The Bw-Tree Key-Value Store and Its Applications to Server/Cloud Data Management in Production

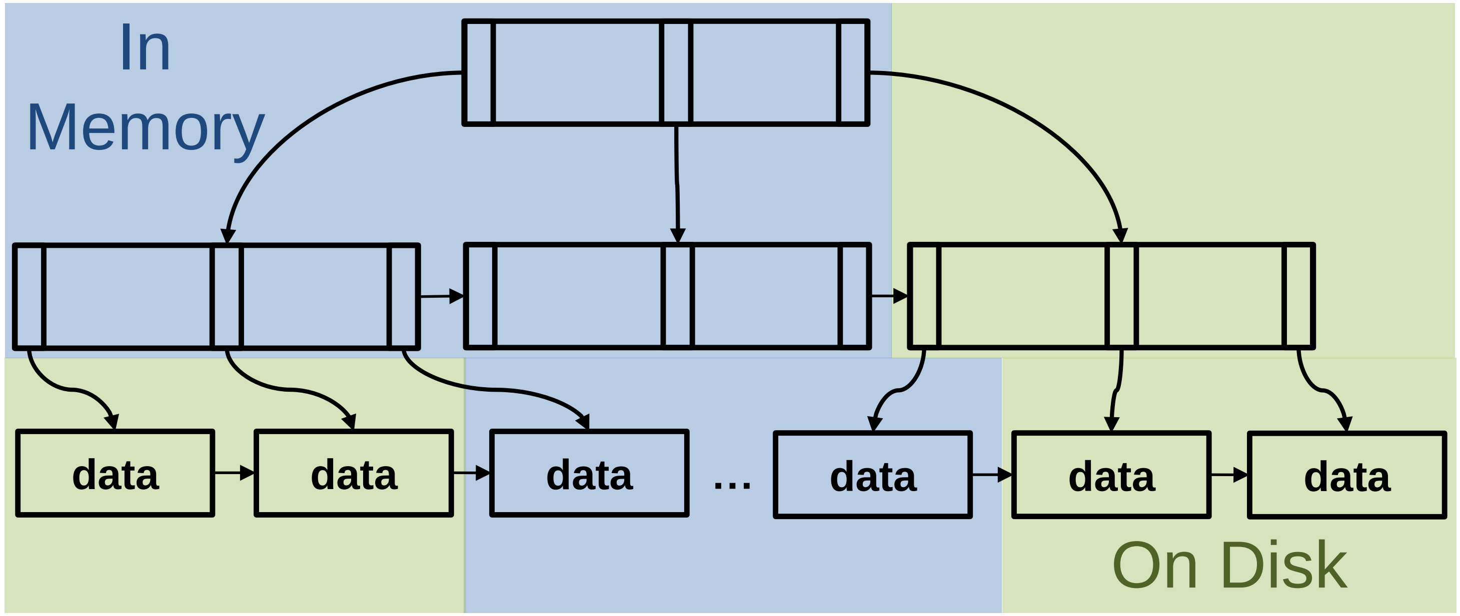
Sudipta Sengupta

Joint work with Justin Levandoski and David Lomet
(Microsoft Research)

And Microsoft Product Group Partners across SQL Server, Azure DocumentDB, and Bing ObjectStore

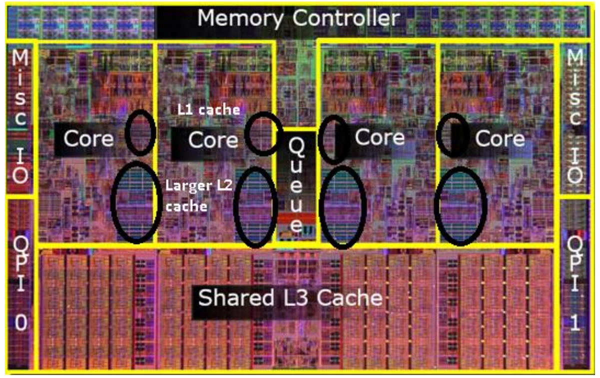
The B-Tree

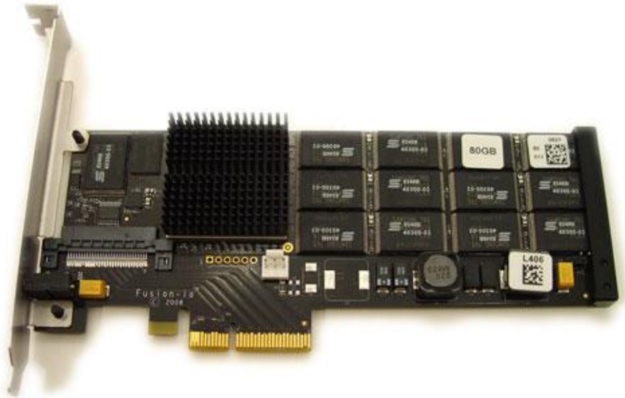
- Key-ordered access to records
- Separator keys in internal nodes (to guide search) and full records in leaf nodes
- Efficient point and range lookups
- Balanced tree via page split and merge mechanisms



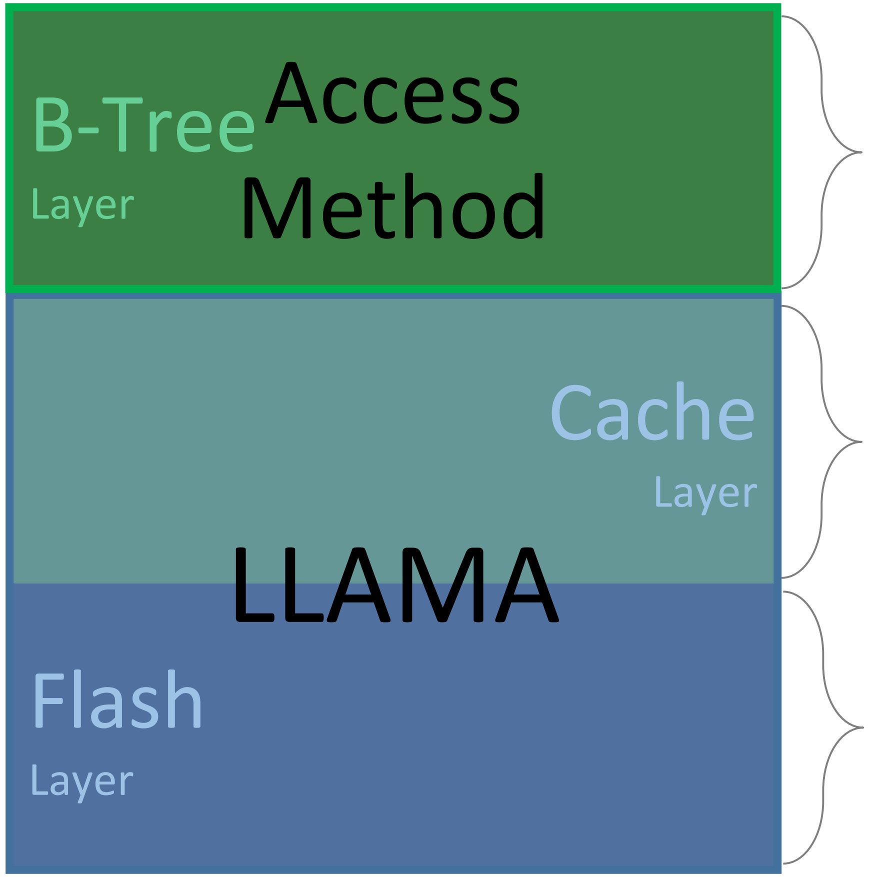
Design Tenets for A New B-Tree

- Lock-free operations for high concurrency
  - Exploit modern multi-core processors
- Log-structured Storage Organization
  - Exploit fast random read property of flash and work around inefficient random writes
- Delta updates to pages
  - Reduces cache invalidation in memory hierarchy
  - Reduces garbage creation and write amplification on flash, increases device lifetime





Bw-Tree Architecture



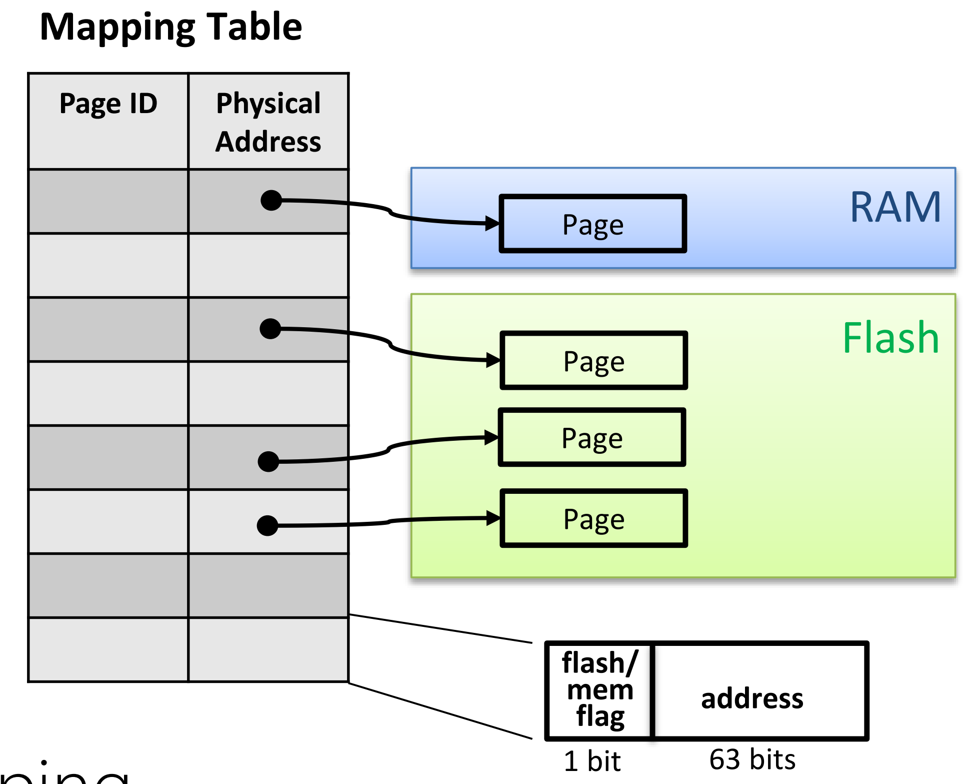
- Expose API
- B-tree search/update logic
- In-memory pages only

- Logical page abstraction for B-tree layer
- Moves pages between memory and flash as necessary

- Reads/Writes from/to storage
- Storage management

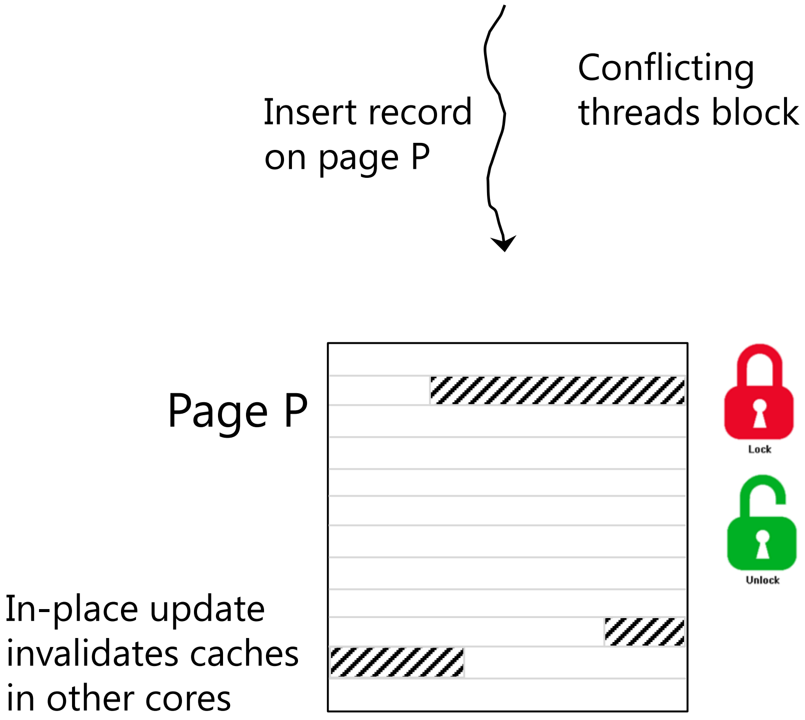
The Mapping Table

- Expose logical pages to the access method layer
  - Translates logical page ID to physical address
- Helps to isolate updates to a *single* page
- Central data structure for multi-threaded concurrency control
- Also used for log-structured store mapping
- Updated in lock-free manner [using compare-and-swap (CAS)]

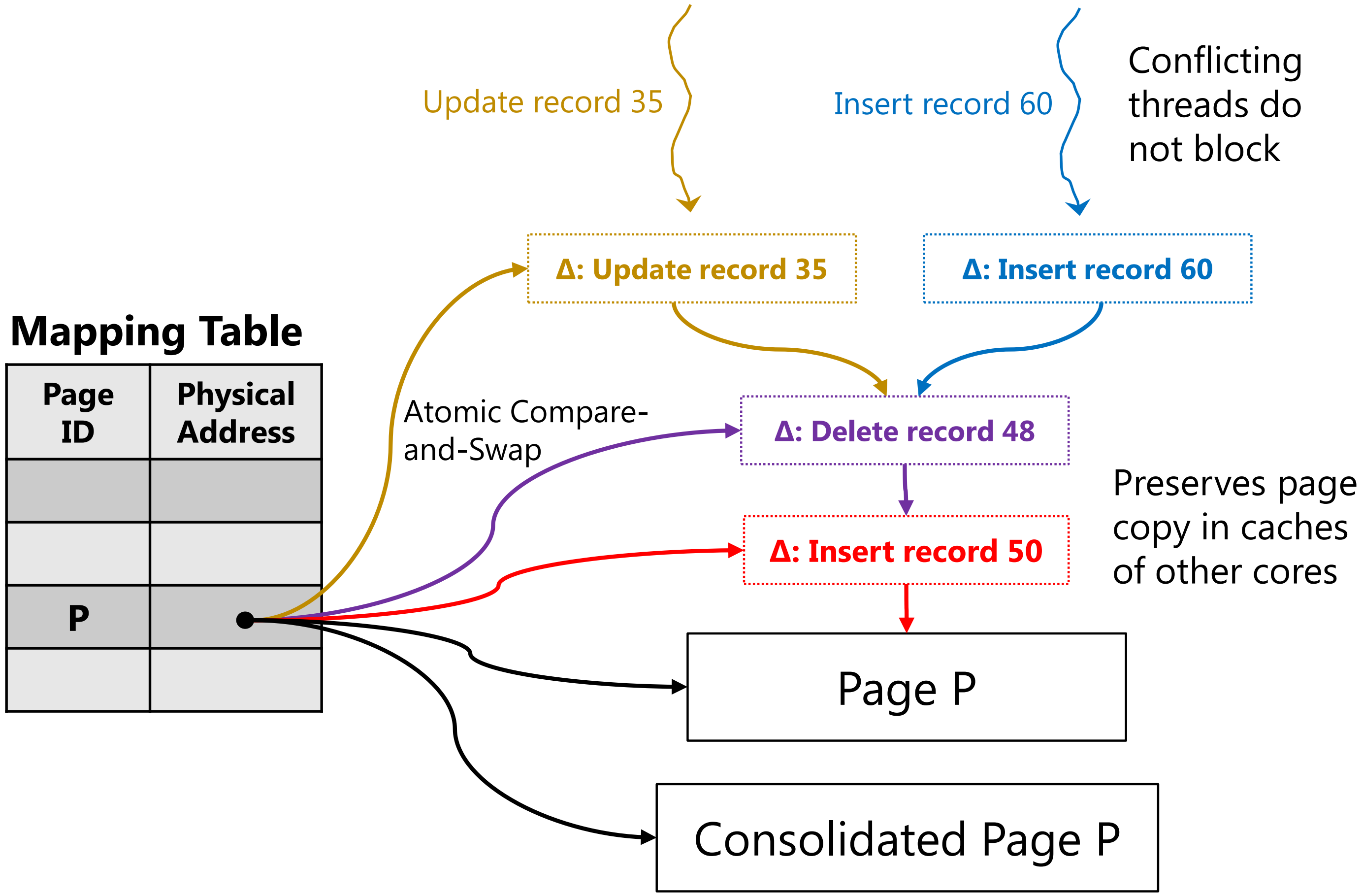


Highly Concurrent Page Updates with Bw-Tree

Classical B-Tree

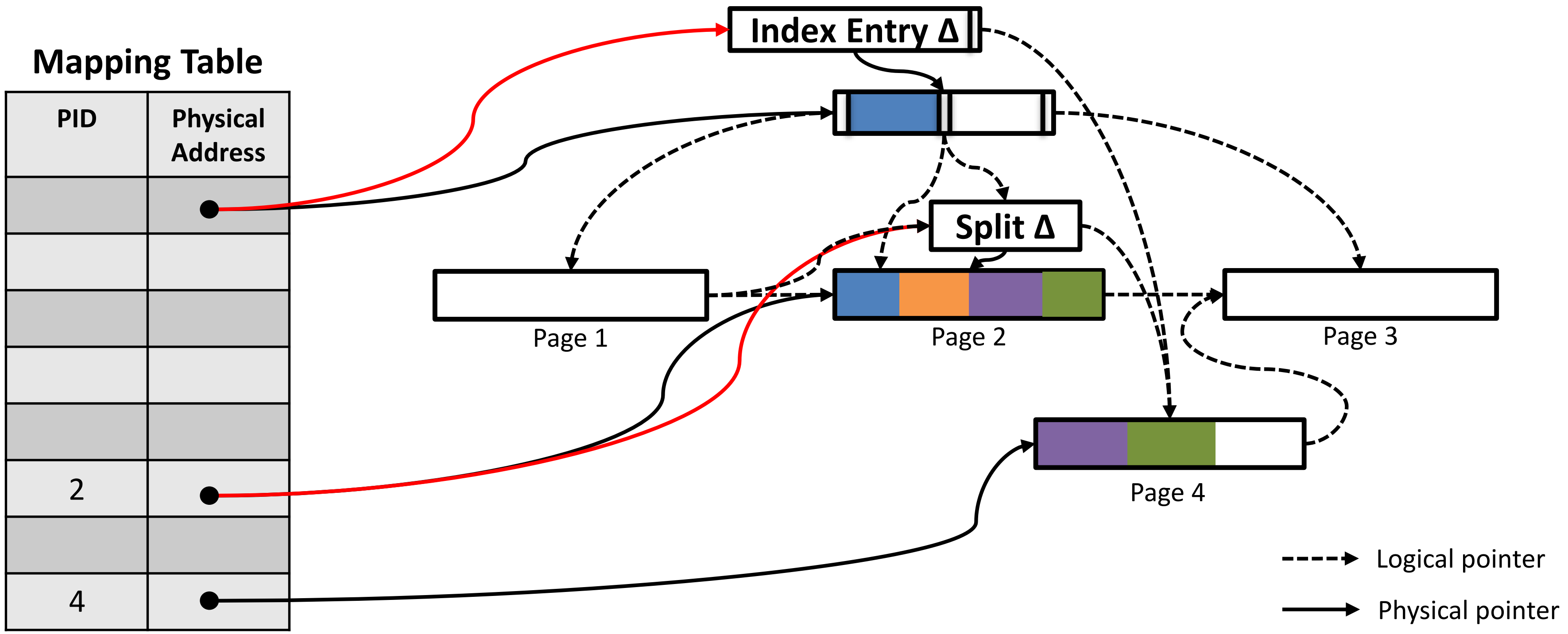


Bw-Tree



Bw-Tree Page Split

- No hard threshold for splitting unlike in classical B-Tree
  - Bw-Tree pages are elastic
- B-link structure allows "half-split" without locking
  - Install split at child level by creating new right page and linking left page to it
  - Install new separator key and pointer at parent level



Flash SSDs: Log-Structured Storage

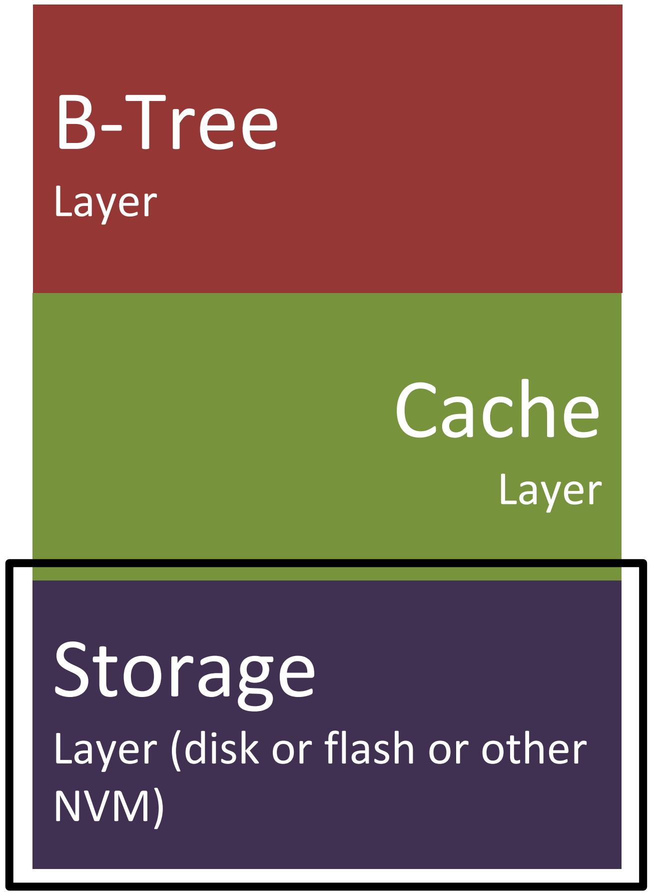
- Exploit benefits of flash and work around its quirks
  - Random reads are fast (10 – 100μsec)
  - Random writes (“in-place updates”) are expensive

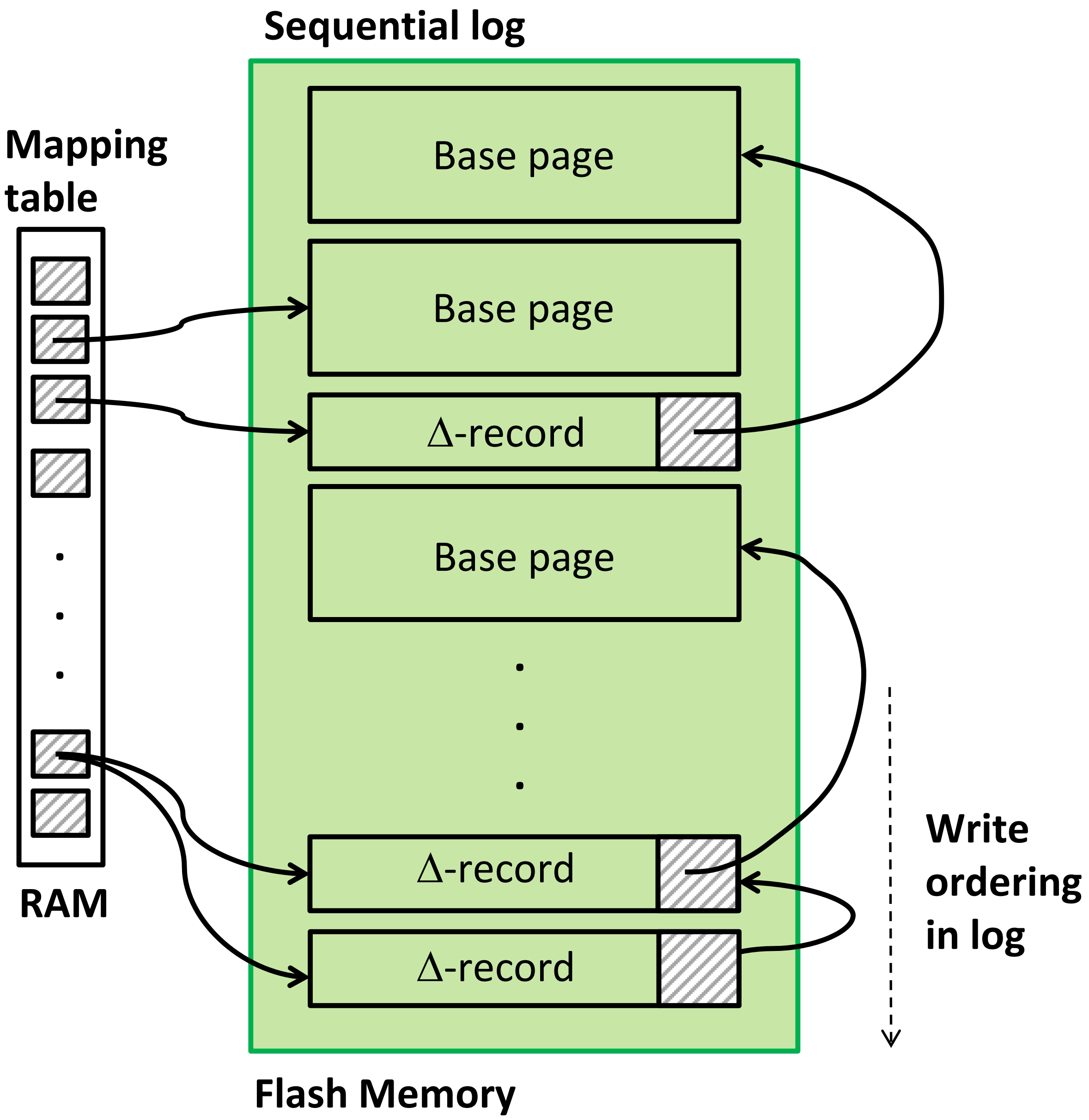
    -> Flash Translation Layer (FTL) cannot hide or abstract away device constraints
- Use flash in a log-structured manner

FusionIO 160GB ioDrive

LLAMA Log-Structured Store

- Suitable for flash + other benefits
- Amortize cost of writes over many page updates
  - Aggregate large amounts of new/changed data and append to the log in a single I/O
- Multiple random reads to fetch a "logical page"
  - Okay for flash, in the order of few tens of usec
- Works well for hard disks also
  - Benefit of amortizing page write cost
  - Random reads incur seek latency but mitigated by capturing working set of pages in RAM



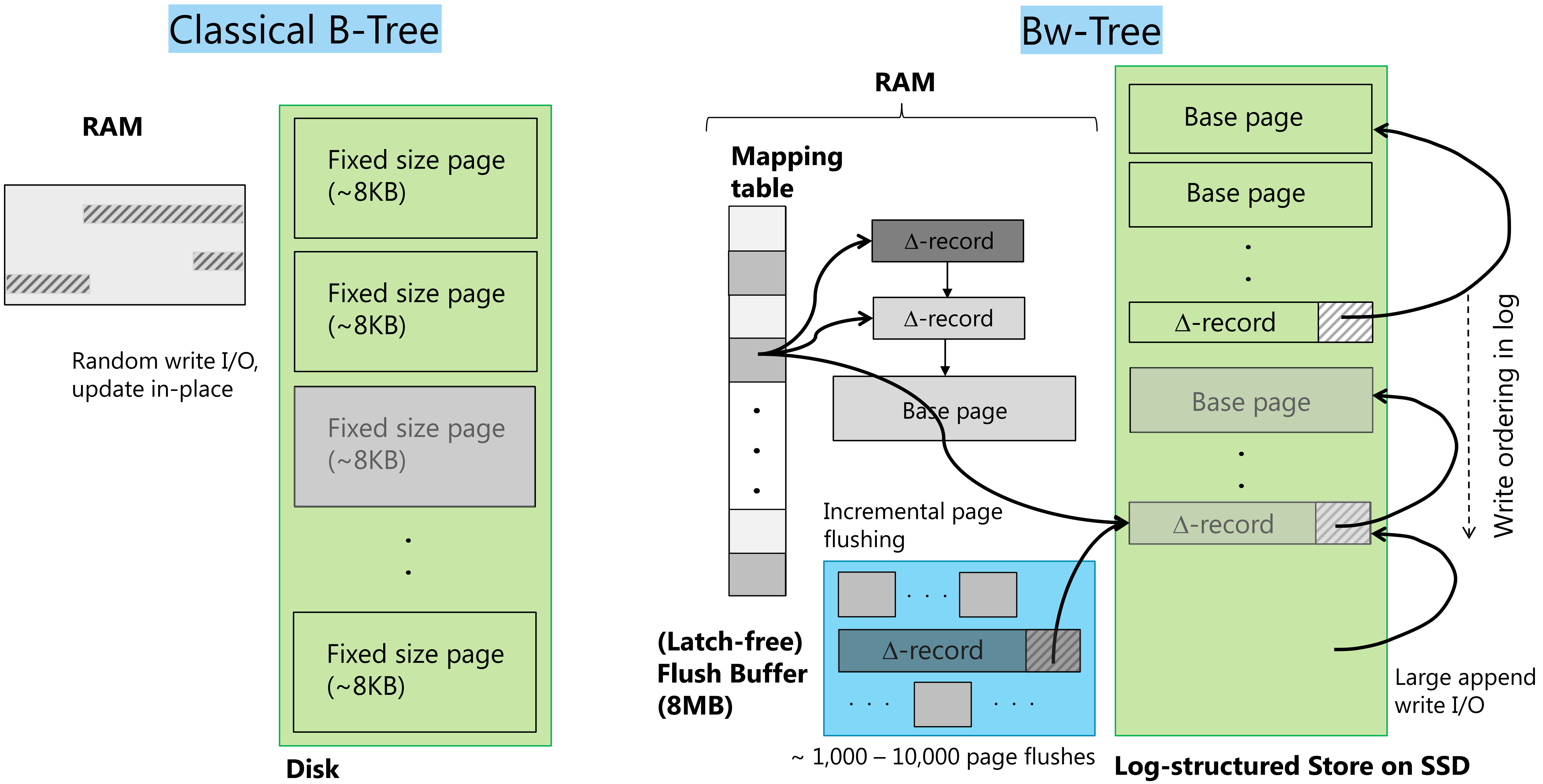


Sequential log
Mapping table
Base page
Base page
Δ-record
Base page
Δ-record
Δ-record
RAM
Write ordering in log
Flash Memory

Departure from Tradition: Page Layout on Flash

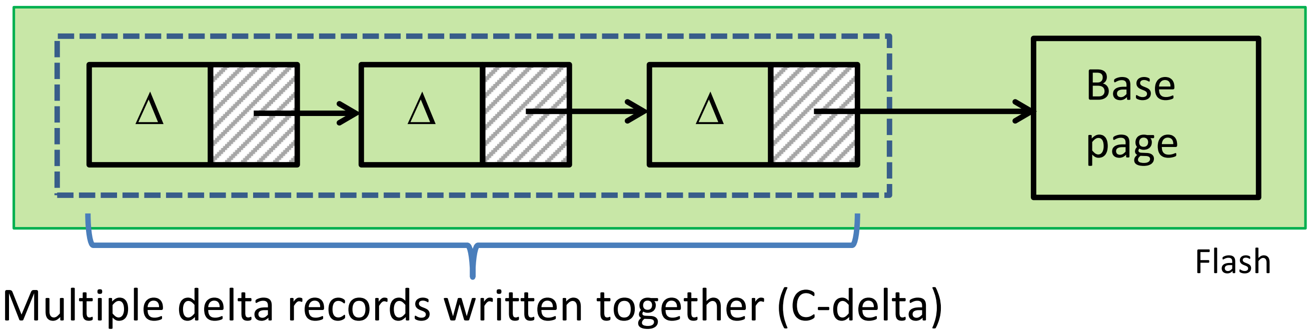
- Logical pages are formed by linking together records on possibly different physical pages
  - Logical pages do *not* correspond to whole physical pages on flash
  - Physical pages on flash contain records from multiple logical pages
- Exploits random access nature of flash media
  - No disk-like seek overhead in reading records in a logical page spread across multiple physical pages on flash
- Adapted from SkimpyStash (ACM SIGMOD 2011)

Write Optimized Storage Organization w/ Bw-Tree



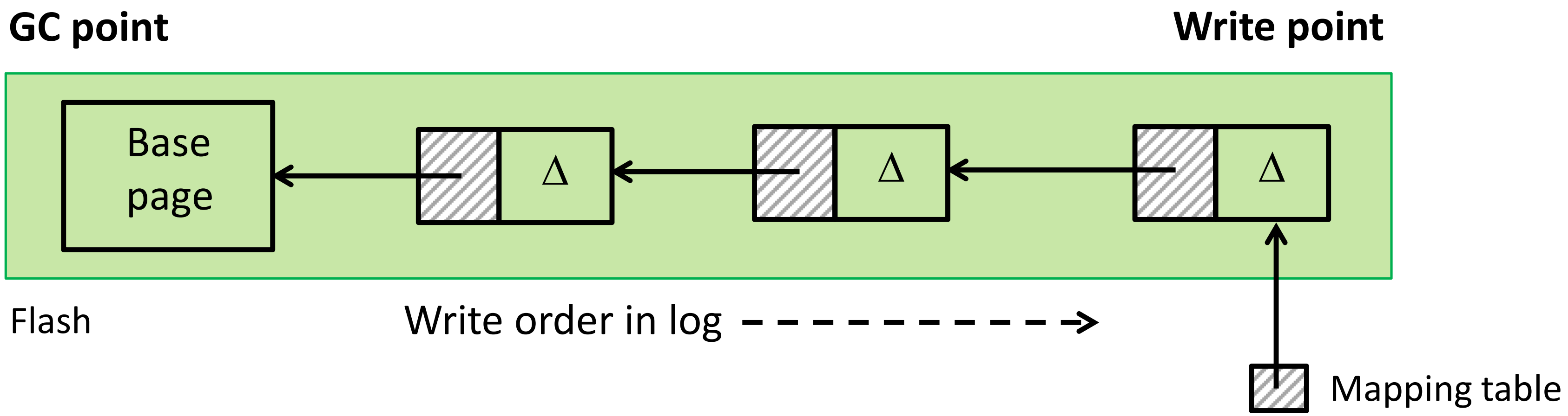
LLAMA: Optimizing Logical Page Reads

- Reading a “logical” page may involve reading delta records from multiple physical pages
  - Probably okay because of fast random access property of flash
  - Mitigated by capturing working set of pages in memory
- But we can reduce read I/Os further
  - Multiple delta records, when flushed together, are packed into a contiguous unit on flash (C-delta)
  - Pages consolidated periodically in memory also get consolidated on flash when they are flushed



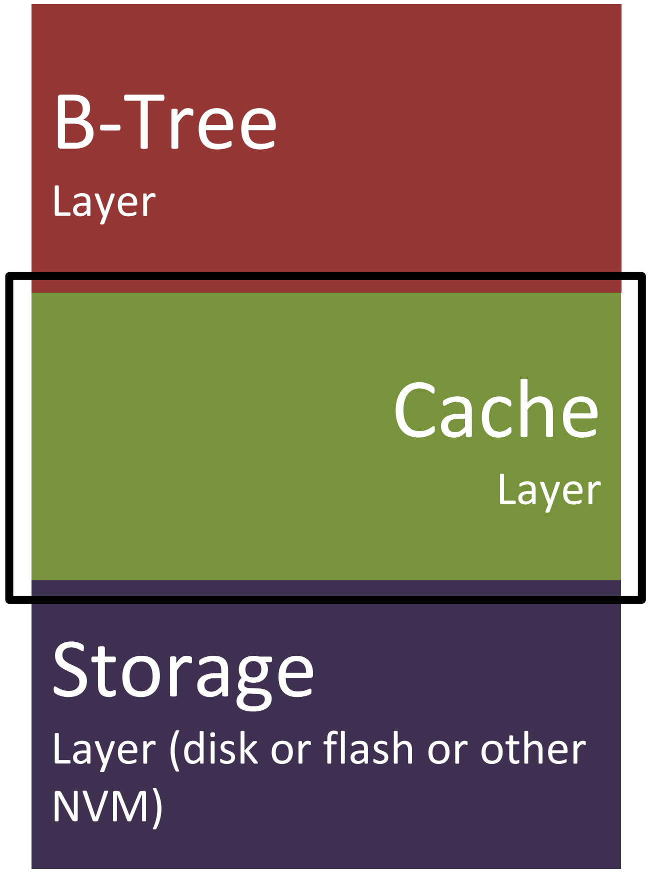
LLAMA: Garbage Collection on Flash

- Two types of record units in the log
  - Valid – Reachable from the flash offset in the mapping table
  - Orphaned – not reachable
- Garbage collection starts from oldest portion of log
  - Earliest written record (base page) on a “logical” page is encountered first
  - Avoid cascaded pointer updates up the chain => relocate entire logical page at a time, use this opportunity to consolidate



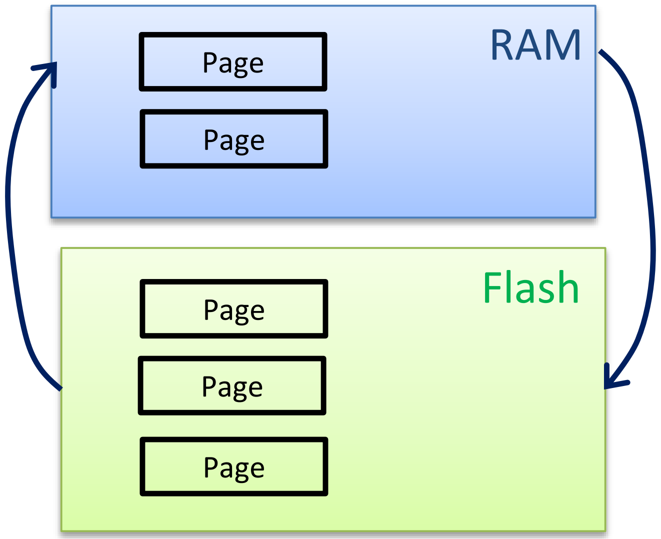
LLAMA: Cache Layer

- Provide abstraction of logical pages to access method layer
  - Mapping table containing RAM pointers or flash offsets
- Read pages into RAM from stable storage
- Flush pages to stable storage
  - Writes to flash ordered through flush buffers
- Swapout pages to reduce memory usage



LLAMA: Page Swapout

- Attempt to swapout pages when memory usage exceeds configurable threshold
- Uses variant of CLOCK algorithm
- Parallel page swapping functionality
  - Each accessor to Bw-Tree does small amount of page swapping work (“CLOCK sweep”) if needed
- RAM pointer replaced by flash offset in mapping table
- Page structure deallocated using epoch based memory garbage collection

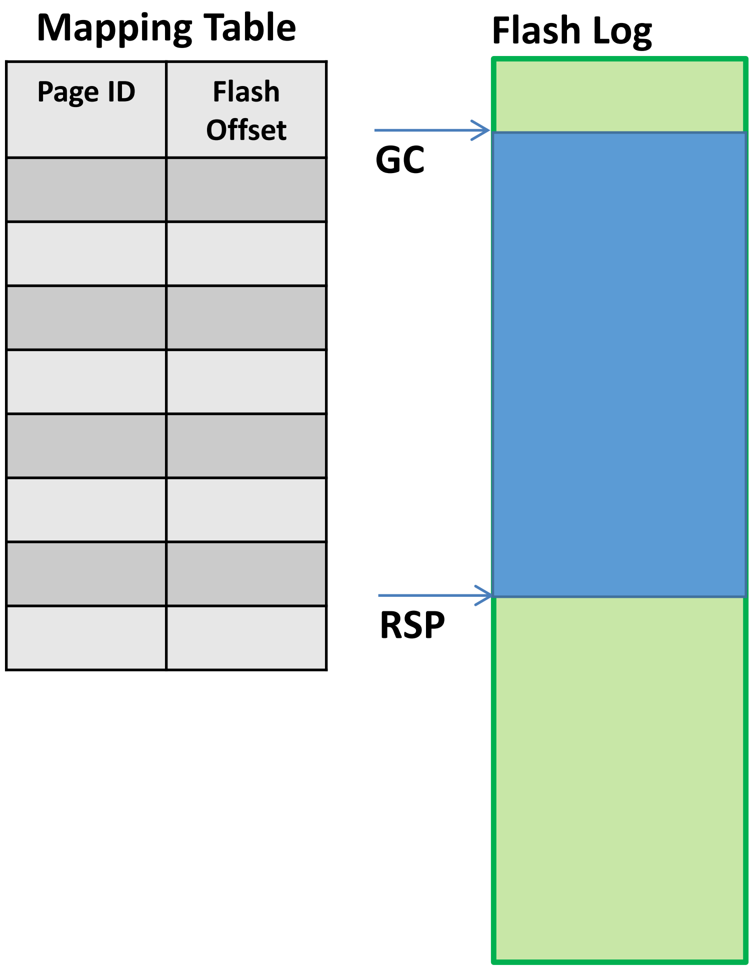


Bw-Tree/LLAMA Checkpointing

- B-Tree layer checkpointing (for durability)
  - Flush pages to flush buffer and subsequently to storage
- LLAMA checkpointing (for fast recovery)
  - Write the mapping table to flash

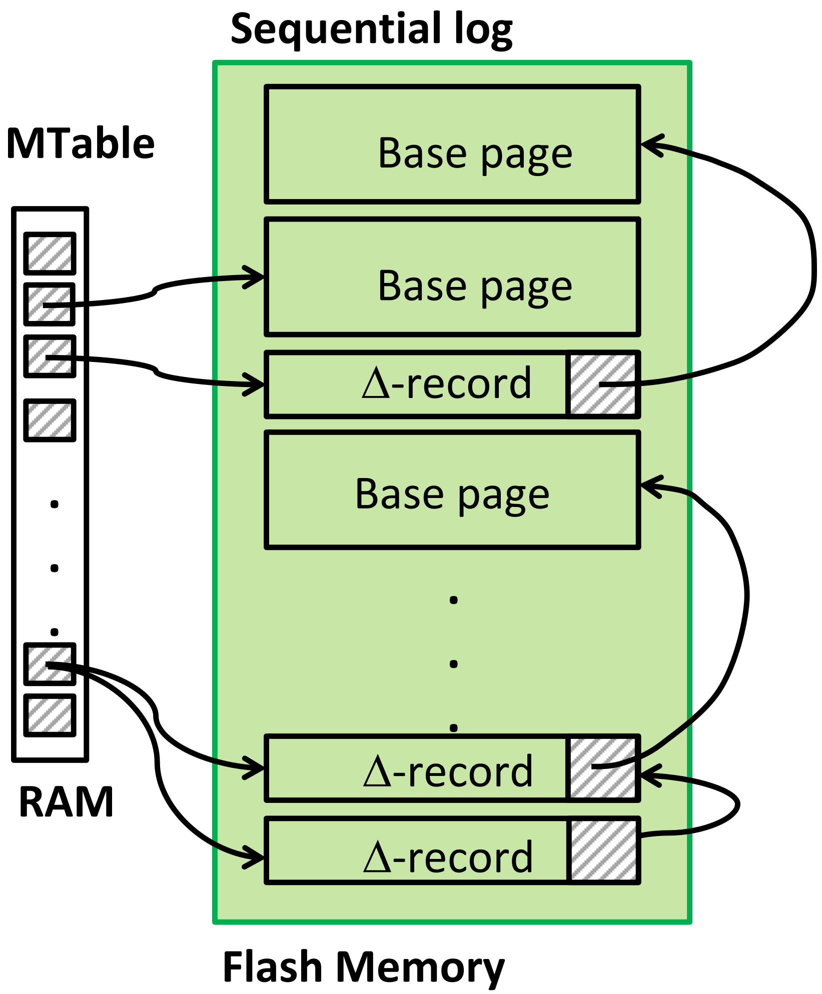
    -> When an entry contains RAM address, obtain flash address from the in-memory page

    -> Unused entries are written as zeroes
  - Record write position in log when the checkpoint started
  - Alternate between two fixed regions on flash for each checkpoint

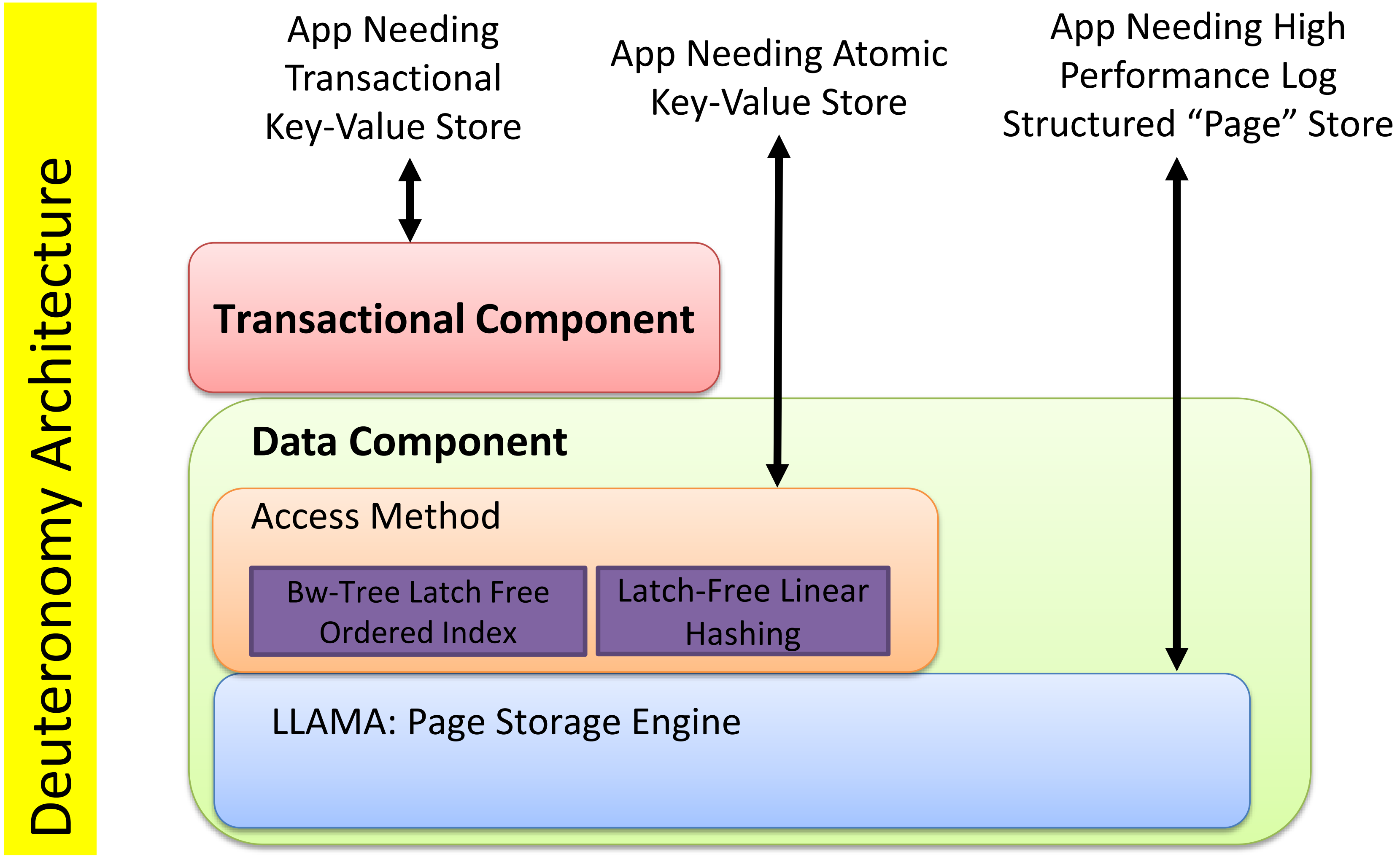


Bw-Tree Fast Recovery

- Restore mapping table from latest checkpoint region
- Scan from log position recorded in checkpoint to end of log
  - Read page ID from C-delta on log and update flash offset in mapping table
- Restore Bw-tree root page LPID
- Optimizations for fast cache warm-up

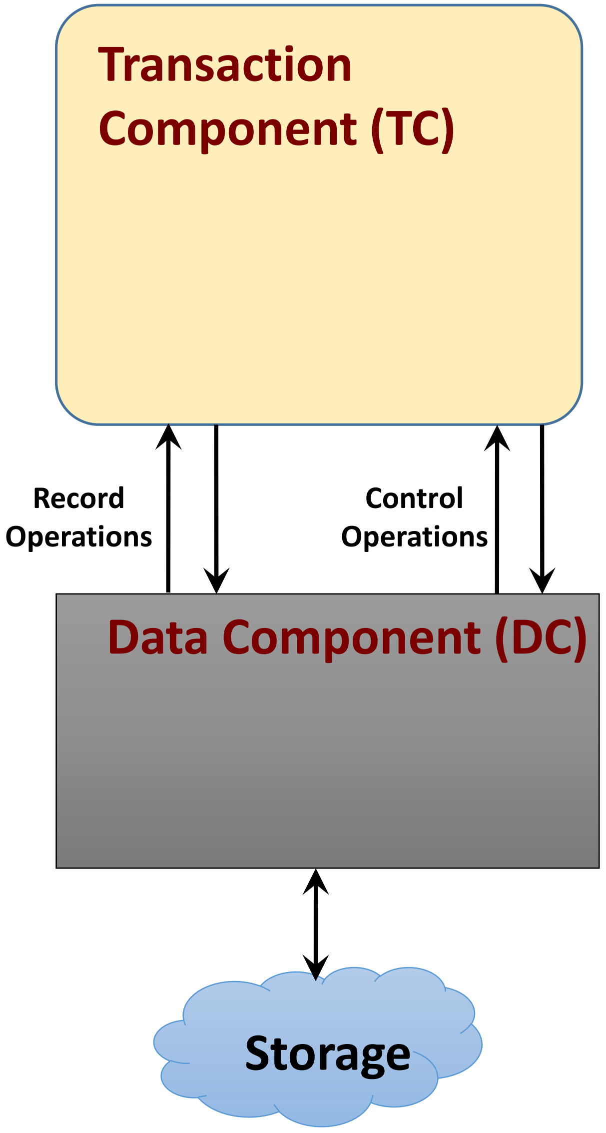


Bw-Tree: Support for Transactions



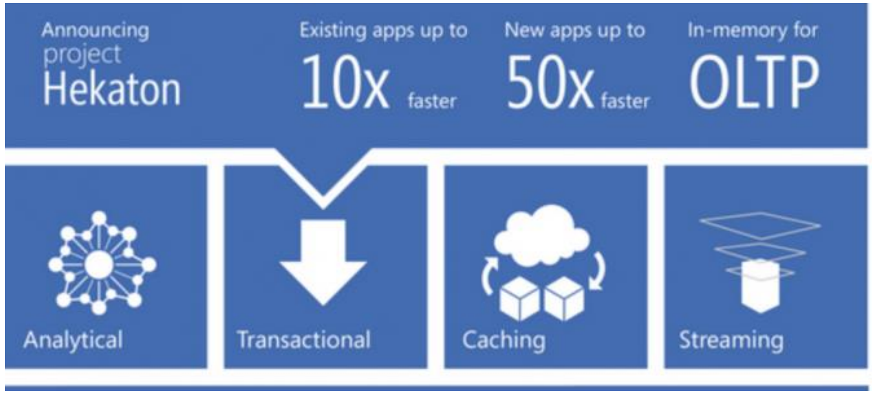
End-to-end Crash Recovery

- Data Component (DC) recovery
  - Bw-Tree fast recovery as described
- Transactional Component (TC) recovery
  - Helps to recover unflushed data at DC “up to” end of stable log (WAL) at time of crash
  - Requires DC to recover to a logically consistent state first

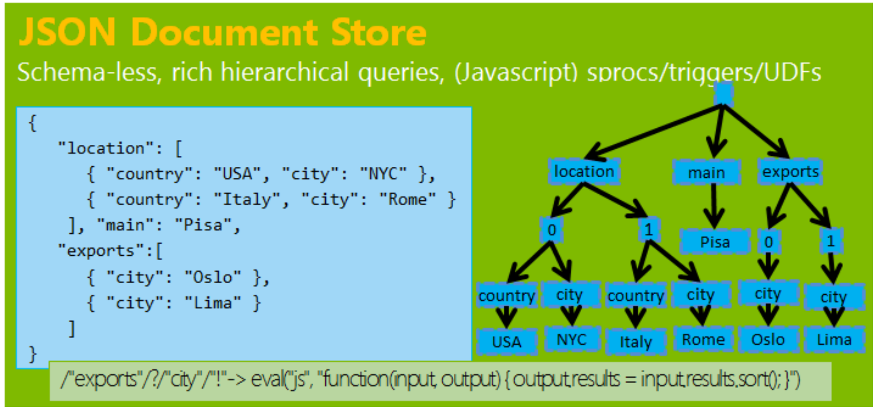


Bw-Tree in Production

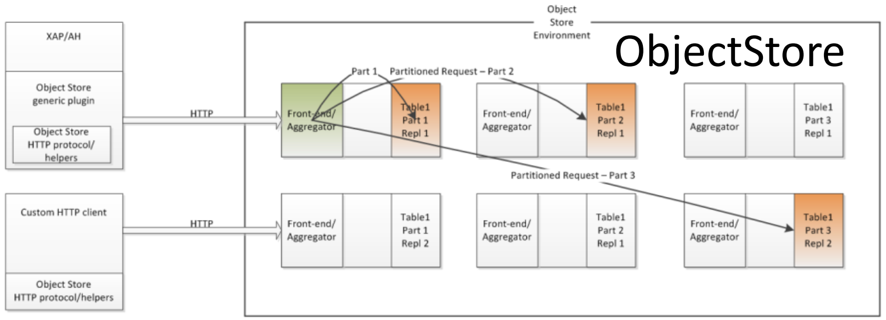
- Key-sequential index in SQL Server Hekaton
  - Lock-free for high concurrency, consistent with Hekaton's overall non-blocking main memory architecture



- Indexing engine in Azure DocumentDB
  - Rich query processing over a schema-free JSON model, with *automatic indexing*
  - Sustained document ingestion at high rates



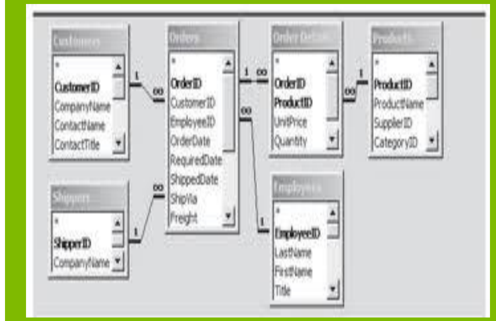
- Sorted key-value store in Bing ObjectStore
  - Support range queries
  - Optimized for flash SSDs



DocumentDB

Relational Stores

Fully schematized, relational queries, transactions (e.g., SQL Azure, Amazon RDS, SQL IaaS)



Key-Value/ Column Family Stores

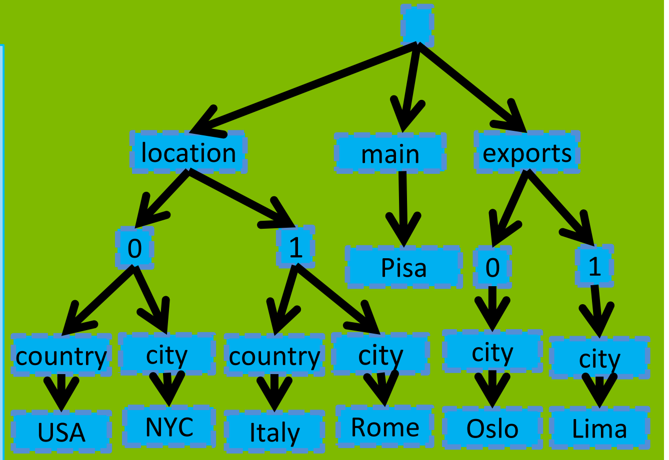
Schema-less with opaque values, lookups on keys (e.g., Azure Tables, HBASE, BigTable, LevelDB, Cassandra, …)

Key	Value
k1	XML
k2	.NET
k3	Java

(JSON) Document Stores

Schema-less, rich hierarchical queries, (Javascript) sprocs/triggers/UDFs (e.g. MongoDB, CouchDB, Espresso, …)

```
{
   "location": [
      { "country": "USA", "city": "NYC" },
      { "country": "Italy", "city": "Rome" }
    ], "main": "Pisa",
   "exports":[
      { "city": "Oslo" },
      { "city": "Lima" }
    ]
}
```




```
/"exports"/?/"city"/"!"-> eval("js", "function(input, output) { output.results = input.results.sort(); }")
```

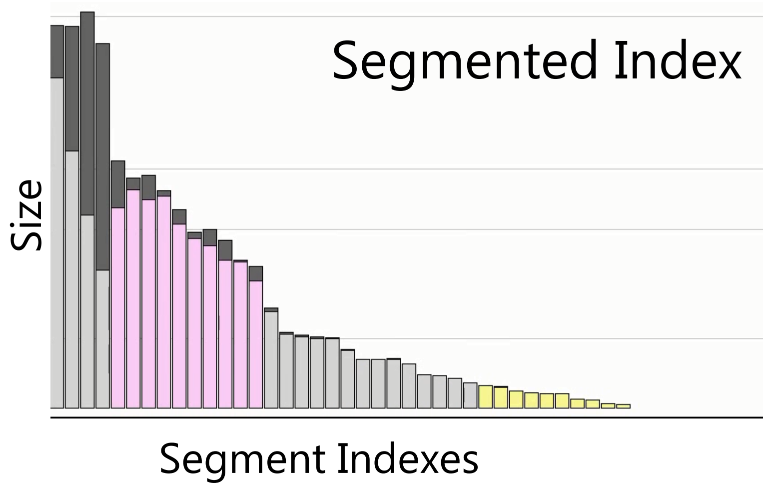
DocumentDB: Differentiated Feature set

- Formal query model optimized for queries over schema-less documents at scale
  - Support for relational and hierarchical projections
- Consistent indexing in face of rapid, sustained high volume writes (optimized for flash SSDs)
- Developer tunable consistency-availability tradeoffs with SLAs
- Low latency, (Javascript) language integrated, transactional CRUD on storage partitions
- Elastic scale, resource governed, multi-tenant PaaS

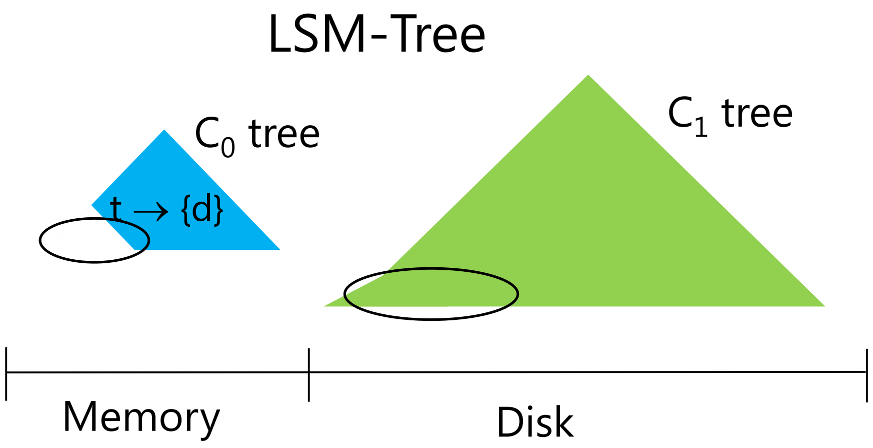
In Search for the ~~Right~~ Write Index

Consistent Indexing over schema-less documents is an overly constrained design space

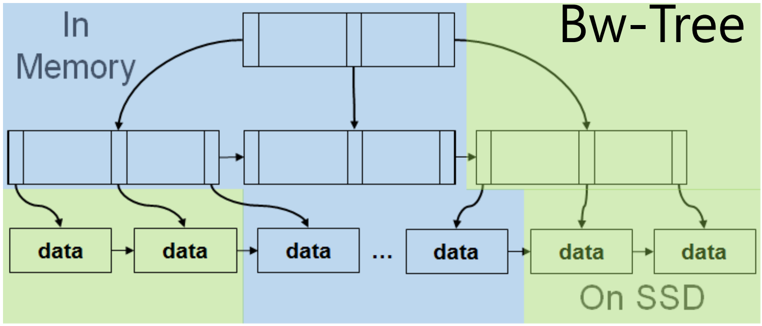
- Sustained high volume writes <u>without</u> any term locality
- Queries should honor various consistency levels
- Extremely high concurrency
- Multi-tenancy with strict, reservation based, sub-process level resource governance



- Segment indexes written one by one as documents arrive
- Queries need to scan multiple segments, hence **tradeoff between query time and freshness**
- Background merge job => leads to **high write amplification** (> 10x).



- Insertions batched in C0 to reduce write I/Os
- Periodic **bulk merge** of key ranges in C0 with those in C1 and made durable to C1
- **No semantic value merge** support
- **Read I/O before write** => **Slows down insertions**



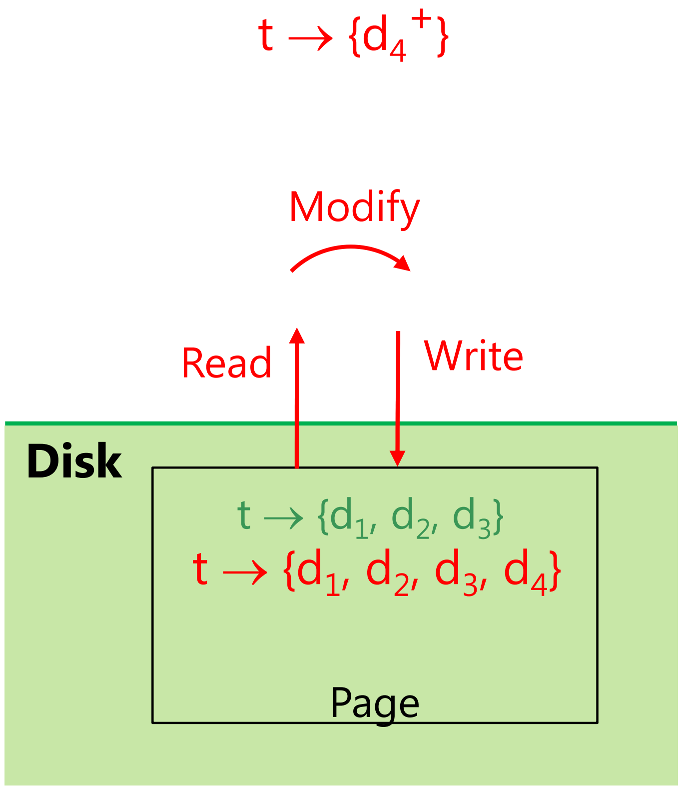
- Highly concurrent: **latch-free**
- **Incremental durability via batch writes**, but no large merges
- Blind incremental updates: **avoid read-before-write**
- **Consistent indexing** in face of **sustained document ingestion**
- **Optimized for SSD** (works well for HDD)
- Flexible **resource governance**

Efficient Index Updates with Bw-Tree

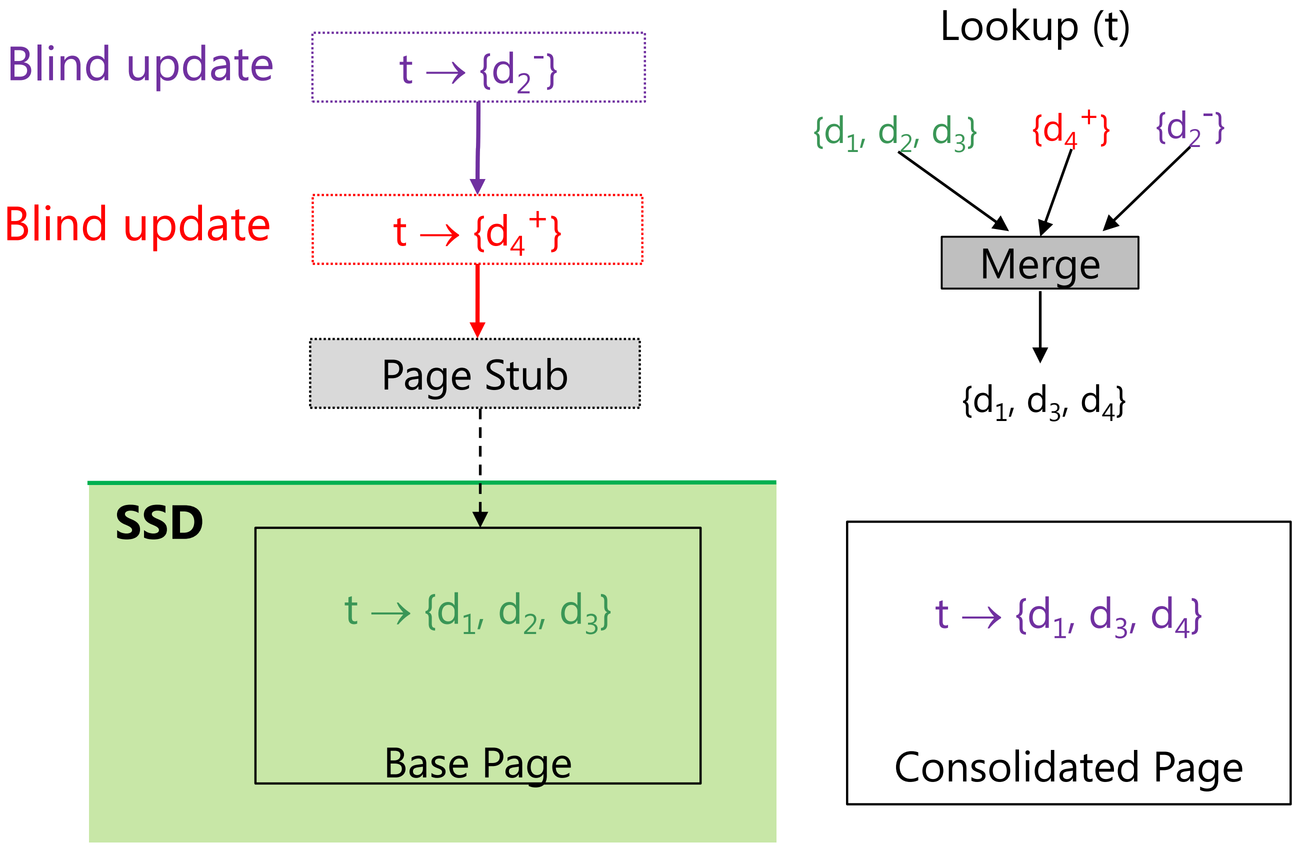
Key Challenges

- Index update: No key locality; cannot afford a Read to do the Write; low Write Amplification
- Queries: Low Read Amplification
- Frugal resource budget

Classical B-Tree

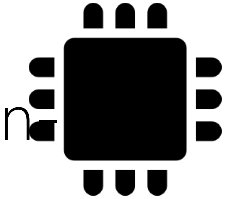


Bw-Tree



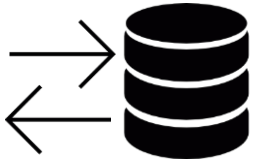
Bw-Tree Resource Governance

- CPU resource governance
  - Threads calling into Bw-Tree do not block (upon I/O or memory page access)
  - Top-level scheduler controls thread budget per replica



- Memory resource governance
  - Dynamically configurable buffer pool limit

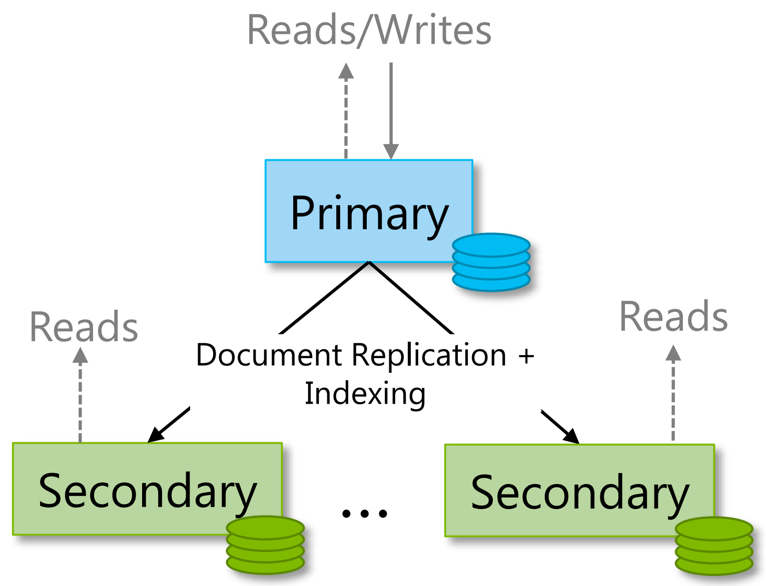
- IOPS resource governance
  - Check resource usage before issuing I/O, retry after dynamically computed timeout interval

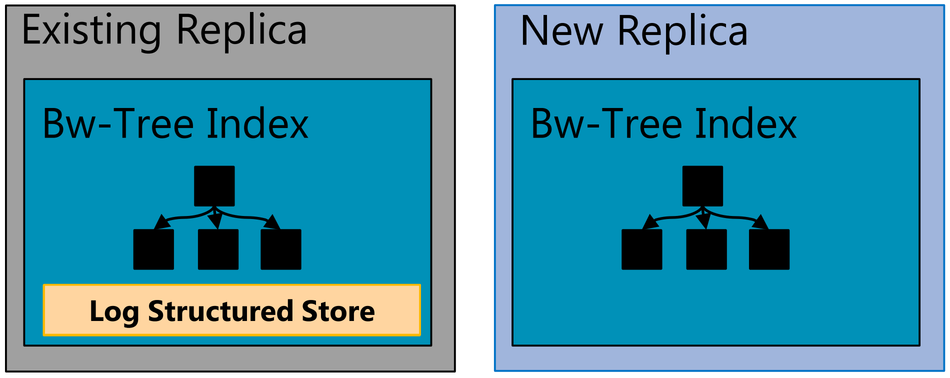


- Storage resource governance
  - LLAMA log-structured store can grow/shrink dynamically
  - Self-adjusting based on logical data size

Bringing up Bw-Tree Replica

- Obtain Bw-Tree physical state stream from primary
  - LLAMA checkpoint file (most recent)
  - Valid portion of LLAMA log (between GC and write points)
- Bring up Bw-Tree using fast recovery
- Catch up with primary
  - Replay logical operations from primary with LSNs upward of last (contiguous) LSN in recovered Bw-Tree





Bw-Tree: Summary

- Classical B-Tree redesigned from ground up for modern hardware and cloud
  - Lock-free for high concurrency on multi-core processors
  - Delta updating of pages in memory for cache efficiency
  - Log-structured storage organization for flash SSDs
  - Flexible resource governance in multi-tenant setting
  - Transactional component can be layered above as part of Deuteronomy architecture
- Shipping in Microsoft's server/cloud offerings
  - Key-sequential index in SQL Server Hekaton
  - Indexing engine in Azure DocumentDB
  - Sorted key-value store in Bing ObjectStore
- Going forward
  - Layer a transactional component on top as per Deuteronomy architecture (CIDR 2015, VLDB 2016)
  - Open-source the codebase