RDBMS Design Using RAM and NVRAM: A Hybrid Architecture

ANUSHKA SRIVASTAVA & JASWINDAR SINGH GILL
Indian Institute of Information Technology Guwahati

Date: 4 March 2025
Guide: Dr. Gautam Barua

Outline

Non-Volatile Random Access Memory (NVRAM)

- **Definition**: A type of non-volatile memory that combines the speed of RAM with the persistence of storage (e.g., Intel Optane, 3D XPoint).
- **Key Characteristics**:
  - **Low Latency**: Significantly faster than traditional disk I/O (nanoseconds vs. milliseconds).
  - **Persistence**: Data remains intact even after power loss, unlike volatile RAM.
  - **Write Endurance**: Higher than SSDs but lower than DRAM.
  - **Byte-Addressability**: Unlike block-based storage, allows fine-grained memory access.
- Relevance to RDBMS: Enables fast, durable storage for databases, bridging RAM and disk performance gaps.

Why RAM and NVRAM Are Better Than RAM and Disk I/O

- **Performance**: NVRAM offers latency closer to RAM (nanoseconds vs. milliseconds for disks), reducing I/O bottlenecks.
- **Persistence**: Unlike RAM, NVRAM retains data without power, eliminating the need for frequent disk writes.
- **Scalability**: By using RAM for fast indexing (e.g., B+ Trees, hash tables) and NVRAM for persistent storage, the system minimizes disk I/O delays. This allows efficient handling of growing datasets without significantly increasing query response times.
- **Cost-Effectiveness**: NVRAM reduces reliance on slow, power-intensive disk storage while maintaining persistence. This lowers the need for large amounts of RAM or frequent disk access, optimizing both hardware costs and energy consumption.

Architecture of the Project

- **Hybrid Design**: Uses RAM for fast indexing and processing, while NVRAM ensures persistence and durability.
- **Software Emulation**: Since true NVRAM (e.g., Intel Optane) is unavailable, we use `mmap()` to simulate persistent memory by mapping a file into virtual memory.
- **Reserving RAM at Boot Time:** Using the memmap boot parameter in Linux (through Grub), a portion of main memory is reserved for use of NVRAM. This RAM becomes accessible as a device file which is then mmap'ed to a user process.
- **Multi-threaded Environment**: Supports concurrent operations with thread-safe data structures and logging mechanisms.
- **Goal**: Achieve low-latency reads/writes, ACID compliance, and scalability for modern database workloads.

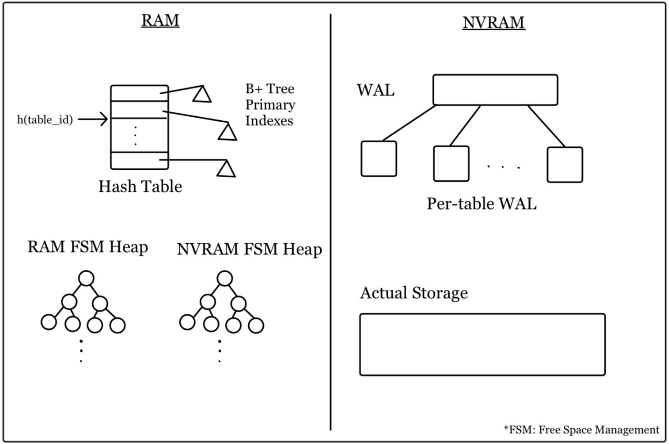
Keys and Values

- Each row of a table is represented by a **key–value pair**.
  - The **key** is the primary key used for the table.
  - The **value** is the contents of the row (including the primary key).
- Each key is composed of two parts: <`table_id, row_id`>.
- There is an **index** stored for each table based on the primary key.
- A **hash table** is used to point to the index of each table.
- Given a key:
  - The `table_id` component is used to access the hash table to obtain a pointer to the root of the index.
  - The `row_id` component of the key is then used to search the index to obtain a pointer to the row contents.

Overview of RAM and NVRAM: Data Structures and Design

- **RAM**:
  - Hash table maps `table_id` to roots of B+ Tree indexes for fast lookups.
  - B+ Trees index tables use `row_id` as key to search for pointer to row contents.
  - Secondary indexes for optimized queries.
  - Free space management for both, B+ Tree and data storage in NVRAM.
- **NVRAM**:
  - Write-Ahead Logs (WALs) ensure durability for table and multi-table transactions.
  - Stores actual data (larger values linked from B+ Tree leaf nodes).

Overview


RAM
NVRAM
h(table_id)
B+ Tree
Primary
Indexes
Hash Table
WAL
. . .
Per-table WAL
RAM FSM Heap
NVRAM FSM Heap
Actual Storage
*FSM: Free Space Management

What Is Stored in RAM

- **Hash Table**: Maps `table_id` to pointers of root of B+ Tree index for fast lookups (`create_table`, `get_row`, `put_row`, `del_row`, `get_nxt_row`).
- **B+ Trees**: Per-table indexing with `row_id` as key; leaf nodes store small data or pointers to NVRAM.
- **Secondary Indexes**: Application-defined indexes for optimized queries on non-primary key attributes.
- **Free Space Management**: Utilizes a heap-based structure to efficiently track and allocate free space for B+ Tree nodes and data stored in NVRAM, minimizing fragmentation.
- **Purpose**: Supports low-latency, in-memory indexing and querying.

Key Table & Row Operations

- **Create Table (`create_table`)**: Compute `hash`, initialize B+ Tree, store root in hash table.
- **Retrieve Row (`get_row`)**: Get B+ Tree root via `hash`, search for `row_id`, fetch from RAM/NVRAM.
- **Insert/Update Row (`put_row`)**: Search B+ Tree, update if exists, else allocate space and insert.
- **Delete Row (`del_row`)**: Find `row_id`, remove from B+ Tree, free NVRAM if needed.
- **Next Row (`get_nxt_row`)**: Find next `row_id` in B+ Tree, return data or `NULL`.

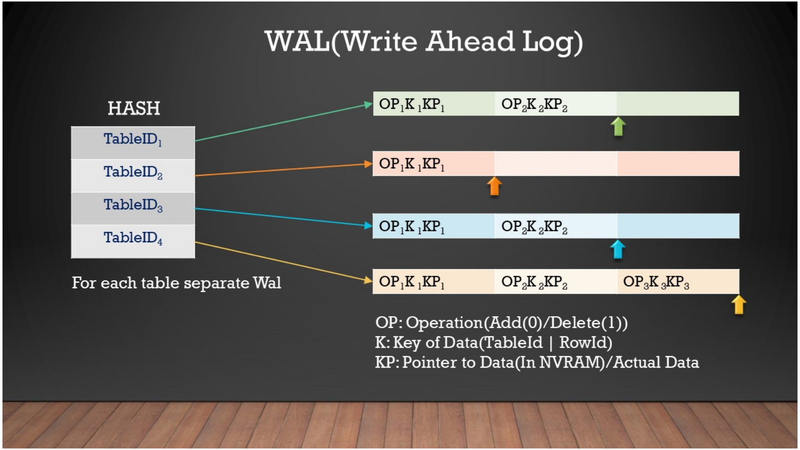
Heap-Based Free Space Management

- **Objective**: Efficiently manage free space using a max heap for fast allocation and deallocation, providing near O(1) operations, efficient merging, and reduced fragmentation.
- Free space management will be done for both RAM and NVRAM, with the data structures for both stored in RAM.
- **Structure Maintained:**
  - Tracking of total available space and a pointer to the max heap.
  - Each node in max heap represents a block of free memory and stores a pointer to the free memory location in NVRAM and its size.
  - Metadata in free blocks is stored at both start and end of each block.
  - Each block of memory has a 1 bit long allocation status (0 for free/1 for filled).
  - If free, contains a pointer back to the heap node.

What Is Stored in NVRAM

- **Write-Ahead Logs (WALs)**:
  - Per-table WALs for individual table operations (ensuring durability).
- **Actual Data Values**: Data stored in NVRAM, with pointers in RAM from the indexes, for access.
- Purpose: Ensure data durability, support crash recovery, and handle persistent storage for large datasets.

Row Data Format



Write-Ahead Log (WAL) in NVRAM

- **Purpose**: Ensures durability and crash recovery by logging changes before applying them to the database.
- **Key Features**:
  - **Table-Wise WAL**: Each table has a separate WAL, hashed by table key for fast access.
  - **Stored in NVRAM**: Eliminates disk I/O bottlenecks and removes the need for explicit flushing.
- **WAL Entry Structure**:
  - **Commit Bit (1-bit)**: Indicates commit status.
  - **Add/Delete Flag (1-bit)**: Specifies insertion or deletion.
  - **Data Key**: Composite key (`table_id` | `row_id`).
  - **Pointer to Data**: Reference to actual data in main storage.

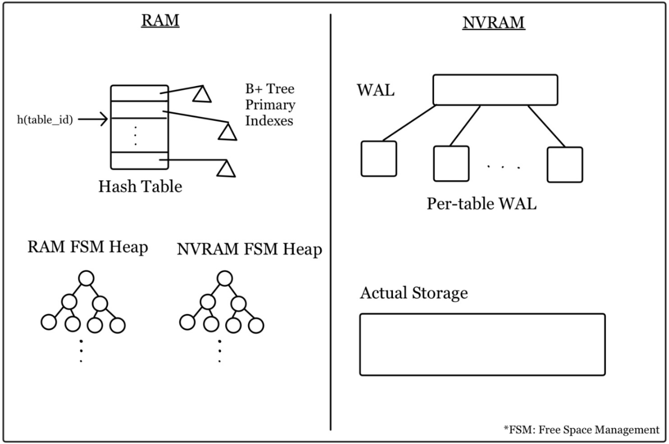
Example

- **Dummy Database**: A table `Users` with columns `row_id` (primary key), `name`, and `age`.
- **Read Operation (`get_row`)**:
  - Client issues `get_row(table_id | row_id)`.
  - Hash table in RAM finds the B+ Tree root for `Users`.
  - B+ Tree in RAM searches for the key:
    - If the data is in RAM, return it directly.
    - If a pointer to NVRAM exists, retrieve the data from there.
  - Return data to client with low latency.

Example

- **Write Operation (`put_row`)**:
  - Client issues `put_row(table_id | row_id, {name: "Alice", age: 30})`.
  - Hash table in RAM locates the B+ Tree root for `Users`.
  - B+ Tree in RAM invokes the free space manager:
    - If sufficient space is available in the leaf node, data is stored in RAM.
    - If not, a pointer to a new allocated NVRAM space is stored instead.
  - WAL in NVRAM logs the operation before applying changes (ensuring durability and atomicity).
  - Data (or pointer) is written to RAM B+ Tree or NVRAM.
  - Upon commit, the WAL entry is marked as committed.

Example


RAM
NVRAM
h(table_id)
B+ Tree
Primary
Indexes
Hash Table
WAL
Per-table WAL
RAM FSM Heap
NVRAM FSM Heap
Actual Storage
*FSM: Free Space Management

Planned Shutdown Algorithm

- **Objective**: Ensure a smooth shutdown by persisting RAM structures (B+ Trees, free space heaps) to NVRAM, reducing recovery time.
- **Steps**:
  1. Pause incoming transactions and complete or rollback any active ones.
  2. Copy the latest RAM-based B+ Trees and free space heaps to NVRAM.
  3. Write a checkpoint to NVRAM, marking a stable state of all tables and indexes.
  4. Flush remaining WAL entries and finalize them to maintain data integrity.
  5. Safely release resources, close NVRAM connections, and shut down the system.
- **Benefits**: Minimizes uncommitted WAL entries, making recovery nearly instantaneous.

Recovery Algorithm

- **Objective**: Restore RAM data structures (B+ Trees, free space heaps) from NVRAM for fast recovery and ensure data consistency.
- **Steps**:
  1. On restart, load the latest B+ Trees and free space heaps from NVRAM into RAM.
  2. Read all WALs (per-table and multi-table transaction logs) from NVRAM.
  3. Identify and apply only committed transactions based on commit markers.
  4. Replay WAL entries to reconstruct changes in B+ Trees and persist values back to NVRAM.
  5. Verify integrity using checksums or hash-based validation techniques.
  6. Clear WALs after successful recovery to start with a clean state.
- **Advantages**: Faster recovery by leveraging stored B+ Trees instead of reconstructing from scratch.

Issues and Planned Solutions for Next Phase

- **Issues**:
  - Concurrency Control: Race conditions in multi-threaded operations on RAM structures and NVRAM WALs.
  - Transaction Management: Ensuring atomicity and consistency across multi-table transactions.
- **Planned Solutions (Next Phase)**:
  - Implement thread-safe locking or lock-free data structures (e.g., atomic operations) for concurrency control.
  - Design transaction managers with two-phase commit or similar for multi-table operations.

Questions?

Thank you for your attention!
Any questions or feedback?