Scientific Research Publishing

Journal of Computer and Communications, 2024, 12, 1-20
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

SQL or NoSQL—Practical Aspect and Rational behind Choosing Data Stores

Sourabh Sethi[1], Sarah Panda[2]

[1]Digital Experience, Infosys Limited, New York, USA
[2]Research and Incubation, Microsoft Inc., Seattle, USA
Email: sourabhsethi@ieee.org, sp3206@columbia.edu

How to cite this paper: Sethi, S. and Panda, S. (2024) SQL or NoSQL Practical Aspect and Rational behind Choosing Data Stores. *Journal of Computer and Communications*, **12**, 1-20.
https://doi.org/10.4236/jcc.2024.128001

Received: April 22, 2024
Accepted: August 4, 2024
Published: August 7, 2024

Abstract

Data storage solutions are a crucial aspect of any application, significantly impacting data management and system performance. This article explores the rationale behind utilizing both SQL and NoSQL databases, addressing key questions about when each type is preferable. The background emphasizes the importance of selecting the appropriate database technology to meet specific application requirements. The purpose of this research is to provide a comprehensive guide for choosing between SQL and NoSQL databases based on various factors, including workload characteristics, scalability needs, and consistency requirements. To achieve this, we examine different strategies for implementing SQL and NoSQL databases in large-scale distributed applications and systems. The research method involves a comparative analysis of the features, advantages, and limitations of both database types. We specifically focus on scenarios involving read-heavy versus write-heavy systems and the trade-offs between availability and consistency. The results of this research indicate that SQL databases, with their relational structure and ACID compliance, are ideal for applications requiring complex queries and data integrity. In contrast, NoSQL databases, offering schema flexibility and horizontal scalability, are better suited for managing extensive datasets and high-velocity data ingestion. In conclusion, the selection of a database depends on the specific needs of the application. SQL databases are preferred for transactional systems with complex relationships, while NoSQL databases excel in scenarios demanding flexibility and scalability. The study provides insights into hybrid approaches, leveraging both database types to optimize system performance.

Keywords

SQLData Stores, NO-SQLData Stores, ACID, BASE, RUM Conjecture

1. Introduction

SQL databases are structured as relational databases, comprising interconnected tables, each with a fixed set of columns. The ability to query across tables facilitates the retrieval of related information. A key requirement for SQL databases is to store data in normalized form to prevent redundancy and ensure consistency across tables.

For instance, consider two tables storing a particular score. If one of the scores is altered due to an update operation, confusion may arise with two different scores in two tables. Normalizing the data helps avoid redundancy and trust issues. SQL data stores offer ACID guarantees, where atomicity ensures a transaction is either entirely successful or entirely rolled back, consistency ensures data consistency before and after a transaction, isolation ensures independence of two transactions, and durability ensures changes persist even after system reboots or crashes. While the fixed schema in SQL has advantages, it may not fit every use case. For instance, a fixed schema might not accommodate various product attributes in an e-commerce website efficiently. A t-shirt and a MacBook Air have vastly different attributes, making a single table impractical. SQL's design for handling millions of records in a single table, not millions of tables, poses challenges with flexible schemas. Additionally, if data sharding or partitioning is required, SQL's advantages diminish. Performing SQL queries across machines after sharding becomes difficult and costly. In addressing these issues, NoSQL databases come into play. For example, when sharding is necessary, NoSQL databases offer solutions. The first step involves choosing a sharding key, and the second step includes performing denormalization. However, denormalization can lead to data redundancy and inefficiencies. Careful consideration of the sharding key is essential to mitigate such problems in NoSQL databases.

Table 1. Database comparison.

Database Name	Type	Fast Writes	Read Performance	Scalability	Data Consistency	Query Complexity	Transaction Support	Security Features	Best Use Cases
MySQL	SQL	Moderate	High	Horizontal	Strong	Moderate	Yes	Moderate	Web applications, Small to medium-sized business systems
PostgreSQL	SQL	High	High	Both	Strong	High	Yes	High	Complex queries, Analytical workloads, Enterprise applications
MongoDB	NoSQL	High	Moderate	Horizontal	Eventual	Simple	Limited	Moderate	Big data, document storage, real-time analytics
Oracle	SQL	High	Very High	Both	Strong	High	Yes	Very High	Large enterprises, High transaction processing, Financial systems
SQL Server	SQL	Moderate	High	Both	Strong	High	Yes	High	Business intelligence, Integrated business solutions, Corporate IT systems

Continued

Cassandra	NoSQL	Very High	Moderate	Horizontal	Eventual	Simple	Limited	Moderate	Large, distributed environments, Time-series data
Redis	NoSQL	Very High	Very High	Horizontal	Eventual	Simple	No	Moderate	Caching, Session storage, Real-time applications
SQLite	SQL	Moderate	Moderate	None	Strong	Moderate	Yes	Moderate	Local storage, Mobile applications, Embedded systems
DynamoDB	NoSQL	High	High	Horizontal	Eventual	Simple	Limited	High	Serverless applications, Web-scale applications
MariaDB	SQL	High	High	Horizontal	Strong	High	Yes	High	Web applications, Database replication, Clustering
Couchbase	NoSQL	High	High	Horizontal	Strong	Moderate	Yes	High	Interactive applications, Mobile and IoT
Neo4j	NoSQL	Moderate	High	Horizontal	Strong	High	Yes	Moderate	Graph-based applications, Network analysis
Elasticsearch	NoSQL	High	Very High	Horizontal	Eventual	Simple	No	Moderate	Search engines, Log analytics, Real-time analysis
HBase	NoSQL	High	Moderate	Horizontal	Strong	Simple	Limited	Moderate	Big data applications, Column-oriented storage
InfluxDB	NoSQL	High	High	Horizontal	Eventual	Moderate	Yes	Moderate	Time-series data, Real-time analytics, Monitoring

These data stores in Table 1 exhibit three key properties: buffering, immutability, and ordering, which are instrumental in describing, memorizing, and expressing various aspects of the storage structure. Selecting the proper physical design—through static auto-tuning, online tuning, or adaptively—and access method has been a key research challenge in data management systems for several decades. The physical organization of data on storage devices (disk, flash, memory, caches) defines and restricts the possible ways that data can be read and updated. As applications evolve rapidly and continuously, the underlying hardware also diversifies and changes quickly with new technologies and architectures. Both trends introduce new challenges in designing data management software. Reviewing existing access method proposals reveals recurring fundamental challenges and design decisions. Specifically, researchers consistently aim to minimize three primary overheads: 1) read overhead (R), 2) update overhead (U), and 3) memory (or storage) overhead (M)—collectively known as the RUM overheads. Deciding which overhead (s) to optimize and to what extent remains a critical part of designing new access methods, especially as hardware and workloads evolve. For instance, in the 1970s, a crucial aspect of every database algorithm was minimizing the number of random disk accesses; 40 years later, a

similar strategy is applied to minimize the number of random accesses to main memory.

A prominent cost model for storage structures, known as the RUM conjecture, factors in three crucial elements: Read, Update, and Memory overhead. According to this conjecture, reducing two of these overheads inevitably worsens the third, necessitating optimization at the expense of one parameter. Comparing storage engines based on these parameters sheds light on their optimization strategies and potential trade-offs. Ideally, a solution would minimize read costs while keeping memory and write overheads low, but achieving this balance is often unattainable, leading to trade-offs. B-Tree structures prioritize read optimization, yet writing to them involves locating records on disk and potentially updating disk pages multiple times, resulting in increased space overhead due to reserved extra space for future updates and deletes. Conversely, LSM (Log-Structured Merge) trees eliminate the need to locate records on disk during writes and do not reserve additional space for future writes. However, in the default configuration of LSM tree-based data stores, reads are more costly since multiple Sorted String Tables (SSTs) must be accessed to retrieve complete records. The table below illustrates how these three properties can be combined to achieve desired characteristics.

Table 2. Storage structure.

Storage Structure	Buffered	Mutable	Ordered
B + Trees	No	Yes	Yes
WiredTiger	Yes	Yes	Yes
La-Trees	Yes	Yes	Yes
COW B-Trees	No	NO	Yes
2C LSM Trees	Yes	No	Yes
MC LSM Trees	Yes	No	Yes
FD-Trees	Yes	No	Yes
BitCask	No	No	No
Wisckey	Yes	No	Yes
BW-Trees	No	No	No

As per the RUM Conjecture [1], developing an access method for a storage system that excels in all three crucial aspects—Reads, Updates, and Memory utilization—simultaneously is deemed unfeasible. This conjecture suggests that optimizing one aspect inevitably comes at the cost of the other two, leading to a competitive triangle akin to the renowned CAP theorem, where the three components are inherently at odds with each other.

Access methods optimized for reads prioritize minimizing read overhead. Examples include indexes offering constant or logarithmic time access, such as hash-based indexes, B-Trees, Tries, Prefix B-Trees, and Skiplists. While these

methods generally provide rapid read access, they often result in increased space overhead and may encounter challenges with frequent updates.

On the other hand, write-optimized structures aim to reduce the write overhead associated with in-place updates by utilizing secondary differential data structures as mentioned in Table 2. The core concept involves consolidating updates and applying them in bulk to the base data. Examples include the Log-structured Merge Tree, Partitioned B-tree (PBT), Materialized Sort-Merge (MaSM) algorithm, Stepped Merge algorithm, and Positional Differential Tree. Notably, write-optimized trees like LA-Tree and FD-Tree focus on leveraging flash storage efficiently while accommodating its limitations, such as the disparity between read and write performance and the finite number of physical updates flash can handle. While such structures generally perform well under updates, they tend to increase read costs and space overhead. Space-efficient access methods aim to minimize storage overhead. Examples encompass compression techniques and lossy index structures like Bloom filters, lossy hash-based indexes such as count-min sketches, bitmaps with lossy encoding, and approximate tree indexing. Sparse indexes, including ZoneMaps, Small Materialized Aggregates, and Column Imprints, also belong to this category. Typically, these methods substantially reduce space overhead but may increase write costs (e.g., due to compression) and occasionally elevate read costs as well (e.g., with a sparse index).

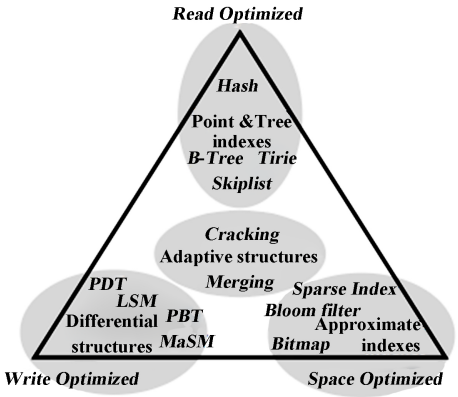

Figure 1. RUM conjecture.

All mentioned data stores in Table 1 could be categorized according to the Storage Structure defined in Table 2 and RUM Conjecture in Figure 1. It is the developer's responsibility to choose the correct data store according to the application's use case and their access pattern.

Sharding and Choosing Shard Key

Consider a banking system where users can have active bank accounts in various cities. The most frequent operations include balance queries, fetching transaction history, retrieving a list of user accounts, and creating new transactions. In this context, selecting an appropriate sharding key is crucial. CityID

would be a suboptimal choice for a sharding key, as users may move between cities, necessitating the migration of data across different city-based shards. Furthermore, the uneven distribution of user populations across cities can lead to load-balancing issues. A more effective sharding key is UserID, as it consolidates all user-related information in a single shard. This approach ensures that operations like balance queries, transaction history retrieval, and user account management are confined to the machine holding that user's data, thereby facilitating efficient load distribution and minimizing inter-machine communication.

In a system akin to Uber, where the primary use case is searching for nearby drivers, CityID appears to be a suitable sharding key. This choice enables searches to be limited to cabs within the user's city, handling common use cases effectively. However, DriverID is not a viable option as nearby drivers could be on any machine, necessitating queries to multiple machines, and incurring high costs. Sharding by PIN CODE is also suboptimal as cabs frequently traverse regions with different pin codes.

Considering the Indian Railway Catering and Tourism Corporation (IRCTC), where the main purpose is ticket booking involving TrainID, date, class, and UserID, the system aims to address double-booked tickets and load balancing, especially during peak times such as tatkal bookings. Date of Booking is an unsuitable sharding key because it would overload the machine handling trains for the next day. UserID poses challenges in preventing the same ticket from being assigned to multiple users, leading to consistency issues. A good sharding key in this case is TrainID, as loads are distributed among trains, addressing the shortcomings of Date and UserID as sharding keys.

For a messaging system resembling Slack, which is group-heavy, where groups may consist of up to 100,000 users, UserID is not ideal due to the need for multiple write operations across different machines for a single message in a group or channel. In this context, GroupID emerges as the best sharding key. It allows for single writes corresponding to messages and events, facilitates storage of all channels of a user in one machine, supports lazy fetch, and enables asynchronous retrieval of unread messages and channel updates. Considering the specific use case of Slack, GroupID proves to be a more logical choice.

Consider the following guidelines when selecting sharding keys:

Aim for a uniform distribution of load across all machines to ensure optimal system performance. Prioritize the efficient execution of the most frequent operations to enhance overall system responsiveness. Minimize the number of machines that need updates during high-frequency operations to maintain database consistency effectively. Strive to minimize redundancy within the system to improve storage efficiency and reduce unnecessary data duplication.

2. Literature Review

BigTable, Dynamo, PNUTS, MongoDB, CouchDB, and Cassandra do not pro-

vide ACID transactions. RAMCloud is an in-memory key-value system that supports only single-object transactions. Google's Percolator, Apache Tephra, and Omid add transactional APIs on top of key-value stores with snapshot isolation. FoundationDB (FDB) supports strictly serializable ACID transactions on a scalable key-value store, which has been used to support flexible schema and richer queries. Similar SQL-over-NoSQL architectures are adopted in Hyder, Tell, and AIM. Many systems establish the serial order among transactions and ensure atomicity and isolation by using the time when all locks are acquired. For example, Spanner uses TrueTime to determine commit timestamps upon acquiring all locks, while CockroachDB employs a hybrid-logical clock combining physical and logical time. Like FDB, several systems order transactions without locks. H-Store, Calvin, Hekaton, and Omid execute transactions in timestamp order. Hyder, Tango, and ACID-RAIN use a shared log to establish ordering, whereas Sprint employs total-order multicast. FDB ensures strict serializability with a lock-free concurrency control combining MVCC and OCC, with the serial order determined by a Sequencer. These databases separate the transaction component (TC) from the data component (DC). Deuteronomy creates virtual resources that can be logically locked in the transaction system, while the DC remains unaware of transactions, their commits, or aborts. Solar combines scalable storage on a cluster of nodes with a single server for transaction processing. Amazon Aurora simplifies database replication and recovery using shared storage. Systems like Tell use advanced hardware to achieve high performance and implement snapshot isolation with a distributed MVCC protocol, while FDB uses commodity hardware with serializable isolation. In FDB, the TC is decomposed into several dedicated roles, and transaction logging is decoupled from the TC, enabling lock-free concurrency management with a deterministic transaction order. Traditional databases tightly couple the transaction and data components. Silo and Hekaton achieve high throughput using a single server for transaction processing, while many distributed databases partition data to scale out. Systems like FaRM and DrTM exploit advanced hardware to enhance transaction performance. FDB adopts an unbundled design with commodity hardware in mind. Traditional databases often use ARIES-based recovery protocols. VoltDB employs command logging, starting recovery from a checkpoint and replaying commands in the log. NVRAM devices have been used to reduce recovery time. Amazon Aurora decouples redo log processing from the database engine using smart storage, leaving only the undo log for the database engine. RAMCloud performs parallel recovery of redo logs across multiple machines, with recovery time proportional to log size. In contrast, FDB completely decouples redo and undo log processing from recovery by separating log servers and storage servers.

3. Datastore Analysis Methodology

There are generally four types of NoSQL datastores: Key-Value datastores, Document DBs, Column-Family Storage, and Graph Databases. Key-Value

NoSQL DBs: Examples include Redis and DynamoDB. In these databases, data is stored in the form of key-value pairs, resembling a hashmap. The values are untyped, akin to a hashmap from string to string. Document DBs: Examples include MongoDB and AWS ElasticSearch. In this type, data is structured in JSON format, where each record is akin to a JSON object with different attributes. This format is beneficial for applications with numerous product categories, offering a tabular structure for efficient searching. Column-Family Storage: Examples include Cassandra and HBase. In this system, the sharding key constitutes the RowID. Each RowID contains multiple column families, akin to tables in SQL databases. Within each column family, multiple strings are stored as records, sorted by timestamp in descending order. This structure allows for efficient prefix searching and retrieval of the top or latest X entries. Column-Family Storage is particularly useful for applications with countable schemas, and it excels in implementing pagination, especially when pagination is required on multiple attributes. These NoSQL databases cater to various application scenarios, offering flexibility and efficiency in handling different types of data structures and access patterns.

Choosing Database

Consider a Twitter hashtag data storage system where the goal is to store the most popular or latest tweets associated with a hashtag. The system must support incremental fetching of tweets, such as retrieving the first 10 tweets initially and then fetching subsequent batches as users scroll through the application. In this context, a Key-Value database is not suitable. The issue is that when fetching information for a particular tweet (key), all associated tweets are retrieved. Even if only 10 tweets are needed, the entire set, potentially comprising 10,000 tweets, is fetched, causing delays and a poor user experience. Conversely, a Column-Family system is more appropriate. By using the tweet as a sharding key, column families like Tweets and Popular Tweets can be utilized. To retrieve posts related to a tweet, queries can access only the first X entries of the tweets column family. Additional tweets can be fetched by specifying an offset and retrieving records from that point, ensuring efficient and incremental data retrieval [2].

Now, consider a system that deals with the live score of matches or sports events. In this scenario, where the goal is to display ongoing score information for a recent event or match, a Key-Value DB is the optimal choice. The simplicity lies in accessing and updating the value corresponding to a particular match per key, making it a lightweight solution. Another example involves the current location of cabs in Uber-like systems. For displaying the live location of cabs, the choice depends on whether location history is required. If location history is needed, a Column-Family DB emerges as the best choice. By using the cab as a sharding key and a column family for location, fetching the first few records of the Location column family for a specific cab suffices. Additionally, new location

records can be seamlessly inserted into the Location column family. On the other hand, if only the current location is needed and historical data is irrelevant, a Key-Value DB becomes the more sensible choice. This approach allows for the straightforward retrieval and updating of the value corresponding to the cab (key).

4. NoSQL Internals

In contrast to SQL, NoSQL is unstructured and lacks a fixed size. Designing a system to ensure efficient updates is crucial. While SQL typically involves both write and read operations taking log (N) time, how can we approach the design of a NoSQL system? Moreover, what adjustments can be made for systems with a heavy emphasis on either reads or writes?

Most NoSQL systems incorporate two types of storage: Write-Ahead Log (WAL) and the current state of data. The Write-Ahead Log is essentially an append-only log capturing every write (new write/update) occurring in the database. Theoretically, starting from scratch, one can replay these logs to reconstruct the final state of the database. Visualize this as a sizable file where entries are only appended, and in most cases, seldom read. If reads are performed, they typically involve requesting a tail of this file, representing entries after a specific timestamp (the last Y number of entries in the file).

Now, if we were to consider fixed-size entries for each row-column pair, as seen in SQL, B-Trees could be employed to store these entries. For simplicity, let us focus on a key-value store. How would one rudimentarily store key-value pairs? A straightforward approach might involve storing all keys and values in a file as shown in Table 3.

Table 3. Key-value.

Key	Value
ID 001	John
ID 002	Karen
ID 005	Bill
ID 003	Scott

Now, envision a scenario where a request is made to update the value of "ID 002" to "Ram". The brute force method would involve searching for "ID 002" in the file and modifying the corresponding value. However, if there is a subsequent read request for "ID 002", the entire file must be scanned again to locate the key "ID 002". This process appears notably sluggish; resulting in both reads and writes being slow. It is crucial to acknowledge that the value is not of a fixed size. Additionally, in situations where multiple threads attempt to update the value of "ID 002", they would need to acquire a write lock, further impeding efficiency. No-Sqldata stores devise a more efficient solution to address these challenges. All new writes were just appended to the file as shown in Table 4.

Table 4. Add updated entry in last.

Key	Value
ID 001	John
ID 002	Karen
ID 005	Bill
ID 003	Scott
ID 002	Ram

This approach introduces the possibility of duplicate keys, but it significantly boosts the speed of my write operations. To address reads, data stores can search for keys from the end of the file and halt at the first matching key encountered, representing the latest entry. Consequently, while reads may remain slow, data stores have successfully accelerated write processes. However, a drawback emerges as this method leads to duplicate entries and may necessitate additional storage. Essentially, this approach implies that each entry is immutable, signifying that once written, an entry is not subject to editing. Consequently, writes no longer require locks. Despite the enhancement in write speed, reads still pose a challenge, operating at O (N) in the worst-case scenario. Is there a way to further optimize this process? What if data stores could establish an index for the keys? Figure 2 having an in-memory index (like a hashmap) that stores the locations of the keys in the file, indicating the offset in bytes to seek and read the latest entry pertaining to a specific key as shown in Table 5.

Table 5. In-memory [key, offset].

Key	Value	In-Memory Key Offset	
ID 001	John	ID 001	0
ID 002	Karen	ID 002	80
ID 005	Bill	ID 005	40
ID 003	Scott	ID 003	60
ID 002	Ram		

This way, Figure 2 shows the read operation flow:

```
def read(key):
  offset = InMemoryHashmap[key]
  bytes = Read `n` bytes from file with key-value starting at offset `offset`
  key, value = parse(bytes)
  return value
```

Figure 2. Read operation.

Now, the write operation is no longer a simple append to the file; it includes an additional step of updating the in-memory hashmap. This modification ensures that reads no longer necessitate scanning the entire file, alleviating the

O(N) constraint. However, a significant flaw arises in assuming that all keys and offsets can fit into memory. In reality, a key-value store may have billions of keys, making it impractical to store such a map in memory. How do No-SQL datastores tackle this challenge? Additionally, considering the need for substantial memory to store duplicate older entries, datastore must address this issue as well. Let us address these concerns one by one. To enhance storage efficiency, one solution is to implement a background process that reads the file, eliminates duplicates, and generates another file while updating the in-memory hashmap with new offsets. Although the idea is sound, its implementation is complex due to the enormity of these files. Quickly identifying duplicates and reading the entire file in chunks present challenges. Datastores reads the file in 100MB chunks, which are structured it as separate files for each chunk instead of a single file? This approach allows the latest chunk to be in memory (referred to as the "memTable"), writable to disk when near full. The latest chunk, being the most frequently written to, is likely to contain the most recent entries for frequently queried items. Notably, MemTable, being an in-memory hashmap, avoids duplicate entries.

Concurrently, node can merge existing immutable chunks (e.g., chunkX, chunkY) into new chunks (e.g., chunkZ). After removing duplicate entries, node delete chunkX and chunkY once chunkZ is created, updating the in-memory hashmap. This process is termed "compaction." While temporary duplicates may exist across older chunks, periodic compaction ensures the consolidation of duplicate entries. The compaction process can be scheduled during off-peak traffic hours to minimize performance impact during peak times. Despite these improvements, the challenge remains regarding the in-memory hash map's size and its potential to exceed available memory. Now that new writes are directed to the memTable, the question arises: is storing keys in a random order truly optimal. How can No-Sql Data stores enhance file searching without relying on a hashmap that stores entries for all keys? Here data store do sorting.

What if the memTable had all entries sorted? One way to achieve this is by utilizing a balanced binary tree, such as a Red-Black Tree, AVL tree, or Binary Search Tree with rotations for balancing. When the mem Table is full and its content is flushed to disk, it can be done in sorted order of keys, similar to how a TreeMap allows iteration in sorted order. These sorted files can be termed SSTables (Sorted String Tables). With a sorted order, binary search becomes feasible within the file. However, performing a binary search in the file poses a challenge because landing on a random byte in the file offers no indication of which key/value this byte corresponds to. To address this, the file can be partitioned into blocks of 64KB each. For example, a 1GB file would consist of approximately 16,000 blocks. In an index associated with each block, one entry is stored, representing the first key in that block. This index, too, maintains sorted entries, resembling a TreeMap. Consider the following schematic representation in **Figure 3**.

Consider Figure 3: Picture a scenario where a request is made for ID-1234. In this case, a binary search would be conducted to find the last entry or the highest entry with a block_key less than or equal to the current key being sought (essentially, the block before index. upper_bound (current_key)). By doing so, it becomes evident in which block the desired key is located, requiring the scanning of only 64KB of data to retrieve the necessary information. It is worth noting that this index is assured to fit within the available memory. The concept we have outlined is commonly referred to as the LSM Tree.

In summary: There is an in-memory MemTable structured with entries stored as a TreeMap. All new writes are directed here, and if a key already exists, the entry in the MemTable is overwritten. A collection of SSTablesis maintained, where keys are sorted and segmented into blocks. Multiple SSTablescan be envisioned as being linked together in a manner reminiscent of a LinkedList, with the newest SSTable positioned at the forefront. An in-memory index of blocks within SSTables is established. Periodically, a compaction process is initiated to merge multiple SSTables into a single SSTable, thereby eliminating duplicate entries. This process closely resembles performing a merge sort on multiple sorted arrays stored on disk.

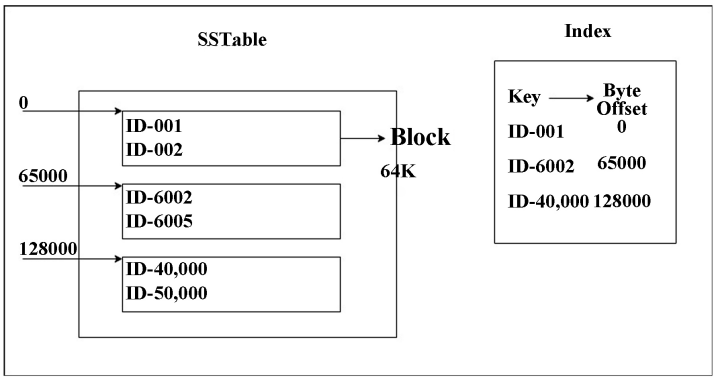

Figure 3. SSTable.

Write Operation: This is plainly an addition/update to the MemTable TreeMapas shown in Figure 4.

```
def write(key, value):
    memTable[key] = value;
```

Figure 4. Write operation.

Read Operations: If the entry is located in the MemTable, excellent! Retrieve and return it. In case it's not found, proceed to the newest SSTable, attempt to locate the entry there (using upper_bound - 1 on the index TreeMap to find the relevant block, and then scanning the block). If the entry is found, return it. If not, move on to the next SSTable and repeat the process (see Figure 5). If the entry is not found in any SSTable, then return "Key does not exist". As shown in Figure 6.

```
def flushProcess():
    if memTable.size() <= THRESHOLD:
        return
    ss_new = new SSTable
    block = []
    key = memTable first key
    for k,v in memTable.items():
        #memTable is a TreeMap. So sorted order.
        newEntry = entry(k, v))
        if size(block) + size(newEntry) > BLOCK_THRESHOLD:
            index[ss_new.id][key] = size(ss_new) + 1 #last byte in ss_new file
            ss_new.write(block) # Flush block to SSTable
            block = [] # Start a new block
            key = k
        block.append(newEntry)

    if size(block) > 0:
        index[ss_new.id][key] = size(ss_new) + 1
        ss_new.write(block)
    ss.writeToDisk() # SSTable is ready.
    listOfSSTables.add(offset of ss, number of blocks in ss).
```

Figure 5. Flush MemTable to disk.

```
def read(key):
    if memTable.hasKey(key):
        return memTable[key]
    for ss in listOfSSTables:
        relevantKeyBlock = the block before index[ss.id].upper_bound(key)
        if key found in scan of block:
            return value
    return "key does not exist"
```

Figure 6. Read operation.

What occurs when the machine containing this entry undergoes a reboot or restart? Given that everything in the MemTable is stored in RAM, it would be lost. The Write-Ahead Log (WAL) comes to our aid in this situation. Prior to resuming operations, the machine must replay logs generated after the last disk flush to reconstruct the accurate state of the MemTable? As all operations are performed in memory, the replaying of logs can be executed rapidly, with the slowest step being the reading of Write-Ahead Log logs from the disk.

SST (Sorted String Table), LSM (Log Structure Merge) Tree, and Memtable extend to column family stores, where updates are appended to a specific column family (CF), and read requests seek the last X entries (last X versions)? The fundamental structure remains mostly unchanged, with a few adjustments: During compaction, merging involves both entries rather than solely relying on the latest entry. In the case of writes, entries are appended in the MemTable to the combination of rowKey and column Family. For reads requesting the last X entries: Check the number of entries available in the MemTable. If there are X entries, return them. If not, continue reading from SSTables until X entries are found or there are no more SSTables left. How does the deletion of a key operate? No-SQLData store employs an approach where deletion is treated as another (key, value) entry, where a unique value is assigned to denote a tombstone. If the most recent value encountered is a tombstone, the system can promptly return “key does not exist”, a read operation for a key that is not found can be

quite costly. Searching through every sorted set implies scanning multiple 64KB blocks before determining that the key does not exist, resulting in a significant amount of work for minimal returns. Adding the probabilistic data structure called as Bloom Filter—a filter that operates as follows: Function: doesKeyExist (key): Returns false: Key definitely does not exist. Returns true: Key may or may not exist. Therefore, if the function returns false, data store can promptly conclude that the "key does not exist" without the need to scan SSTables. The accuracy of bloom function directly correlates with the level of optimization achieved. Additionally, a crucial prerequisite is that the Bloom Filter must be space-efficient, fitting into memory while utilizing as little space as possible.

5. Acid vs Base Datastores

The CAP theorem asserts that achieving both consistency and availability in a partition-tolerant distributed system is impossible, especially during temporary communication breakdowns. The ACID model ensures system consistency, making it well-suited for businesses engaged in online transaction processing (e.g., financial institutions) or online analytical processing (e.g., data warehousing). ACID transactions guarantee consistent states, and choosing a relational database management system like MySQL, PostgreSQL, Oracle, SQLite, or Microsoft SQL Server is a reliable way to ensure ACID compliance. While some NoSQL database management systems, such as Apache's Couch DB or Apache's Casandra, exhibit a certain degree of ACID compliance, the overall philosophy of NoSQL contradicts strict ACID rules. Consequently, NoSQL databases are not recommended for environments that demand strict adherence or total order broadcasting. The emergence of NoSQL databases introduced a flexible approach to data manipulation, leading to the creation of a new model reflecting these properties: Basically Available: BASE-modeled NoSQL databases prioritize data availability by distributing and replicating it across database cluster nodes. Soft State: The BASE model deviates from enforcing immediate consistency, leaving it to developers to manage. Eventually Consistent: BASE achieves consistency over time, allowing data reads even before full consistency is reached. Marketing and customer service companies engaged in sentiment analysis favor BASE's elasticity for social network research, handling vast amounts of unstructured data found in social network feeds. Similar to how SQL databases are predominantly ACID-compliant, NoSQL databases tend to align with BASE principles. MongoDB, Cassandra, Redis, Amazon DynamoDB, and Couchbase are popular NoSQL solutions. Choosing between ACID and BASE depends on project considerations. ACID-compliant databases are preferable for those valuing consistency, predictability, and reliability due to their structured nature. Conversely, those prioritizing growth might opt for the BASE model, offering scalability and flexibility at the expense of developer familiarity with its limitations.

6. Database Abstraction

Various abstraction layers are available in the Java/Spring ecosystem, such as

Spring Data JPA and Spring Data JDBC, which emphasize the development of business rules and models by concealing the internal implementation details of databases. This approach aligns with the dependency inversion principle. Our code relies on the abstraction layers provided by Spring Data interfaces, effectively concealing the intricate low-level details of different implementations. As application developers, our primary focus is on crafting business and enterprise applications while providing simplified configurations, allowing the Spring abstraction to handle Object-Relational Mapping (ORMs) and data stores [3]. Within this ecosystem, Spring Data JDBC, a component of the extensive Spring Data family, simplifies the implementation of JDBC-based repositories. This module offers enhanced support for data access layers based on JDBC, streamlining the development of Spring-powered applications that harness data access technologies.

Similarly, Spring Data JPA, another component of the broader Spring Data family, facilitates the seamless implementation of repositories based on the Java Persistence API (JPA) [4]. It simplifies the process of building Spring-powered applications that rely on data access technologies. Creating a data access layer for an application can be a cumbersome task, often involving the writing of extensive boilerplate code for even the most straightforward queries. When additional features like pagination, auditing, and other commonly required options are added, the complexity can become overwhelming. Spring Data JPA aims to address this challenge by significantly reducing the effort required for implementing data access layers to only the essential elements. As a developer, one can define repository interfaces using various techniques, and Spring will automatically handle the wiring. Additionally, developers have the flexibility to use custom finders or employ query-by-example, with Spring taking care of generating the queries for developers [5].

7. Transaction Management

All databases offer transaction support, but they vary in their levels of isolation and consistency models to ensure high availability, which encompasses both liveliness i.e. availability and safety, i.e. correctness such as consistency [6]. There are four levels of Isolation, Read Uncommitted, Read Committed, Repeatable Read, and Serializable, and four consistency models like Linearity, Causal Consistency, Eventual Consistency, and Majority Quorum, are crucial features supported by general-purpose databases. **Table 6**: Isolation Vs Efficiency compares Isolation level and efficiency i.e. Liveliness or availability. Read committed is a very popular isolation level, it is the default setting in Oracle 19c, PostgreSQL, SQL Server 2022, MemSQL.

[1] The Read Committed isolation level presents an anomaly known as non-repeatable reads or read skew. To address this issue, Snapshot Isolation emerges as a prevalent solution. The concept revolves around each transaction accessing a consistent snapshot of the database, ensuring that the transaction perceives all data committed in the database at the transaction's initiation. Even if subsequent transactions alter the data, each transaction exclusively observes

the old data from the specific point in time. Snapshot isolation enjoys widespread adoption and is supported by databases such as PostgreSQL, SQL Server 2022, and MySQL with the InnoDB Storage engine. Similarly, Repeatable Reads entail anomalies like Lost Updates, Write Skew, and Phantom Reads, which can be mitigated through Serializable isolation. Weak isolation levels provide partial protection or correction against these anomalies but necessitate manual handling by the application developer using explicit locking. Only Serializable isolation guarantees defense against all anomalies. Achieving Serializable isolation can be approached through three different methods: executing transactions in a literal serial order, employing Two-Phase Locking, and implementing Serializable Snapshot Isolation at the cost of compromising liveness or availability.

Table 6. Isolation vs Efficiency.

Isolation Level	Efficiency	Implementation	Explanation
Read Uncommitted	Highest	Single Data Entry	Only need a single entry in the database and it is overwritten whenever there is an update.
Read Committed	Average	Local Copy of changed values	If clients are making an update to a key then the older value of the key stays in the database and the newer value is kept in the local copy till the commit finally goes through.
Repeatable Read	Average	Versioning Of Unchanged Value	Transaction takes the value that it cares about but transaction are not changing and keep a version of them. For every key txn will store all the values that it has ever has in different transaction commits. It achieves my Multi Version Concurrency Control.
Serializable	Low	Queued Locks	Transaction uses causal ordering here. If two transactions use queries, foe the same key then they must be ordered. Transactions that do not have any conflict can run concurrently.

We have delved into Transaction Isolation levels, but it's equally essential to explore data consistency levels when selecting the appropriate data store for specific use case. Among the four consistency levels are linearizability, eventual consistency, causal consistency, and Quorum. Table 7: Consistency vs Efficiency compares consistency levels with respect to efficiency.

Table 7. Consistency vs Efficiency.

Level	Consistency	Efficiency
Linearizability	Highest	Lowest
Eventual Consistency	Lowest	Highest
Causal Consistency	Higher than eventual Consistency but lower than Linearizability	Higher than Linearizability but lower than eventual consistency
Quorum	Configurable	Configurable

At Linearizability level of consistency, datastores objective is to display all database changes up to the moment of the current read request [7]. This entails ensuring that all alterations occurring in the database prior to the read operation are accurately reflected in the query results.

For example, suppose initially txn had x = 10.

Transaction [
update x to 13
update x to 17
read x --> Returns 17
update x to 1
read x --> Returns 1
]

To achieve linearizability, txn use a single-threaded single server, ensuring that every read and write request is ordered. For example, a read operation on 'x' would execute only after updating its value to 17. This approach is essential for systems requiring perfect consistency and high reliability.

With eventual consistency, a read request might initially return stale data, but it will eventually provide the latest data, as long as no new updates occur. During the period of returning stale data, the system is not fully consistent, but it will become consistent over time. This can be achieved by processing read and write requests in parallel using multiple servers or concurrently using multiple threads. For instance, if a write request is made before a read request, the read request might still be processed first.

Whereas causal consistency requires that if a previous operation is related to the current operation, the previous operation must be executed before the current one.

For example
Transaction [
i. update x = 20
ii. update y = 10
iii. read x
iv. update x = 2
v. read y]

The value of the read operation for x depends entirely on the prior update operation where x is set to 20. Therefore, it is crucial that the update operation for x is completed before the read operation. Conversely, the update operation for y does not affect the value of x. Thus, the order of execution regarding y in relation to the read operation for x does not matter. As a result, the first, third, and fourth operations will be processed on one server/thread, while the second and fifth operations will be handled on another server/thread. Causal consistency is superior to eventual consistency because operations related to the same key are processed in order, ensuring better consistency. It also offers advantages over linearizability because it doesn't require waiting for all previous operations to finish, thus improving availability. However, causal consistency faces difficulties

with aggregation operations [8]. In the quorum consistency model, data stores work with multiple replicas of the database that may not always be in sync. When performing a read query, data is fetched from all replicas, and the most appropriate values (such as the majority value or the latest updated value) are returned. This approach relies on some form of consensus within the distributed system and usually achieves eventual consistency. For example, consider three replicas where initially x = 20 in all. If the value is updated to x = 40 in the second replica and this replica then crashes, a read request would return x = 20 from the remaining two replicas, providing outdated data. However, this inconsistency is temporary because once the second replica is restored, the correct result is obtained. To enforce strong consistency, we can specify a minimum number of replicas from which data must be read. This can be achieved using the formula R + W > N, where:

- R represents the minimum number of replicas required for reading data.
- W denotes the number of replicas involved in writing data.
- N signifies the total number of replicas.

For example, with N = 5 and W = 2, R should exceed 3 (i.e., R ≥ 4). If data cannot be retrieved from at least 4 replicas, an error response is generated. This approach offers fault tolerance, and by adjusting the values of R, W, and N, we can establish either an eventually consistent system (R + W ≤ N) or a strongly consistent one (R + W > N) [9]. However, utilizing quorum has its drawbacks: It necessitates multiple replicas, resulting in higher costs. In cases where the number of replicas is even, it can lead to the split-brain problem hence to eliminate split-brain problem, number of nodes in cluster should be 2N + 1.

8. Research Result

This research employs a comparative analysis methodology to examine the features, advantages, and limitations of SQL and NoSQL databases. The study evaluates these database types based on their performance in various application scenarios, particularly focusing on read-heavy and write-heavy systems, scalability, and consistency requirements. Data collection involved an extensive literature review, including academic papers, industry reports, and case studies. Practical experiments were conducted to assess database performance, using benchmark tools to simulate different workloads. The collected data was analyzed to identify trends and draw conclusions about the optimal use cases for SQL and NoSQL databases. The findings are presented with detailed comparisons and recommendations, providing a comprehensive guide for selecting the appropriate database technology based on specific application needs.

Our research explores the nuances between ACID and BASE data stores, highlighting the trade-offs dictated by the CAP theorem. ACID-compliant databases such as MySQL, PostgreSQL, Oracle, SQLite, and Microsoft SQL Server ensure consistency and reliability, making them suitable for online transaction processing and analytical applications. These systems are pivotal for environments where data integrity and predictability are paramount. Conversely,

BASE-modeled NoSQL databases like MongoDB, Cassandra, Redis, Amazon DynamoDB, and Couchbase prioritize availability and scalability, making them advantageous for applications handling vast amounts of unstructured data, such as those in marketing and social network analysis. In the Java/Spring ecosystem, abstraction layers like Spring Data JPA and Spring Data JDBC facilitate the development of business applications by concealing database implementation details. This allows developers to focus on business logic rather than data access boilerplate code. These frameworks streamline the creation of robust, scalable applications by handling complexities like pagination, auditing, and query generation automatically. Transaction management in databases is a critical aspect examined in our study, with various isolation levels (Read Uncommitted, Read Committed, Repeatable Read, and Serializable) and consistency models (Linearizability, Eventual Consistency, Causal Consistency, and Quorum) being assessed. Our findings indicate that Read Committed is the most popular isolation level due to its balance between data consistency and efficiency. However, Snapshot Isolation is widely adopted to address anomalies in Read Committed transactions, ensuring a consistent view of data for each transaction.

We also delve into the effectiveness of different consistency models. Linearizability offers the highest consistency but at the cost of efficiency. Eventual Consistency provides the highest efficiency, though it may initially return stale data. Causal Consistency ensures operations related to the same key are processed in order, offering a compromise between consistency and availability [10]. Quorum Consistency balances consistency and efficiency by relying on consensus across multiple replicas. Ultimately, the choice between ACID and BASE databases, as well as the appropriate isolation and consistency levels, depends on the specific needs of the application, whether it prioritizes data integrity or scalability and availability.

9. Conclusion

We have explored data storage and retrieval Models such LSM Tree, MemTable. Replication, Sharding, Transaction Support, Consistency Model, Database abstraction by frameworks, tunable consistency through features like Quorumin the paper. It is the developer's responsibility to evaluate their use case and choose the correct data store to solve their specific problem by analyzing such as need of High Availability vs High Consistency, High Latency vs High Consistency, Tunable Consistency and Availability Model.

Future Scope

Exploring consistency and consensus in clusters of replicated sharded databases, to address the distributed nature of next-generation enterprise systems, presents significant opportunities for research. While beyond the scope of this paper, avenues worth investigating include master-slave replication and multi-master replication using PAXOS and RAFT algorithms. Additionally, protocols like the GOSSIP protocol in leaderless replicated clusters are noteworthy. This challenge

traces back to Lamport's 1985 research papers and remains relevant, warranting further exploration to identify the most suitable conflict resolution and fault-tolerant algorithms for replicated sharded clusters in distributed databases.

Acknowledgments

We extend our appreciation to all the developers we have collaborated with and their valuable contributions have played a crucial role in identifying data store solutions in the market and implementing diverse data store solutions across various enterprise applications over the years. Authors would like to specially thank to Apper, Saravanan, VP-Morgan Stanley for valuable comments and proofreading this article.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Athanassoulis, M., Kester, M.S., Maas, L.M., RaduStoica, *et al.* (2016) Designing Access Methods: The RUM Conjecture. *EDBT*, **2016**, 461-466.

[2] Kleppmann, M. (2017) Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Inc.

[3] Gierke, O. (2012) Spring Data JPA-Reference Documentation. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[4] Shivakumar, S.K. and Sethii, S. (2019) Introduction to Digital Experience Platforms. In: Building Digital Experience Platforms. Apress. https://doi.org/10.1007/978-1-4842-4303-9_1

[5] Pollack, M, Gierke, O., Risberg, T., *et al.* (2012) Spring Data: Modern Data Access for Enterprise Java. O'Reilly Media, Inc.

[6] Meier, A., and Kaufmann, M. (2019) SQL & NoSQL Databases. Springer Fachmedien Wiesbaden.

[7] Venkatraman, S., Kaspi, K.F.S. and Venkatraman, R. (2016) SQL versus Nosql Movement with Big Data Analytics. *International Journal of Information Technology and Computer Science*, **8**, 59-66. https://doi.org/10.5815/ijitcs.2016.12.07

[8] Parker, Z., Poe, S. and Vrbsky, S.V. (2013). Comparing NoSQL MongoDB to an SQL DB. *Proceedings of the* 51*st ACM Southeast Conference*, Savannah, 4-6 April 2013, 1-5. https://doi.org/10.1145/2498328.2500047

[9] Shivakumar, S.K., and Sethii, S. (2019) Building Digital Experience Platforms: A Guide to Developing Next-Generation Enterprise Applications. APress.

[10] Petrov, A. (2019) Database Internals: A Deep Dive into How Distributed Data Systems Work. O'Reilly Media.