In-Memory Database

Xueqing Gong

xqgong@sei.ecnu.edu.cn

華東師範大學
EAST CHINA NORMAL UNIVERSITY

Background

- Much of the history of DBMSs is about dealing with the limitations of hardware.
- Hardware was much different when the original DBMSs were designed:
  - Uniprocessor (single-core CPU)

So why not just use a "traditional" disk-oriented DBMS with a really large cache?

memory.

  - Structured data sets are smaller (e.g., tables with schema).
  - Unstructured or semi-structured data sets are larger (e.g., videos, log files).

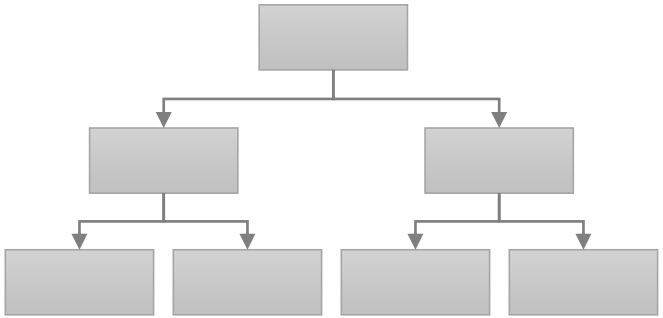
Disk-Oriented DBMS

- The primary storage location of the database is on non-volatile storage (e.g., HDD, SSD).
  - The database is stored in a **file** as a collection of fixed length blocks called **slotted pages** on disk.
- The system uses an in-memory **buffer pool** to cache pages fetched from disk.
  - Its job is to manage the movement of those pages back and forth between disk and memory.

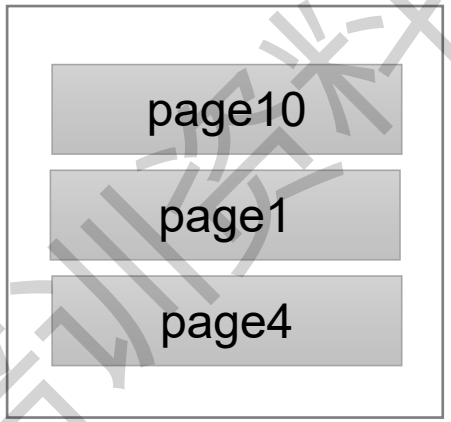
Buffer Pool

- When a query accesses a page, the DBMS checks to see if that page is already in memory:
  - If it's not, then the DBMS must retrieve it from disk and copy it into a **frame** in its buffer pool.
  - If there are no free frames, then find a page to evict guided by the **page replacement policy**.
  - If the page being evicted is dirty, then the DBMS must write it back to disk.
- Once the page is in memory, the DBMS **translates** any on-disk addresses to their in-memory addresses.

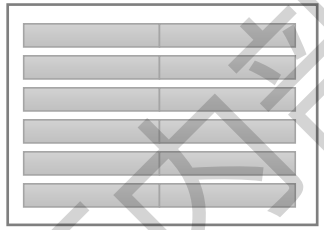
Disk-Oriented Data Organization



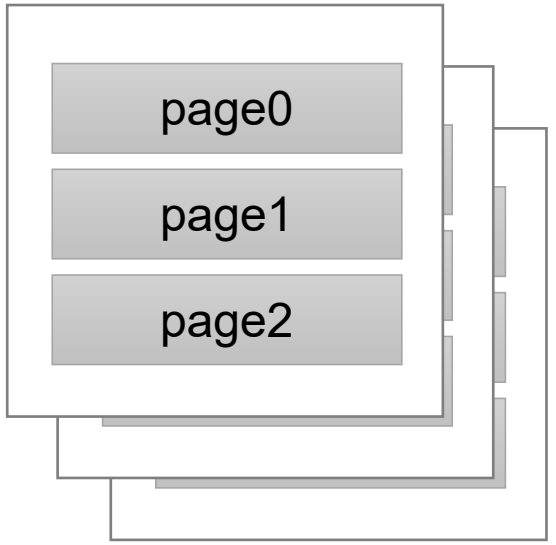
Index



Buffer Pool



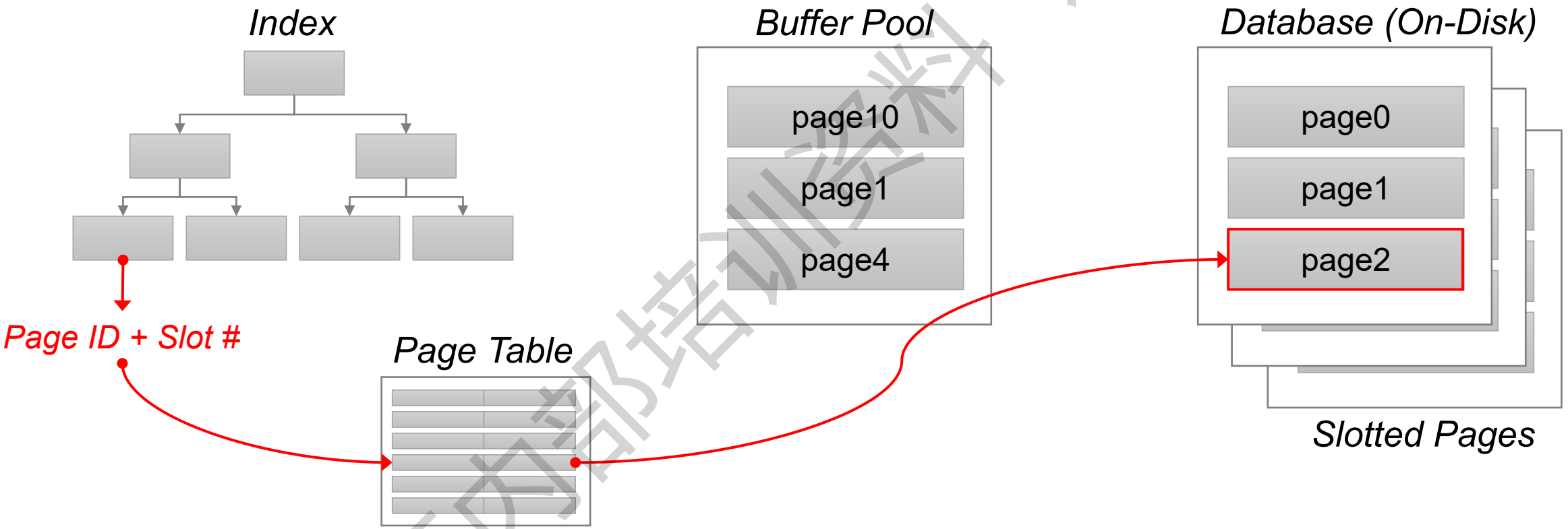
Page Table



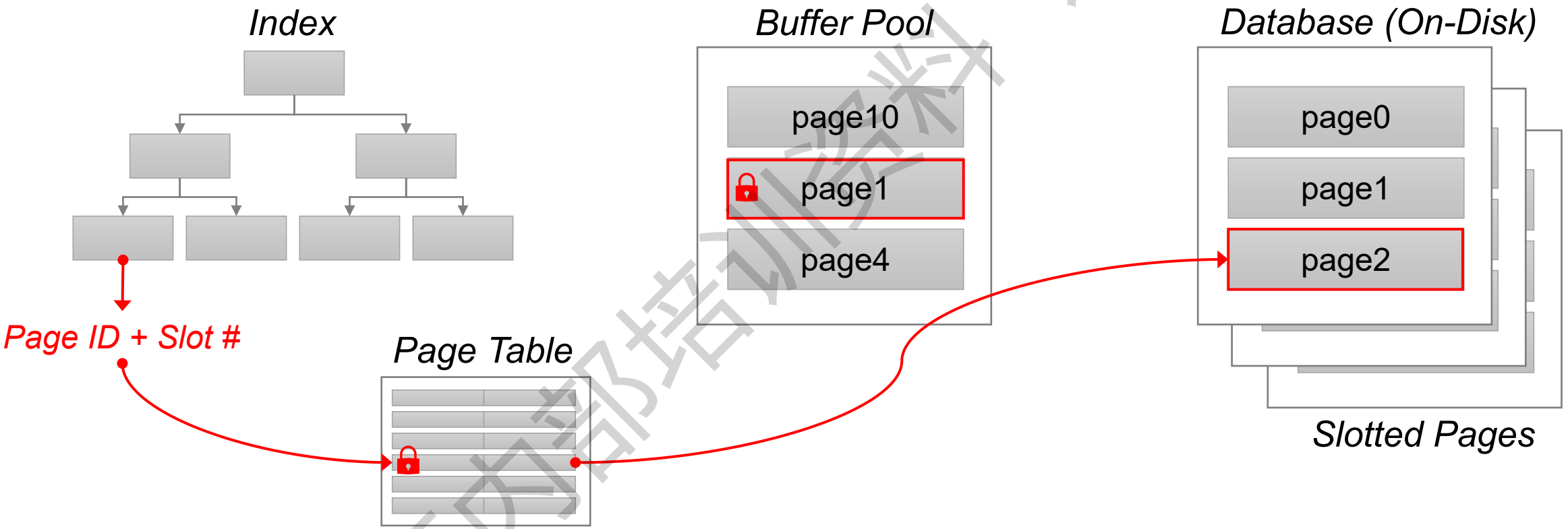
Database (On-Disk)

Slotted Pages

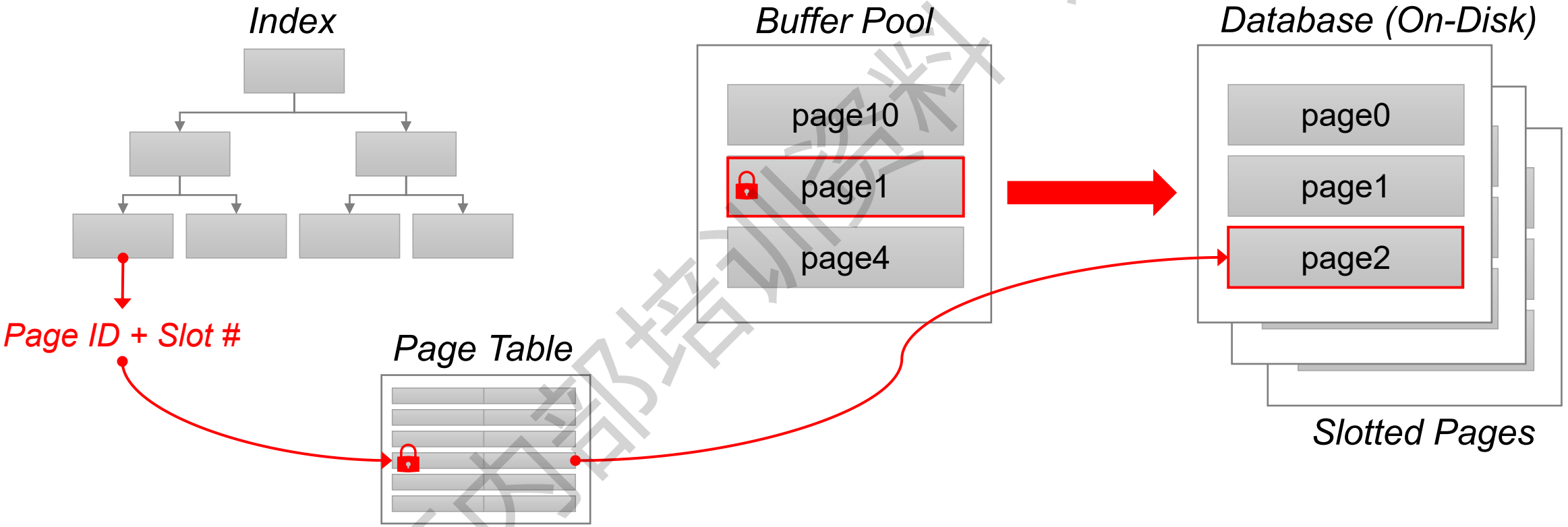
Disk-Oriented Data Organization



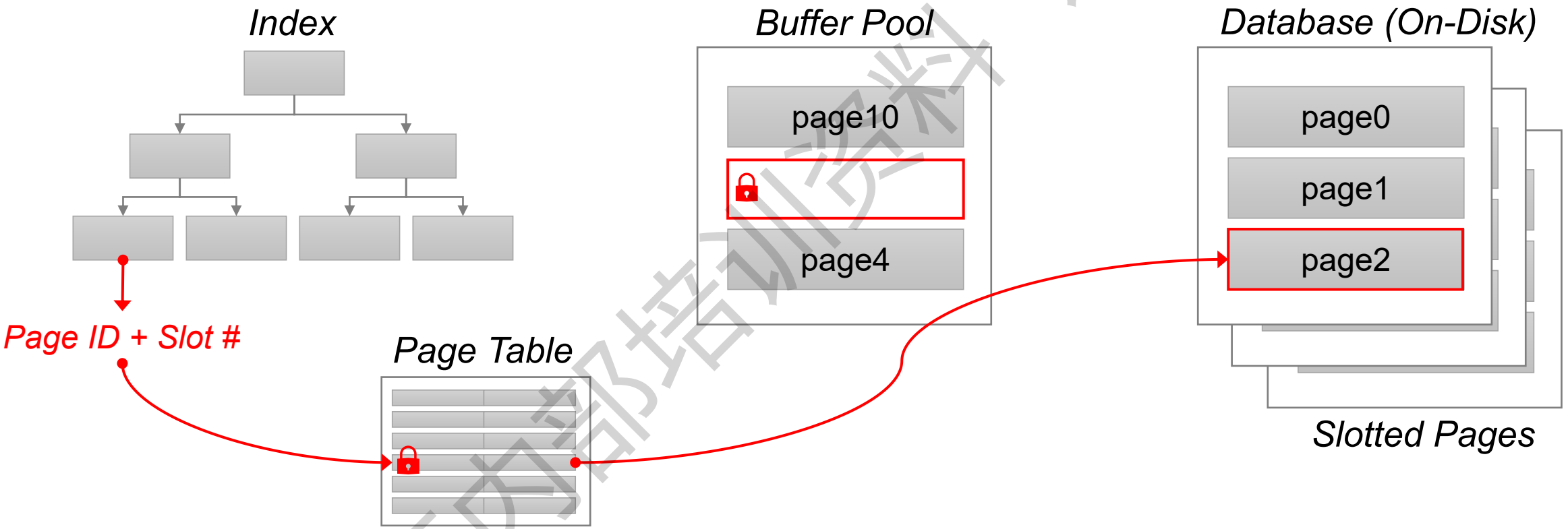
Disk-Oriented Data Organization



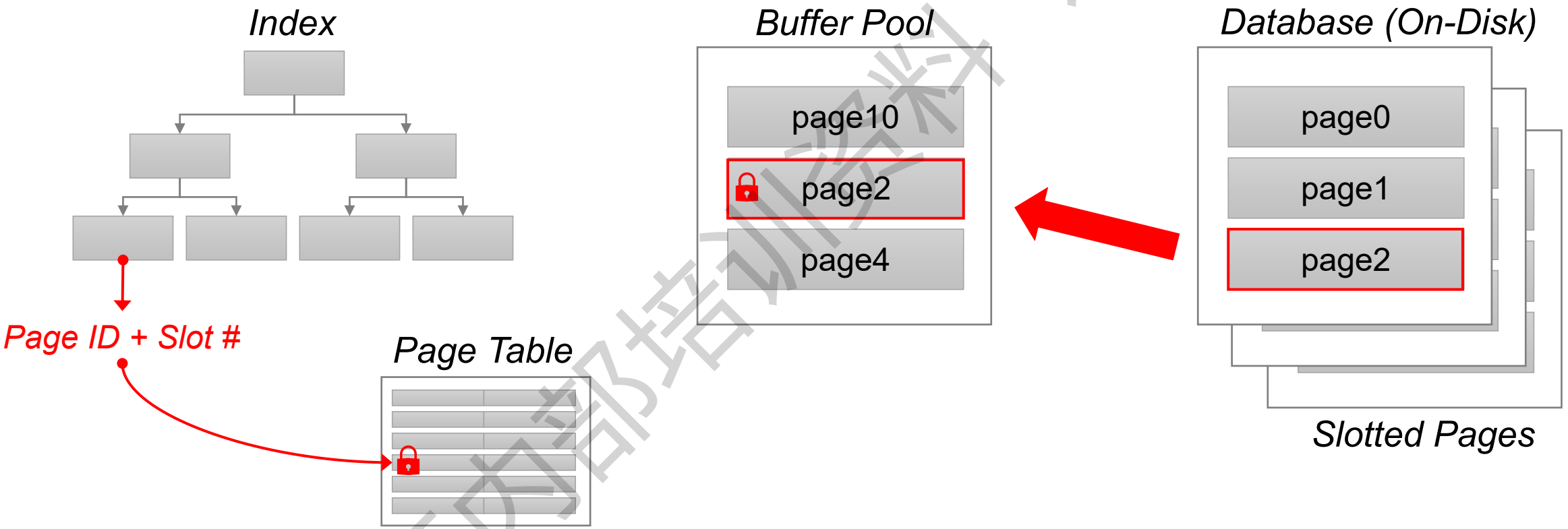
Disk-Oriented Data Organization



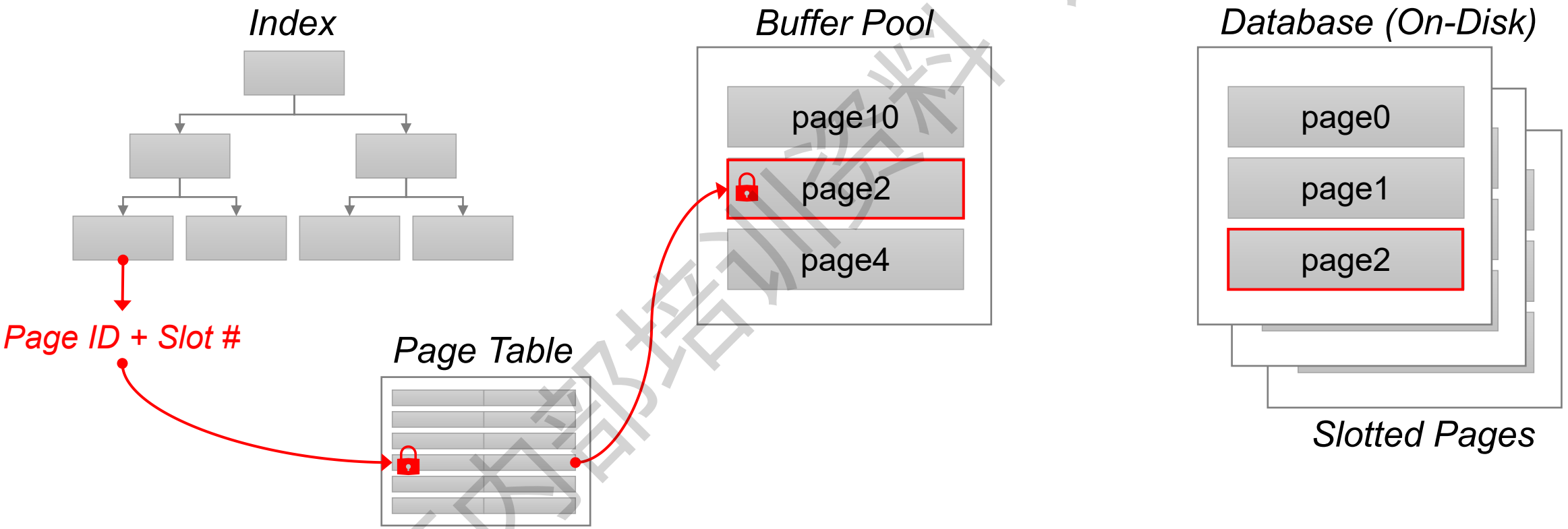
Disk-Oriented Data Organization



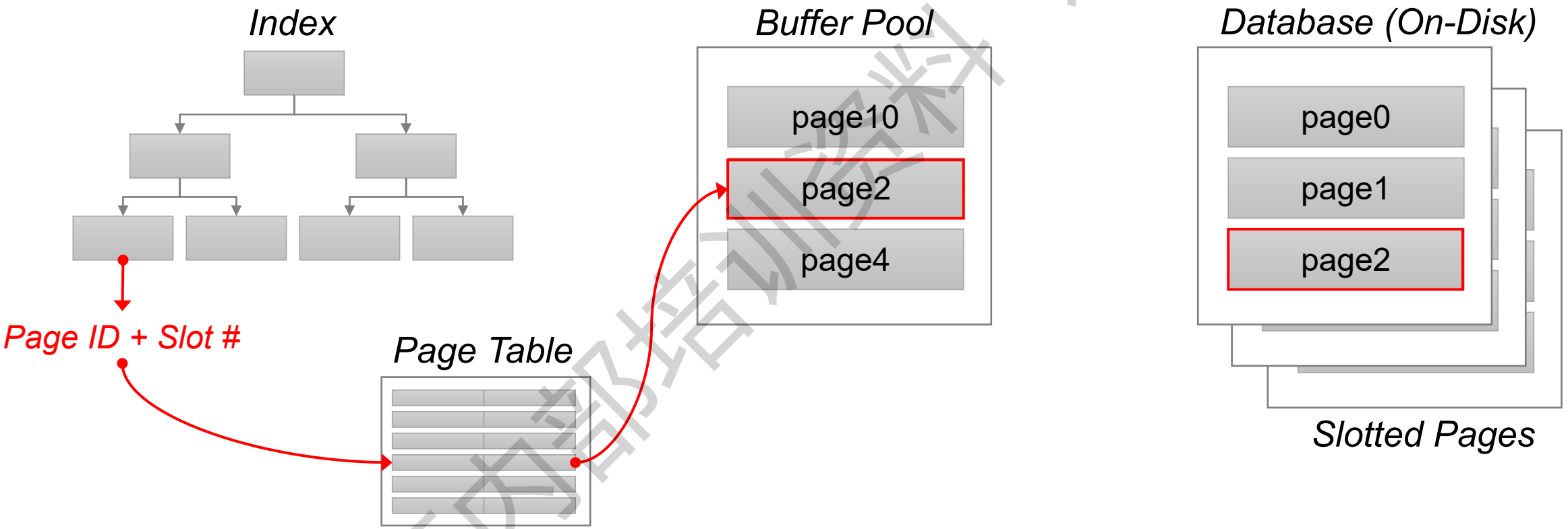
Disk-Oriented Data Organization



Disk-Oriented Data Organization



Disk-Oriented Data Organization



Buffer Pool

- Every tuple access goes through the **buffer pool manager** regardless of whether that data will always be in memory.
  - Always **translate** a tuple's record id to its memory location.
  - Worker thread must **pin** pages to make sure that they are not swapped to disk.

Concurrency Control

- The systems assumes that a txn could **stall** at any time whenever it tries to access data that is not in memory.
- Execute other txns at the same time so that if one txn stalls then others can keep running.
  - Set locks to provide **ACID** guarantees for txns.
  - Locks are stored in a separate data structure (**lock table**) to avoid being swapped to disk.

Locks vs. Latches

- Locks
  - Protects the database's logical contents (e.g., tuple, table) from other txns.
  - Held for txn duration.
  - Need to be able to rollback changes.
- Latches
  - Protects the DBMS's internal physical data structures (e.g., page table) from other threads.
  - Held for operation duration.
  - Do not need to be able to rollback changes.

A survey of B-tree locking techniques.
ACM Trans. Database Syst. 35, 3, Article 16 (July 2010).

Locks vs. Latches

	Locks	*Latches*
Separate …	User transactions	Threads
Protect …	Database contents	In-memory data structures
During …	Entire transactions	Critical sections
Modes …	Shared, exclusive, update, intention, escrow, schema, etc.	Read, writes, (perhaps) update
Deadlock …	Detection & resolution	Avoidance
… by …	Analysis of the waits-for graph, timeout, transaction abort, partial rollback, lock de-escalation	Coding discipline, "lock leveling"
Kept in …	Lock manager's hash table	Protected data structure

Fig. 2. Locks and latches.

Logging & Recovery

- Most DBMSs use **STEAL** + **NO-FORCE** buffer pool policies.
  - STEAL: DBMS can flush pages dirtied by uncommitted transactions to disk.
  - NO-FORCE: DBMS is not required to flush all pages dirtied by committed transactions to disk.
  - So all page modifications have to be flushed to the **write-ahead log (WAL)** before a txn can commit.
  - Each log entry contains the before and after images of modified tuples.
  - Lots of work to keep track of log sequence numbers (LSNs) all throughout the DBMS.

Disk-Oriented DBMS Overhead

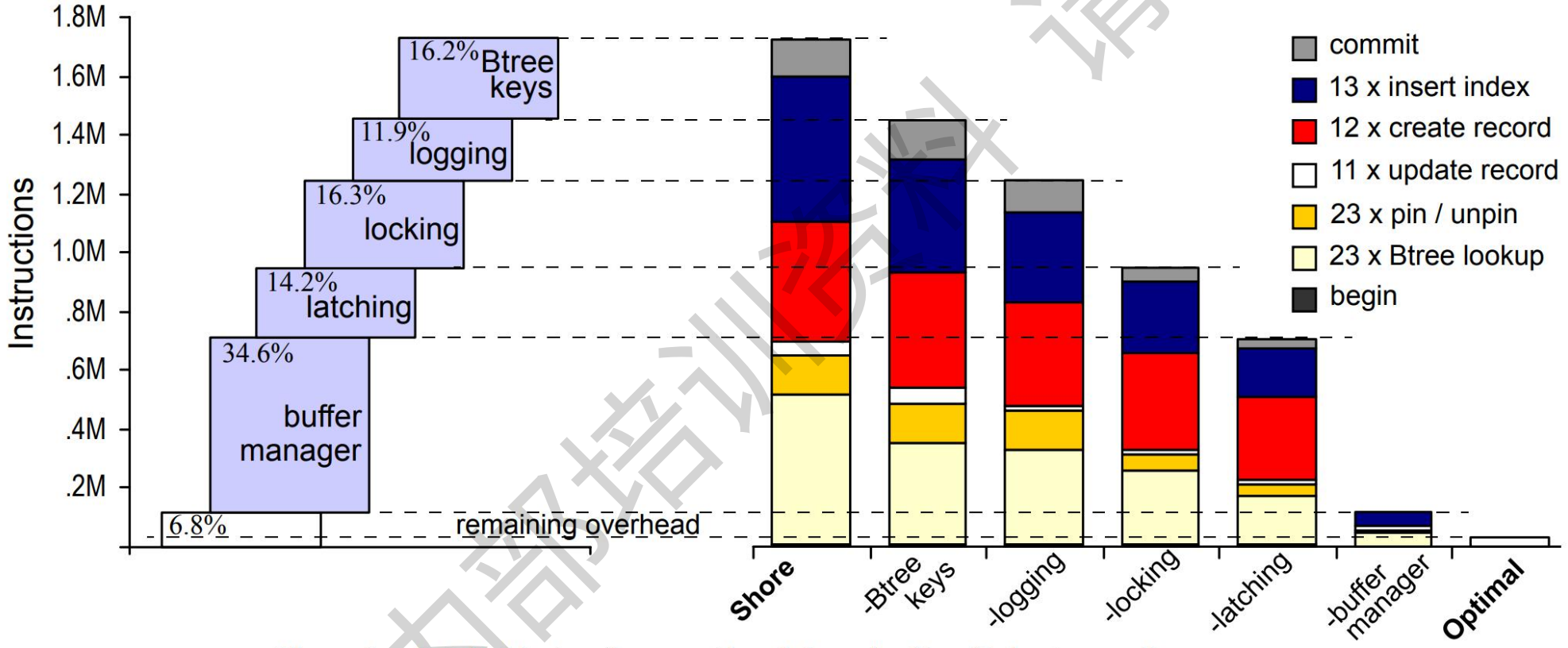


Figure 6. Detailed instruction count breakdown for New Order transaction.

OLTP through the looking glass, and what we found there.
SIGMOD '08, 981–992.

In-Memory DBMS

- Disk-oriented DBMSs do **a lot of extra stuff** because they are predicated on the assumption that data has to reside on disk
- In-memory DBMSs **maximize performance** by optimizing these protocols and algorithms
  - Assume that the **primary storage** location of the database is **permanently** in memory.
- Early ideas proposed in the 1980s but it is now feasible because DRAM prices are low and capacities are high.
- First commercial in-memory DBMSs were released in the 1990s.
  - Examples: TimesTen, DataBlitz, Altibase

Historical Overview

Summary of Previous Research in In-Memory DBMSs

The Early Years: 1984 ~ 1994

- Assume buffer pool fits in memory
  - Group commit/fast commit optimizations from University of Wisconsin
  - IMS Fastpath memory-resident optimizations
- Direct memory access to records
- Main-memory optimized indexing methods (T-Trees)
- Durability and recovery
  - Functional partitioning of engine into runtime processor and recovery processor
  - Redo-only logging: avoid space overhead for undo bytes
- Partitioned main-memory databases (PRISMA)
- Concurrency
  - Course-grained locking

The Early Years: 1984 ~ 1994

- 1976: IMS Fastpath [1] (published 1985) main-memory resident database optimizations
  - Fine-grained record-level locking
  - Install record updates at commit time
  - Group commit
- 1984: DeWitt, Katz, Olken, Shapiro, Stonebraker, Wood [2]
  - Assume database fit in main-memory buffer pool
  - Access method optimizations, join techniques, group/fast commit, parallel log and checkpoint I/O
- 1986: MM-DBMS from University of Wisconsin [3-5]
  - Use of pointers for direct record access (no buffer pool)
  - T-Trees: memory optimized indexing
  - Partitioned recovery method: main processor and recovery processor

The Early Years: 1984 ~ 1994

- 1987: MARS from SMU and System M from Princeton [6, 7]
  - Partitioned into database processor and recovery processor: lazy copy of volatile updates to storage
  - Showed benefits of avoiding undo logging in main-memory systems
- 1987: IBM Office by Example [8]
  - Main-memory optimizations for read-mostly data: inverted indexes using memory pointers
- 1988: TPK from University of Wisconsin [9]
  - Executed transactions serially using collection of specialized threads (input, output, execution, recovery)
- 1988: PRISMA [10, 11]
  - Two-phase locking and two-phase commit over partitioned main-memory database
- 1991: IBM Starburst Memory Resident Storage Component [12]
  - Used many techniques from MM-DBMS
  - Concurrency: single latch to protect table and indexes (serial execution), direct addressing of lock data

New Millennium: 1994 ~ 2005

- Commercial systems targeting specialized workloads (e.g., telecom)
  - Dali from Bell Labs, later DataBlitz
  - ClustRa
  - HP Smallbase, later TimesTen
- Multi-core optimizations
  - P*-Time
  - Lock-free implementation techniques

New Millennium: 1994 ~ 2005

- 1994: Dali from Bell Labs [13, 14] (later DataBlitz[15])
  - Gave applications direct shared access to main-memory records
  - Data replication (fast failover), redo-only logging (reduce I/O), fuzzy action-consistent checkpoint scheme (reduce lock contention)
- 1995: ClustRa Distributed Main-Memory DBMS [16]
  - High performance and availability using fully replicated main-memory DBMS at each node (2-safe)
  - Targeted telecom workloads
- 1997: System K from NYU[17]
  - In-memory partitions assigned to CPU cores (no distribution across machines)
  - Transactions run serial within partition with logical logging

New Millennium: 1994 ~ 2005

- 1999: HP Smallbase and TimesTen [18-20]
  - Unique architectural design: used as both server and linkable library
  - Fully ACID-compliant, with ability to relax for performance and use elsewhere (e.g., mid-tier cache)
  - Used T-Trees initially, multi-versioning, along with other main-memory optimizations
- 2004: P*Time[21]
  - Light-weight OLTP database optimized for CPU cache consciousness
  - Use of lock-free index read protocol (OLFIT)
  - Fine-grained parallel logging
  - Eventually became the main-memory OLTP engine in SAP HANA

Previous Techniques

- Direct memory pointers vs buffer pool indirection
  - Modern systems avoid page based indirection for performance reasons
- Data partitioning
  - Some modern systems like H-Store/VoltDB choose to partition the database (e.g., across cores, machines)
- Lock-free (as much as possible), cache-conscious data structures
- Coarse-grain locking
  - Possibly OK on early systems due to few cores
  - Not used today due to bottlenecks with raw parallelism on modern machines
- Functional partitioning
  - Functional handoff between threads (input, output, recovery duties, durability) is not done in (most)modern systems

Modern Hardware Environment

- RAM prices have made a **fall** since the previous era of main-memory database systems
  - Servers can be configured with up to 12TB of RAM
  - Accommodate most OLTP databases
- Multi-core CPUs
  - Clock speeds have stopped increasing; each generation of CPU increases parallelism on a chip
  - Intel Xeon E5-2699 v3 supplies 18 cores (36 hardware threads)
  - AMD's upcoming Zen chip is rumored to go up to 32 cores
- Multi-socket machines
  - Multiple multi-core CPUs within a single machine adds even more parallelism
  - These machines display NUMA behavior
    - Separate memory/cache for each processor
    - Processor can access memory from another processor through interconnect (e.g., Intel QPI)
    - Memory access is non-uniform: accessing local memory is faster than accessing remote memory

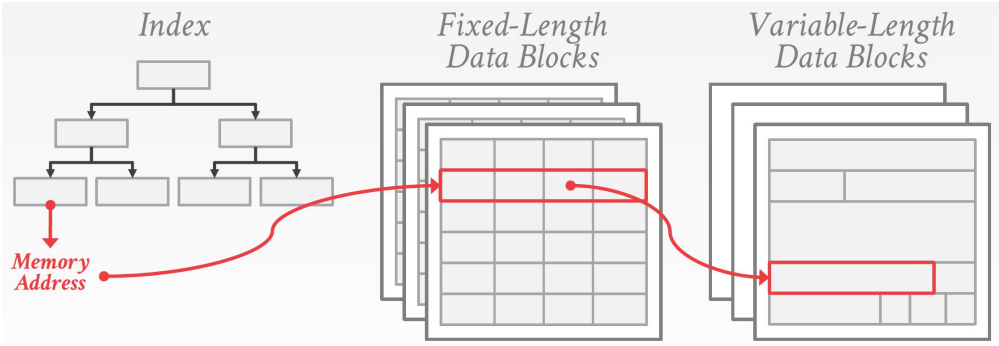
Main-Memory Optimizations

- Remove overheads of
  - Buffer pool
  - Lock manager
- Making progress in eliminating runtime recovery overhead
  - Records live in memory, but still need to log to Disk/SSD
  - NVRAM will help greatly
- Aggressive compilation
  - All transactions/queries compiled to byte code; no interpretation
- Scalable high-performance indexing methods
  - Latch-freedom coupled with memory-optimized layout

Data Organization

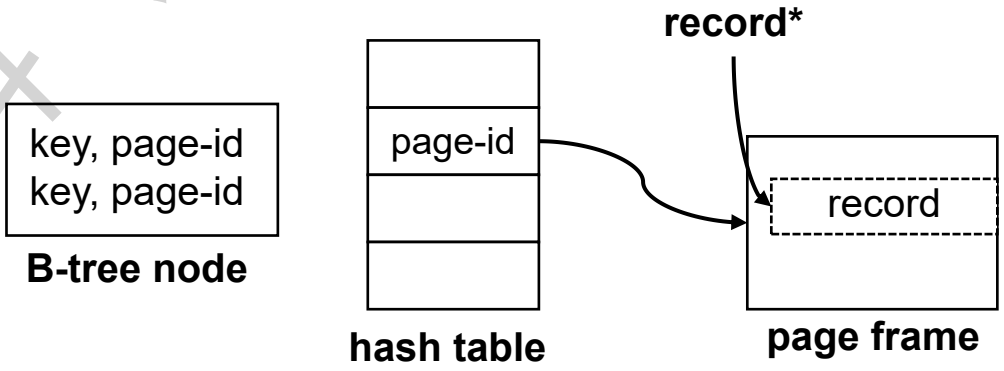
In-Memory Data Organization

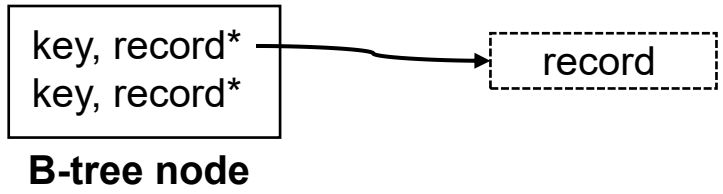
- An in-memory DBMS does not need to store the database in slotted pages but it will still organize tuples in blocks/pages
  - Direct memory pointers vs. tuple identifiers
  - Separate pools for fixed-length and variable-length data
- Can result in order of magnitude performance improvements
  - Avoids page-based indirection to resolve record pointer
  - Avoids page latch overhead



Avoiding Page Indirection

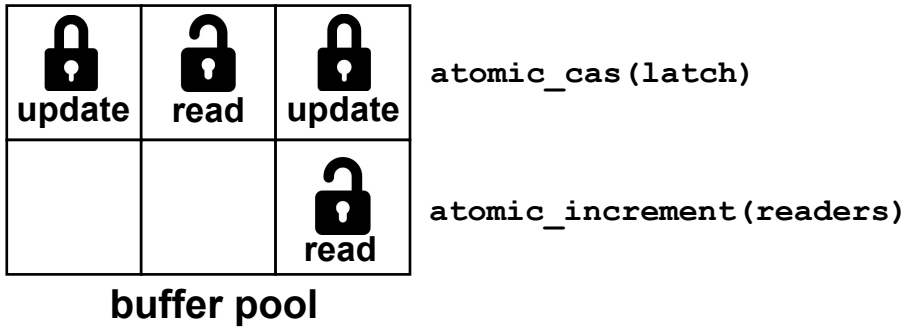
- Classic relational architectures **require two layers of indirection** to resolve pointer to record
  - Hash table access to resolve page frame in buffer pool
  - Calculate pointer to record using offset within page
- No indirection in In-Memory Systems
  - Internal structures like indexes store **memory pointers**
  - Magnitude performance improvement by removing buffer pool





Avoiding Page Latching

- Accessing a page in a classical database design requires latch on page frame
  - Readers: shared latch
  - Writers: update latch (possibly exclusive)



- Latching leads to overhead
  - Internal manipulation of latch requires **atomic operation**
  - Shared read latch: update of reference count
  - Update latch: variable to mark exclusion
  - Commercial systems have several latch types, so thread could end up acquiring several latches!

Organization Choices

- Partitioning
  - **Partitioned systems**: Disjoint partitioning of the database, assign partition to node, core, etc.
    - Simple per-node implementation (serial execution, no concurrent data structures)
    - Partition management burden: rebalance for hotspots
  - **Non-partitioned systems**: Any thread/core can access any record in the database
    - Increased implementation complexity (e.g., concurrent data structures)
- Multi-Versioning
  - Allows for high concurrency; important in multi-core environments
  - Readers allow to execute uncontested, do not block writers
- Row/Columnar Layout
  - Reasonable OLTP performance on in-memory columnar layout;
  - not true for disk-based column stores

Organization Choices

	Partitioned	Multi-Versioned	Row/Columnar
Hekaton	No	Yes	Row
HyPer	No	Yes	Hybrid
SAP HANA	No	Yes	Hybrid
H-Store/VoltDB	Yes	No	Row

Hekaton

- Non-partitioned system
  - Any thread can access any record
  - Entirely **lock-free** engine implementation for thread-safety
- Multi-versioned
  - Records have begin and end timestamps
  - Timestamps define record visibility for concurrency control
- Row-oriented
  - Records in memory with **no clustering**, etc.
  - Up to 8 hash or range indexes can be built over records
  - Records contain a number of pointer link fields to manager overflow/duplicate chains within indexes

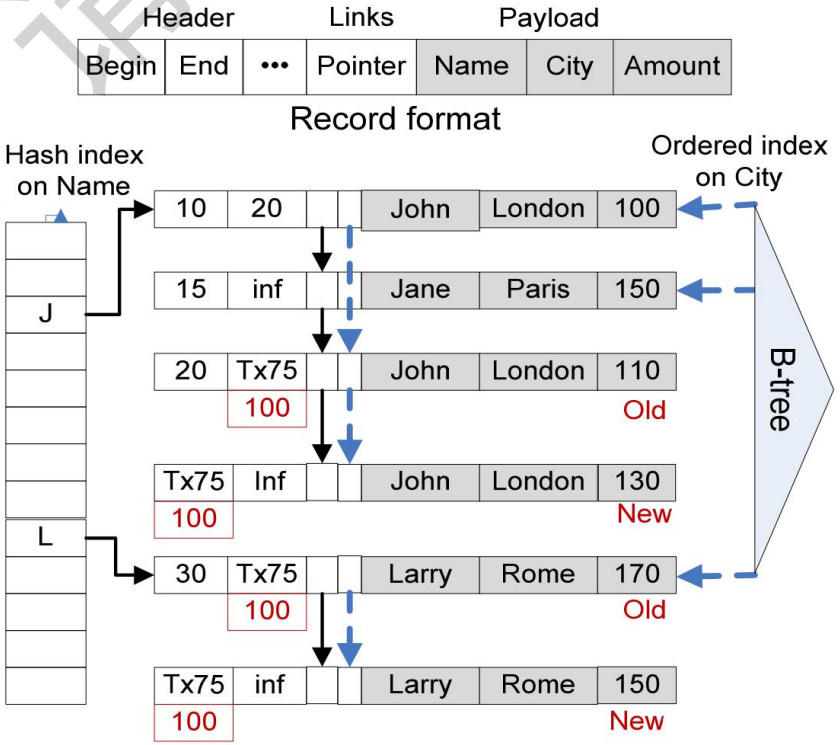


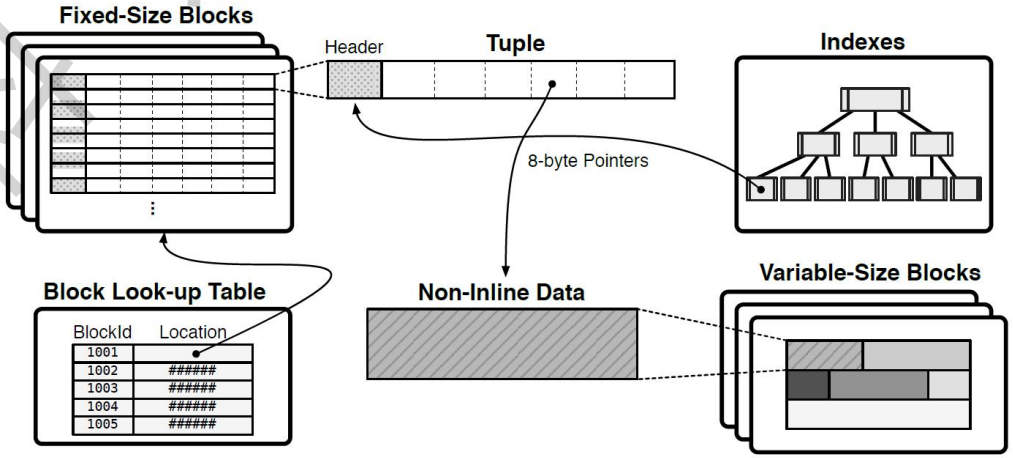
Figure 2: Example account table with two indexes. Transaction 75 has transferred $20 from Larry's account to John's account but has not yet committed.

Hekaton: SQL server's memory-optimized OLTP engine. SIGMOD '13, 1243–1254.

H-Store/VoltDB

https://hstore.cs.brown.edu/

- Partitioned system
  - Distributes data across compute nodes (or cores)
  - Serial execution at partitions: avoids concurrency control
- Two-tiered architecture
  - Transaction coordinator
  - Execution engine: data storage, indexing, and execution.
- Single-version Row Store
  - Execution engines maintain single version of records
  - Storage divided into pools for **fixed-size** and **variable-size** blocks, with fixed-pool as primary storage
  - All tuples are fixed size (per table) to ensure they remain **byte-aligned**
  - Fields larger than 8-bytes stored as variable-length blocks, all other fields stored inline in tuple
  - Lookup table translates block id (4 bytes) to physical location (8 bytes), allows engine to address blocks using 4 bytes.

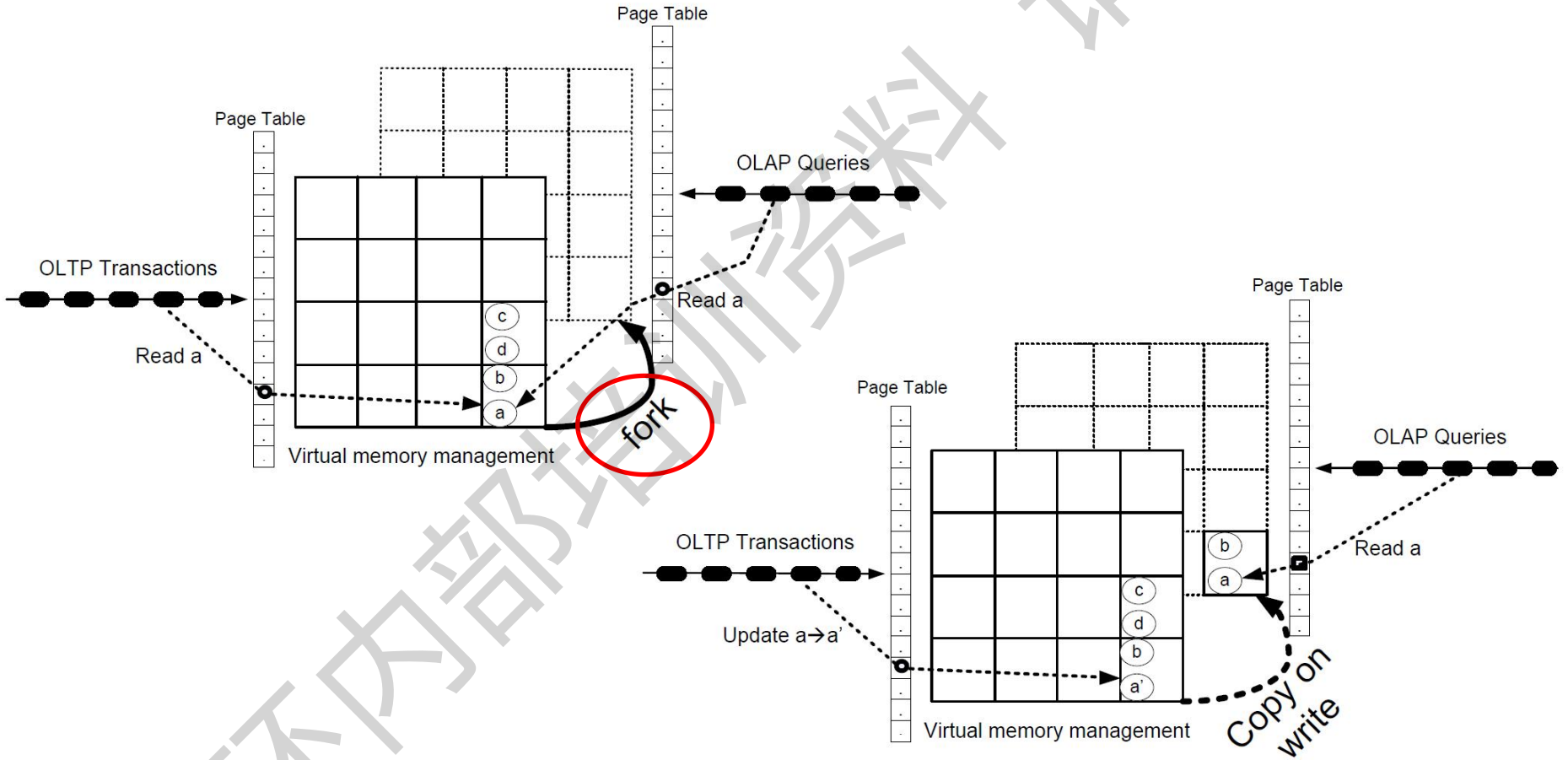


HyPer

- Not partitioned
  - Any transaction can touch any record
- Versioned
  - OLAP queries run over virtual memory snapshots
  - Support transient per-record undo buffers to support MVCC
- Hybrid row/column layout
  - hybrid record layout that clustered frequently-accessed columns together
  - Most commonly configured as a column store in most experiments

Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. SIGMOD '15, 677–689.

Hyper Copy-on-Write Mechanism



Hyper

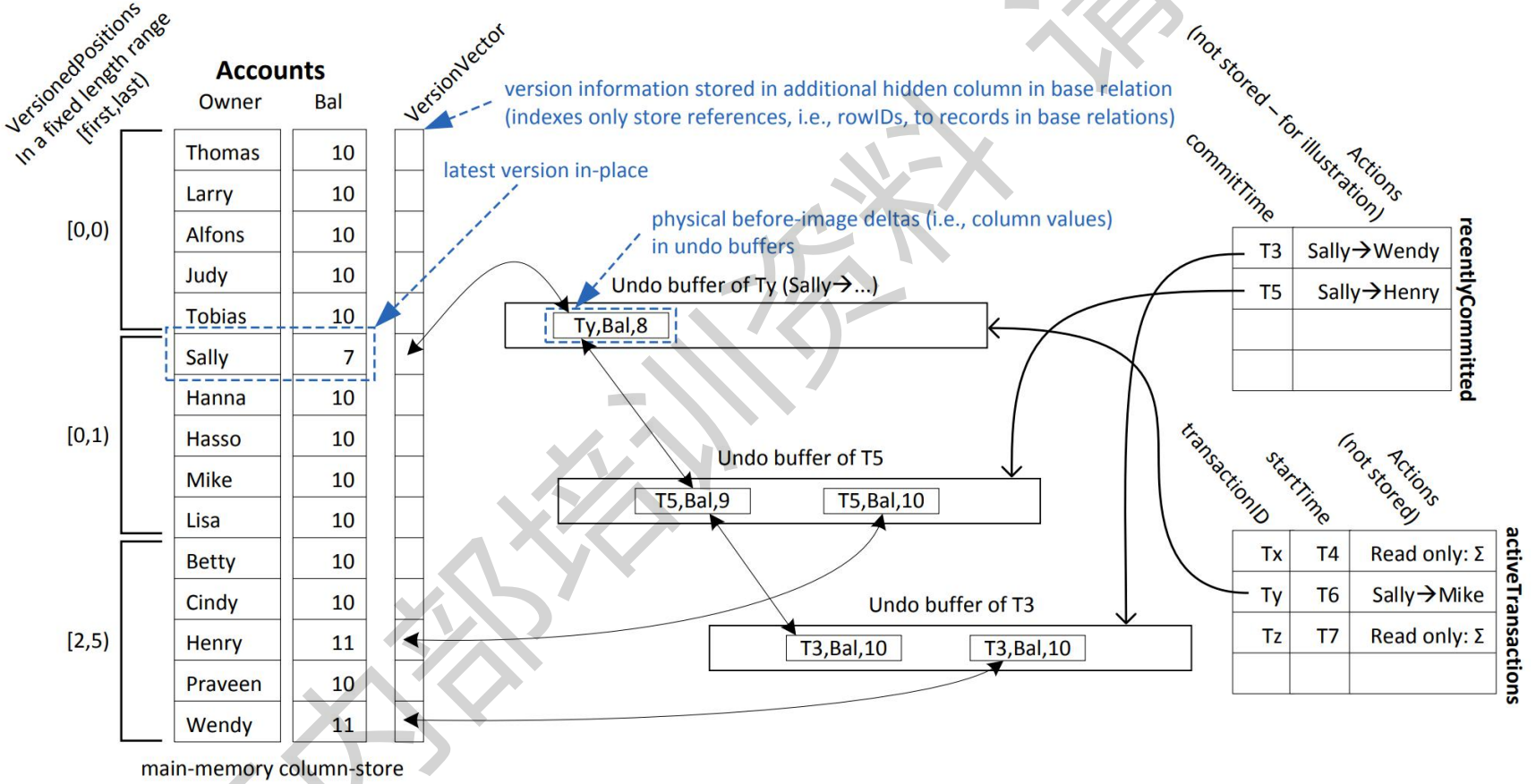


Figure 1: Multi-version concurrency control example: transferring $1 between Accounts (from → to) and summing all Balances (Σ)

SAP HANA

- Not partitioned
  - Any thread can access any record
- Versioned (row-to-column)
  - Versioning persists throughout entire lifetime
- Hybrid storage format
  - Three stages of physical record representation
    - L1-Delta: Write-optimized row format, no data compression (10K to 100K rows/node)
    - L2-Delta: Column format, unsorted dictionary encoding (~10M rows)
    - Main: Column format, highly compressed and sorted dictionary encoding

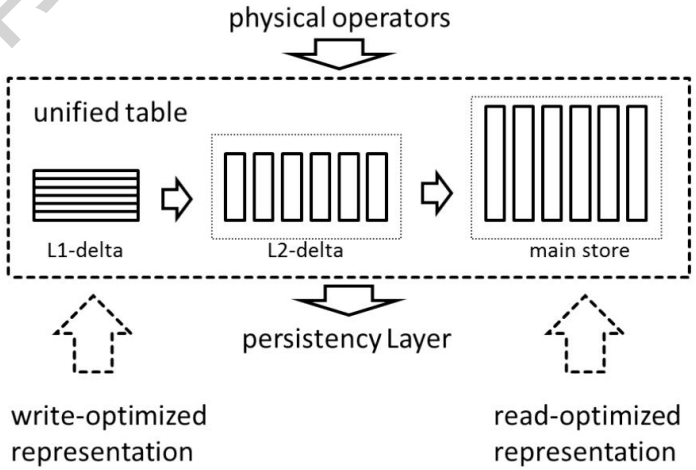


Figure 4: Overview of the unified table concept

Efficient transaction processing in SAP HANA database: the end of a column store myth. SIGMOD '12, 731–742.

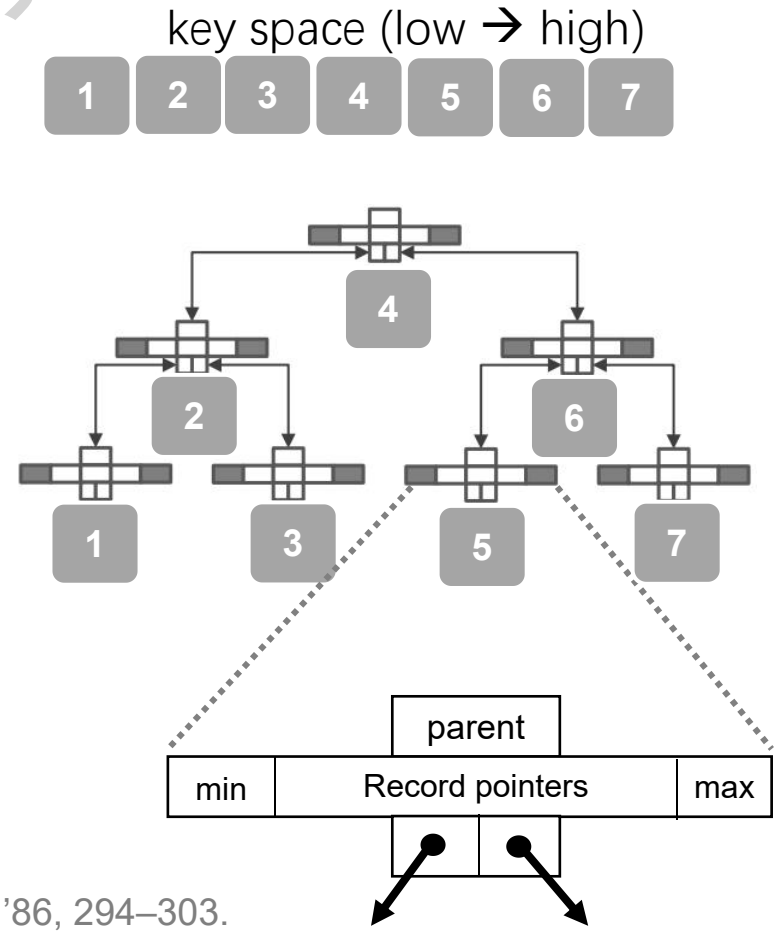
Indexing

Indexing

- Cache-awareness
  - Since the mid-90's CPU and RAM **speed gap** has been an issue
  - Indexing techniques created to keep as much **data in CPU cache** as possible
- Multi-core and multi-socket parallelism
  - Modern multi-core machines have massive amount of raw parallelism
  - Current focus is to enable high performance parallel indexing methods

T-Trees (Self-Balancing Binary Search Tree)

- Index in the MM-DBMS project at U. Wisconsin (1986)
  - Based on AVL tree; nodes store record pointers, sorted by key value for that node
  - Internal nodes index data; similar to B-Tree, not a B+-Tree
  - Min/Max boundary values define key space for a node
- Advantages
  - Low memory overhead (important in 80s)
- Disadvantages
  - Rebalance, scans, pointer dereference
- Used in early commercial main-memory systems (e.g, TimesTen)



A Study of Index Structures for Main Memory Database Management Systems. VLDB '86, 294–303.

Cache-Awareness

- 1990s: cache stalls show up as major **bottleneck**
  - CPU speeds at the time increasing at 60% per year
  - Memory speeds improving 10% per year
- Improvement of cache behavior in database system become a priority

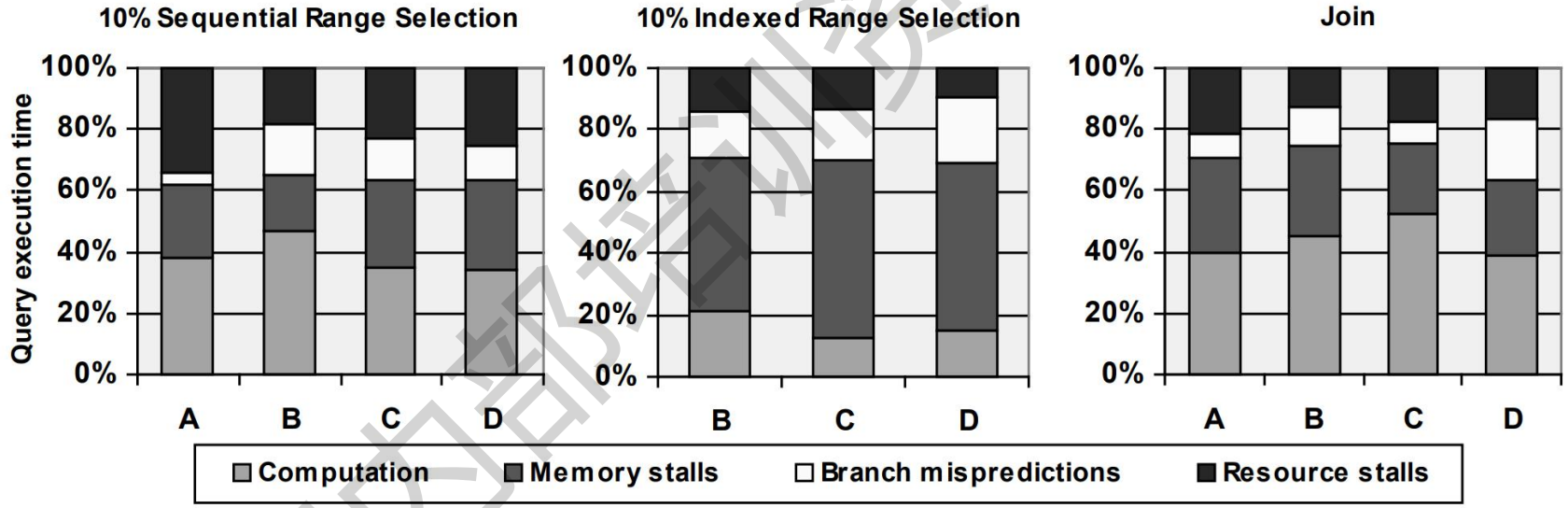
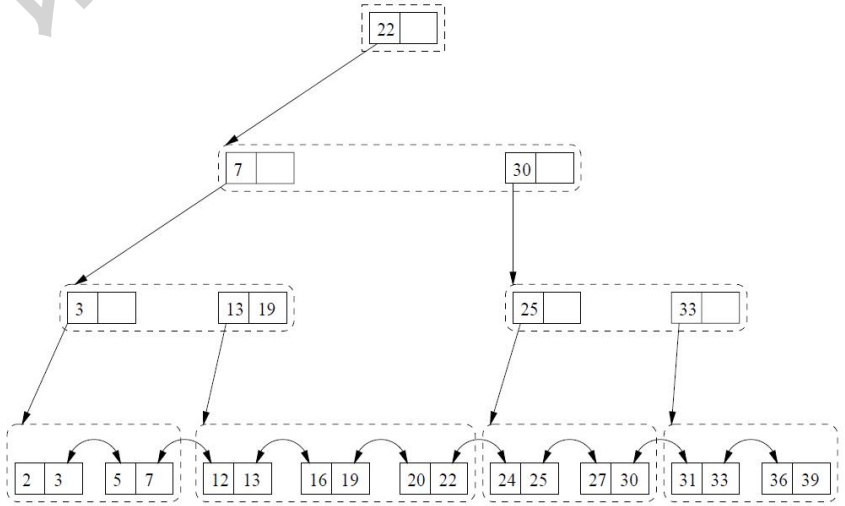


Figure 5.1: *Query execution time breakdown into the four time components.*

DBMSs on a Modern Processor: Where Does Time Go? VLDB, pp. 266-277, 1999

CSB+-Tree

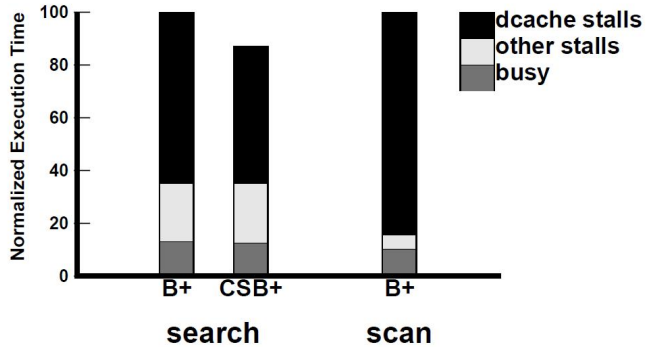


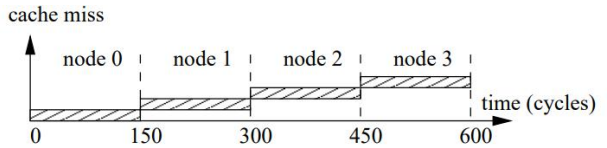
- Size nodes to cache line
- Basic idea: reduce node pointer storage
  - Similar to B+-Tree in spirit
  - All child nodes of a parent placed into node group
  - Parent stores single pointer to node group
  - Nodes within a node group can be accessed using offset
- Splits
  - Whole node groups allocated to accommodate new node
  - Entire node group copied over to new memory (pointer at parent replaced with new group pointer)
- Optimizations
  - Segmented node groups: essentially more node groups with more pointers stored at parent
  - Whole nodes groups: avoid reallocation by pre-allocating max node group space

Making B+-Trees Cache Conscious in Main Memory, SIGMOD, pp. 475-486, 2000

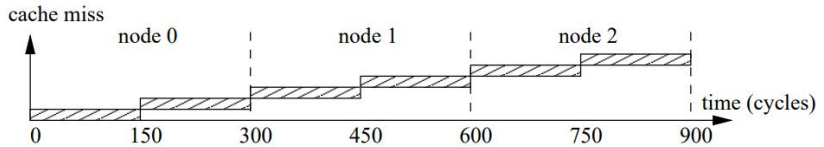
Pb+-Trees

- Prefetching on B+-Tree search and scans
  - Prefetching allows for increased fanout of tree node
  - Leads to shallower trees and better performance
  - 1.5x improvement for search, 6x improvement for scans
- Search requires very little change
  - Prefetch all lines that comprise a node
  - Single “expensive” cache miss when traversing from parent to child
- Scan implements a **jump pointer array**
  - Leaf nodes contain offset into pointer array
  - Can calculate N subsequent prefetch addresses from jump array

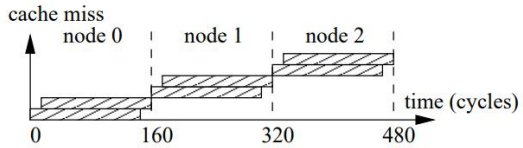




(a) Four levels of one-cache-line-wide nodes.



(b) Three levels of two-cache-lines-wide nodes.

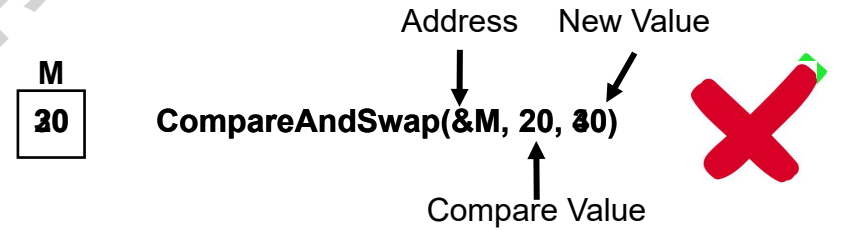


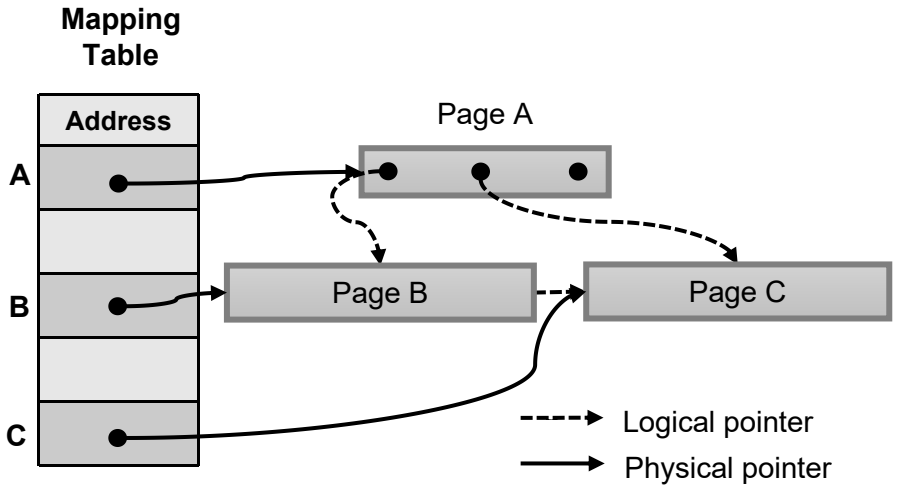
(c) Part (b) with prefetching.

Improving Index Performance through Prefetching, SIGMOD, pp. 235-246, 2001

Hekaton: Bw-Tree

- B+-Tree range index used in Hekaton
  - Completely latch-free like the rest of the database engine
  - Copy-on-write “delta” updates reduce cache invalidation
- Uses compare-and-swap atomic operation to change state
- Pages are logical; identified by mapping table index
  - Translates logical page id to memory address
  - Important for latch-free behavior
  - Isolates updates to a single page in the index (no pointer propagation)

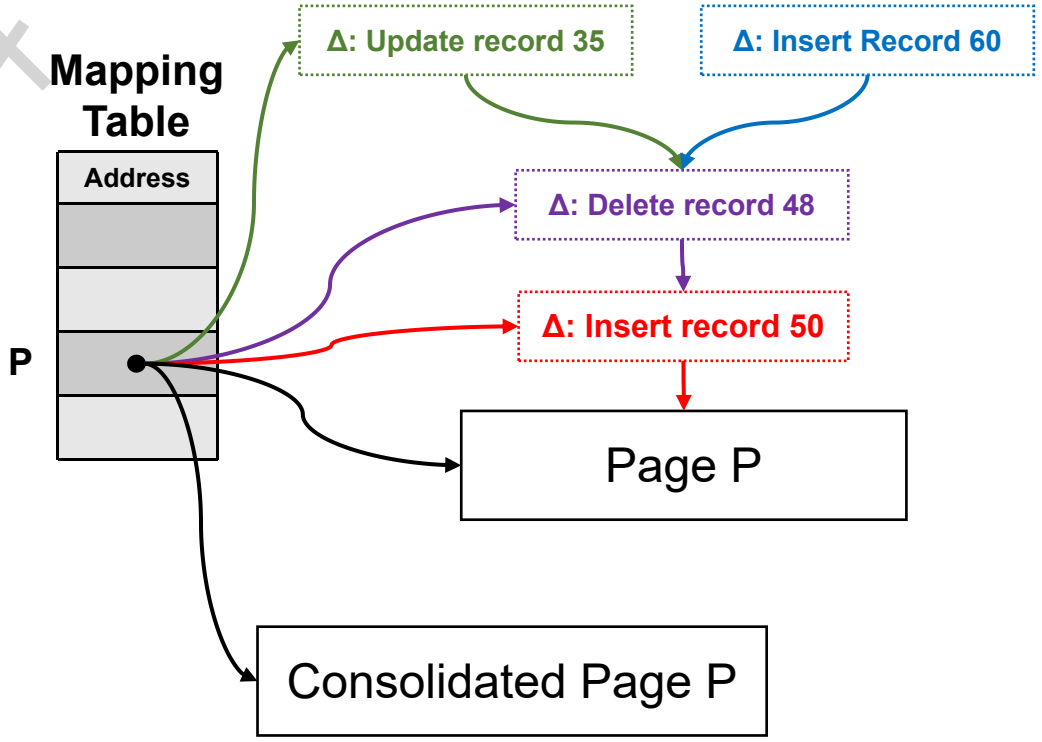




The Bw-Tree: A B-Tree for New Hardware Platforms, ICDE, pp. 302-313, 2013

Hekaton: Bw-Tree (2)

- Each page update produces a new address in the form of a delta record
- Install new page address in mapping table using compare-and-swap
- Only one winner on concurrent update to the same address
- Eventually install new consolidated page with deltas applied once a threshold has been met
- Structure modifications also latch-free



HyPer: Adaptive Radix Tree

- Traditional radix tree
  - Poor storage utilization: Max fanout was maintained in nodes
  - Height is determined by the length of the search key
- ART has an adaptive node structure
  - Allows for different node sizes within the index
  - Addresses trade-off between tree height vs. space efficiency
- Path compression
  - Collapses nodes with only a single child pointer
  - Nodes store prefix of key bytes in node header; prefix of length reduces index height by n levels
- Original proposal did not tackle multi-thread concurrency

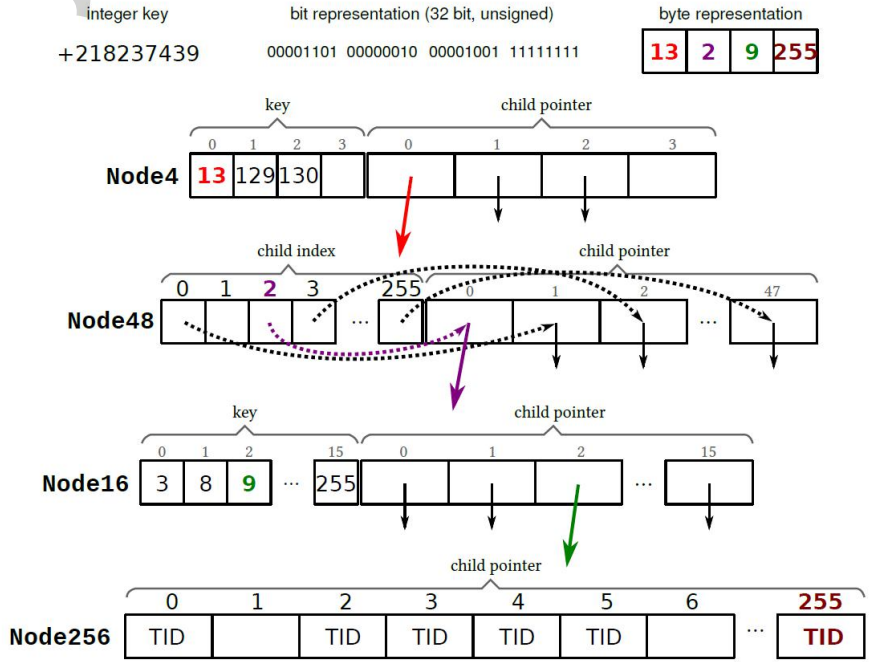


Figure 4.12: A sample path with adaptive nodes in the radix tree

The Adaptive Radix Tre: ARTful Indexing for Main-Memory Databases, ICDE, pp. 38-49, 2013

HANA P*Time: OLFIT on B+-Trees

- P*Time uses a B+-Tree index (hashing supported as well)
- OLFIT(optimistic latch free index access protocol)
  - address index concurrency focusing on CPU cache coherence issues due to latch/lock protocols
- Associate latch plus version number with B+-Tree node
- B+-Tree node updates
  - Acquire latch, update content, increment version, unlatch
- B+-Tree node reads (**no latching**)
  - Copy version to local register r
  - Read node content, validate latch is unset and node version is same as local copy, otherwise re-read node
- Node splits handled by B-link tree strategy

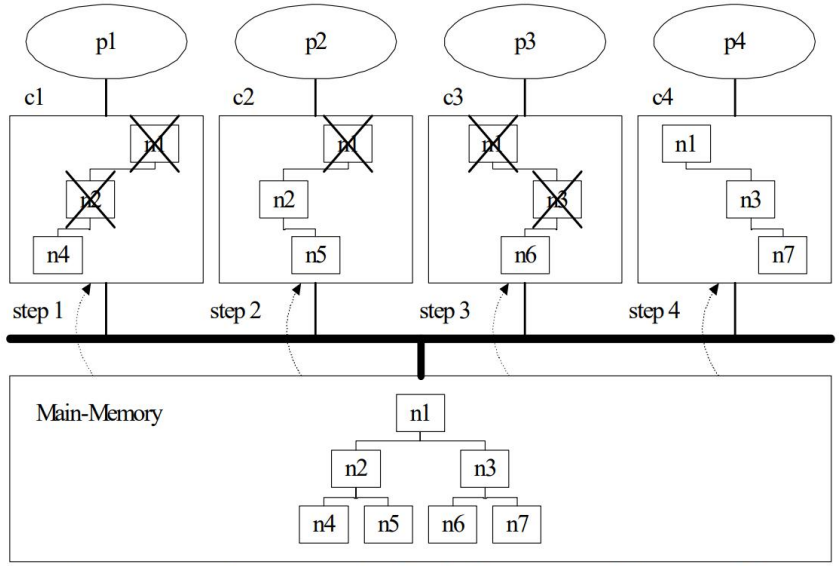
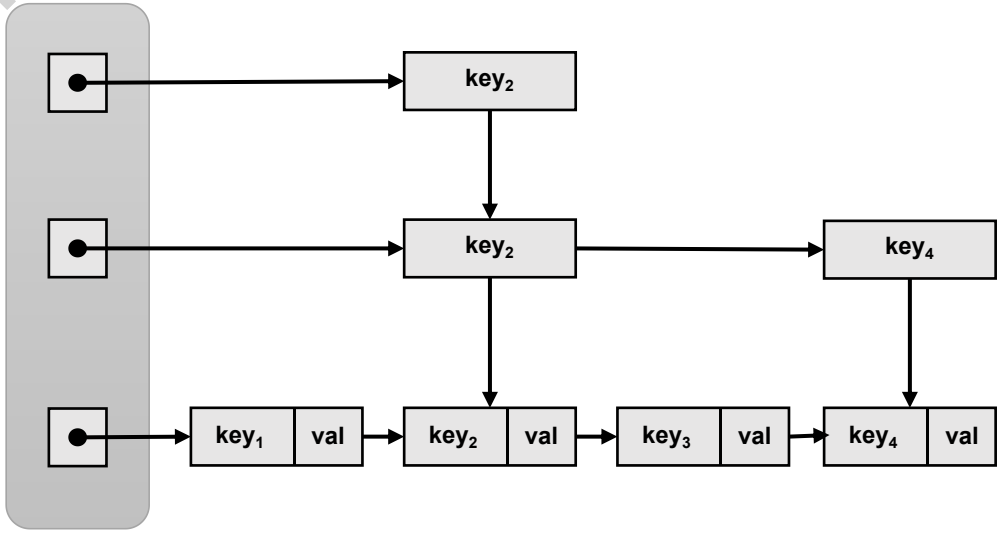


Figure 1. Coherence Cache Miss Caused by Latching

Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems
VLDB, pp. 181-190, 2001

Skiplists

- A linked list of sorted records with “express lanes” built at higher levels to enable fast search
- Bottom layer is always a complete list
- Key for an item at level *i* appears at level ***i+1*** with probability ***p***
  - Enables $(log_{1/p}n)/n$ search time
- Can be made **latch-free**
  - Compare-and-swap insert at base linked list
  - Once successful, build up “tower” at higher levels
- Skiplists used in MemSQL database engine

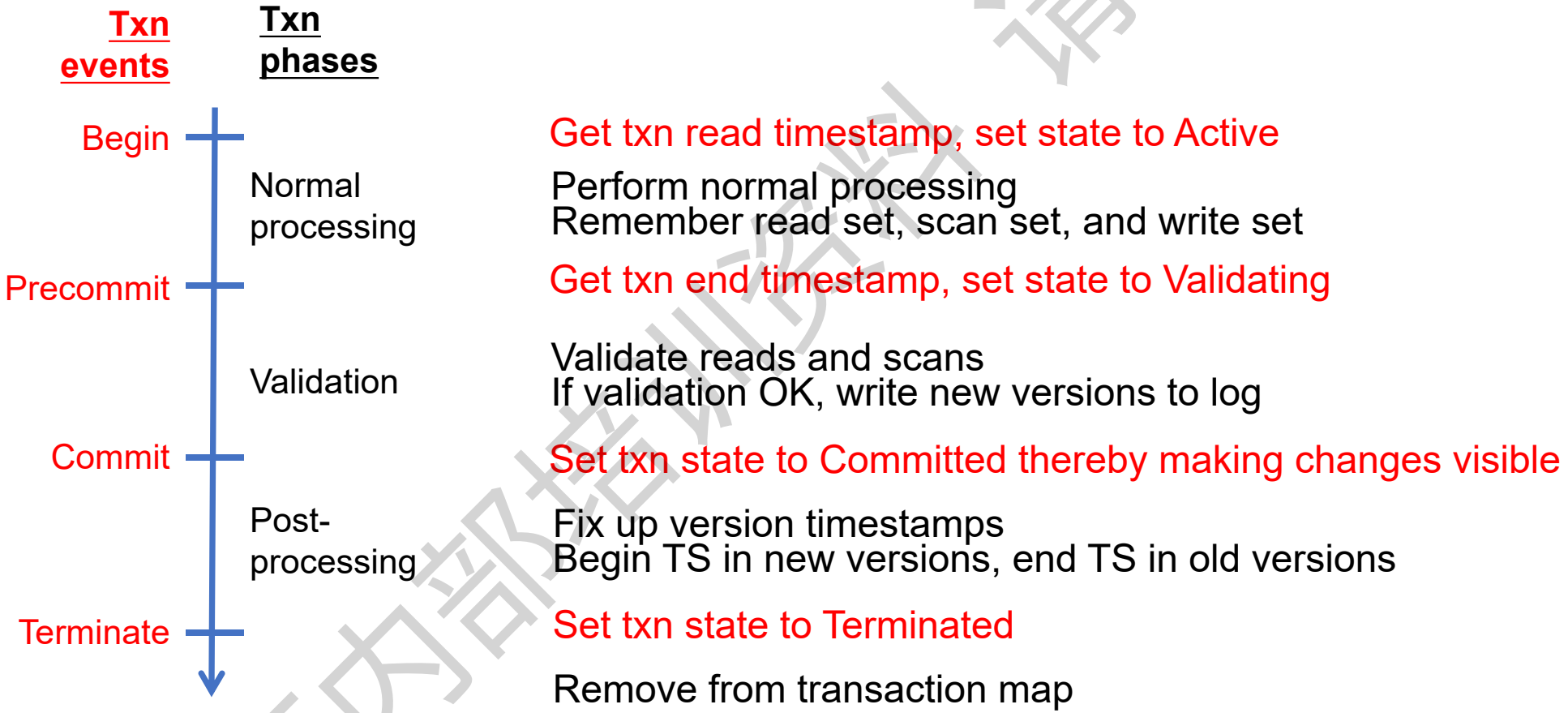


Concurrency Control &Durability

Main CC Approaches

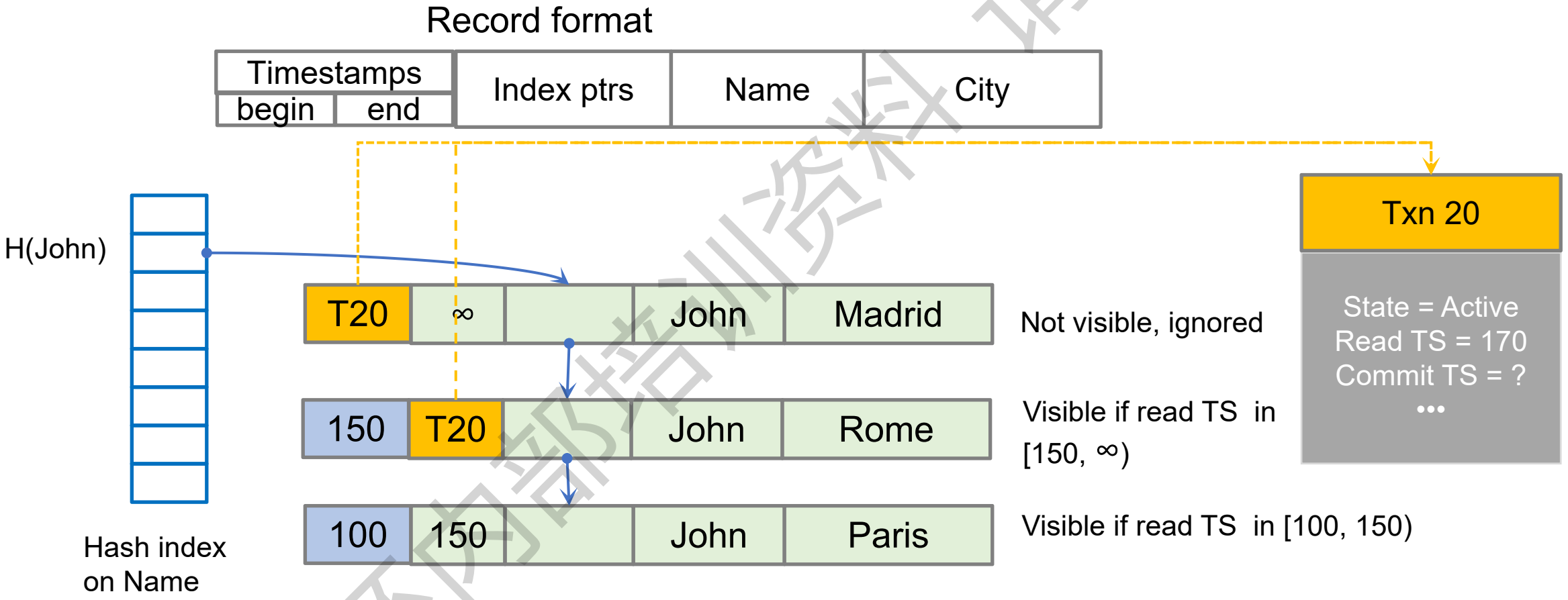
- Multi-version optimistic concurrency control (MVCC)
  - Multi-versioning: updates create new versions
    - + Readers read old versions and never block updaters → higher concurrency
    - - Updates more expensive; requires garbage collecting obsolete versions
  - OCC: No locks, check for interference with other transactions before committing
    - + Much cheaper than traditional locking; Scales to high numbers of cores
    - - High contention may cause high abort rates
- Partitioned serial execution (PSE): Partition DB by core, execute serially
  - + No locks or validation needed; Very fast for single-partition transactions
  - - Multi-partition transactions lock partitions in X mode → greatly reduced throughput
  - - Unpredictable performance

Hekaton's optimistic MVCC

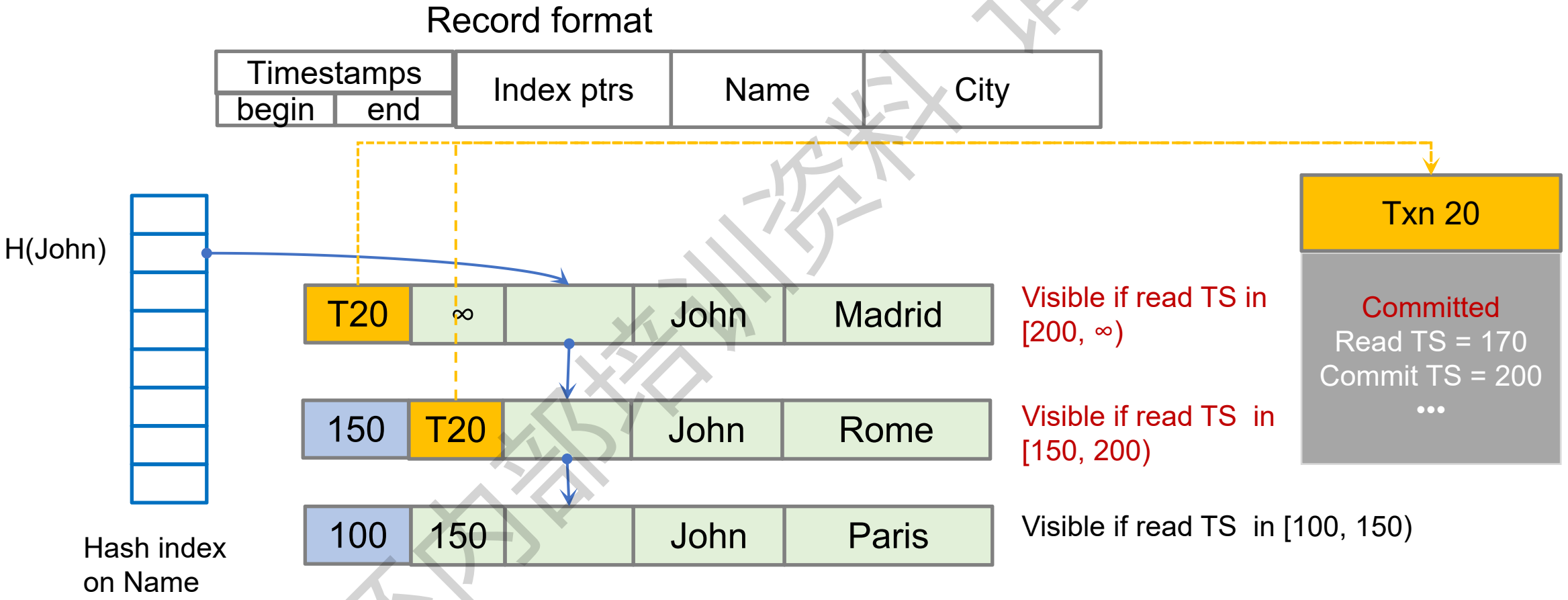


High-Performance concurrency control mechanisms for main-memory databases, VLDB 2011

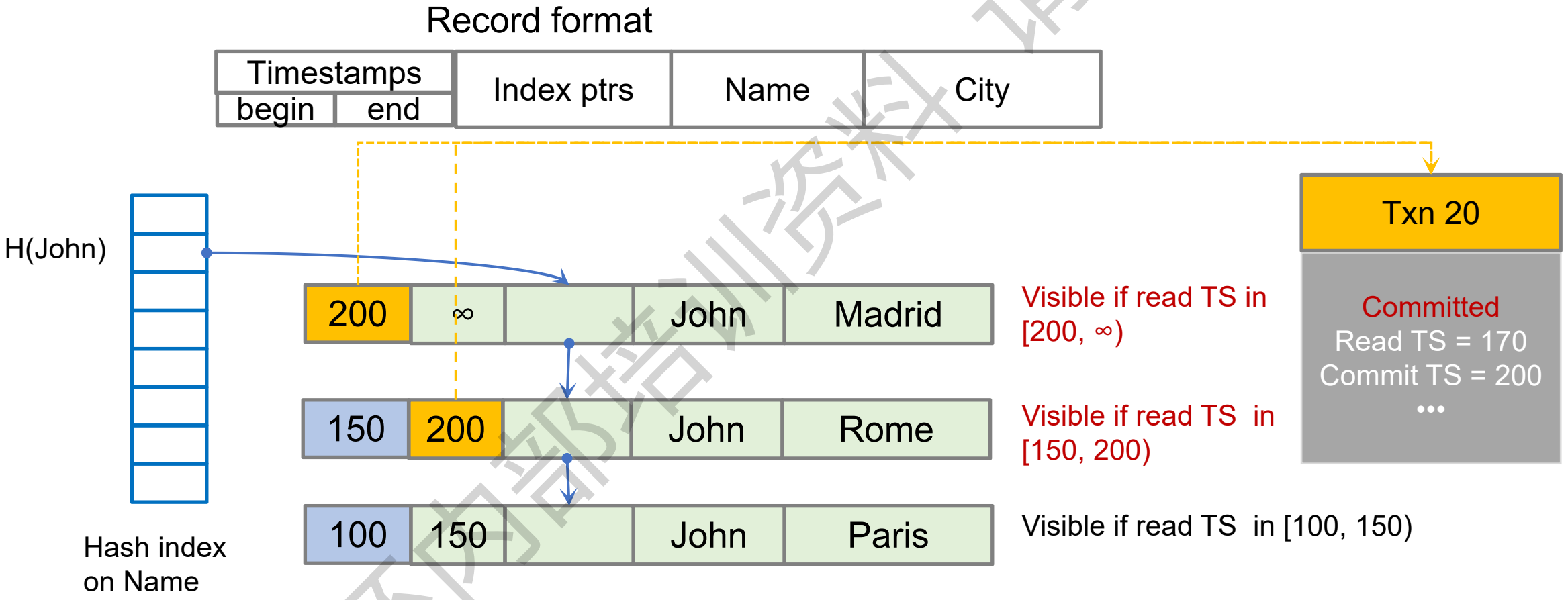
Record Versions and Visibility



Record Versions and Visibility



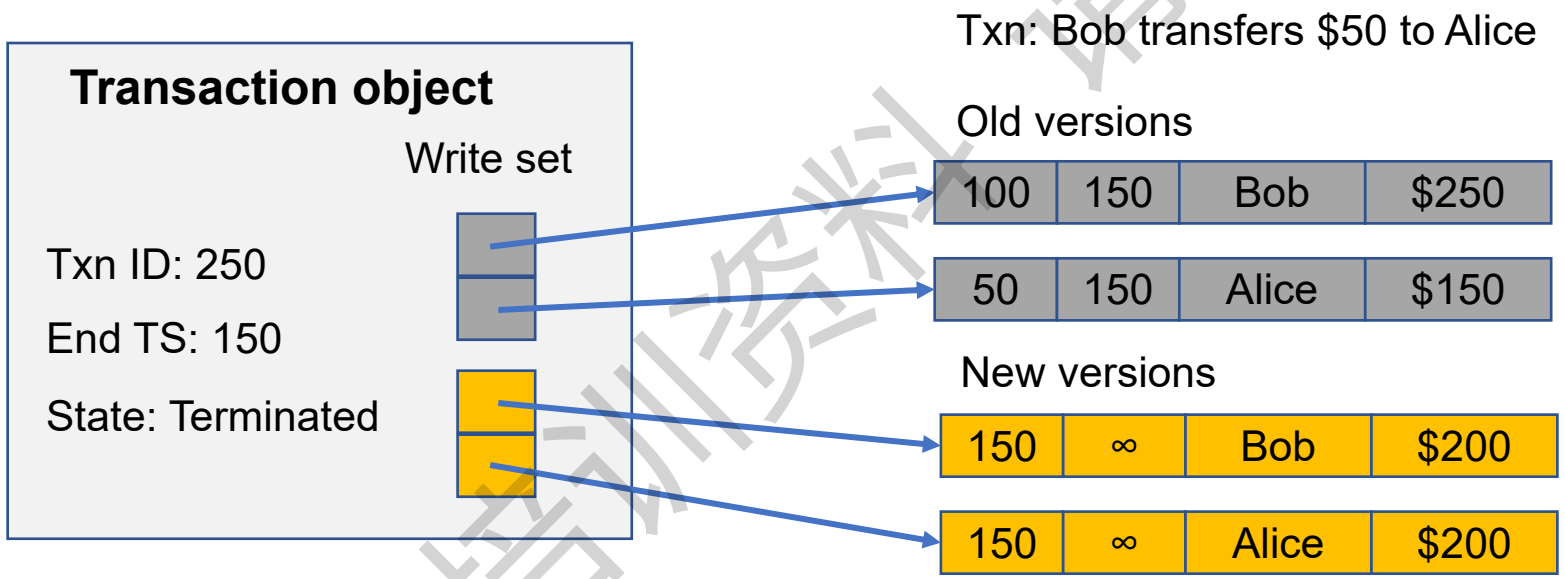
Record Versions and Visibility



Transaction Validation (Hekaton)

- Read stability
  - Check that each version read remains visible as of the end of the transaction
- Phantom avoidance
  - Repeat each scan checking whether new versions have become visible since the transaction began
- Read-only transaction: no validation required
- Updated transactions: extent of validation depends on isolation level
  - Snapshot isolation: no validation required
  - Repeatable read: read stability
  - Serializable: read stability, phantom avoidance

Old Versions Discarding



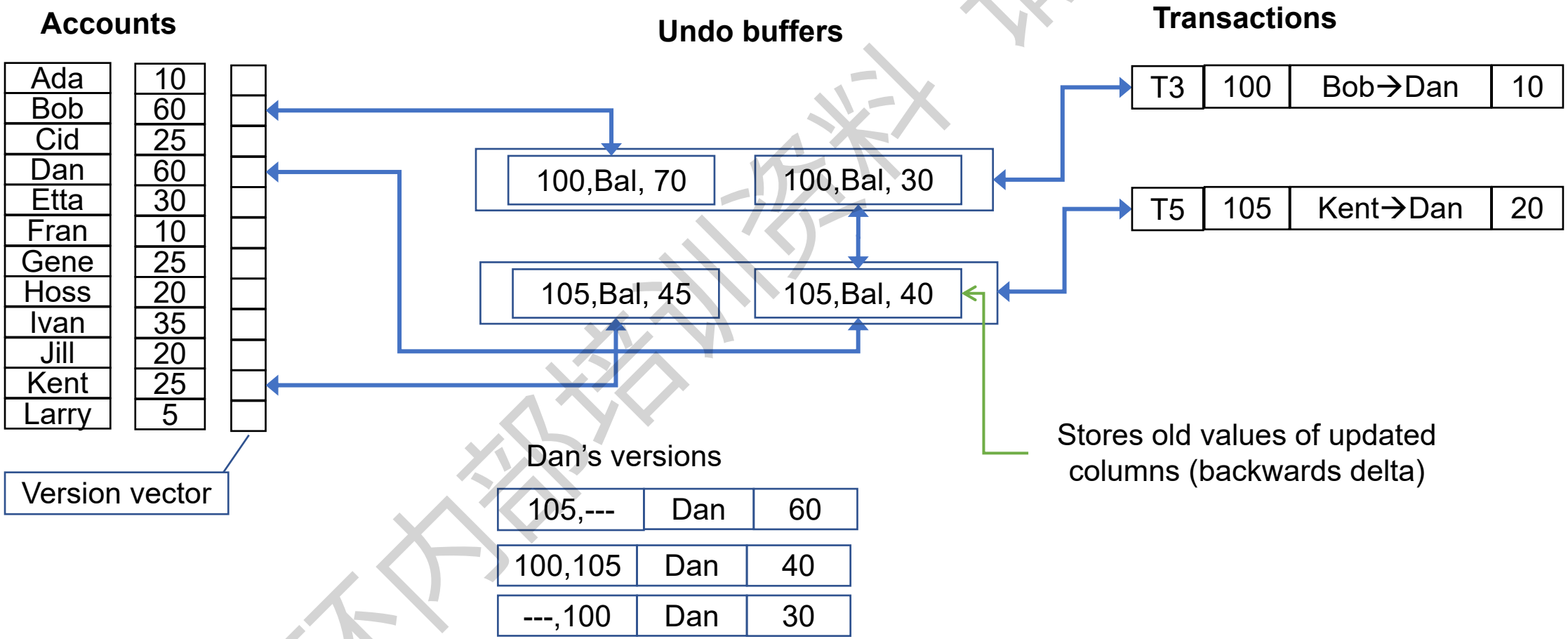
- **Can discard the old versions as soon as the read time of the oldest active transaction is over 150**
  - Old versions easily found – use pointers in write set
  - Two steps: unhook version from all indexes, release record slot

Hyper's MVCC

- Three main differences compared with Hekaton's implementation:
  - Records are updated **in place**; older version reached through a linked list
  - Validation done be checking recent updates against the read predicates of the current transaction
  - Commit processing currently done serially (get commit timestamp, validation, writing to log).

Fast Serializable Multi-Version Concurrency Control for Main-Memory Databases
SIGMOD 2015

Version Storage in Hyper



Transaction Validation in Hyper

- A transaction performs reads as of its start time. If it were to repeat its reads as of its commit time, would it get exactly the same result? If so, the transaction is serializable.
- Hyper's validation approach:
  - A transaction logs its
    - Read predicates (point lookup and range scan predicates)
    - Inserts, deletes, and updates.
  - At commit time, we check all **recent updates**, inserts, and deletes against the read predicates of the transaction.

High-level Validation Algorithm

Tv = transaction to be validated

For each transaction **Tx** whose commit time is between Tv's **start time** and **commit time** do

For every insert, delete, or update done by Tx do

- **Insert:** if the new record satisfies any of Tv's read predicates, we have a **phantom** so abort Tv
- **Delete:** if the deleted record satisfies any of Tv's read predicates, it belonged to Tv's read set but has now been deleted so abort Tv
- **Update:** if the old or the new version satisfies any of Tv's read predicates, the update might change Tv's read result so abort Tv . (Overly conservative but safe – can be refined).

End

End

Note: The actual Hyper implementation has many refinements and optimizations

Concurrency Control in HANA

- Uses Pessimistic Multi-Versioning (MVCC with row-level write locks)
  - Used by both the column store and the row store (P*Time)
- Rows contain timestamps that determine their visibility
  - Timestamps are updated lazily after commit
- A transaction must acquire a write lock before updating or deleting a row.
  - The lock is held until the transaction commits or aborts
- Deadlock detection is necessary

https://archive.sap.com/documents/docs/DOC-57101

CC in H-Store/VoltDB

- Database partitioned per core with serial execution within a partition
- Each partition protected by a single exclusive lock
- Before operating on a partition, a transaction must acquire the partition lock
- A transaction is aborted when it needs access to a new partition.
  - It's restarted when it has acquired locks on the required partitions.
  - So a transaction may be aborted and restarted multiple times.
- Performance:
  - ++ very fast for single-partition transactions
  - -- slow for multi-partition transactions
  - -- sensitive to load skew (hotspots)

The VoltDB Main Memory DBMS
IEEE Data Engineering Bulletin, 36(2):21–27, 2013.

Predetermining Serialization Order

- Daniel Abadi and his students have explored CC approaches where
  - Transaction serialization order is determined before execution
  - Transaction read and write sets are known in advance
- VLL (Very Lightweight Locking)
  - Lock information stored with records (counts of exclusive and shared locks)
  - All locks requested before transaction execution begins
  - Execution begins when all locks have been granted
- BOHM – serializable multi-version CC protocol
  - Determines serialization order and creates (placeholder) new version prior to execution
- Calvin – distributed transactions without a distributed commit protocol

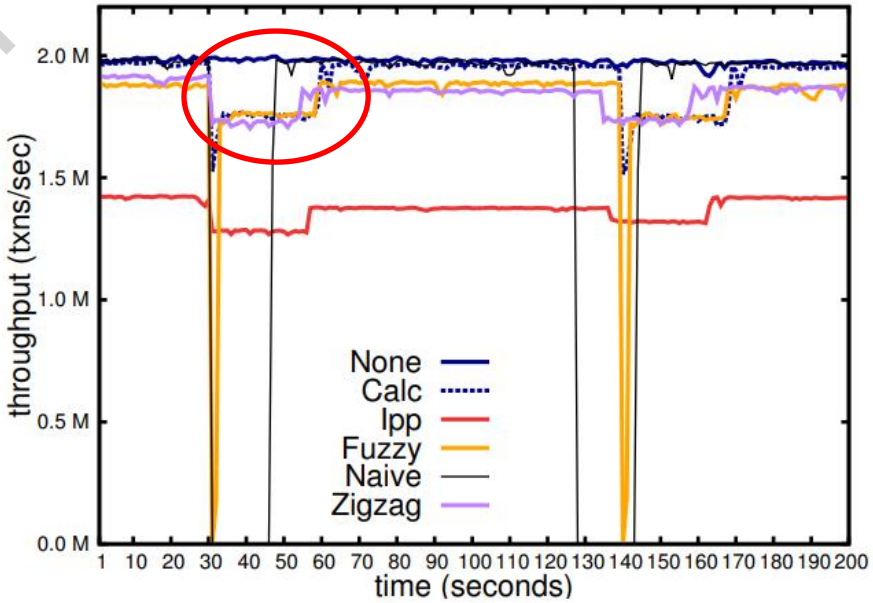
Calvin: Fast Distributed Transactions for Partitioned Database Systems, SIGMOD 2012
Lightweight Locking for Main-Memory Database Systems, PVLDB 2013
Rethinking Serializable Multiversion Concurrency Control, PVLDB 2015

Durability

- Basic approach still based on **logging** and **checkpointing**
- A checkpoint must contain all rows but not necessarily **index data**
  - Much larger than checkpoints in disk-based systems
- Optimize logging for high throughput and low latency
  - Minimize log write latency
  - Reduce log volume, e.g. redo logging only, don't log index updates
  - Use fast log devices
  - Spread log over multiple devices
- Recovery time dominated by rebuilding the database and indexes from a checkpoint
  - Max out IO bandwidth
  - Parallelize, parallelize!

CALC Checkpointing Algorithm

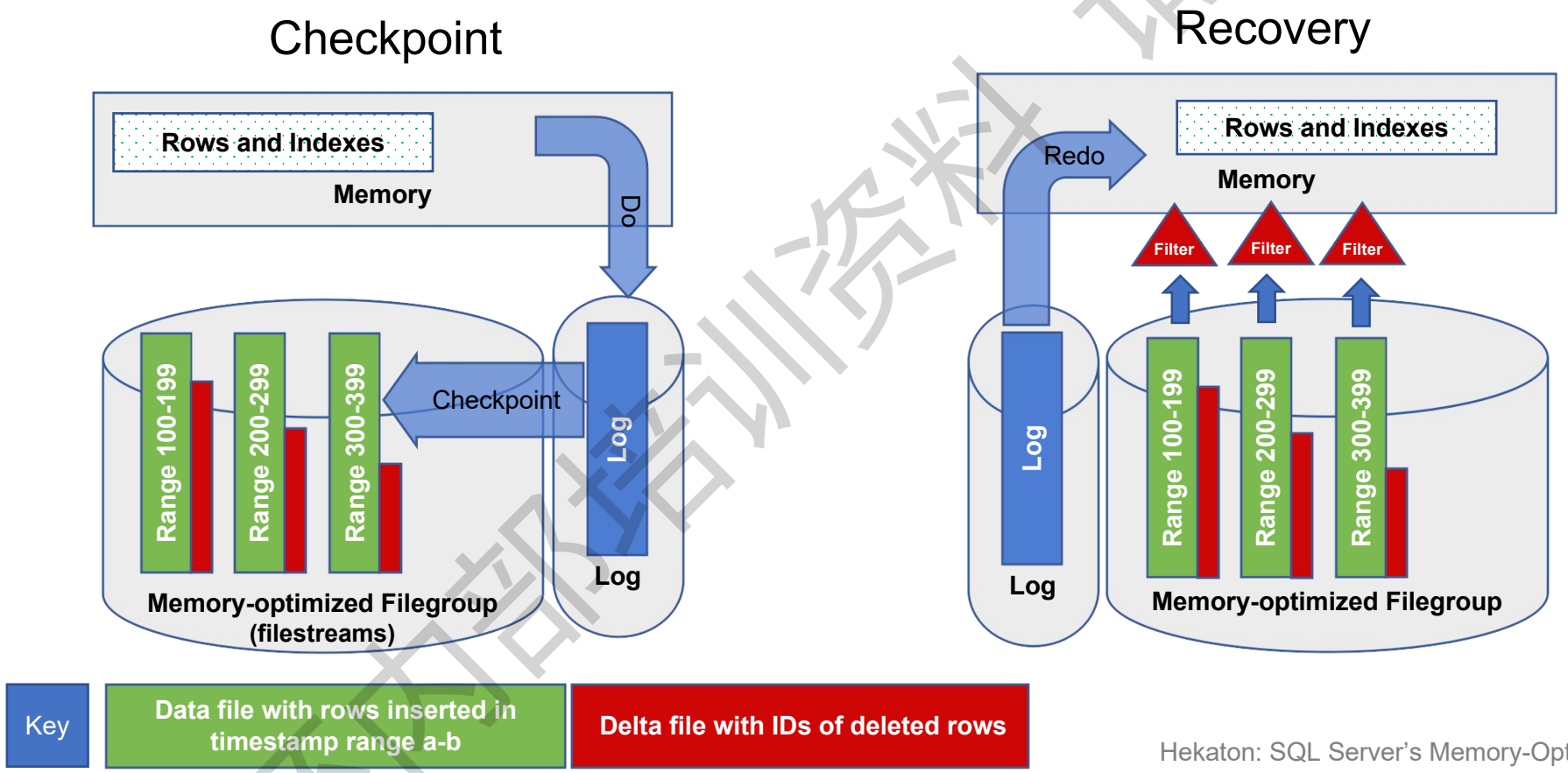
- Creates transaction-consistent checkpoints asynchronously
  - For systems without multiversioning
  - Complete checkpoint – all records
  - Partial checkpoint – only updates since last checkpoint
- Maintains two versions while checkpoint is taken
  - Live version
  - Stable version - version as of checkpoint time
- Partial checkpoints require recording which records have changed since last checkpoint
  - Bit map, hash table, …
- About 10% slowdown during checkpointing
  - No more than fuzzy checkpointing
- Low memory overhead



(a) Throughput over time, no long xacts

Low-Overhead Asynchronous Checkpointing in Main-memory Database Systems
SIGMOD 2016

Hekaton Checkpoint and Recovery



Hekaton: SQL Server's Memory-Optimized OLTP Engine
SIGMOD 2013

Other Systems

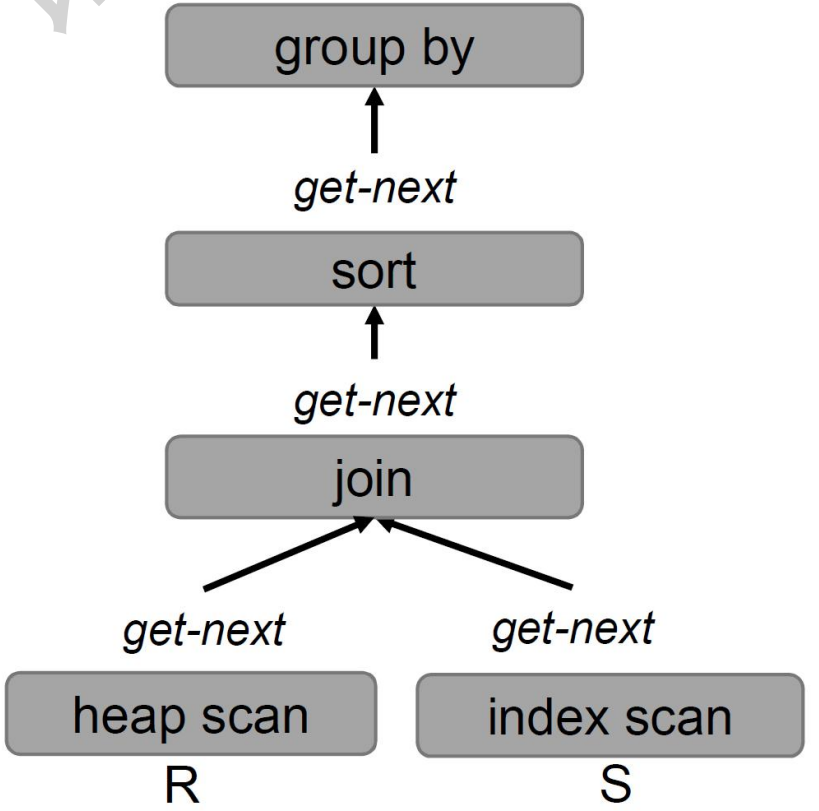
- Typically just redo information is logged
  - Value logging: updated version of each record is logged
  - Command logging: log procedure calls including parameters (VoltDB)
- Periodically write out a consistent snapshot of the database
  - Indexes may or may not be included
  - Hyper uses it OS-based (fork) snapshot mechanism
  - H-Store/VoltDB switches to a copy-on-write mode so old versions are retained. Also needs to ensure consistency across partitions.
- Recovery is a two-step process
  - Rebuild the database including indexes from the latest checkpoint (dominates recovery time)
  - Apply the tail of the log
    - Value logging: copy in new values
    - Command logging: re-execute transactions – must guarantee exactly the same result

Rethinking Main Memory OLTP Recovery
ICDE 2014

Query Processing and Compilation

Traditional Volcano-Style Processing

- The query processing component of a DBMS
  - "upper" layers: authorization, parsing, and query optimization
  - **"lower" layers**: execute the physical query plan
- Iterator Model
  - Each operator implements a generic interface
    - *init, get-next, close*
  - Arbitrary combination of operators
- Overhead in IMDB
  - General-purpose code must handle various scenarios
    - Interpret the bytes within a tuple at runtime, casting them to the appropriate data type and performing runtime error checks as necessary
  - Get-next function called for every tuple (millions of times)
  - Get-next call is usually virtual (or call via function pointer)
    - Expensive
    - Degrades branch prediction on modern processors
  - Poor code locality



Query Processing in IMDB

- Compile queries and transactions into machine code
  - avoid interpretation
  - avoid unnecessary runtime overhead
  - make efficient use of the CPU cache hierarchy

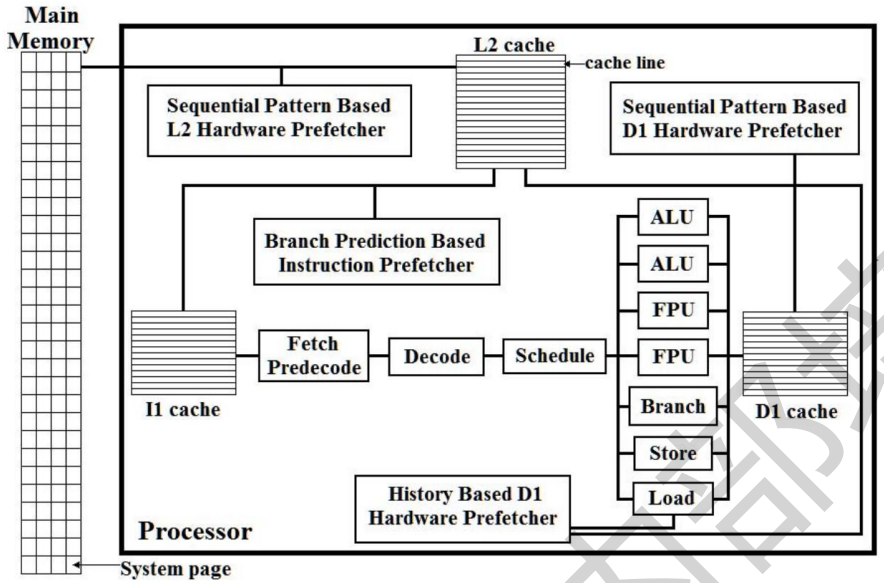


Fig. 1. The architecture of modern CPUs

- HIQUE (Holistic Integrated Query Engine)

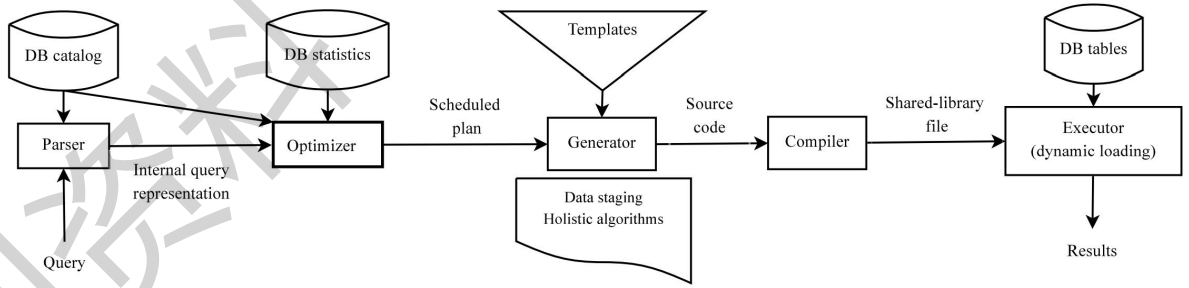
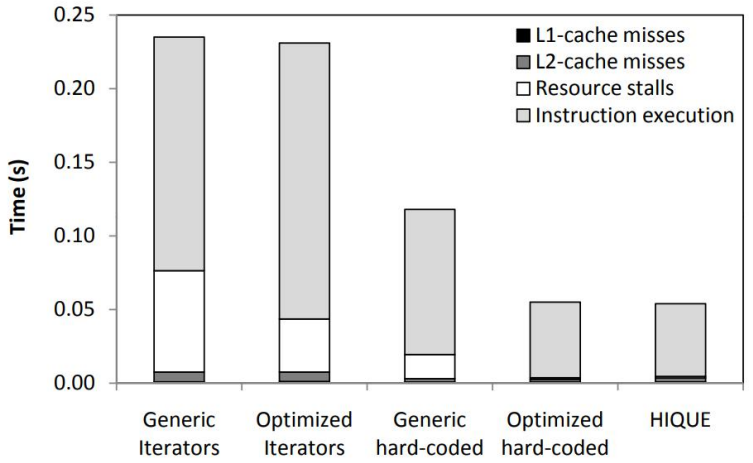


Fig. 2. Holistic query engine overview



(a) Execution time breakdown for Join Query #1

Generating Code for Holistic Query Evaluation
ICDE, pp. 613-624, 2010

Query Compilation in Hekaton

- Compile queries directly into machine code
  - Compile once; execute many times (typical OLTP scenario)
- Do as much as possible at compile time
  - Types known at compile time; avoids interpretation
  - Collapse logic into single function as much as possible
- Schema compilation
  - Compile table definition when created
    - Hekaton storage engine treats records as opaque objects (use callbacks for comparison, calculating hash values, etc.)
    - Interoperation with SQL-Server interpreted query executor
- Optimize T-SQL query then compile to C (then to machine code)
  - Leverage existing compiler technology within Microsoft
- 3-4X observed speedup

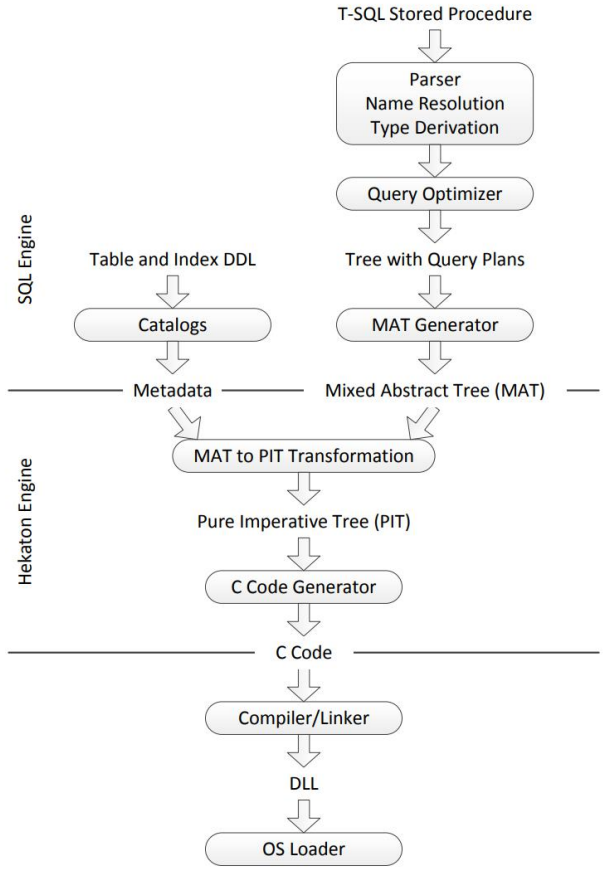


Figure 1: Architecture of the Hekaton compiler.

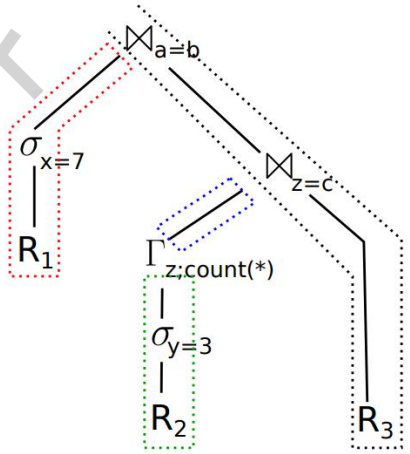
Compilation in the Microsoft SQL Server Hekaton Engine
IEEE Data Eng. Bull. 37(1): 22-30 (2014)

Query Compilation in HyPer

- Data centric: maximize data locality by keeping attributes in CPU registers as long as possible
- Query broken into pipeline fragments
  - Pipeline defined by materialization points in plan
  - Tuple passes through all operators in pipeline before materialization into next pipeline breaker
- LLVM backend and JIT compilation
- Operators partially implemented in C++ while performance and query-specific logic is generated

```
select  *
from    R1,R3,
        (select  R2.z,
                 count(*)
         from    R2
         where   R2.y=3
         group by R2.z) R2
where   R1.x=7
and     R1.a=R3.b
and     R2.z=R3.c
```

(a) Example SQL Query



(b) Execution Plan

```
initialize memory of ⋈a=b, ⋈c=z, and Γz
for each tuple t in R1
  if t.x = 7
    materialize t in hash table of ⋈a=b
for each tuple t in R2
  if t.y = 3
    aggregate t in hash table of Γz
for each tuple t in Γz
  materialize t in hash table of ⋈z=c
for each tuple t3 in R3
  for each match t2 in ⋈z=c [t3.c]
    for each match t1 in ⋈a=b [t3.b]
      output t1 ∘ t2 ∘ t3
```

(c) Compiled query (pseudo-code)

Figure 1: From SQL to executable code

Compiling Database Queries into Machine Code
IEEE Data Eng. Bull. 37(1): 3-11 (2014)

Query Compilation in MemSQL

- Uses specialized programing language MemSQL Plan Language (MPL)
- SQL operator trees are converted directly to MPL abstract syntax trees
- MPL are translated to MemSQL Bytecode
- MemSQL Bytecode can interpreted or transformed to LLVM bitcode for compilation to machine code

```
memsql> select * from t where j > 0;
Empty set (0.05 sec)

memsql> select * from t where j > 0;
Empty set (0.05 sec)
```

There is no additional latency on the first request because the query is **interpreted**.

```
memsql> select * from t where j > 0;  -- With code generation
Empty set (0.08 sec)

memsql> select * from t where j > 0;  -- Using the cached plan
Empty set (0.02 sec)
```

The additional **latency** on the first request

The subsequent request is more than twice as **fast** as the interpreted execution

https://docs.memsql.com/v7.1/concepts/code-generation/

Systems

Notable In-Memory DBMSs

- Oracle TimesTen
- Dali / DataBlitz
- Altibase
- P*TIME
- **SAP HANA**
- **VoltDB / H-Store**
- **Microsoft Hekaton**
- MIT/Harvard Silo
- **TUM HyPer**
- MemSQL
- solidDB
- Apache Geode

solidDB

- Founded 1992 in Helsinki, Finland, bough by IBM in 2007, sold to UNICOM in 2014
- Hybrid database system: both disk-based and main-memory optimized engine
- Indexing using Vtrie (variable-length trie)
- Uses pessimistic locking
- Snapshot consistent checkpoints for recovery

IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability
IEEE Data Eng. Bull. 36(2): 14-20 (2013)

Oracle TimesTen

- Began as research project at HP Labs named Smallbase
  - Spun off into separate company in mid 1990s and acquired later by Oracle in 2005
- Flexible engine deployment
  - Standalone DBMS engine
  - Transactional cache on top of Oracle RDBMS
  - In-memory repository for BI workloads
- Concurrency control through locking
- Row-level latching to handle write-write conflicts
- Uses write-ahead logging and checkpointing for durability

Oracle TimesTen: An In-Memory Database for Enterprise Applications
IEEE Data Eng. Bull. 36(2): 6-13 (2013)

Altibase

- Founded in 1999 in South Korea
  - Large customer base spanning telecom, financial, and manufacturing companies
- Stores records on pages
  - Checkpoints written at page granularity
  - Compatibility with disk-based engine
- Multi-versioned concurrency control
- Uses write-ahead logging and checkpointing for durability/recovery
  - Latch-free checkpointing process when writing page data to checkpoint file

MemSQL

- Hybrid database aimed at high-performance transactions and analytics
- Designed to scale out on commodity hardware
  - Aggregator node interface for query routing
  - Leaf nodes provide in-memory storage and query processing
- Latch-free skiplists for indexing
- Multi-version concurrency control
- Row locks for read committed and snapshot isolation concurrency control
- Durability through flushing redo-only transaction log
- Query compilation using LLVM using MemSQL Programming Language (MPL)

MemSQL FAQ
https://docs.memsql.com/v7.1/introduction/faqs/memsql-faq/

Silo

- High-performance main-memory database system built on top of the MassTree
- Reduces atomic writes to hotspots
  - Key to performance on multi-core, multi-socket machines
  - Unique timestamp generation (classic example of a hotspot)
    - Use epoch-based approach: global epoch E (incremented every so often) occupies high-order bits of each transaction
- Serializable concurrency through tracking read sets and installing at commit time
  - Phantom protection through versioning and tracking range index leaf nodes
- Exploits multi-core parallelism throughout durability and recovery design
  - Redo-only logging in parallel across multiple disks
  - Parallel checkpointing

Speedy Transactions in Multicore In-Memory Databases
SOSP, pp.18-32, 2013

P*TIME

- Korean in-memory DBMS from the 2000s.
- Performance numbers are still impressive.
- Lots of interesting features:
  - Uses differential encoding (XOR) for log records.
  - Hybrid storage layouts.
  - Support for larger-than-memory databases.
- Sold to SAP in 2005. Now part of HANA.

P*TIME: HIGHLY SCALABLE OLTP DBMS FOR MANAGING UPDATE-INTENSIVE STREAM WORKLOAD
VLDB, pp. 1033-1044, 2004.

Dali / DataBlitz

- Developed at AT&T Labs in the early 1990s.
- Multi-process, shared memory storage manager using memory-mapped files.
- Employed additional safety measures to make sure that erroneous writes to memory do not corrupt the database.
  - Meta-data is stored in a non-shared location.
  - A page's checksum is always tested on a read; if the checksum is invalid, recover page from log.

DALI: A HIGH PERFORMANCE MAIN MEMORY STORAGE MANAGER
VLDB, pp. 48-59, 1994

Summary

Bottlenecks in IMDB

- The primary storage location of the database is in memory
  - Don’t need to retrieve data from disk
  - Disk I/O is no longer the slowest resource
- IMDBs account for other bottlenecks
  - Locking/latching
  - Cache-line misses
  - Pointer chasing
  - Predicate evaluation
  - Data movement and copying
  - Networking (between application and DBMS)

The end of an architectural era: (it’s time for a complete rewrite).
VLDB ’07, pages 1150–1160, 2007.

	Disk-based DB	IMDB
Data Organization	• Slotted pages (Disk & Buffer) • Continuous pages on Disk • Using record ID	• Splits tuples into fixed-length and variable-length pools • Don't need to store tuples continuous • Using direct pointers to the fixed-length data for each tuple
Concurrency Control	• Assumes that a transaction could stall at any time • Store locking information separate from the data (lock table)	• Still use either a pessimistic or optimistic concurrency control schemes • Store locking information together with the data
Indexes	• B+-tree index	• Use data structures optimized for fast in-memory access • Latch-free • Will not log index updates (Rebuild the indexes)
Query Processing	• tuple-at-a-time iterator model • init, get-next, close	• Sequential scans are no longer significantly faster than random access • Compile queries and transactions into machine code
Logging and Recovery	• Use WAL • Use STEAL + NO-FORCE buffer pool policies	• Still needs WAL on non-volatile storage • Use more lightweight logging schemes • Still takes checkpoints

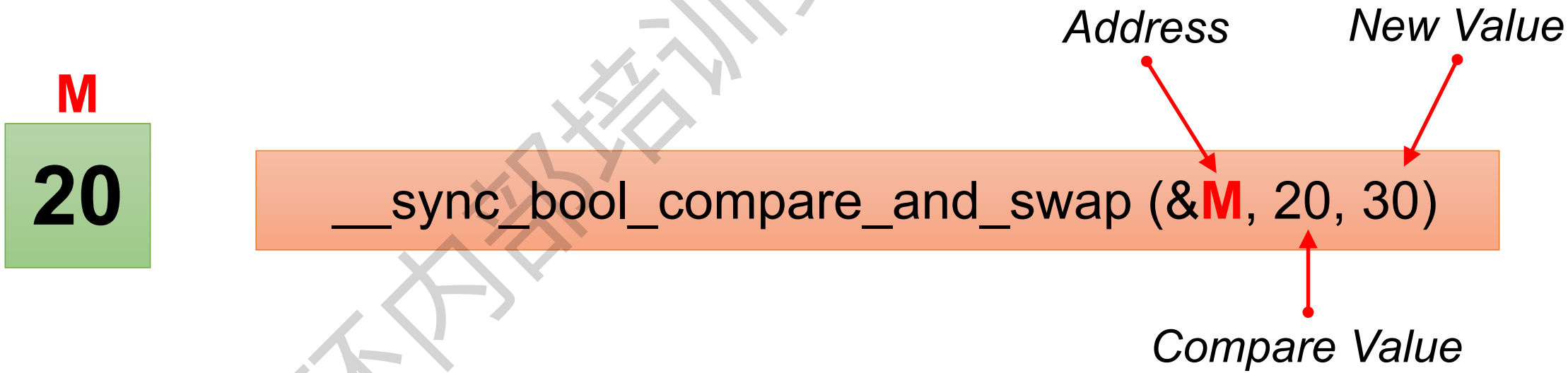
1. Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. IEEE Data Engineering Bulletin, 8(2):3–10, 1985.
2. David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. SIGMOD Record, 14(2):1–8, 1984.
3. Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In Proceedings of the International Conference on Very Large Data Bases, VLDB, pages 294–303, 1986.
4. Tobin J. Lehman and Michael J. Carey. Query Processing in Main Memory Database Management Systems. In Proceedings of the ACM Conference on Management of Data, SIGMOD, pages 239–250, 1986.
5. Tobin J. Lehman and Michael J. Carey. A Recovery Algorithm for a High-Performance Memory-Resident Database System. Proceedings of the ACM Conference on Management of Data, SIGMOD, pages 104–117, 1987.
6. Margaret H. Eich. A Classification and Comparison of Main Memory Database Recovery Techniques. In Proceedings of the International Conference on Data Engineering, ICDE, pages 332–339, 1987.
7. Kenneth Salem and Hector Garcia-Molina. Checkpointing Memory-Resident Databases. In Proceedings of the International Conference on Data Engineering, ICDE, pages 452–462, 1989.
8. Dina Bitton, Maria Hanrahan, and Carolyn Turbyfill. Performance of Complex Queries in Main Memory Database Systems. In Proceedings of the International Conference on Data Engineering, ICDE, pages 72–81, 1987.
9. Kai Li and Jeffrey F. Naughton. Multiprocessor Main Memory Transaction Processing. Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, DPDS, pages 177–187, 1988.
10. Peter M. G. Apers, Martin L. Kersten, and Hans Oerlemans. PRISMA Database Machine: A Distributed, Main-Memory Approach. In Proceedings of the International Conference on Extending Database Technology, EDBT, pages 590–593, 1988.

11. Peter M. G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W. P. J.Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB: A parallel main memory relational DBMS. IEEE Transactions on Knowledge and Data Engineering, TKDE, 4(6):541–554, 1992.
12. Tobin J. Lehman, Eugene J. Shekita, and Luis-Felipe Cabrera. An Evaluation of Starburst's Memory Resident Storage Component. IEEE Transactions on Knowledge and Data Engineering, TKDE, 4(6):555–566, 1992.
13. H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In Proceedings of the International Conference on Very Large Data Bases, VLDB, pages 48–59, 1994.
14. Philip Bohannon, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, S. Seshadri, and S. Sudarshan. The Architecture of the Dalí Main-Memory Storage Manager. Multimedia Tools Appl., 4(2):115–151, 1997.
15. Jerry Baulier, Philip Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, Henry F. Korth, Peter McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, Rajeev Rastogi, S. Seshadri, Abraham Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. DataBlitz Storage Manager:Main Memory Database Performance for Critical Applications. In Proceedings of the ACM Conference on Management of Data, SIGMOD, pages 519–520, 1999.
16. Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In Proceedings of the International Conference on Very Large Data Bases, VLDB, pages 469–477, 1995.
17. Authur Whitney, Dennis Shasha, and Stevan Apter. High Volume Transaction Processing without Concurrency Control, Two Phase Commit, SQL or C++. In Proceedings of the International Workshop on High Performance Transaction Systems, HPTS, 1997.
18. Times-Ten Team. In-Memory Data Management for Consumer Transactions The Times-Ten Approach. In Proceedings of the ACM Conference on Management of Data, SIGMOD, pages 528–529, 1999.
19. Times-Ten Team. In-Memory Data Management in the Application Tier. In Proceedings of the International Conference on Data Engineering, ICDE, pages 637–641, 2000.
20. Times-Ten Team. Mid-tier Caching: The TimesTen Approach. In Proceedings of the ACM Conference on Management of Data, SIGMOD, pages 588–593, 2002.
21. Sang Kyun Cha and Changbin Song. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In Proceedings of the International Conference on Very Large Data Bases, VLDB, pages 1033–1044, 2004.

谢谢！

Compare-and-Swap

- Atomic instruction that compares contents of a memory location **M** to a given value **V**
  - If values are equal, installs new given value **V'** in **M**
  - Otherwise operation fails

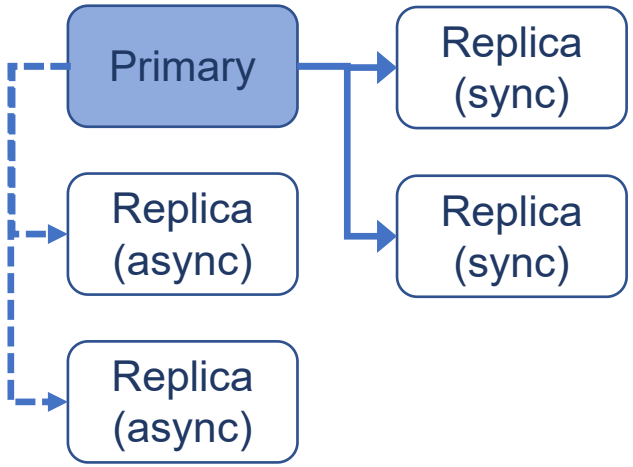


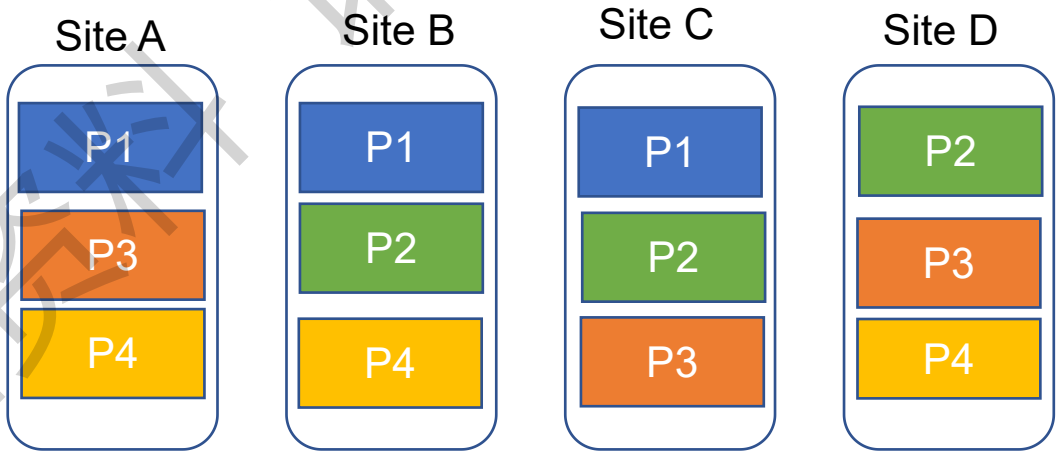
Concurrency Control in Silo

- Main-memory database from MIT and Harvard designed
  - To be multicore and NUMA friendly; To avoid all centralized contention points
  - So readers never write to shared memory
- Indexes are a cache-friendly variant of B-trees (Masstree) with versioned nodes
- Commit timestamps not drawn from a centralized counter
  - High-order bits from a slow-moving central counter, low-order bits from a local counter
- Single-version optimistic CC with three-phase commit protocol
  - Phase 1: lock all records in the transaction's write set (lock bit embedded in record header)
  - Phase 2: validate reads by checking that record timestamps haven't changed
  - Phase 3: apply all updates, release record lock as soon as the record has been updated
- Problem: a transaction doesn't see its own updates!
  - Unless it checks every read against its write set - very expensive

Speedy Transactions in Multicore In-Memory Databases, SOSP, 2013.

Primary HA Architectures





Primary plus failover replicas

- Fast failover to a replica if primary fails
- Read-only load can be offloaded to replicas

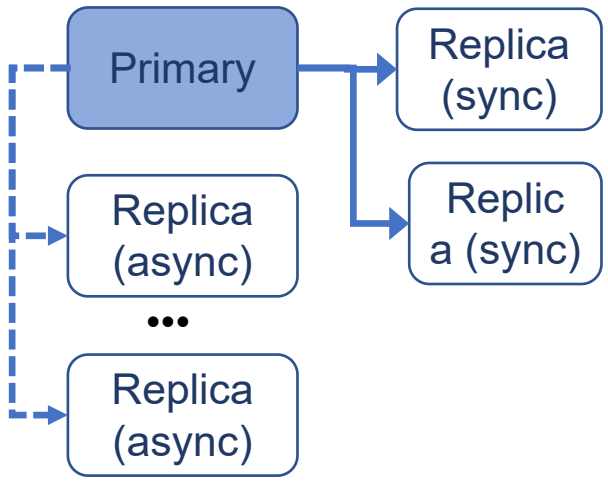
Partitioned and replicated database

- Service remains available even if two sites are lost

Primary + Failover Replicas

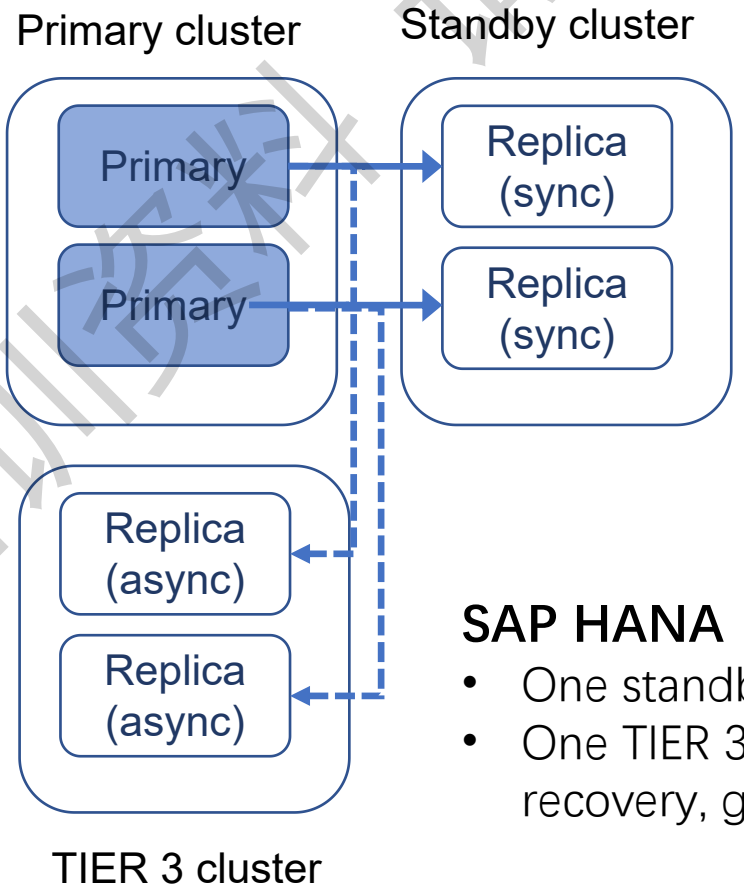
- **Replication protocol**
  - Primary ships log to replicas
  - Replica writes the log records to its log and send an ack to the primary (safest option)
  - Replica updates its database copy
- **Commit options:** The primary commits a transaction when
  - All replicas have written the transaction's log records to their logs (foolproof)
  - All replicas have the log records in memory (data loss if all replicas lose power)
  - When the log records have been sent (data loss likely on failover)
- **Failover** – typically completes in a few seconds
  - A replica detects missed heartbeats from the primary and begins failover
  - The replicas select a leader that becomes the new primary
  - The new primary and the replicas synchronize the state of their database copies
  - The primary begins accepting connections
- Work can be **offloaded** to replicas
  - Read-only transactions, database back-ups, shipping data to additional replicas, ….
- Used by SQL Server, SAP HANA, Hyper/Scyper

SQL Server and HANA HA Configuration



SQL Server

- Max 8 replicas, 2 sync
- Read-only offloading
- Backup offloading



SAP HANA

- One standby replica (sync) close by
- One TIER 3 async replica for disaster recovery, geodistributed

Partitioned and Replicated Database

- With K replicas, up to K-1 sites can fail and the DB still remains available
- When the master site detects that a site has failed, it starts the process of rebuilding the failed site
- The new site rebuilds the required partitions, catches up with the update activity, and rejoins
  - Copy partitions from other sites and catch up by applying their logs
- If the master fails, a new master (leader) is selected
- Is logging to durable storage really needed when we have N copies in RAM?
  - Most customers are conservative and insist on logging and backup
- Used by VoltDB – each transaction is executed at K sites
  - Must be deterministic, producing exactly the same result at each site

Timestamp Allocation

- Mutex
  - This is the worst option. Mutexes are always a terrible idea.
- Atomic Addition
  - Use compare-and-swap to increment a single global counter.
  - Requires cache invalidation on write.
- Batched Atomic Addition
  - Use compare-and-swap to increment a single global counter in batches.
  - Needs a back-off mechanism to prevent fast burn
- Hardware Clock
  - The CPU maintains an internal clock (not wall clock) that is synchronized across all cores.
  - Intel only. Not sure if it will exist in future CPUs.(2014)
- Hardware Counter
  - Single global counter maintained in hardware.
  - Not implemented in any existing CPUs.

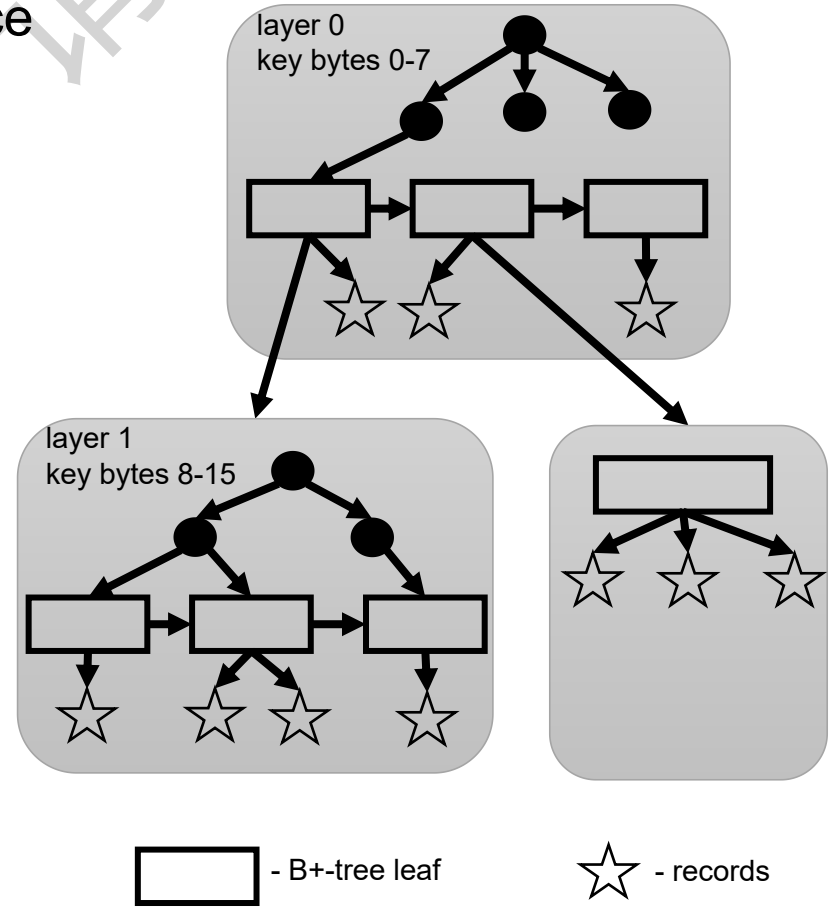
Staring into the abyss: an evaluation of concurrency control with one thousand cores.
VLDB '14, P.209–220, November 2014.

CSB+ and Pb+ Trees

- Cache line is the “unit of transfer” in main-memory system
  - Size B+-tree nodes accordingly
- CSB+-Tree
  - Remove as many B+-tree pointers as possible; use space to **pack keys together**
  - All child nodes of a B+-tree parent placed into **node group**
  - Parent stores **single** pointer to node group
  - Nodes within a node group can be accessed using **offset**
- Pb+-Tree
  - Explores effect of prefetching on B+-Tree search and scans
  - Prefetching allows for increased fanout of tree node beyond the “unit of transfer”
  - Leads to shallower trees and better performance
  - 1.5x improvement for search, 6x improvement for scans

MassTree

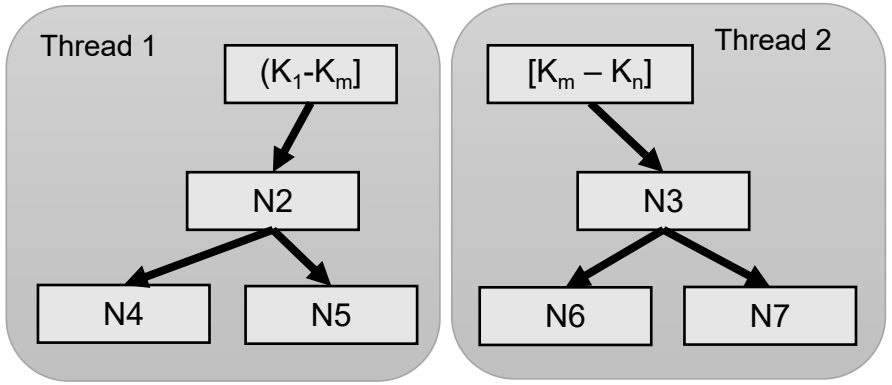
- Trie of B+-Trees; each layer indexed by 8-byte key slice
- Concurrency design leads to high performance on modern hardware
  - B+-tree nodes versioned
  - writer-writer coordination: fine-grained spinlock bit in version word
  - reader-writer coordination: optimistic concurrency
    - Writers use aligned writes; mark version word dirty before writing
    - Reader snapshots version word on node it is reading
    - Before returning read value, validates whether version is dirty or different from snapshot
    - Reader retries if validation fails



Cache Craftiness for Fast Multicore Key-Value Storage
EuroSys, pp. 183-196, 2012

Partitioned Latch-Free Indexing

- PLP
  - Multi-rooted B+-tree partitions key range
  - Single-threaded execution within partition
    - Thread given exclusive access to partition
    - No thread-level concurrency issues within partition
  - Must deal with hotspots and imbalance
- PALM
  - Latch-free batched B+-Tree design
  - Read phase performed first
  - Updates partitioned amongst threads to guarantee no contention and correct ordering of queries in batch
  - Structure modifications proceed in lock step and redistributed to guarantee single-thread execution



PLP: Page Latch-free Shared-everything OLTP
PVLDB 4(10): 610-621, 2011

PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors
PVLDB 4(11): 795-806, 2011