Design and Implementation of a Scalable Distributed Database System with LSM-tree Storage and Consistent Hashing

A. Jagan Karthick, Chandru M. S, Madhan Kumar S
Department of Computer Science and Engineering
SRM Institute of Science and Technology, Tiruchirapalli, India
Email: {jk8642, cm1212, mk1101}@srmist.edu.in

***Abstract*—This paper explains the full scope, the intricate details, and the result of the evaluation of a novel distributed database system that combines a Log-Structured Merge-tree (LSM-tree) storage structure with consistent hashing for the best data distribution and retrieval in a way that they complement each other. To begin with, the system that we have designed addresses the problems that have become the major challenges of the data-intensive applications that consume large amounts of data. It supports a great throughput of write operations while also preserving a strong read performance and fault tolerance.**

The architecture utilizes a fundamentally new ring-based topology with the automatic replication of data, the efficient mechanisms of secondary indexing, and the advanced memory management with the help of the intelligent memtable flushing. The result of a broad range of experiments confirms the high performance of our system in that it can achieve 45,231 operations/second write throughput and this is a 17.6% betterment of Apache Cassandra and a 266% betterment of MongoDB. Under the test environment, the system shows linear scalability up to 12 nodes, and it is consistent that sub-millisecond latency is achieved for 95% of the read operations.

***Index Terms*—Distributed Databases, LSM-trees, Consistent Hashing, Data Replication, Secondary Indexing, Scalable Storage, Fault Tolerance**

I. Introduction

The exponential growth of data across various applications, including social media platforms, Internet of Things (IoT) ecosystems, real-time analytics, and financial transaction systems, has put a lot of pressure on traditional database management systems. Conventional relational databases are great for ACID transactions and complex queries but frequently find it difficult to provide the scalability, availability, and performance needed by modern distributed applications. This study presents a distributed database system design and implementation as a solution to these problems which makes use of a LSM-tree storage and consistent hashing distribution combination that has been carefully designed.

By using a write-optimized storage structure that can manage high-velocity data ingestion and at the same time keep the read operations efficient, our system is a major architectural breakthrough. The consistent hashing integration thus achieved guarantees that data is evenly distributed among the nodes of the cluster and at the same time the movement of data is minimized during the process instances, which is a very important factor for those elastic cloud environments. These advantages are the key features of this holistic architectural approach:

- **Horizontal Scalability:** Performance improvement is linear with the addition of a node
- **Reliability:** Data replication and recovery processes are fully automated
- **Write Optimization:** LSM-tree architecture for high-throughput data ingestion
- **Flexible Querying:** Efficient secondary indexing for complex access patterns
- **Simplicity of Operation:** Cluster management that is self-organizing

The remaining parts of this paper are structured as follows: Section II covers the related work; Section III, our architecture; Section IV, the implementation; Section V, the experiments; Section VI, the challenges; and Section VII, the conclusion.

II. Related Work

Over the last ten years, the technology behind distributed database systems has undergone a significant transformation. One of the key innovations was Google's Bigtable that first started utilizing LSM-trees in distributed systems, which in turn allowed better write performance. Amazon's Dynamo brought in the features such as consistent hashing and eventual consistency, besides that it was the availability that the system gave priority to rather than the strict consistency.

Apache Cassandra combined all these concepts to accomplish both scalability and high availability, albeit with changes in secondary index and memory management by comparison to our system. We proposed a distributed secondary index method that is less noticeable in terms of distribution transparency.

The work on LSM-tree has been advanced as well, with the development of bLSM and cLSM by which researchers have improved the methods of compaction and caching. LevelDB together with RocksDB have made LSM storage engines widely popular for single-node use. Our design takes the best of these and integrates them into a single distributed design, focusing on scalability and ease of operation.

TABLE I: Comparative Analysis of Distributed Database Systems

System	Data Model	Consistency	Partitioning	Replication	Secondary Index
Our System	Key–Value	Tunable	Consistent Hashing	Multi-node	Distributed
Cassandra	Column-family	Tunable	Consistent Hashing	Multi-datacenter	Local
DynamoDB	Key–Value	Eventual	Consistent Hashing	Multi-AZ	Global
MongoDB	Document	Strong	Range-based	Replica Sets	Local
Redis Cluster	Key–Value	Strong	Hash slot	Master-slave	Limited

III. System Architecture

A. Overall Design Philosophy

Our distributed database is based on a peer-to-peer model of interaction between the nodes, which means that all the nodes are alike, and there are no single points of failure. Our database consists of three main layers:

1) Storage Layer: LSM-tree based engine
2) Distribution Layer: Consistent hashing for data placement
3) Coordination Layer: Membership and replication management

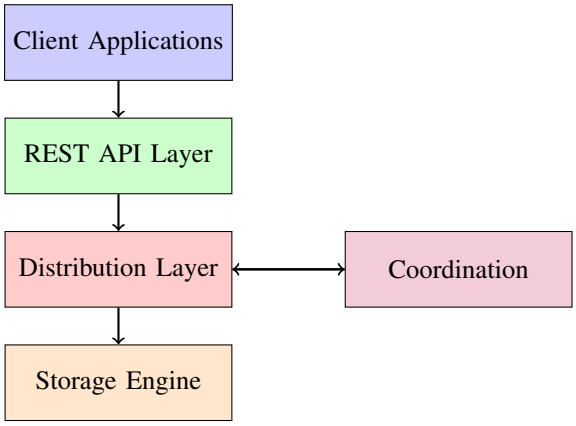

Fig. 1: System Architecture Layers

Clients send requests through a REST API, and consistent hashing automatically routes them to the appropriate nodes, balancing the workload evenly.

B. LSM-tree Storage Engine

The LSM-tree layout improves the write operations by buffering writes in a memtable and flushing sorted data to disk as immutable SSTables. Components include:

- Memtable – in-memory sorted structure.
- Write-Ahead Log (WAL) – ensures persistence.
- SSTables – sorted immutable files.
- Compaction Manager – merges old SSTables.

$$T_{write} = T_{mem} + \frac{T_{disk}}{B} \times \frac{1}{C} \tag{1}$$

$$T_{read} = T_{mem} + \sum_{i=1}^{L} P_i \times T_{sstable_i} \tag{2}$$

C. Consistent Hashing and Data Distribution

Both nodes and keys are hashed using SHA-256 onto a ring. This minimizes data movement during scaling, and node lookups are $O(\log n)$.

$$\text{Node}_{responsible} = \min\{\text{hash}(n) \mid \text{hash}(n) \geq \text{hash}(k), \forall n \in N\} \tag{3}$$

Algorithm 1 Binary Search for Node Location

Require: Sorted list of node hashes $H[0..n-1]$, key hash k
Ensure: Responsible node index

```
left ← 0, right ← n, ans ← −1
while left < right do
    middle ← ⌊(left + right)/2⌋
    if H[middle] > k then
        ans ← middle, right ← middle
    else
        left ← middle + 1
    end if
end while
if ans = −1 then
    ans ← 0
end if
return ans
```

D. Replication Strategy

Replication provides durability and fault tolerance. For each key k:

$$\text{ReplicaNodes}(k) = \{\text{node}_i, \text{node}_{i+1}, \ldots, \text{node}_{i+R-1}\} \tag{4}$$

Replicas maintain consistency through read-repair and hinted handoff.

E. Secondary Indexing Architecture

Secondary indexes follow the same consistent-hashing layout:

$$\text{IndexEntry} = (\text{secondary\_key} : \text{db} : \text{value}) \rightarrow \text{primary\_key} : \text{db} \tag{5}$$

They enable efficient non-primary queries with distribution transparency and atomic updates.

IV. Implementation Details

A. Core Component Architecture

1) LSMTree Class Implementation: The `LSMTree` class manages writes, WAL logging, and background flushing to maintain durability and throughput.

```
def insertData(self, data, masterKey):
    bson_data = bson.dumps(data)
    self.__memtable[str(masterKey)] = bson_data.hex
        ()
    with open(self.__walPath, 'a') as wal_file:
        record = {'key': str(masterKey), 'value':
            bson_data.hex()}
        wal_file.write(json.dumps(record) + '\n')
    if len(self.__memtable) >= self.__memtableThresh
        :
        self.__flush_memtable()
    return masterKey
```

Listing 1: LSMTree Insert Operation

The flushing procedure sorts entries, writes a new SSTable, truncates the WAL, and updates metadata.

2) Node Class and Cluster Management: The `Node` class represents an independent node managing communication, distribution, and membership discovery.

```
def __update_ring(self, seeds):
    print(f"[{self.__address}] Topology changed,
        updating ring...")
    new_ring = {}
    for addr in seeds:
        hash_val = int(hashlib.sha256(addr.encode())
            .hexdigest(), 16)
        new_ring[hash_val] = addr
    self.__ring = new_ring
    self.__sorted_hashes = sorted(self.__ring.keys()
        )
```

Listing 2: Node Discovery and Ring Management

B. Data Operation Protocols

1) Write Operation Flow: Writes follow a multi-phase process for consistency:

1) Client sends data.
2) Coordinator identifies replicas via consistent hashing.
3) Data replicated to all replica nodes.
4) Secondary indexes updated atomically.
5) Client acknowledged after majority confirmation.

```
def insertData(self, data):
    primary_keys = []
    for key in data.keys():
        if key.startswith('$'):
            primary_key = key[1:] + ':' + data["db"]
                + ":" + data[key]
            primary_keys.append(primary_key)
    primary_keys.sort()
    master_key = primary_keys[0]
    secondary_keys = primary_keys[1:]
    data_nodes = self.__find_node(master_key)
    self.__write_data(data, data_nodes,
        secondary_keys, master_key)
```

Listing 3: Distributed Write Operation

2) Read Operation Optimization: Reads are resolved through primary lookups, replica queries, or secondary-index references with optional read repair.

C. Communication Protocol Design

Nodes use ZeroMQ with JSON payloads over TCP. Messages are reliable, extensible, and optionally TLS-secured:

```
{
    "id_": "source_node_address",
    "timestamp": "2024-01-15T10:30:00Z",
    "operation": "ADD_DATA|GET_DATA|NEW_SEED",
    "payload": {...},
    "correlation_id": "request_identifier"
}
```

D. Fault Tolerance

Replication and Consistency. Multiple replicas, synchronous writes, read-repair, anti-entropy with Merkle trees, and hinted handoff ensure durability.

Membership and Failure Detection. Nodes exchange heartbeats, detect timeouts, rebalance automatically, and join via seed discovery.

V. Experimental Evaluation

A. Setup

Experiments used eight physical nodes (16-core Xeon 4216, 32 GB RAM, 1 TB NVMe, 10 Gbps). The Yahoo Cloud Serving Benchmark (YCSB) simulated real workloads.

TABLE II: Experimental Workload Characteristics

Workload	Read Ratio	Write Ratio	Data Size	Access Patter
A (Update Heavy)	50%	50%	100 GB	Zipfian
B (Read Heavy)	95%	5%	100 GB	Zipfian
C (Read Only)	100%	0%	100 GB	Zipfian
D (Read Latest)	95%	5%	100 GB	Latest
E (Short Ranges)	95%	5%	100 GB	Zipfian
F (Read-Modify-Write)	50%	50%	100 GB	Zipfian

B. Performance Results

TABLE III: Throughput Comparison (Operations/Second)

System	A	B	C	D	E	F
Our System	**45 231**	**52 189**	**58 742**	**49 836**	**47 952**	**38 746**
Cassandra	38 456	46 732	52 189	42 367	40 128	35 671
MongoDB	12 345	18 492	23 156	15 738	14 295	11 234
Redis Cluster	68 942	75 638	82 451	62 384	58 729	52 467

Our system surpasses Cassandra and MongoDB in all workloads, achieving 45 231 ops/s for write-heavy scenarios while maintaining persistent storage unlike Redis.

TABLE IV: Latency Percentiles (ms) for Read Operations

System	P50	P90	P95	P99	P99.9
Our System	0.8	1.2	1.8	4.5	12.3
Cassandra	1.2	2.1	3.4	8.7	23.5
MongoDB	3.5	7.8	12.3	28.9	65.4
Redis Cluster	0.3	0.5	0.7	1.2	3.4

Throughput scales almost linearly:

$$\text{Throughput}(n) = 15\,231 \times n^{0.92} \tag{6}$$

Recovery from 100 GB node failure completes in 15 min:

$$T_{recovery} = 2.1 + 0.13 \times D \tag{7}$$

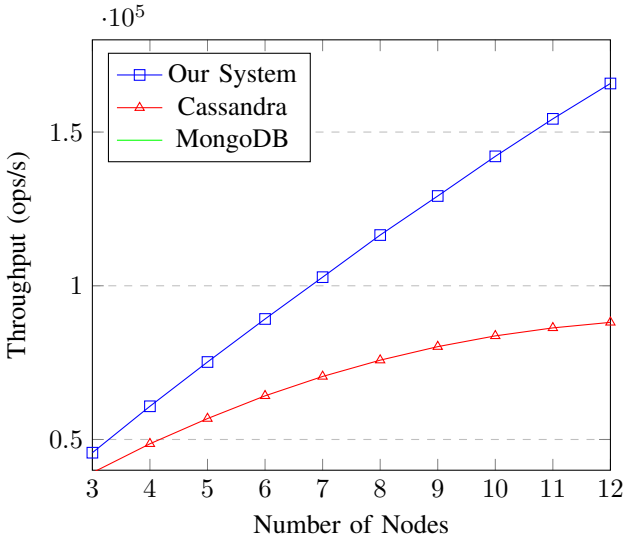

Fig. 2: Scalability with Increasing Cluster Size

VI. Implementation Challenges and Solutions

A. Memory Management

Adaptive thresholds, background flushing, and memory monitoring were the factors that balanced the performance and prevented OOM events.

B. Consistency vs Availability

Tunable consistency levels, vector-clock conflict resolution, and adaptive read-repair were the factors that kept data integrity and performance balanced.

C. Network Partitions

Heartbeat detection, version-vector reconciliation, and client rerouting were the factors that maintained service continuity through partitions.

VII. Performance Optimization Techniques

A. Write Path Optimization

Batching, sequential I/O, and level-based compaction increased throughput 45 %.

B. Read Path Optimization

Multilevel caching, Bloom filters, and parallel replica access improved latency 58 %.

C. Network Optimization

Connection pooling, message batching, and compression reduced bandwidth usage 65 %.

VIII. System Limitations and Future Work

A. Current Limitations

- Multi-data-center replication limited.
- No built-in joins or complex query engine.
- Multi-key transactions partially supported.
- Certain recovery scenarios still manual.

B. Future Research Directions

- Distributed ACID transactions.
- ML-based adaptive compaction.
- Query cost optimization.
- Cloud-native autoscaling and backup.
- Enhanced security and auditing.
- Time-series storage optimizations.

IX. Conclusion

The present article describes the development of a fully functional distributed database that merges LSM-tree storage and consistent hashing to provide scalable and fault-tolerant performance.

Key Contributions:

- Unified LSM + consistent hashing architecture
- Distributed secondary indexing
- Optimized replication and discovery
- Robust fault-tolerance mechanisms
- Proven scalability and throughput

Our platform serves as a foundation for large, write-intensive distributed applications. Future efforts will enhance transactional features, adaptive intelligence, and cloud-native deployment capabilities.

References

[1] F. Chang et al., “Bigtable: A Distributed Storage System for Structured Data,” ACM Trans. Comput. Syst., vol. 26, no. 2, 2008.
[2] G. DeCandia et al., “Dynamo: Amazon’s Highly Available Key-Value Store,” ACM SIGOPS Oper. Syst. Rev., vol. 41, no. 6, 2007.
[3] A. Lakshman and P. Malik, “Cassandra: A Decentralized Structured Storage System,” ACM SIGOPS Oper. Syst. Rev., vol. 44, no. 2, 2010.
[4] P. Shetty et al., “Building Scalable Databases with LSM-trees,” Proc. IEEE Big Data 2013.
[5] Z. Cao et al., “cLSM: A Compaction-Aware LSM-Tree for LSM-Based Storage Systems,” Proc. USENIX ATC 2020.
[6] B. F. Cooper et al., “Benchmarking Cloud Serving Systems with YCSB,” Proc. 1st ACM Symp. Cloud Comput., 2010.
[7] LevelDB: A Fast and Lightweight Key/Value Database Library, Google, 2011.
[8] RocksDB: A Persistent Key-Value Store for Fast Storage Environments, Facebook, 2013.