University of Cyprus - Department of Computer Science

EPL646: Advanced Topics in Databases

Georgiou Zacharias and *Paschalides Demetris*
[zgeorg03, dpasch01]@cs.ucy.ac.cy

***Reference**:*
Let's Talk About Storage Recovery Methods for Non-Volatile Memory Database Systems, Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015.In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM, New York, NY, USA, 707-722. DOI: https://doi.org/10.1145/2723372.2749441

Overview

1. Introduction
2. Background Work
   - 2.1. Motivation
3. DBMS Testbed and NVM-Aware Engines
   - 3.1. In-Place Updates Engine
   - 3.2. Copy-on-Write Updates Engine
   - 3.3. Log-structured Updates Engine
4. Experimental Analysis
   - 4.1. Benchmarks
   - 4.2. Runtime Perf
   - 4.3. Reads & Writes
   - 4.4. Recovery
   - 4.5. Execution Time Breakdown
   - 4.6. Storage Footprint
5. Conclusion

Introduction

- New OLTP applications support larger number of concurrent users/systems because of the scale to which ingest information.
- Performance is affected by how fast the system reads/writes data.
- DBMSs always dealt with the tradeoffs between ***volatile*** and ***nonvolatile*** storage devices because recovery logs need to be written in non-volatile devices
- HDDs and SSDs are such devices but are slow and support bulk data transfers as blocks.

Introduction

- DRAM would be great for such workloads but it is a volatile memory and also consumes a lot of energy

“ **DRAM - D**ynamic **R**andom **A**ccess **M**emory is a type of memory that is typically used for data or program code that a computer processor needs to function. DRAM is a common type of random access memory used in personal computers, workstations and servers. “ - *Wikipedia*

- Flash-Based SSDs have better storage capabilities and less energy consumption but slower than DRAM
- Flash-Based SSDs also have Block-Based access methods, writing single byte the DBMS must write the change as a block - *4KB*
- Problematic for OLTPs since they do a lot of small changes

Introduction

- NVM - Non-Volatile Memory will fundamentally change the dichotomy between memory and durable/persistent storage in the DBMSs

“ **Non-Volatile Memory** is a type of computer memory that can retrieve stored information even after having been turned off and back on - power cycled. The opposite of non-volatile memory is volatile memory which needs constant power in order to prevent data from being erased. “ - *Wikipedia*

- NVM is a blend of flash-based SSD and DRAM and provides low latency persistent reads and writes
- Usage of NVM-only hierarchy in OLTP DBMS will be evaluated in this paper presentation

About Data Structures

- **SSTable:**
  - Sorted string table, key-value storage sorted by keys
  - On-disk data structure and is always immutable
  - Immutable - appropriate only for storing static data. Bloom filters are used to reduce reads
  - Internally contains a sequence of blocks, each block is 64 KB in size
- **Bloom Filter** [1]**:**
  - Space-efficient probabilistic data structure
  - A query returns either "possibly in set" or "definitely not in set"
  - Sufficient core memory, an error-free hash used to eliminate all unnecessary disk accesses.
  - Fewer than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set
- **MemTable:**
  - An in-memory SSTable with the contents loaded in RAM
- **LSM Tree:**
  - A data structure with performance characteristics that make it attractive for providing indexed access to files with high insert volume, such as transactional log data

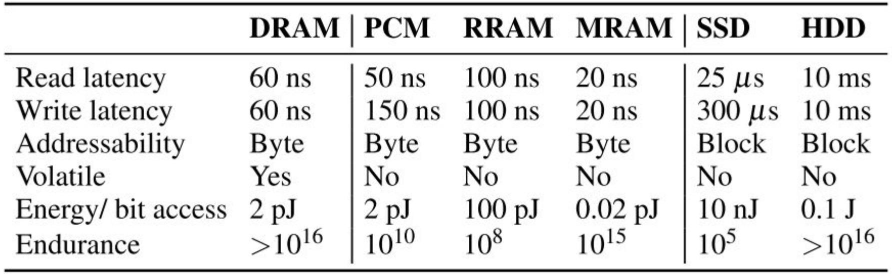
Motivation

Disk-oriented e.g., IBM’s System R [2]

- Manage blocks of tuples on disk using in-memory cache

Memory-oriented e.g., IBM’s IMS/VS Fast Path [3]

- Updates on in-memory data and relies on the disk to ensure durability



	DRAM	PCM	RRAM	MRAM	SSD	HDD
Read latency	60 ns	50 ns	100 ns	20 ns	25 μs	10 ms
Write latency	60 ns	150 ns	100 ns	20 ns	300 μs	10 ms
Addressability	Byte	Byte	Byte	Byte	Block	Block
Volatile	Yes	No	No	No	No	No
Energy/ bit access	2 pJ	2 pJ	100 pJ	0.02 pJ	10 nJ	0.1 J
Endurance	$>10^{16}$	10^{10}	10^{8}	10^{15}	10^{5}	$>10^{16}$

Previous studies have shown that the overhead of managing this data movement for OLTP workloads is considerable

NVM technologies, remove tuple transformation and propagation costs **through byte-addressable loads and stores with low latency**

- Unlike DRAM, all writes to the NVM are **potentially durable** and therefore a DBMS can access the tuples directly in the NVM after a restart or crash without the need to reload the database

Previous work showed that the two architectures achieve almost the same performance when using NVM because of the overhead of logging [4]

Authors seek to understand the characteristics of different storage and recovery methods

NVM Hardware Emulator

NVM storage devices are prohibitively expensive and only support small capacities

- **Use of a NVM hardware emulator**
- Emulator supports tunable read latencies and read/write bandwidths

Allocator Interface

- POSIX **malloc** interface
- Use of hardware write barrier primitive (SFENCE) to guarantee the durability of writes to NVM
- **NEGATIVE**: No naming mechanism that is valid after a system restart

Filesystem Interface

- POSIX **filesystem** interface (read/write)
- Emulator exposes a NVM-backed volume to the OS through an optimized for non-volatile memory filesystem
- **POSITIVE:** Supports a naming mechanism that ensures file offsets are valid after restart
- **NEGATIVE:** Requires the application's writes to go through kernel's VFS

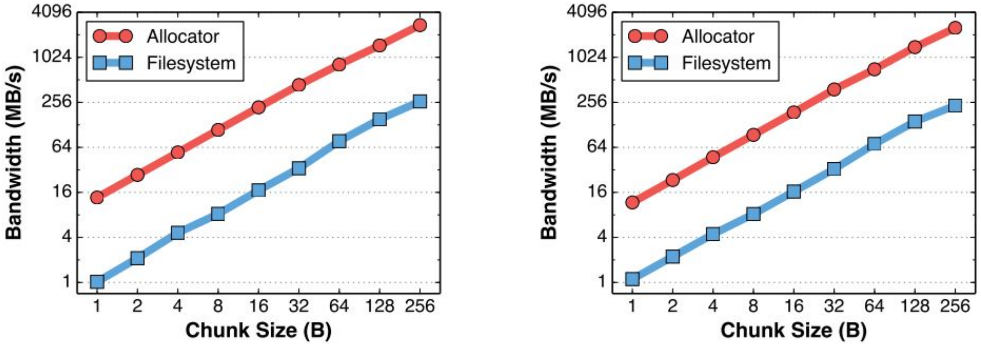
NVM-aware Memory Allocator

Should provide durability

- Necessary because the changes made by a transaction to a location on NVM may still reside in volatile CPU caches when the transaction commits
- This is achieved using ***CLFLUSH*** and ***SFENCE*** instructions

Must provide a naming mechanism

- Ensure that pointers still point to valid locations after a system restart
- Achieved by the use of non-volatile pointers

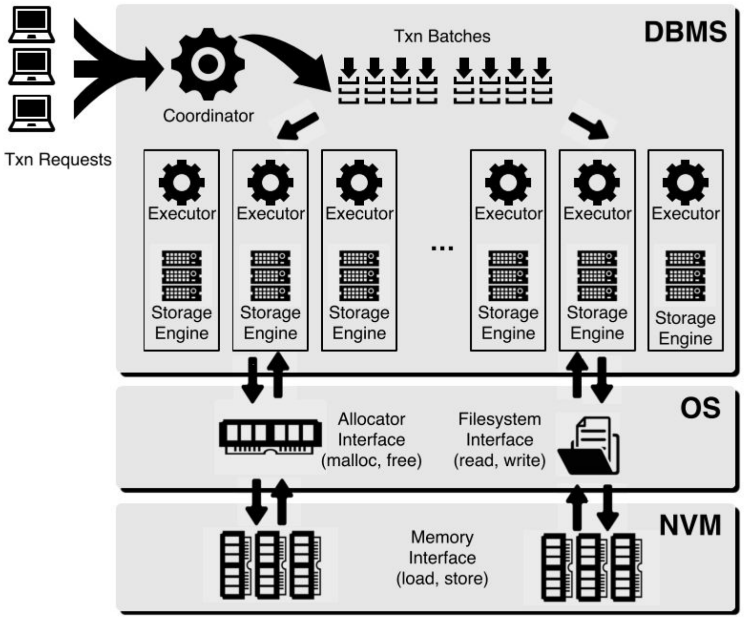


(a) Sequential Writes

(b) Random Writes

DBMS Testbed

- Developed a lightweight DBMS to evaluate different storage architecture designs for OLTP workloads
- The DBMS's internal coordinator receives incoming transaction requests from the application and then invokes the target stored procedure
- As a transaction executes in the system, it invokes queries to read and write tuples from the database
- These requests are passed through a query executor that invokes the necessary operations on the DBMS's active storage engine
- The DBMS uses *pthreads* to allow multiple transactions to run concurrently in separate worker threads
- 3 Storage Engines for durable updates are implemented


Txn Requests
Coordinator
Txn Batches
DBMS
Executor
Storage Engine
...
OS
Allocator Interface (malloc, free)
Filesystem Interface (read, write)
NVM
Memory Interface (load, store)

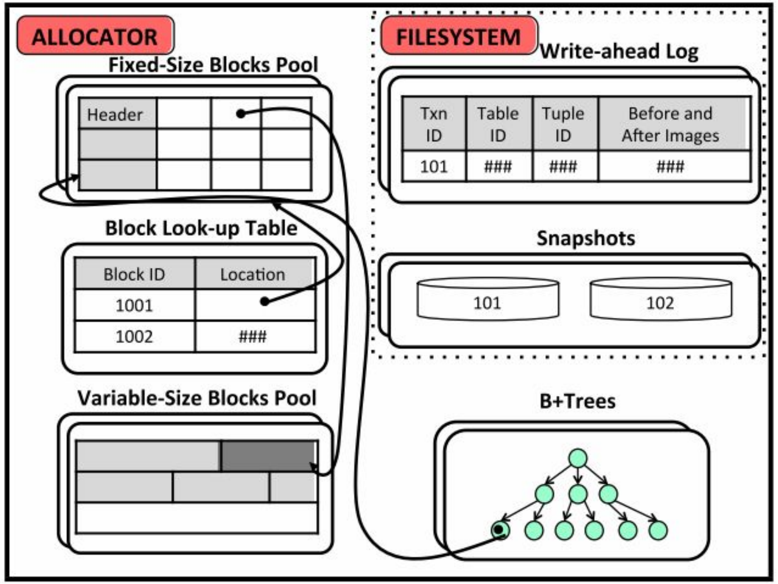
In-Place Updates Engine - InP

The most common storage engine strategy in DBMSs and the most efficient method of applying changes

- The system writes the new value directly on top of the original one
- Based on VoltDB [5], a memory oriented DBMS
- Uses STX B+tree library for its indexes [6]

Storage

- Both fixed-sized and variable-length blocks
- Unsorted tuples within blocks
- A list of unoccupied tuple slots for each table is maintained
- The engine uses the allocator interface to maintain the indexes and stores them in memory



Block ID	Location
1001	
1002	###

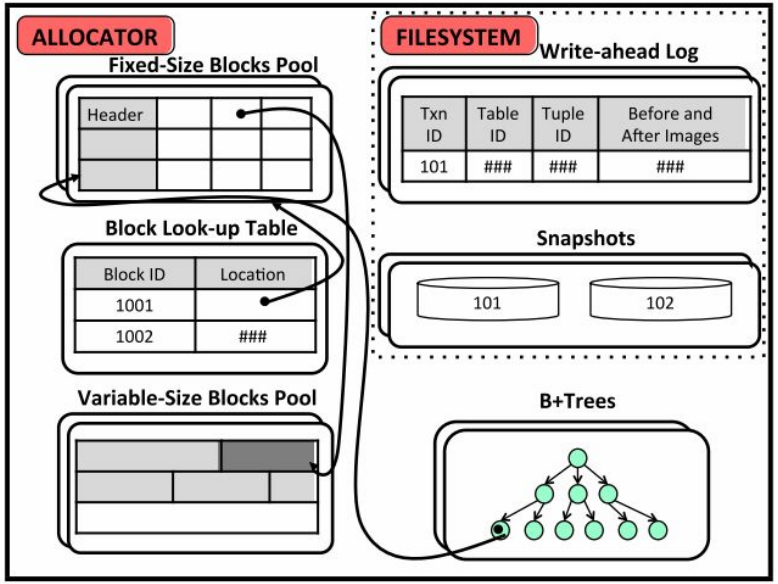
Txn ID	Table ID	Tuple ID	Before and After Images
101	###	###	###

(a) In-place Updates (InP)

In-Place Updates Engine - InP

Recovery

- InP maintains a durable **W**rite-**A**head **L**og - **WAL** in the file system
  - WAL records the transactions’ changes before they are applied
- **ARIES** recovery protocol is used
  - The engine periodically takes checkpoints that are stored on the filesystem to bound recovery latency and reduce the storage space
  - Authors compress (*gzip*) the checkpoints to reduce their storage footprint on NVM
- Changes made by uncommitted transactions at the time of failure are not propagated to the database



(a) In-place Updates (InP)

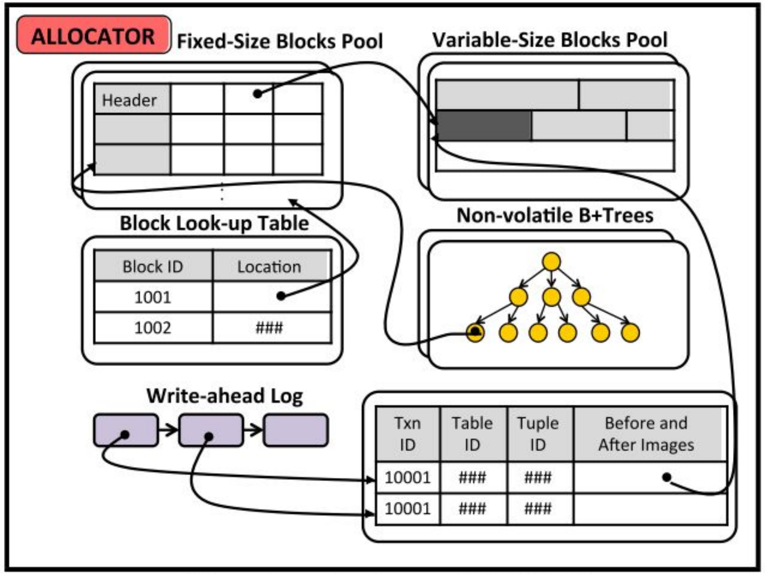
In-Place Updates Engine - NVM - InP

The InP engine’s logging infrastructure assumes that the system’s durable storage device has orders of magnitude higher write latency compared to DRAM

- Increases the mean response latency as transactions need to wait for the group commit operation
- NVM-InP engine only records a non-volatile pointer to the tuple in the WAL, rather than copying the tuple to the WAL

Storage

- The engine stores the WAL as a **non-volatile linked list**
- Appends using an **atomic write**
- After all of the transaction’s changes are safely persisted, the engine truncates the log



Block ID	Location
1001	
1002	###

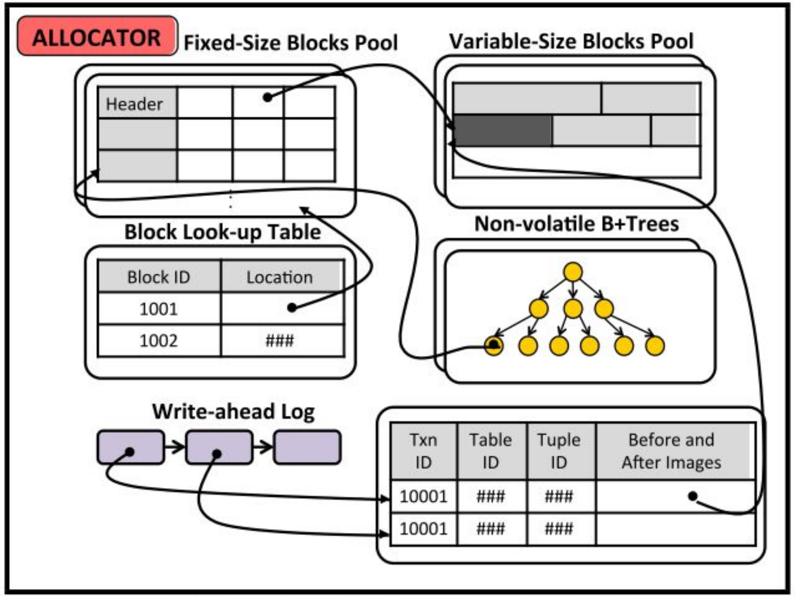
Txn ID	Table ID	Tuple ID	Before and After Images
10001	###	###	
10001	###	###	

(a) In-place Updates (NVM-InP)

In-Place Updates Engine - NVM - InP

Recovery

- Committed transactions are durable after the system restarts because the NVM-InP engine immediately persists the changes made by a transaction when it commits
- No need to replay the log during recovery
- Changes of uncommitted transactions may be present in the database because the memory controller can evict cache lines containing those changes to NVM at any time
    - It needs to undo those transactions using the WAL
- As this recovery protocol does not include a redo process, **the NVM-InP engine has a short recovery latency** that only depends on the number of uncommitted transactions



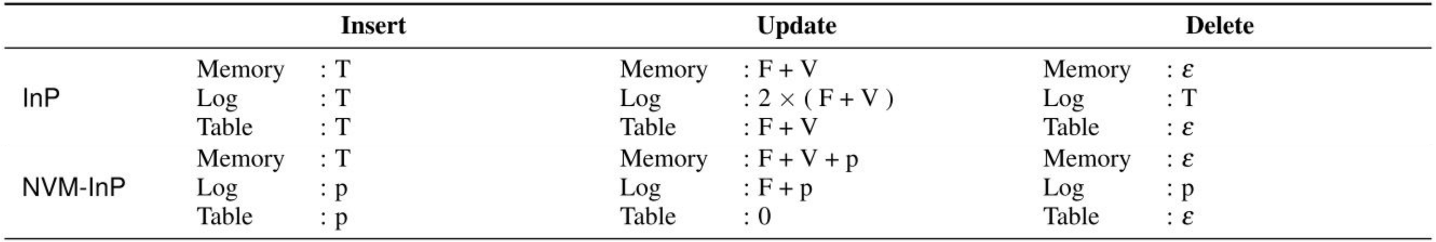
Block ID	Location
1001	●
1002	###

Txn ID	Table ID	Tuple ID	Before and After Images
10001	###	###	●
10001	###	###	

(a) In-place Updates (NVM-InP)

In-Place Updates Engine - NVM - InP

INSERT	UPDATE	DELETE	SELECT
■ Sync tuple with NVM ■ Record tuple pointer in WAL ■ Sync log entry with NVM ■ Mark tuple state as persisted ■ Add tuple entry in indexes	■ Record tuple changes in WAL ■ Sync log entry with NVM ■ Perform modifications on the tuple ■ Sync tuple changes with NVM	■ Record tuple pointer in WAL ■ Sync log entry with NVM ■ Discard entry from table and indexes ■ Reclaim space at the end of transaction	■ Find tuple pointer using index/table ■ Retrieve tuple contents



	Insert		Update		Delete	
InP	Memory	: T	Memory	: F + V	Memory	: ε
	Log	: T	Log	: 2 × (F + V)	Log	: T
	Table	: T	Table	: F + V	Table	: ε
NVM-InP	Memory	: T	Memory	: F + V + p	Memory	: ε
	Log	: p	Log	: F + p	Log	: p
	Table	: p	Table	: 0	Table	: ε

T: size of the tuple

F: size of fixed-length field

V: size of variable-length field

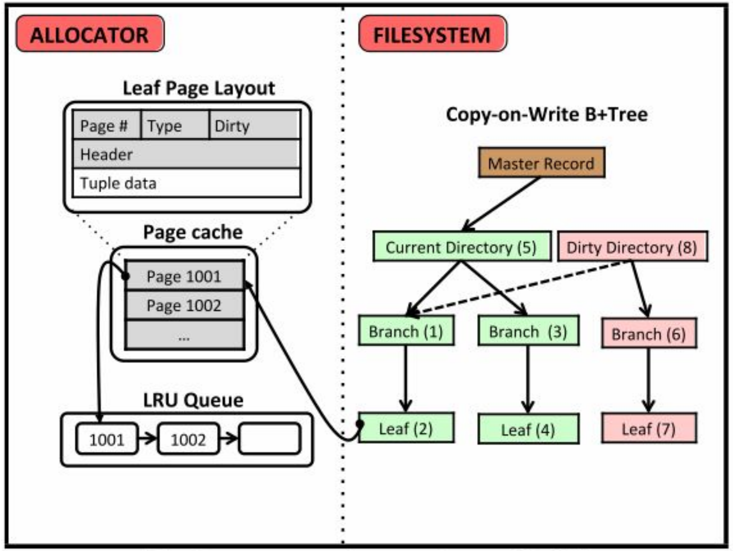
p: size of the pointer

ε: small fixed-length writes to NVM

Copy-on-Write Updates Engine - CoW

Creates a copy of the tuple and then modifies that copy

- As the CoW engine never overwrites committed data, it **doesn't need to record changes** in a WAL for recovery
- Uses different look-up directories for accessing the versions of tuples in the database, known as ***shadow paging*** in IBM's System R [7]
  - **Current directory** points to the most recent versions of the tuples and committed transactions
  - **Dirty directory** points to tuples being modified
- The engine maintains a master record that always points to the current directory
  - Ensure that the transactions are isolated from the effects of uncommitted transactions

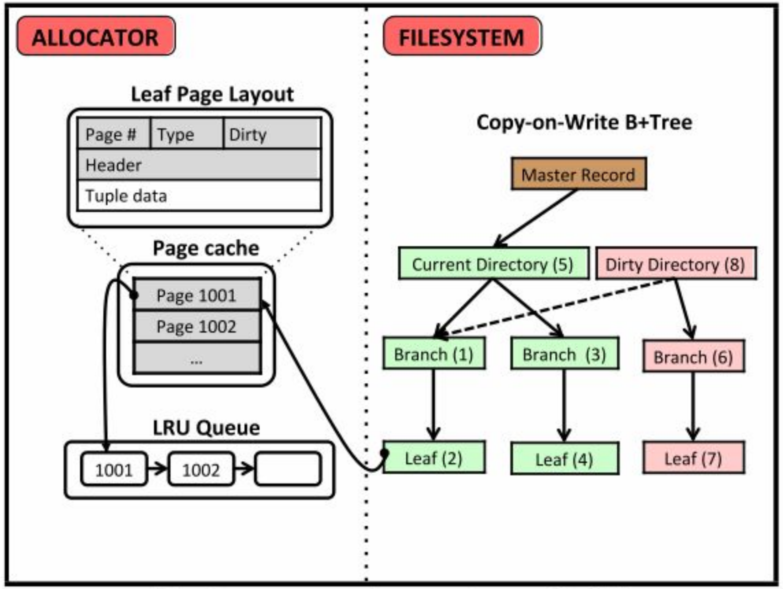


(b) Copy-on-Write Updates (CoW)

Copy-on-Write Updates Engine - CoW

Storage

- CoW engine stores the directories on the filesystem
- Each database is stored in a separate file
- **Downside:** CoW engine creates a new copy of tuple even if a transaction only modifies a subset of the tuple's fields
- **Downside:** The engine needs to keep track of references to tuples from different versions of the copy-on-write B+tree to reclaim the storage space consumed by old unreferenced tuple versions
  - Increasing wear on the NVM device thereby **reducing its lifetime**



(b) Copy-on-Write Updates (CoW)

Recovery

- **No recovery process**: If DBMS crashes before the master record updated then changes present in the dirty directory are not visible after restart

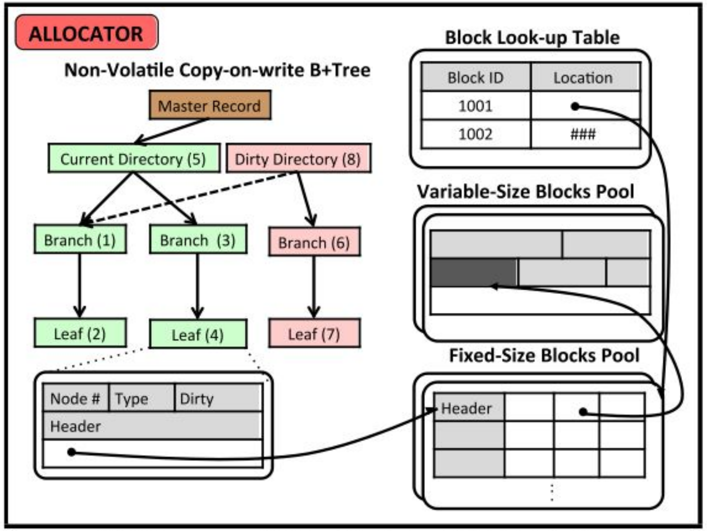
Copy-on-Write Updates Engine - NVM CoW

The original CoW engine stores tuples in self-containing blocks without pointers in the B+tree on the filesystem

- High overhead of propagating modifications to the dirty directory
- Expensive writes (go through the kernel’s VFS path)

Optimizations

- Use of **non-volatile copy-on-write B+tree**
- **Directly persists the tuple copies** and only records non-volatile pointers in the directory
- Use of a **lightweight durability mechanism** to persist changes in the B+tree

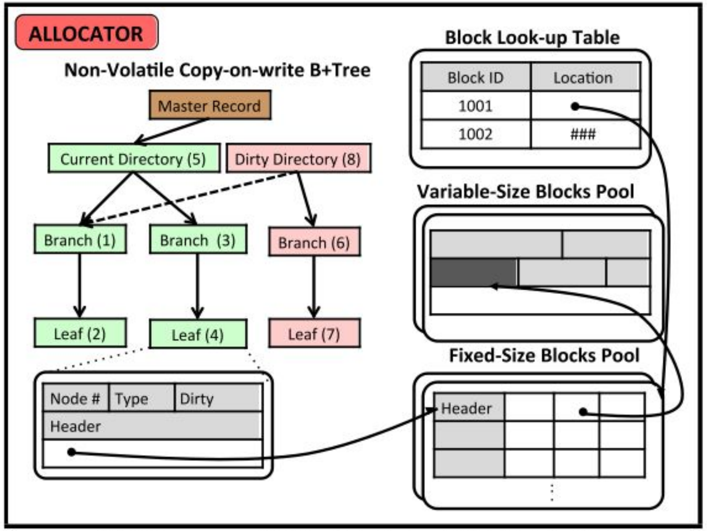


(b) Copy-on-Write Updates (NVM-CoW)

Copy-on-Write Updates Engine - NVM CoW

Storage

- The engine maintains the durability state of each slot in both pools similar to the NVM-InP engine
- The NVM-CoW engine stores the current and dirty directory of the non-volatile copy-on-write B+tree using the allocator interface
- It avoids the transformation and copying costs incurred by the CoW engine



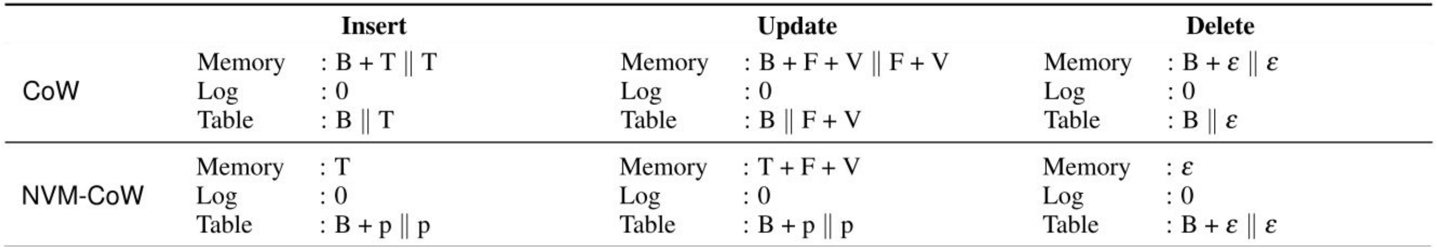
(b) Copy-on-Write Updates (NVM-CoW)

Recovery

- No recovery process as it never overwrites committed data
- Storage space consumed by the dirty directory at the time of failure is **asynchronously reclaimed** by the NVM-aware allocator

Copy-on-Write Updates Engine - NVM CoW

INSERT	UPDATE	DELETE	SELECT
■ Sync tuple with NVM ■ Store tuple pointer in WAL ■ Update tuple state as persisted ■ Add tuple entry in secondary indexes	■ Make a copy of the tuple ■ Apply changes on the copy ■ Sync tuple with NVM ■ Store tuple pointer in dirty dir ■ Update tuple state as persisted ■ Add tuple entry in secondary indexes	■ Remove tuple pointer from dirty dir ■ Discard entry from secondary indexes ■ Recover tuple space immediately	■ Locate tuple pointer in appropriate dir ■ Fetch tuple contents from dir



		Insert		Update		Delete
CoW	Memory	$: B + T \parallel T$	Memory	$: B + F + V \parallel F + V$	Memory	$: B + \varepsilon \parallel \varepsilon$
	Log	$: 0$	Log	$: 0$	Log	$: 0$
	Table	$: B \parallel T$	Table	$: B \parallel F + V$	Table	$: B \parallel \varepsilon$
NVM-CoW	Memory	$: T$	Memory	$: T + F + V$	Memory	$: \varepsilon$
	Log	$: 0$	Log	$: 0$	Log	$: 0$
	Table	$: B + p \parallel p$	Table	$: B + p \parallel p$	Table	$: B + \varepsilon \parallel \varepsilon$

T: size of the tuple

F: size of fixed-length field

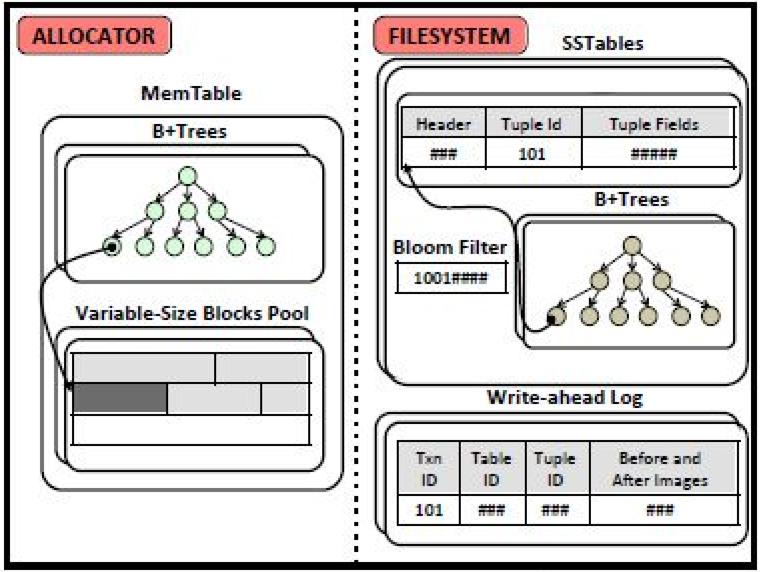
V: size of variable-length field

p: size of the pointer

ε: small fixed-length writes to NVM

Log-structured Updates Engine

- Uses log-structured update policies by utilizing LSM Trees [8]
- Each LSM Tree consists of a collection of runs of data [9]
- Each run contains an ordered set of entries with changes performed on tuples
- Runs reside in volatile and durable memory e.g. *MemTable* and *SStable*
- Design based on Google’s LevelDB [10] - uses leveled LSM Tree, each level with changes of a single run



Header	Tuple Id	Tuple Fields
###	101	#####

Txn ID	Table ID	Tuple ID	Before and After Images
101	###	###	###

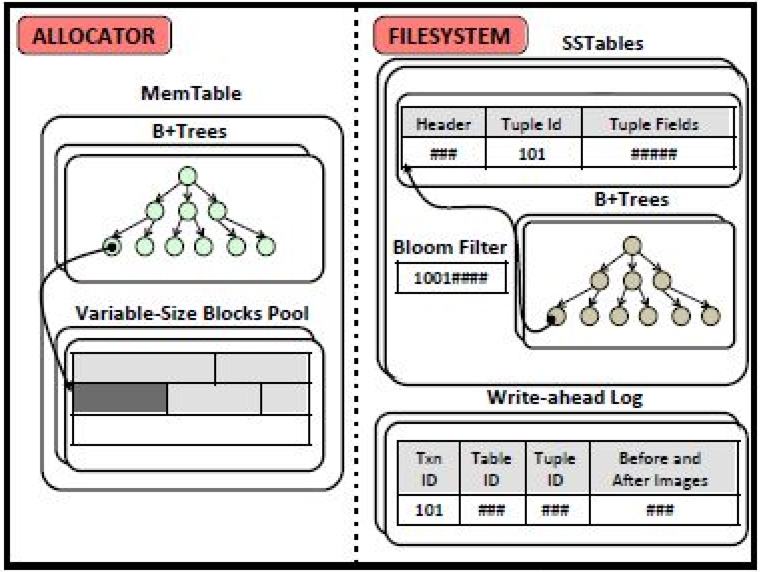
Log-structured Updates Engine

Storage:

- When size of *MemTable* exceeds the threshold, the engine flushes it in *SSTable* to filesystem
- Performs well with write-intensive workloads by reducing random writes
- Poor performance in high read amplification - index look up all runs [11]
- Solution: periodic compaction process for merging subsets of *SSTables* into a new one

Recovery:

- MemTable is volatile, thus *WAL* is maintain in order to recover from logs
- Replays the log and then remove changes that are uncommitted



Header	Tuple Id	Tuple Fields
###	101	#####

Txn ID	Table ID	Tuple ID	Before and After Images
101	###	###	###

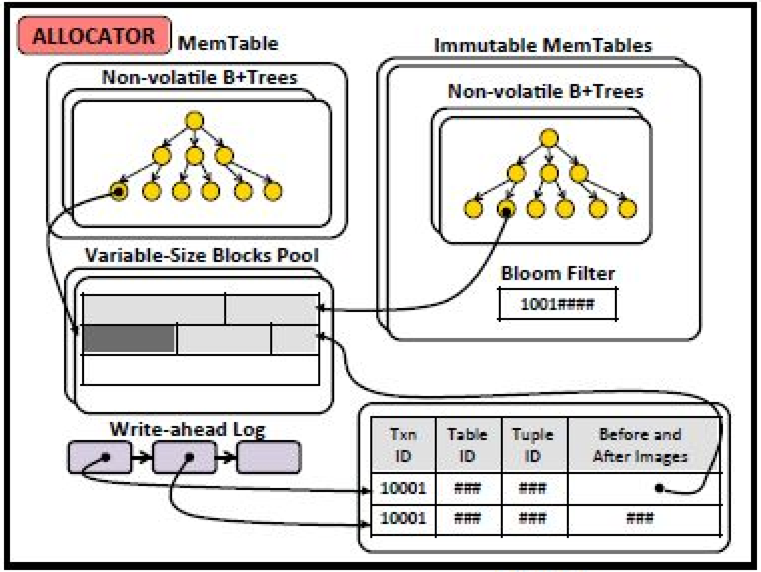
Log-structured Updates Engine - NVM-Log

Storage:

- Batches all writes in MemTable to reduce random accesses on durable storage [9, 12]
- Original Log-Structure Update Engine incurs large overhead from flushing MemTable and compacting SSTable
- Instead of flushing, marks existing as immutable and creates new *MemTable*
- Logging overhead lower - less data and maintains *WAL* with allocator interface

Recovery:

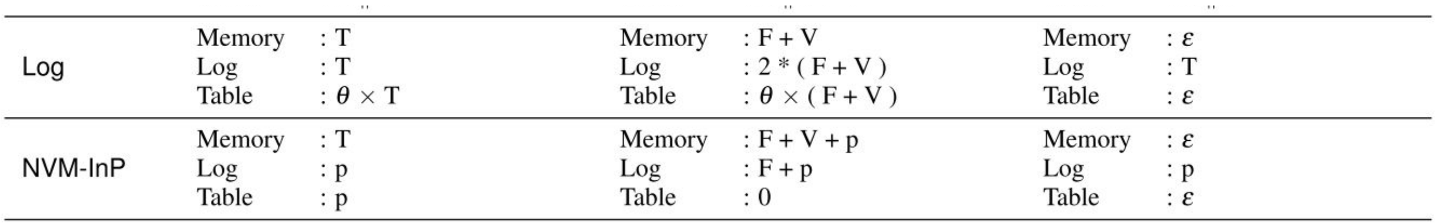
- NVM-aware recovery protocol does not rebuild MemTable but undo the effects of uncommitted transactions



Txn ID	Table ID	Tuple ID	Before and After Images
10001	###	###	
10001	###	###	###

Log-structured Updates Engine - NVM-Log

INSERT	UPDATE	DELETE	SELECT
■ Sync tuple with NVM ■ Record tuple pointer in WAL ■ Sync log entry with NVM ■ Mark tuple state as persisted ■ Add tuple entry in MemTable	■ Record tuple changes in WAL ■ Sync log entry with NVM ■ Perform modifications on the tuple ■ Sync tuple changes with NVM	■ Record tuple pointer in WAL ■ Sync log entry with NVM ■ Mark tuple ***tombstone*** in MemTable ■ Reclaim space during compaction	■ Find tuple entries in relevant LSM runs ■ Rebuild tuple by coalescing entries



Log	Memory	: T	Memory	: F + V	Memory	: ε
	Log	: T	Log	: 2 * (F + V)	Log	: T
	Table	: $\theta \times$ T	Table	: $\theta \times$ (F + V)	Table	: ε
NVM-InP	Memory	: T	Memory	: F + V + p	Memory	: ε
	Log	: p	Log	: F + p	Log	: p
	Table	: p	Table	: 0	Table	: ε

T: size of tuple
F: size of fixed-length field
V: size of variable-length field
θ: LSM compaction factor
ε: small fixed length size
p: size of a pointer

Benchmarks

- **YCSB:** Yahoo key-value workloads for web-based companies transactions
  - **Transaction types:**
    - Read actions to retrieve single tuple based on primary key
    - Update transactions to modify single tuple based on primary key
  - **Workload mixtures:**
    - Read-Only - 100% Read
    - Heavy-Read - 90% Read and 10% Write
    - Balanced - 50% Read and 50% Write
    - Write-Heavy - 10% Read and 90 % Write
  - **Skew Levels:**
    - Low Skew - 50% of the transaction access 20% of the tuples
    - High Skew - 90% of the transaction access 10% of the tuples
- **TPC-C:** Industry standard for evaluating OLTP systems
  - Five transaction types from wholesale suppliers

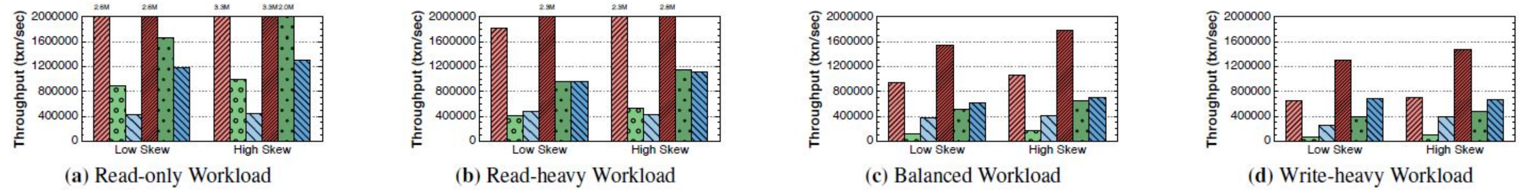
Runtime Performance

- NVM's latency impact analysis on the performance of the storage engines
- Run the benchmarks under three different latency configurations:
  - Default DRAM latency configuration - 160 ns
  - A low NVM latency configuration - 2x higher than DRAM with 320 ns
  - A high NVM latency configuration - 8x higher than DRAM with 1280 ns

▨ InP
▨ CoW
▨ Log
▨ NVM-InP
▨ NVM-CoW
▨ NVM-Log

YCSB Benchmark Performance Analysis

DRAM Latency



(a) Read-only Workload

(b) Read-heavy Workload

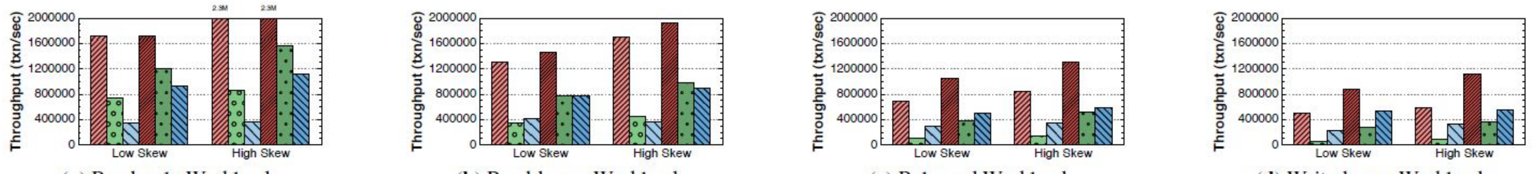
(c) Balanced Workload

(d) Write-heavy Workload

Runtime Performance

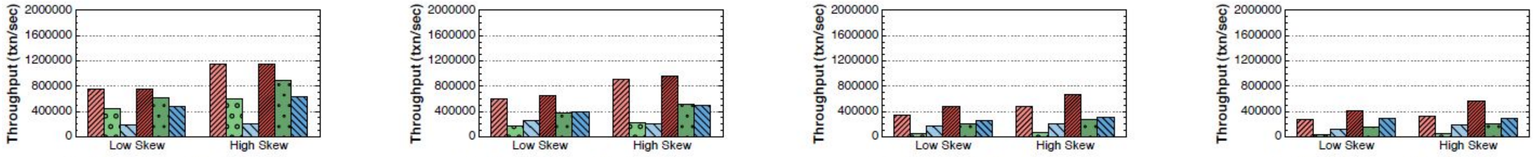
YCSB Benchmark Performance Analysis

NVM Low Latency

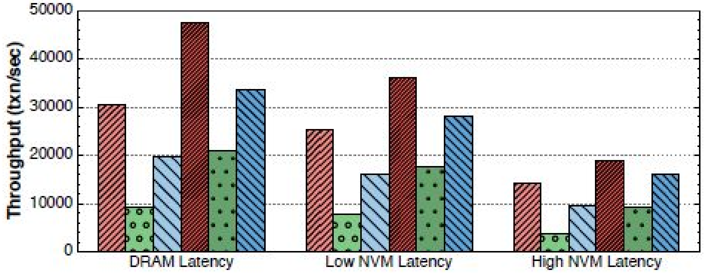


(a) Read-only Workload (b) Read-heavy Workload (c) Balanced Workload (d) Write-heavy Workload

NVM High Latency



(a) Read-only Workload (b) Read-heavy Workload (c) Balanced Workload (d) Write-heavy Workload



The performance of the engines for TPC-C benchmark for all three NVM latency settings

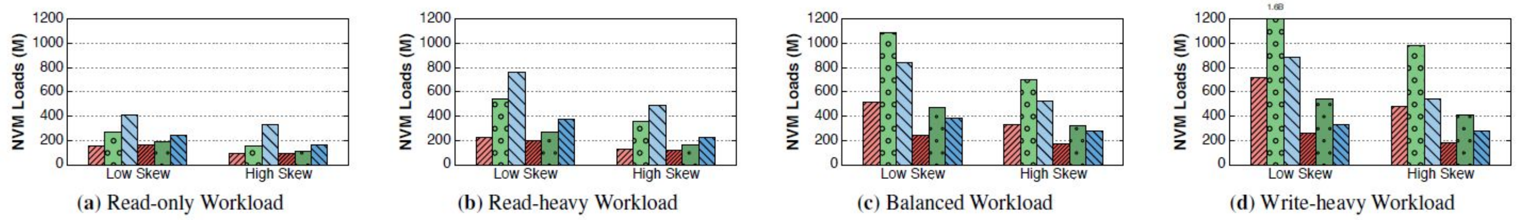
▨ InP
▨ CoW
▨ Log
▨ NVM-InP
▨ NVM-CoW
▨ NVM-Log

Reads and Writes

- Measure the times that the storage engines access the NVM device while executing the benchmark
- Important because different NVM technologies have limits on write cycles per bit
- Measure using hardware performance counters that track the number of loads (reads) and stores (writes)

YCSB Benchmark Performance Analysis

YCSB NVM Reads



(a) Read-only Workload

(b) Read-heavy Workload

(c) Balanced Workload

(d) Write-heavy Workload

▨ InP

▨ CoW

▨ Log

▨ NVM-InP

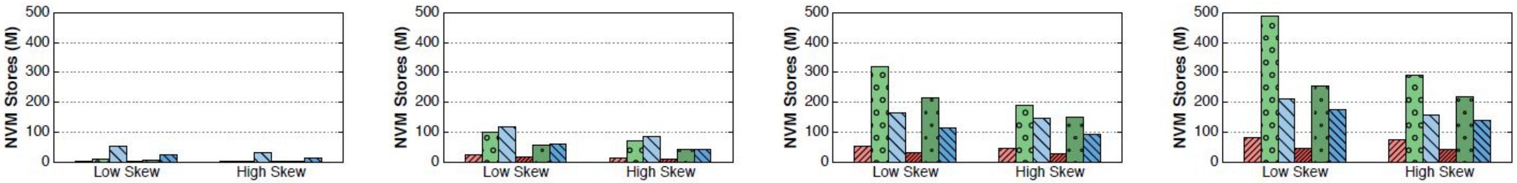
▨ NVM-CoW

▨ NVM-Log

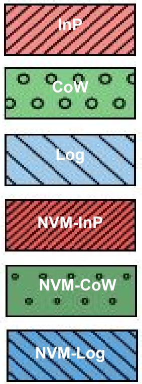
Reads and Writes

YCSB Benchmark Performance Analysis

YCSB NVM Stores

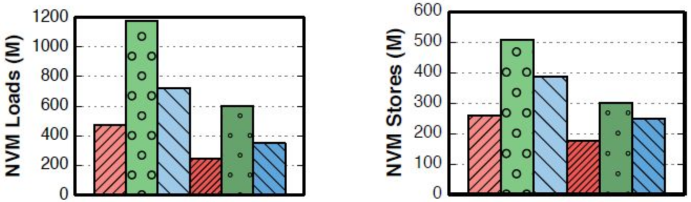


(a) Read-only Workload (b) Read-heavy Workload (c) Balanced Workload (d) Write-heavy Workload



TPC-C Benchmark Performance Analysis

TPC-C NVM Loads

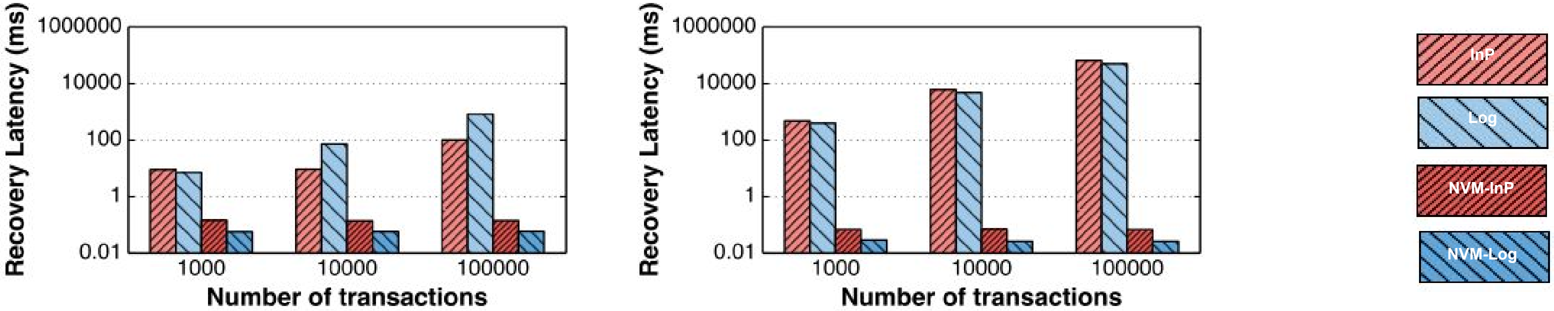


(a) Reads (b) Writes

The number of load and store operations executed by the storage engines while running the TPC-C benchmark

Recovery

- For each benchmark authors execute a fixed number of transactions and then force a hard shutdown - ***SIGKILL***



(a) YCSB

(b) TPC-C

- The latency of the InP and Log engines **grows linearly**
  - Redo the effects of committed transactions before undoing the effects of uncommitted transactions

- NVM-aware engines' **recovery time is independent of the number of transactions executed**
  - Only undo the effects of the transactions that are active at the time of failure

The NVM-aware engines have **a short recovery** that is always **less than a second**

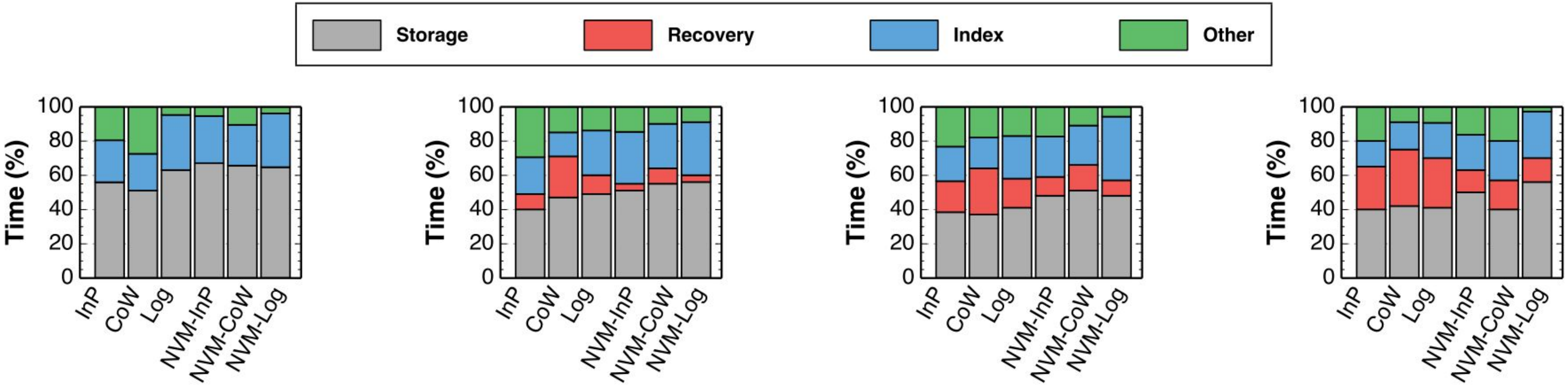
Execution Time Breakdown

- Analyze the time that the engines spend in their internal components during execution
- Only YCSB with low skew and low NVM latency configuration
- Use event-based sampling with the *perf-framework* to track the cycles executed within the engine’s components

Categories

- Storage Management operations ⬜
- Recovery Mechanisms 🟥
- Index accesses and maintenance 🟦
- Other components 🟩

Execution Time Breakdown



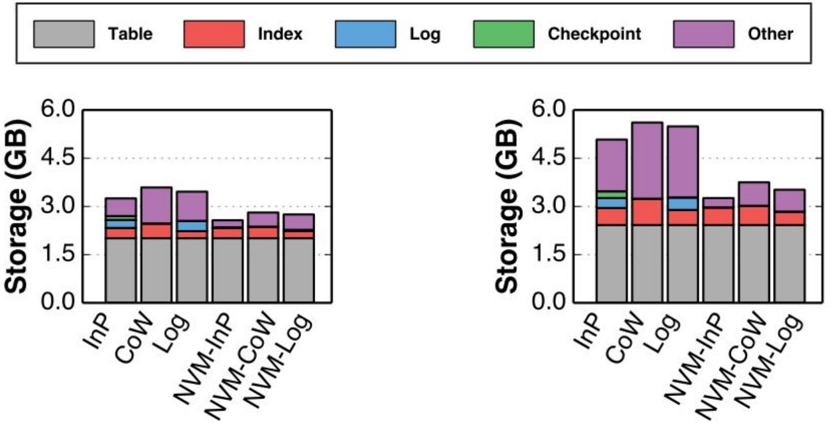
(a) Read-only Workload **(b)** Read-heavy Workload **(c)** Balanced Workload **(d)** Write-heavy Workload

- NVM-aware engines spend **13-18%** of their time on recovery tasks compared to traditional engines **(33%)**
- Engines performing copy-on-write updates spend a higher proportion of time on recovery-related tasks
  - Cost of creating and maintaining the dirty directory for large databases, even using efficient B+tree
- Log and NVM-Log spend a higher fraction of their time on index look-ups
  - They perform multiple index look-ups on the LSM tree to reconstruct tuples

Storage Footprint

The amount of space that it uses for storing tables, logs, indexes, and other internal data structures

- This metric is important because we expect that the first NVM products will initially have a higher cost than current storage technologies



(a) YCSB Storage

(b) TPC-C Storage

- CoW engine has high overhead (creating directories and copying tuples)
- NVM-aware engines consume 17-21% less storage space than traditional engines

- 31-38% smaller storage footprints
- More significant because TPC-C is write-intensive with longer running transactions

Conclusion

- NVM-aware engines outperform the traditional engines by up to **5.5** times while reducing the number of writes to the storage more than half on write-intensive workloads
- NVM access latency has the most impact on the runtime performance, more than the amount of skew or the modifications number
- The NVM-aware engines perform fewer store/write operations than the traditional ones, thus extending NVM device lifetimes.
- Smaller storage footprint due to allocator interface usage with non-volatile pointers for data structures
- NVM-InP engine overall performs better across the workload mixtures, skew settings and NVM latencies
- NVM-CoW engine did not perform well for write-intensive workloads - better fit for DBMSs' that support non-blocking, read-only transactions
- NVM-Log essentially performs in-place updates like NVM-InP without additional overhead

References

[1] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM. 1970.

[2] M. M. Astrahan, M.W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths,W. F. King, R. A. Lorie, P. R. McJones, J.W. Mehl, G. R. Putzolu, I. L. Traiger, B.W. Wade, and V.Watson. System R: relational approach to database management. ACM Trans. Database Syst., 1(2):97–137, June 1976

[3] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS Fast Path. Technical report, Tandem, 1985

[4] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. In ADMS@VLDB, 2014.

[5] VoltDB. http://voltdb.com.

[6] T. Bingmann. STX B+ tree C++ template classes. http://panthema.net/2007/stx-btree/

[7] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system R database manager. ACM Comput. Surv., 13(2):223–242, June 1981

References

[8] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. ACM Trans. Comput. Syst., 10(1):26–52, Feb. 1992

[9] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil. The log-structured merge-tree (lsm-tree). Acta Inf., 33(4):351–385, June 1996.

[10] J. Dean and S. Ghemawat. LevelDB. http://leveldb.googlecode.com.

[11] Apache Cassandra. http://datastax.com/documentation/cassandra/2.0/.

[12] LevelDB. Implementation details of LevelDB. https: //leveldb.googlecode.com/svn/trunk/doc/impl.html.