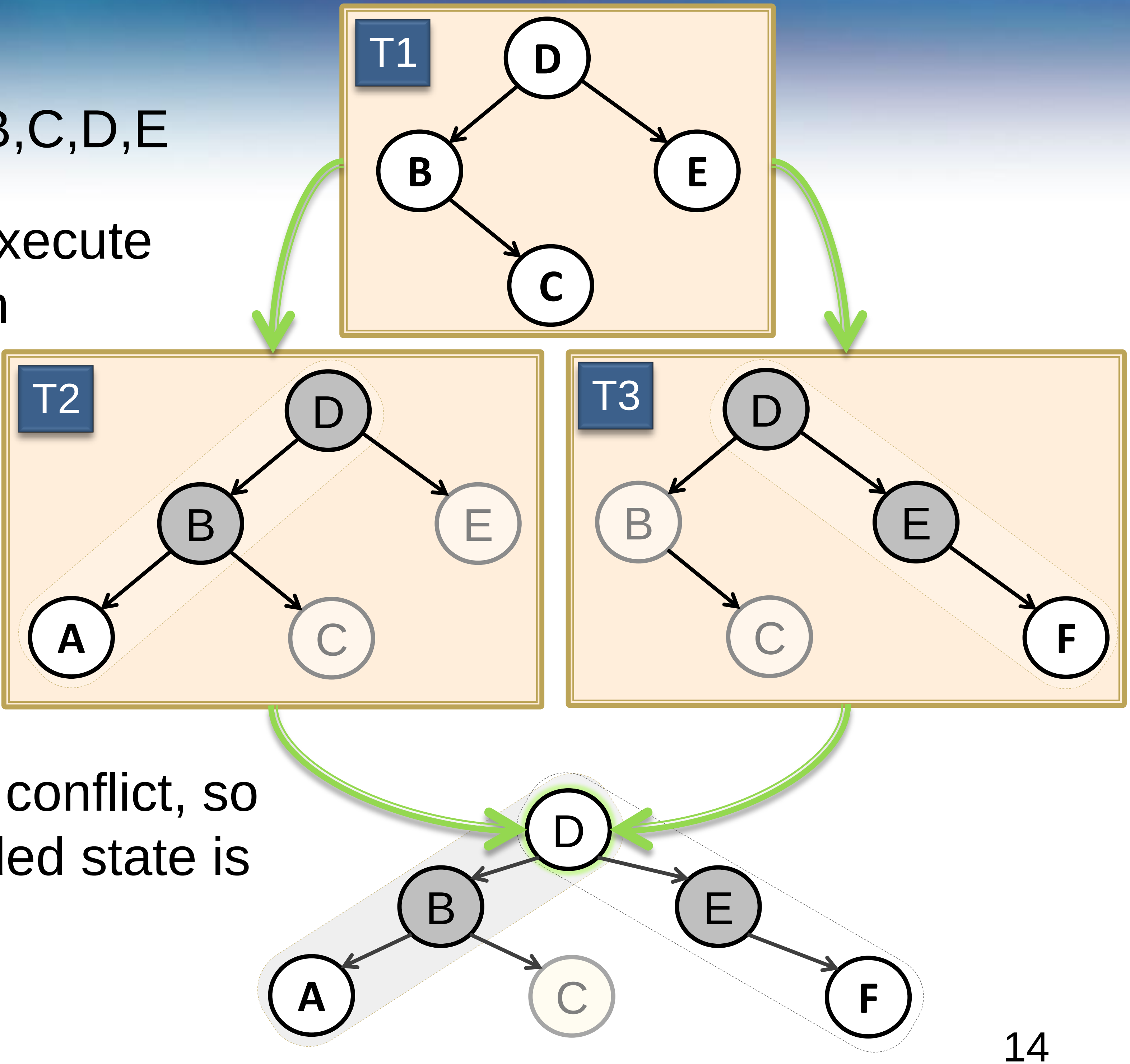
Intention record

Meld

New version of database state

- Must be computationally efficient
- Must be deterministic
  - Must produce the same sequence of states on all servers

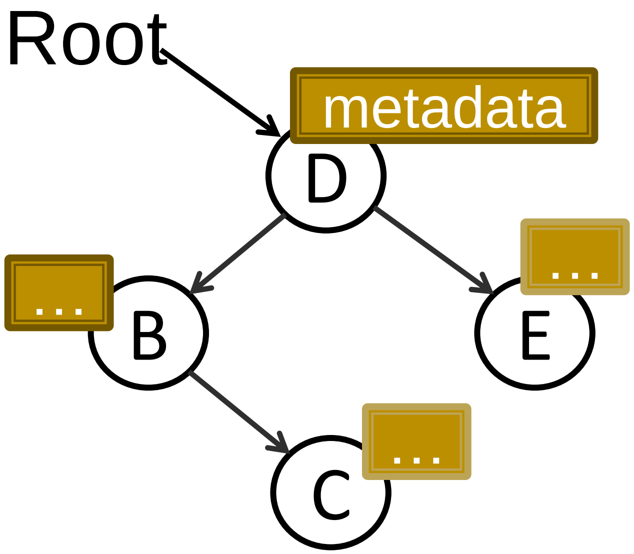
Running Example



- T1 creates keys B,C,D,E
- T2 and T3 then 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

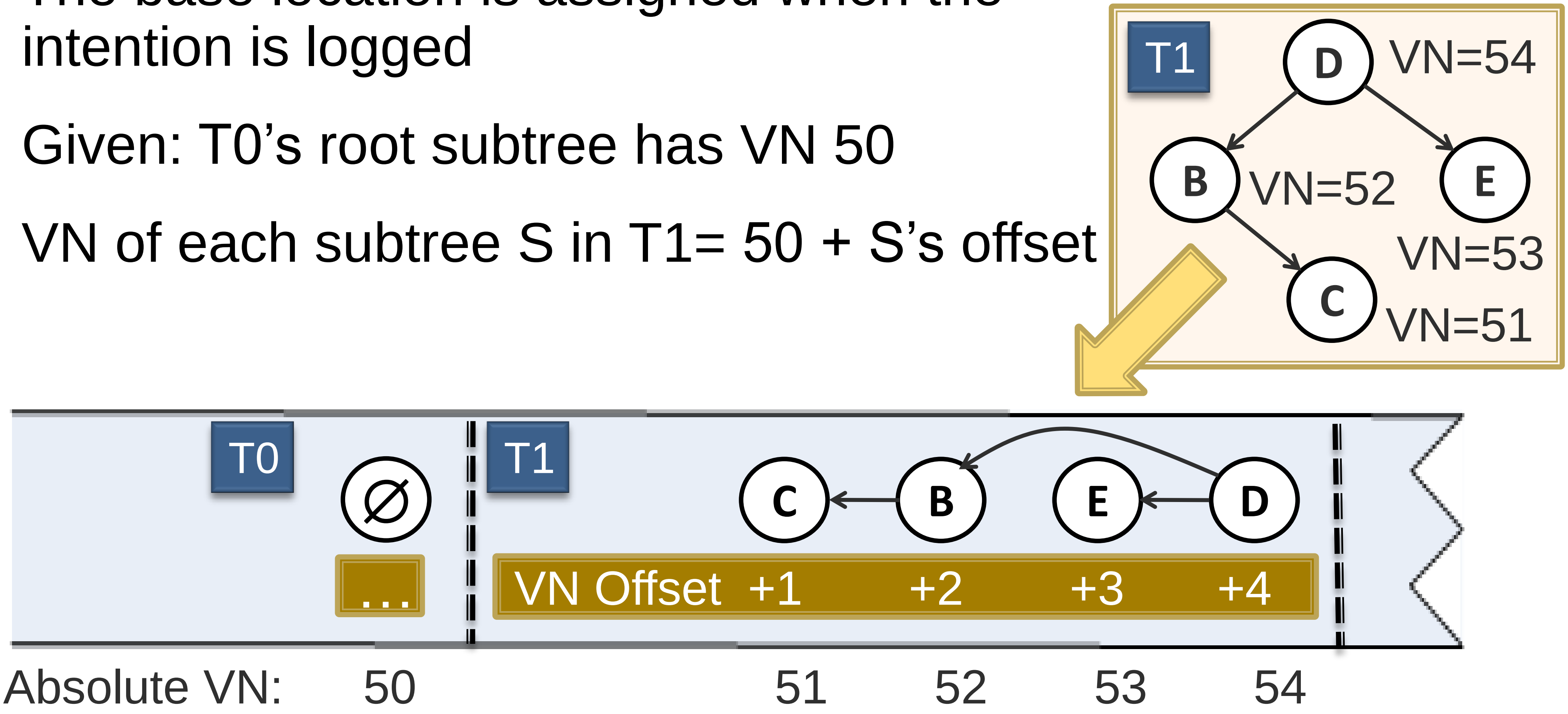
Node Metadata
- version of the subtree
- dependency info



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

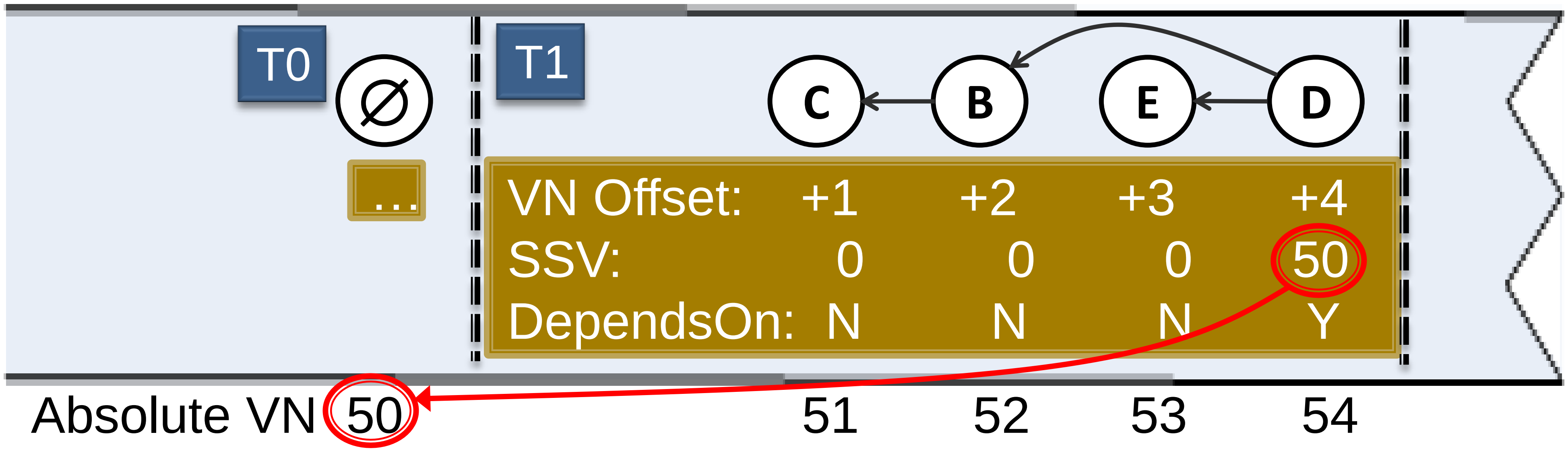
Lazy VN Assignment

- We need to avoid synchronization when assigning VNs
- Stored as offsets from the base location of their intention
- The base location is assigned when the intention is logged
- Given: T0’s root subtree has VN 50
- VN of each subtree S in T1= 50 + S’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*) = Y if T depends on *n* not having changed while T executed

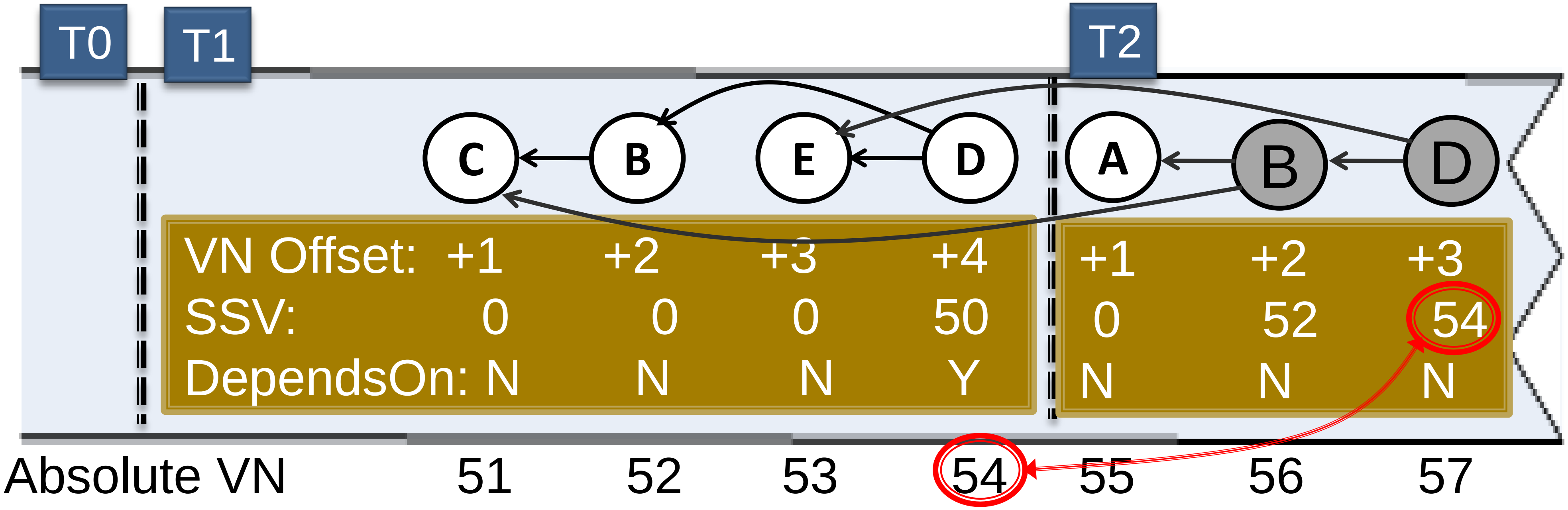


VN Offset:	+1	+2	+3	+4
SSV:	0	0	0	50
DependsOn:	N	N	N	Y

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

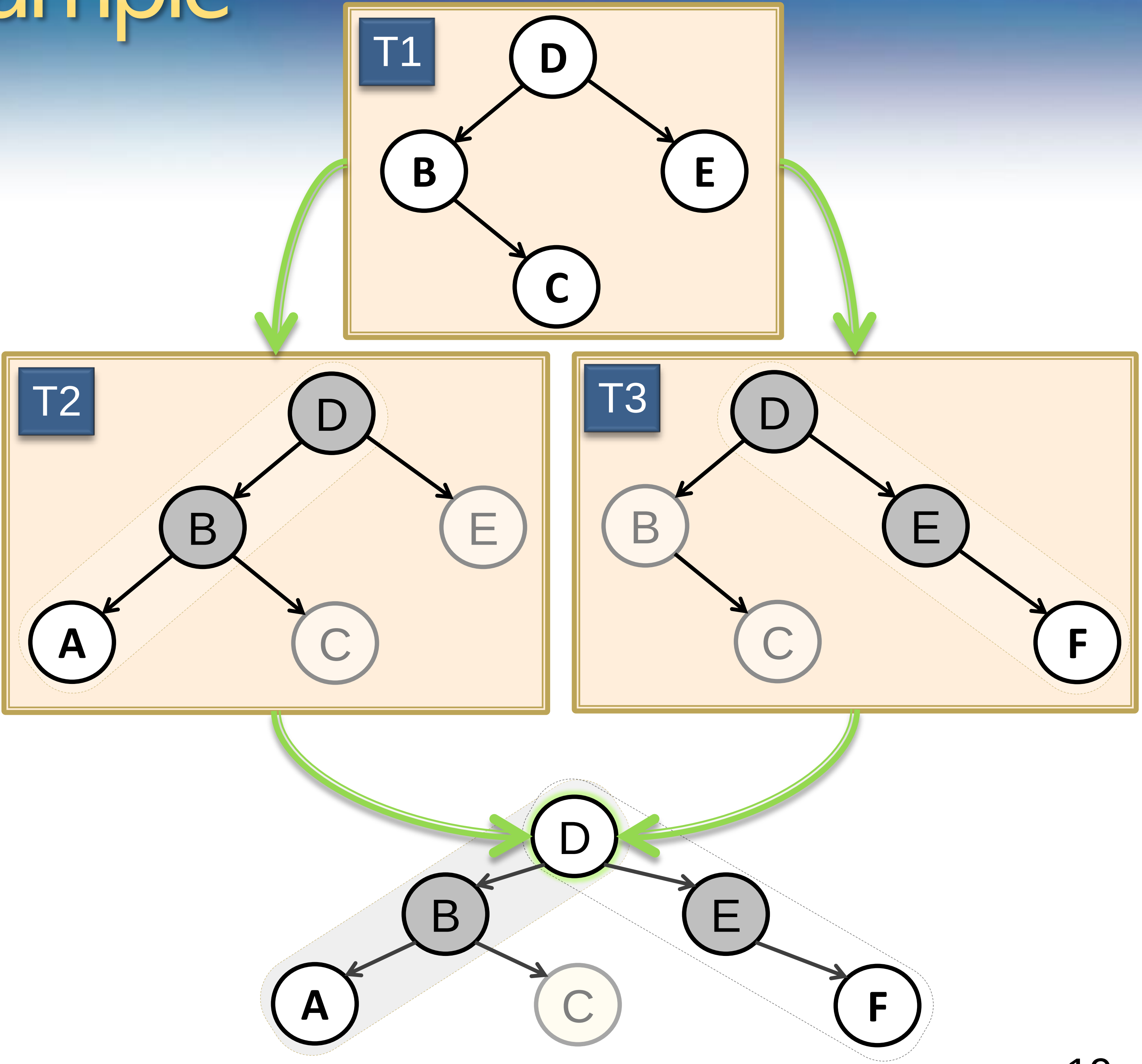
Serial Intentions

- 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.



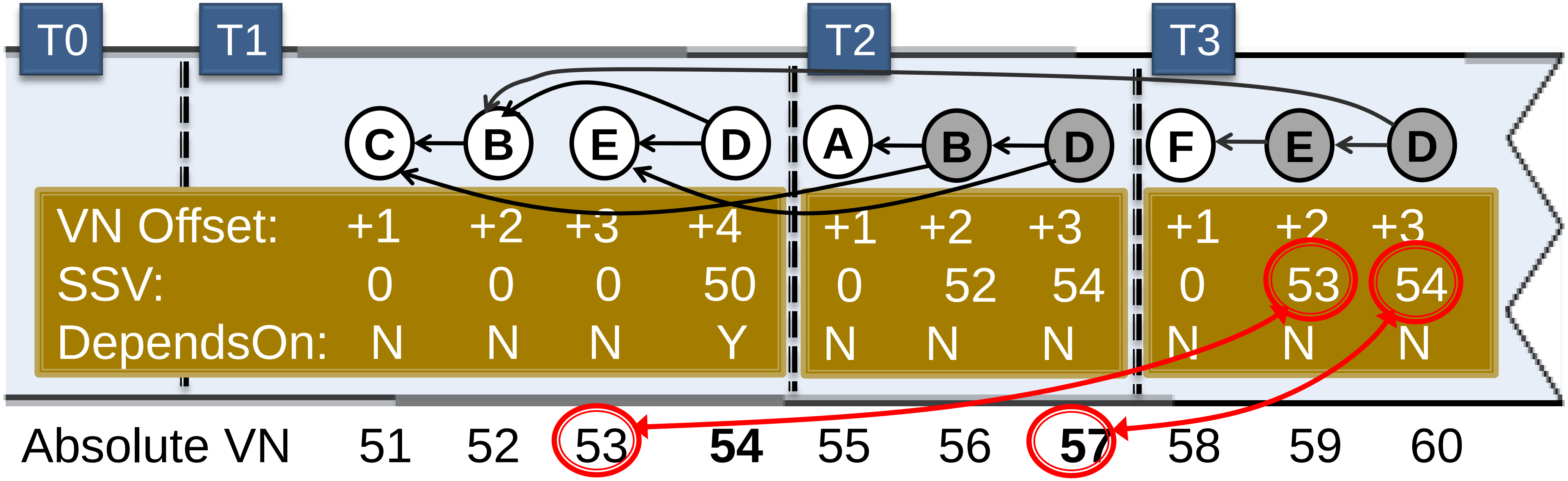
VN Offset:	+1	+2	+3	+4	+1	+2	+3
SSV:	0	0	0	50	0	52	54
DependsOn:	N	N	N	Y	N	N	N
Absolute VN	51	52	53	54	55	56	57

Running Example



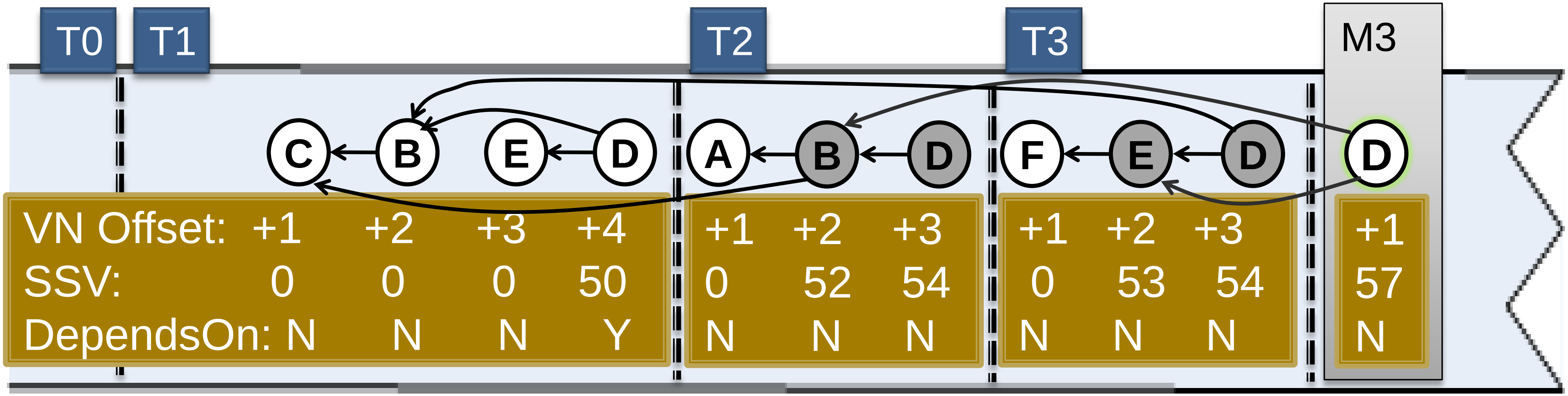
Concurrent (= non-serial) Intentions

- 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
- If there are no conflicts, then since T3 is concurrent, meld creates an **ephemeral intention** to merge T3's state



Ephemeral Intentions

- A committed concurrent intention produces an ephemeral intention (e.g. M3)
  - It's created deterministically in memory on all servers.



Absolute VN	51	52	53	54	55	56	57	58	59	60	61
VN Offset:	+1	+2	+3	+4	+1	+2	+3	+1	+2	+3	+1
SSV:	0	0	0	50	0	52	54	0	53	54	57
DependsOn:	N	N	N	Y	N	N	N	N	N	N	N

- It logically commits immediately after the intention it melds.
- To meld the concurrent intention T3 above, we need to consider metadata only on the root node D.

Garbage Collecting Ephemeral Intentions

- Most are automatically trimmed
- Each committed intention I trims the previous ephemeral intention with either a persisted node (if I is serial) or an ephemeral node (if I is concurrent).
- To track them use an ephemeral flag (or count) on each node that has ephemeral descendants
- Periodically run a flush transaction
  - It copies a set of ephemeral nodes that have no reachable ephemeral nodes
  - It makes the original ephemeral nodes unreachable in the new committed state.
  - It has no dependencies, so it can’t conflict

Other Important Details

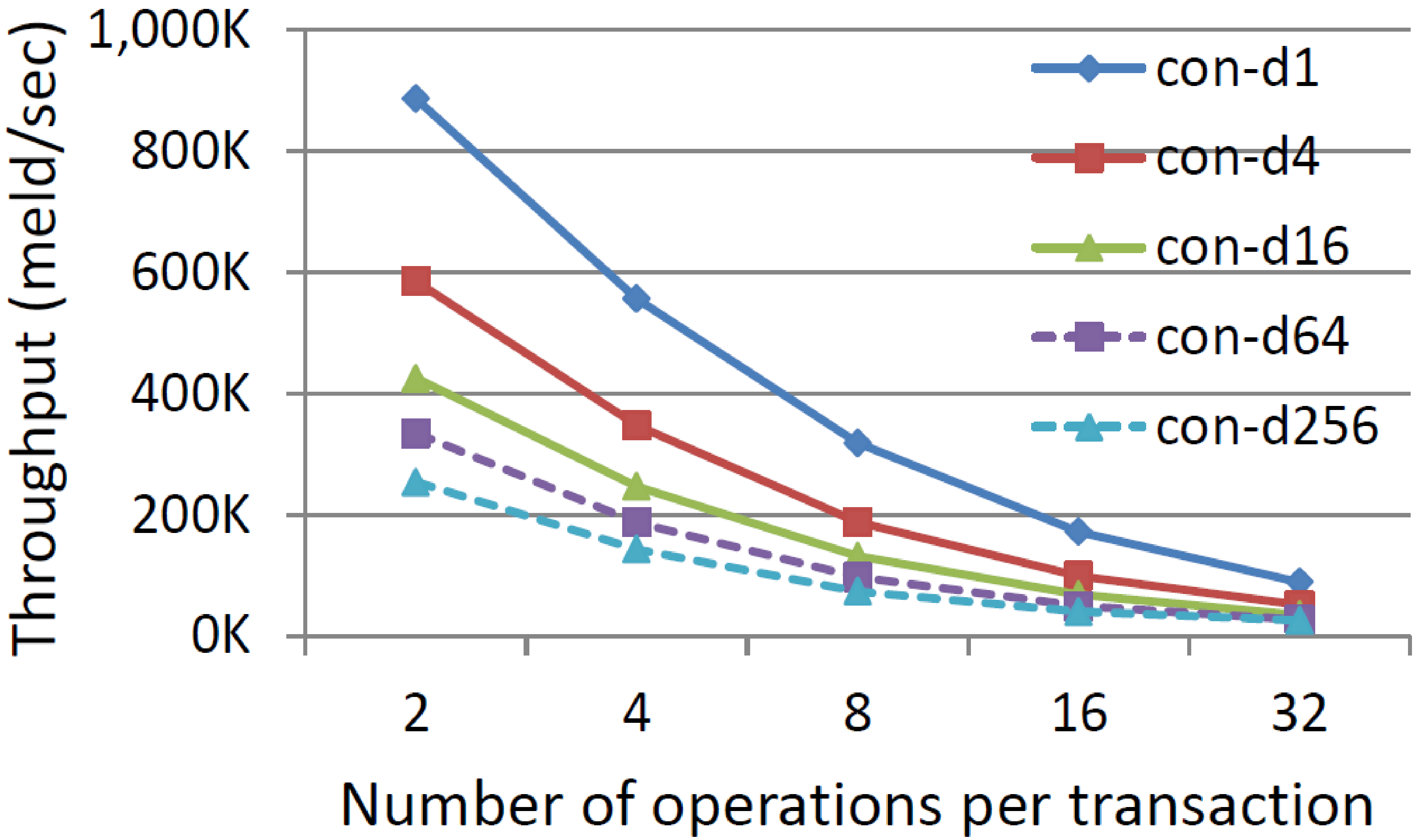
- Phantom avoidance
- Asymmetric meld operations
  - Necessary in common case when subtrees do not align
  - Uses a key-range as a parameter to the top-down recursion
- Deletions
  - Use tombstones in the intention header
- Checkpointing and recovery

Performance

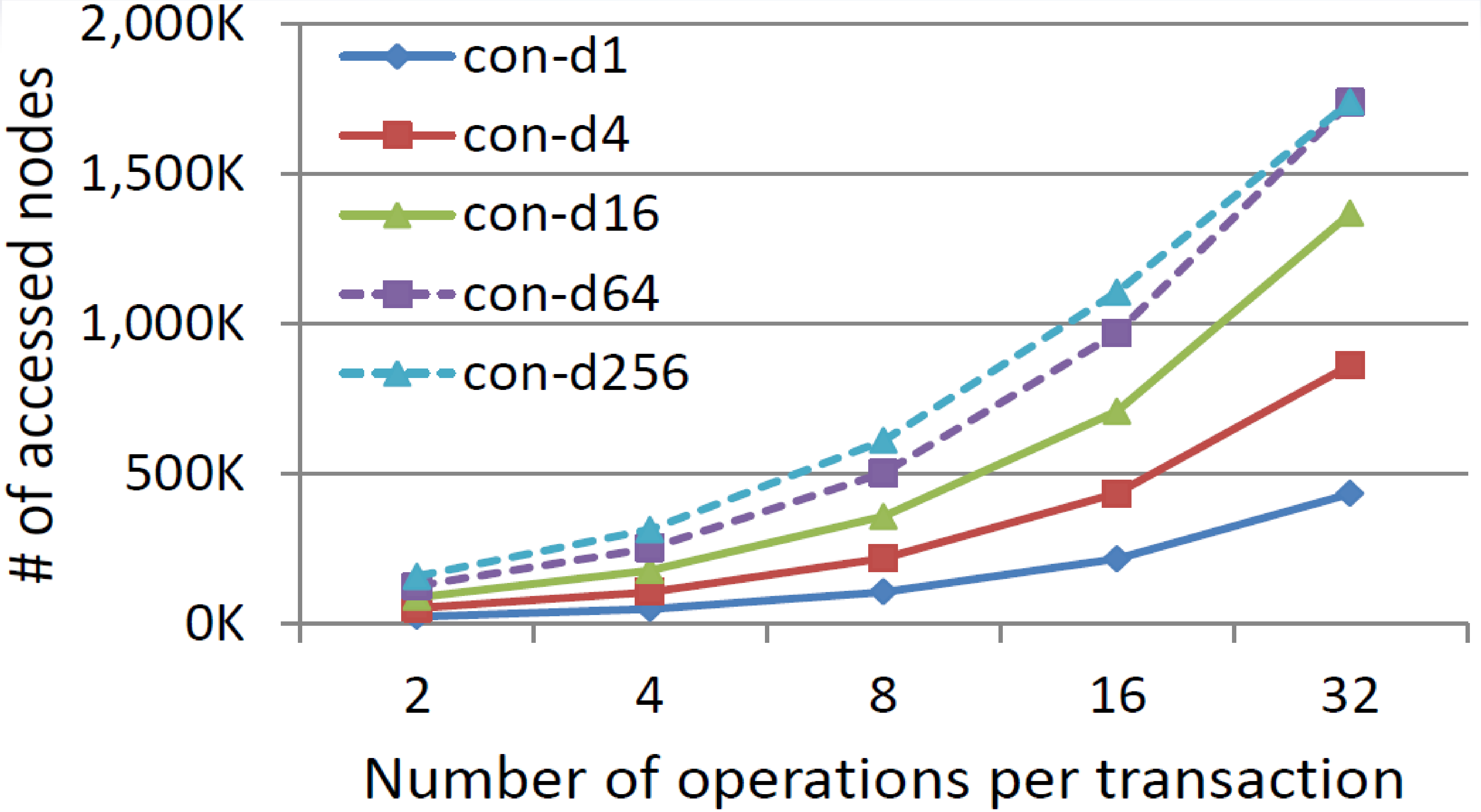
- Focus here is on meld throughput only
  - For latency, see the paper
  - We count committed and aborted transactions
- Experiment setup
  - 128K keys, all in main memory. Keys and payloads are 8 bytes.
  - Serializable isolation, so intentions contain readsets
  - De-serialize intentions on separate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

Throughput

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

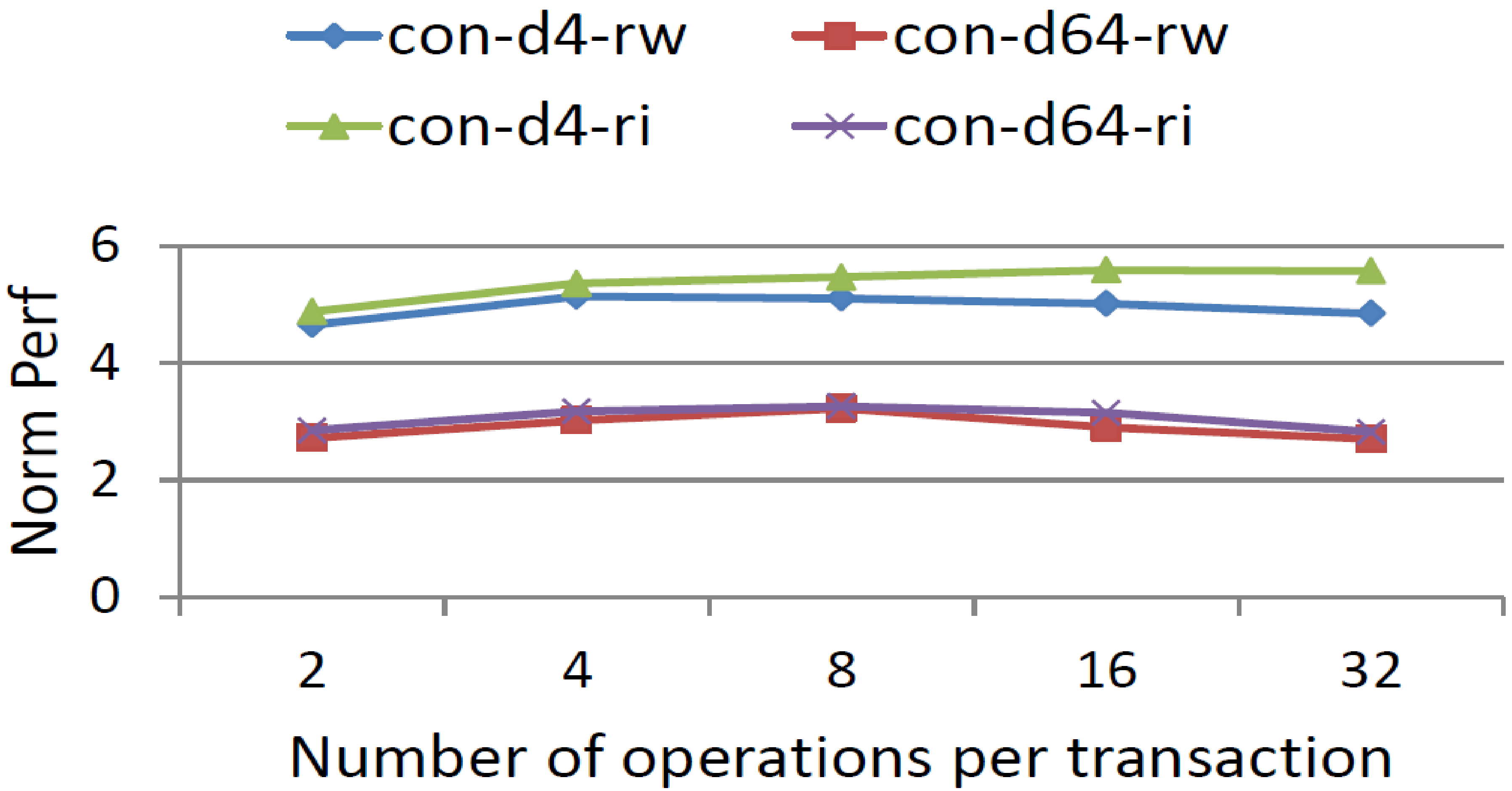


Number of Nodes Accessed



Meld Performance vs. Brute Force

- Brute force = traverse the whole tree



Related Work

- 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

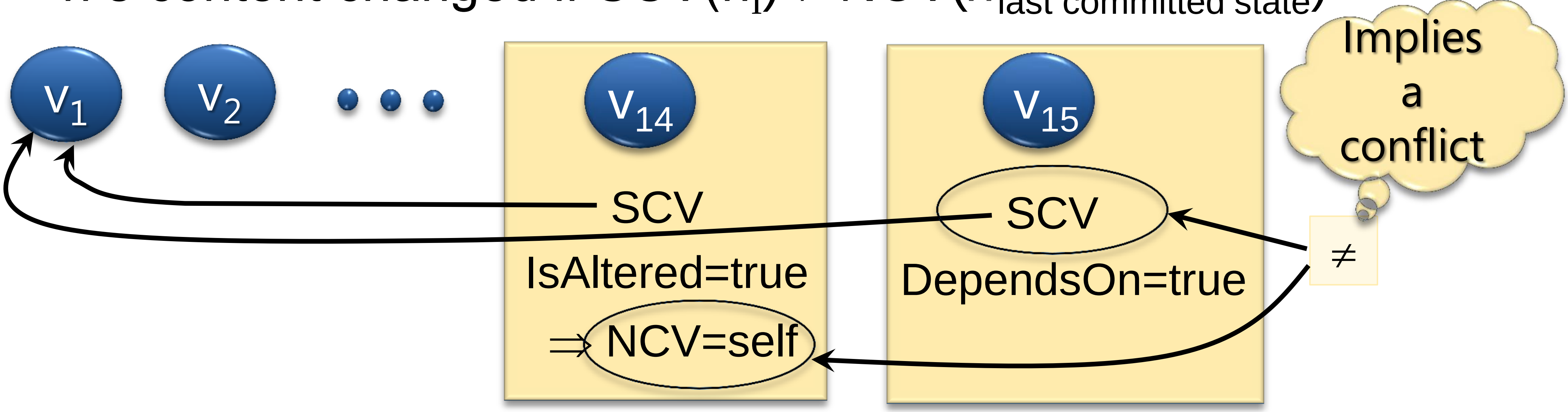
Summary

- 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

Backup Slides

Metadata for Conflict Testing

- Consider node *n* in Intention I
- SCV(*n*) = VN of the node that first generated the payload in n's predecessor
- Altered(*n*) = true if *n*'s payload differs from its predecessor's
- DependsOn(*n*) = true if I depends on *n*'s predecessor's content
- NCV(*n*) = if Altered(*n*) then VN(*n*) else SCV(*n*)
- *n*'s content changed if SCV(n_I) $\neq$ NCV($n_{\text{last committed state}}$)



V_1 V_2 • • •

V_{14}
SCV
IsAltered=true
$\Rightarrow$ NCV=self

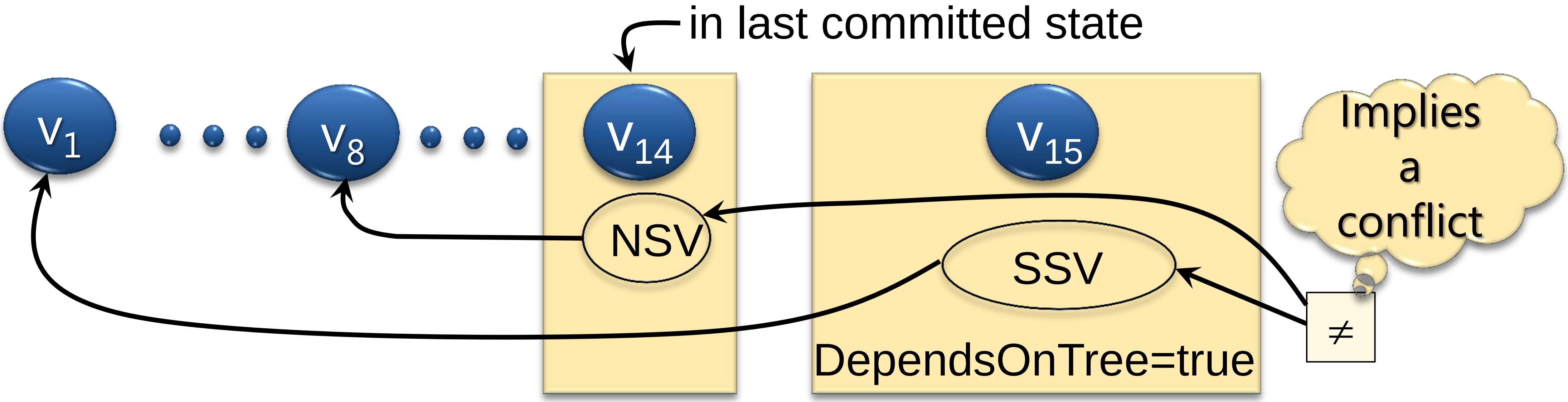
V_{15}
SCV
DependsOn=true

$\neq$

Implies a conflict

Metadata for Detecting Phantoms

- Again consider node n in Intention I
- DependsOnTree(n) = true if I depends on n's subtree not having changed
- NSV(n) = oldest version of n whose subtree is exactly subtree(VN(n))
- DependsOnTree(n) & NSV($n_{\text{last committed state}}$)$\neq$SSV($n$) $\Rightarrow$ a conflict
- Can extend DependsOnTree with DependencyRange of keys



Computing NSV(n)

- SubtreeIsOnlyReadDependent(n) = true iff
  none of n's descendants are updated in I
  (i.e., have Altered = true).
  - Analogous to an intention-to-read lock
  - Avoids traversing entire tree when a descendent of n has DependsOnTree=true and NSV($n_{\text{last committed state}}$) = SSV($n$).
- It also enables computing NSV
  - NSV(n) = (SubtreeIsOnlyReadDependent(n) = true) $\Rightarrow$ SSV(n)
    else VN(n)