# Design Optimization of Computing Systems

Assignment 3


## DOCS-DB (Part-A)

**Submitted by:**

**Bratin Mondal (21CS10016)**
**Swarnabh Mandal (21CS10068)**
**Somya Kumar (21CS30050)**

*Department of Computer Science and Engineering,*
*Indian Institute of Technology Kharagpur*

# 1 Introduction to DocsDB

DocsDB is a lightweight, high-performance key-value storage engine tailored for efficient storage and retrieval operations. It employs the **Log-Structured Merge Tree (LSM Tree) architecture**, which is particularly well-suited for write-heavy workloads due to its ability to batch writes in memory before persisting them to disk. DocsDB supports three fundamental operations:

- **SET**: Insert or update a key-value pair.
- **GET**: Retrieve the value associated with a key.
- **DEL**: Delete a key-value pair.

The implementation of DocsDB is structured into three main phases:

- **Database Engine Design**: Focuses on building and optimizing the core engine using efficient data structures, storage techniques, and caching mechanisms.
- **Network Infrastructure**: Enhances communication efficiency with high-performance networking tools such as network namespaces and DPDK.
- **Integration**: Combines the database engine with the network stack to create a unified, high-performance system.

This document outlines the key design decisions and implementation strategies that contribute to DocsDB's efficiency and performance.

# 2 Design Choices

The design of DocsDB is driven by the need to maximize efficiency, scalability, and resilience in handling key-value data. Each choice—ranging from data structures to concurrency control mechanisms—is aimed at minimizing disk I/O, optimizing in-memory operations, and ensuring durability. These design considerations allow DocsDB to excel in write-heavy workloads while maintaining fast reads and robust system performance.

## 2.1 Optimized Workload and Reduced Disk Access

- DocsDB is optimized for **write-heavy workloads**, where incoming writes are handled in-memory before being flushed to disk. This approach minimizes disk I/O during write operations, enhancing overall performance.
- **Bloom Filters**, a probabilistic data structure, reduce unnecessary disk access. They efficiently test for the possible presence of keys. If a key is not in the database, the Bloom Filter guarantees this without requiring a disk lookup. For potential matches, a disk lookup confirms the key's presence.

## 2.2 Data Structures Used

DocsDB's performance relies on carefully selected data structures:

- **Primary Storage:** The *Log-Structured Merge Tree* (LSM Tree) organizes data into levels and uses sequential writes to disk, optimizing for high write throughput. Periodic **compaction** merges and organizes data, enabling efficient reads.
- **Auxiliary Structure:** *Bloom Filters* optimize read performance by enabling quick existence checks. Integrated with the LSM Tree, they avoid unnecessary disk reads while maintaining reliability.

### 2.3 Concurrency Control

Concurrency in DocsDB is managed using a flexible and configurable thread pool, ensuring optimal utilization of system resources and smooth handling of concurrent operations. Each thread in the pool is assigned a specific role, which enhances overall efficiency and performance:

- **Write Threads:** These threads handle incoming write operations, buffering data in the **Memtable**. By batching and organizing writes, they minimize contention and improve write throughput.
- **Read Threads:** Dedicated to serving user queries, these threads prioritize low-latency access. They coordinate with the caching layer and check the **Memtable**, **Immutable Memtable**, and **SSTables** sequentially to quickly return results.
- **Compaction Threads:** Responsible for background compaction tasks, these threads merge and reorganize **SSTables**, remove stale data (including tombstones), and optimize disk usage without disrupting foreground operations.

This thread-based model supports high levels of parallelism, leveraging multiple CPU cores to handle various operations concurrently.

### 2.4 Additional Optimizations

DocsDB incorporates several advanced optimizations to enhance both performance and durability:

- **Write-Ahead Logging (WAL):** To ensure durability, all operations are recorded in the *Write-Ahead Log* before being applied to the in-memory **Memtable**. This mechanism allows the system to recover seamlessly from unexpected crashes or failures by replaying the logged operations.
- **In-Memory Memtables:** Active writes are temporarily buffered in an in-memory structure called the **Memtable**. This approach significantly reduces the frequency of disk writes, minimizes I/O overhead, and improves overall write performance. When the Memtable reaches its capacity, it is converted into an **Immutable Memtable** and written to disk as a **Sorted String Table (SSTable)**.
- **Background Compaction Threads:** Compaction tasks, such as merging and reorganizing **SSTables**, are performed by dedicated background threads. These threads operate independently of foreground read and write operations, ensuring consistent performance while maintaining efficient disk storage by removing stale data and consolidating fragmented entries.
- **Caching Mechanisms:** Frequently accessed data is stored in memory through a multi-layer caching system. By reducing the need for repeated disk access, these caches significantly lower read latency and enhance the overall query performance, particularly for workloads with high read frequencies.
- **Efficient Disk Usage:** DocsDB employs sequential writes for **SSTables** and avoids random I/O patterns, thereby taking full advantage of modern storage devices' performance characteristics. Periodic compactions further optimize storage by removing redundancies and merging overlapping data.

These optimizations collectively enable DocsDB to handle demanding workloads with high throughput and reliability while maintaining low latency for both reads and writes.

## 3 Data Path Components and Core Operations

DocsDB is designed to efficiently handle write, read, and delete operations while ensuring durability, low latency, and optimal disk usage. The following sections detail the components and workflows that enable these capabilities.

## 3.1 Data Path Components

The data path in DocsDB is divided into three main components, each optimized for specific tasks:

### 3.1.1 Write Path Components

The write path ensures data durability and efficient write operations:

- **Write-Ahead Log (WAL):** Logs every write operation to guarantee durability and support crash recovery.
- **Memtable:** A mutable, in-memory structure that temporarily stores active writes, enabling fast updates without immediate disk writes.
- **Immutable Memtable:** A read-only version of the Memtable that is flushed to disk as a **Sorted String Table (SSTable)** once it reaches capacity.
- **SSTables:** Persistent, sorted, on-disk files that store data in an optimized format for fast reads and efficient disk usage.

### 3.1.2 Read Path Components

The read path is designed for efficiency and minimal latency:

- **Bloom Filters:** Probabilistic data structures that quickly check if a key might exist, avoiding unnecessary disk lookups for non-existent keys.
- **Memtable and Immutable Memtable:** In-memory structures that are queried first to retrieve the requested key.
- **SSTables:** Queried sequentially, starting from the newest to the oldest, only if the key is not found in memory.

### 3.1.3 Compaction Components

Compaction processes maintain storage efficiency and eliminate redundant or stale data:

- **Compaction Manager:** Oversees the merging and reorganization of **SSTables** to reduce fragmentation and remove stale entries such as tombstones.
- **Level-Based Storage:** Organizes **SSTables** into hierarchical levels, enabling efficient compaction and reducing query latency.

## 3.2 Core Operations

DocsDB supports three core operations: writes, reads, and deletes. Each workflow is optimized for performance and data integrity.

### 3.2.1 Write Workflow

- Log the write operation in the **Write-Ahead Log (WAL)** to ensure durability.
- Add the key-value pair to the **Memtable** for fast, in-memory processing.
- Convert the **Memtable** to an **Immutable Memtable** and flush it to disk as an **SSTable** once it reaches capacity.
- Perform background compaction to merge overlapping **SSTables** and remove stale data.

#### 3.2.2 Read Workflow

- Check the **Bloom Filter** to determine if the key might exist in an **SSTable**.
- Search the **Memtable** and **Immutable Memtable** for the requested key.
- If the key is not found in memory, sequentially query the **SSTables**, starting from the newest.
- Return the value if the key is found; otherwise, return an error indicating the key does not exist.

#### 3.2.3 Delete Workflow

- Add a ”tombstone” marker to the **Memtable** to signify the key's deletion.
- Ensure tombstoned keys are excluded from subsequent read operations.
- Remove tombstones and the associated data during the next compaction cycle, freeing up storage space.

These components and workflows work in unison to provide a high-performance, reliable key-value storage system that is optimized for both write-heavy and read-heavy workloads.

## 4 Merge and Compaction

The compaction process in DocsDB is critical for maintaining efficient disk usage, optimizing query performance, and ensuring storage scalability. By leveraging background threads and level-based organization, DocsDB ensures that the storage system remains efficient, even as data grows over time. Below are the key components of the merge and compaction process:

- **Compaction Manager:** The **Compaction Manager** is responsible for coordinating the merging of **SSTables**. It ensures that redundant data is consolidated, stale entries and tombstones are removed, and the overall disk space is optimized.
- **Compaction Process:** Older **SSTables** are merged into fewer, larger files. This process reduces storage fragmentation and removes stale data that no longer contributes to query results. The merging process also eliminates tombstone markers, which signify deleted keys, ensuring that unnecessary data does not occupy disk space.
- **Level-Based Organization: SSTables** are organized into levels, with each level containing sorted files of increasing size. This level-based approach facilitates faster reads and more efficient merges. During compaction, files from one level are merged with those from the next level, reducing the number of files to be scanned during queries.
- **Thread Efficiency:** The compaction process is handled by background threads, which operate independently of foreground operations. The number of threads allocated for compaction is configurable, allowing the system to balance performance and resource usage. This flexibility ensures that DocsDB can maintain optimal performance under varying workload conditions without overburdening system resources.

These strategies ensure that DocsDB remains efficient and scalable, even as the volume of data grows. By intelligently merging **SSTables** and removing stale data, the system reduces I/O overhead and keeps read latencies low.