Journal of Autonomous Intelligence (2024) Volume 7 Issue 3
doi: 10.32629/jai.v7i3.936

ORIGINAL RESEARCH ARTICLE

Concurrency versus consistency in NoSQL databases

Sonal Kanungo[1,*], Rustom D. Morena[2]

[1] *DPG School of Technology & Management, MDU Rohtak, Gurgaon 122001, India*

[2] *Department of Computer Science, Veer Narmad South Gujarat University, Surat (Gujarat) 395007, India*

*** Corresponding author:** Sonal Kanungo, sonalkanungo@gmail.com

ABSTRACT

With the advent of cloud services, the proliferation of data has reached unprecedented levels. The load distribution across multiple servers, driven by web and mobile applications, has become a defining characteristic of contemporary data management. In contrast to this surge in data complexity, traditional relational databases have proven inadequate in handling vast amounts of unstructured data due to their inherent focus on structured data models. Additionally, the concept of clustering, vital for efficient unstructured data management, eluded relational databases, rendering them ill-equipped for customized clustering techniques and the optimal execution of queries. SQL (Structured Query Language) databases earlier emerged as a groundbreaking solution, introducing the relational database model that organized data into structured tables. They employed ACID (atomicity, consistency, isolation, durability) properties to maintain data integrity and enabled intricate querying through SQL. However, as applications grew in complexity, SQL databases encountered hurdles in handling various data types, rapid data expansion, and concurrent workloads. The limitations of SQL databases propelled the rise of NoSQL (Not Only Structured Query Language) databases, which prioritized adaptability, scalability, and performance. NoSQL databases embraced diverse data models such as documents, key-values, column families, and graphs, enabling effective management of structured, semi-structured, and unstructured data. The transition to NoSQL databases was justified by several factors; horizontally scaled across nodes, handling extensive read-write operations effectively, Agile development of accommodating changing data structures without schema constraints, optimization for specific tasks, providing low-latency access and high throughput, dynamic schemas aligned with modern iterative development, promoting adaptability, and adeptly managed diverse data types, spanning text, geospatial, time-series, and multimedia data. These databases are purposefully designed to accommodate the escalating demands of data storage. Notably, this data emanates from diverse nodes and is susceptible to concurrent access by numerous users. However, a critical challenge surfaces as the data present on one node may diverge from its counterpart on another node replica. In this context, the simultaneous execution of database operations, while preserving the integrity of the data, emerges as a pivotal concern. Maintaining data consistency amid concurrent access hinges upon the synchronization of operations across all replica nodes. Achieving this synchronization necessitates the adoption of a robust concurrency control technique. Concurrency control acts as the linchpin for upholding accuracy and reliability within a system where operations unfold concurrently. Hence, the focal point of this investigation lies in examining the assorted concurrency control methodologies employed by NoSQL systems. The objective is to dissect the intricate interplay between concurrency and consistency, shedding light on the strategies these systems employ to strike an optimal balance between the two. In summation, as the landscape of data management witnesses an era of exponential growth catalyzed by cloud services, the dynamics of load distribution and unstructured data have necessitated a departure from traditional relational databases. NoSQL databases have risen to the fore, demonstrating the ability to grapple with these challenges. However, the quest for concurrent data access without compromising data consistency propels the exploration of various concurrency control methods. The aim of this study is to look at some of the different concurrency control approaches employed by NoSQL systems, highlighting how they priorities concurrency and consistency.

Keywords: multiversion; NoSQL; locking; optimistic; MongoDB; Cassandra; DynamoDB

ARTICLE INFO

Received: 7 July 2023
Accepted: 8 September 2023
Available online: 28 December 2023

1. Introduction

The volume of data increases significantly with the Internet of Things, the Web2.0, or next generation operational applications such as online shopping, gaming, etc. These applications and social-networking websites such as Facebook, Twitter and LinkedIn demand larger storage, support, and maintenance. These applications collect data in terabytes and petabytes from millions of its users. This data is accessed by large number of concurrent users.

Traditional relational databases have requirement of fixed schema; therefore, they are not able to handle cloud data. Relational database management system is losing its significance because of issues like fixed data model, scaling of semi-structured data storage, etc. NoSQL databases are coming into picture as solution to these problems as they are designed to handle ever increasing large data storage needs.

NoSQL are non-relational databases. A NoSQL database provides simple design with horizontal scalability. NoSQL databases offer different data structures than traditional databases, which gives more flexibility than relational database tables[1].

NoSQL provides high availability, geographic distribution, and horizontal scaling for the database. Many industries are now migrating towards NoSQL solutions. NoSQL database provides schema less structure, it allows data to grow dynamically and horizontally with data replication collections, sharding and clusters[2]

The NoSQL database can be divided into four categories column-oriented, key-value pair, document-orientated, and graph-based. Each type has its own set of characteristics and limitations.

Key-value stores data as pairs of attribute names (keys) and values. The data store in a schema-less way. In key-value relationship, the data can be any objects like such as integer, array, files, images, etc. Column oriented stores store data in columns. This will result in to fast execution because only the required columns are accessed in the database. The applications like aggregation are faster since data are stored in small blocks instead of tables.

Document stores store data in files. These files are data structures known as documents and implement specific pattern. Each file has automatically generated unique key.

In graph databases store data stored in graph structure in place of tables. Nodes and edges are used to store data. The complex joins and indexing can be implemented easily because there is no strict schema[3].

Distinct NoSQL database categories which emerged are:

- Document stores: MongoDB and Couchbase stored data in dynamic, JSON (JavaScript object notation)-like documents, facilitating schema evolution.

- Key-value stores: Redis and Amazon DynamoDB excelled in high-speed key-value storage, catering to real-time applications.
- Column-family stores: Apache Cassandra specialized in distributing and managing vast datasets across multiple nodes, ideal for scalability.
- Graph databases: Neo4j and Amazon Neptune facilitated intricate relationship analysis through graph-based data representation.

Users can select the database that best suits their product requirements. Column-based databases include Cassandra, H-Base, and Hypertable. Redis, Dynamo, and Riak Redis are key-value store database. CouchDB, Amazon SimpleDB, MongoDB, Riak, Lotus Notes, and MongoDB, are well-known document-based database. Infinite Graph, Neo4J, FlockDB, OrientDB are examples of graph-based databases.

We have identified three NoSQL databases and are discussing some of the properties in the following **Table 1**.

Table 1. Properties of NoSQL databases.

Characteristics	**MongoDB**	**Cassandra**	**DynamoDB**
Language	C, C++	Java	Java
Fault tolerance	Replication	Replication, partitioning	Replication, partitioning
Data model	JSON based document store. Up to 16MB document size.	Big table	Limited key-value store with JSON support Maximum 400KB record size
Protocol	TCP/IP	Custom API, thrift, reset	HTTPs
Data storage	Volatile memory, file, system.	HDFS	DynamoDB accelerator (DAX) tables
Replication mode	Master-slave replication	Maser-slave replication	Master-slave replication
Preferable for	Laptop, mainframe, hybrid cloud, managed cloud service, MongoDB atlas database as a service can be deployed on AWS (Amazon Web Services), Azure and GCP.	Real-time access, do bulk operation	Only available on AWS. No support for on-premises deployments locked-in to a single cloud provider.

Cloud based applications need complex data models. For these applications with frequent updates the consistency of database is of pivotal concern. This consistency will be achieved by maintaining interaction between the concurrent transactions and this control is attained through mechanisms known as concurrency-control techniques.

NoSQL databases are not able to deliver consistency with higher availability at same time. This was first defined by Brewer in CAP theorem[4]. The distributed system can recommend only two out of the three desired qualities that are C-Consistency, A-Availability, and P-Partition-tolerance[5]. The CAP theorem says, 'along with network partition, one can choose either consistency or availability'. The CAP theorem is providing two types of consistency: strong consistency and eventual consistency.

- Strong consistency

  In strong consistency, an application will read all the writes that were successfully accepted in the system.

- Eventual consistency

  In an eventual consistent database, there is no fixed order of the replicated messages. Therefore, the system can return temporarily inconsistent data.

NoSQL compromise consistency in support of speed, partition tolerance and availability. Most NoSQL databases give idea of eventual consistency where updates are circulated to all nodes. Here, the result the queries may not give latest updated data, the given data may not be accurate or it can be an old[2].

In this document we have studied various concurrency control techniques, their categorisations and comparison on performance, rollback, and deadlock handling. Further we have explored how different NoSQL databases handle concurrency control techniques to preserve consistency. In the literature review several documents were studied and their results are evaluated and utilized for this research. For further detailing, comparative study between MongoDB, Cassandra and DynamoDB was done and results were categorized under various heads. Finally in the conclusion, the crucial features of each of the NoSQL database are enumerated in order to support the end user to make an informed choice of choosing the write database.

2. Concurrency control techniques

There are various types of concurrency control techniques. These techniques can be categorized as pessimistic (lock based) or optimistic (without locking). In lock based protocols a lock is applied to fetch the data item. The approval is provided to access a data item if and only if when there is a lock applied on that data item. The lock will be applied in two possible modes: one, the exclusive mode and other, the shared mode[6]. A transaction can concede a lock on an item if the demanded lock has compatibility with locks previously hung on that item by different operations. The shared locks on an item can be applied by a number of transactions. At the point when the there is an exclusive lock on the data, A lock on that data can't be implemented by any other transaction. A lock is not allowed in this situation, and the requested transaction should wait until all incompetent locks held by other transactions are released before proceeding. The inappropriate acquiring and releasing locks will prompt inconsistency and deadlocks[7]. Therefore, some restrictions are imposed to avoid conflicts of different transactions to access same data and forced them to execute in serial order. This restriction can actualize by Two Phase Locking techniques[8]. Refer **Table 2** for details.

The Timestamp Ordering Protocol strategies ensures that timestamp order is given to each transaction when it starts, and conflicts will resolve by timestamp ordering. This protocol gives deadlock freedom since transactions do not wait for one another as algorithms utilizes timestamps for conflict resolutions[8]. The timestamp is unique for each transaction. Two types of time stamps are used such as read time stamp and write timestamp. The highest timestamp of any completed write transaction is write-timestamp, whereas the highest timestamp of the latest read transaction is read-timestamp. The conflict is tackled by the serializability of transaction order. The transaction whose timestamp is more than the read and write timestamp of data can access this data else the process will abort and start again.

In Optimistic Concurrency Control (validation) technique, the transaction executes with the anticipation that everything will go well during validation. The protocol is optimistic, hoping that conflicting transaction will not happen[9]. The validation-based protocols work with presumption that the read and write conflict of transaction will occur rarely. This protocol has given unrestrained access to the shared data objects. If a conflict arises during the validation phase, the transaction will abort and restart. This leads to re-do of all the work done up to validation stage. This is the major disadvantage of optimistic algorithm.

The Multiversion Concurrency Control provides flexibility and let read-only transaction to read a previous, but consistent version of the data[10]. Read operations read a previous version of the item[11]. Multiversion protocol do not overwrite previous versions values and these versions are always available for snapshot read[12]. This protocol will not reject processes that appear too late[13]. A read generally rejects as the data it is intending to read has now been changed by some other transaction[14]. This rejection of read can be prevented by preserving the older data item[15].

In same case we want to take group of data items as one individual unit, in such case transaction must lock each data item in the database. Locking of each data will be time consuming and tiresome process. It can be solved by issuing a single lock to the full database by implementing Multiple Granularity Locking[16]. In

Multiple Granularity technique, the locks will be obtained in top-down (root-to-leaf) approach and released as base up (leaf-to-root) order. Different types of locks are used in this technique. When the node is locked in an intention mode explicit locking will be implemented on trees' lower level. Intention locks will grant to all the ancestors of a root node before explicit locking of root node. For share mode the intention-shared mode implemented on node imposed the explicit locking at a lower level of the tree. In the same way when root node is locked in intention-exclusive mode exclusive-mode or shared-mode locks will apply at a lower level of the tree.

The two-phase commit[17] Cattell protocol works with distributed database. In a distributed system all sites must come to an agreement in performing a transaction. All nodes will commit the transaction or else abort happens because of this protocol. It conveys a signal to all nodes during first phase to commit and waits for their response to the agree message. The coordinator accepts an agreed message from all nodes in the second phase. Now coordinator writes a committed record in its log and all the nodes receives the commit message. However, if the transaction fails it sends an aborting message. Further, coordinator will wait indefinitely for all the acknowledgements. The coordinator sends an abort message if all the agreement messages are not received. When the nodes accept a commit message, all the locks will be released, and an acknowledgement will be sent to the coordinator. In case of failure the transaction will undo with abort message. The undo log releases the locks from resources and sends an acknowledgement.

Locking or pessimistic protocols are ideal for applications that needs updates very frequently, whereas optimistic protocols are suitable for read-only applications since read-only operations don't have any additional locking overheads. Locking techniques reduce performance because transactions that are incompatible with each other are blocked[14]. Deadlocks are resolved by restart of transactions which decrease the performance and may observe thrashing. The prevention and identification of deadlocks in locking mechanisms is substantially more complicated and expensive. Detecting a deadlock should be considered as part of maintenance overhead of the lock. The efficient deadlock-free locking protocols is the basic requirement of pessimistic protocol for databases that are always able to handle a large number of concurrent users.

Optimistic or validation protocols abort blocked transaction. They don't keep them for waits. The performance degradation occurs due to rollback of conflict transactions. Another major problem is starvation. In Optimistic protocol, commit of transaction is done only after validation phase as soon as conflicts occur. In this method, the systems abort more transactions than previous methods because it checks timestamps in last stage.

Table 2. Comparison of different concurrency control techniques[7].

	2PL	**Timestamp**	**Optimistic**	**Multiversion 2PL**
Performance	Locking protocols are good for update-intensive applications. The performance is degraded with standard locking because blocking is done if transactions are not compatible with each other.	Timestamps are used to decide the older-younger relationships. Timestamp can give better results if some available information about the transactions.	For read only optimistic protocols are good. This is because there are no unnecessary overheads of locking of read-only.	Multiversion follows the approach for maintaining a number of versions of a data item and allocates the right version to a read operation of a transaction.
Rollback	A large number of transitions will be roll back because of conflicts.	There is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.	They abort blocked transaction rather than sending them for waits.	In multiversion scheme a read operation is never rejected. Read can be given an old value of a data item, even though read is always possible.

Table 2. (*Continued*).

	2PL	Timestamp	Optimistic	Multiversion 2PL
Deadlock handling	The deadlocks are found in most locking protocols. Starvation is also possible if concurrency control manager is badly designed.	Timestamp protocol ensures freedom from deadlocks, as no transaction has to wait for other.	No deadlocks in optimistic approach.	To resolve deadlocks caused by certify-locks, the system should force one or more transactions to give up enough of their certify-locks to break the deadlock; these transactions can try later to get these locks back.

3. Concurrency control techniques and consistency maintained by various NoSQL database management system

NoSQL frameworks generally don't offer ACID (Atomicity, Consistency, Isolation, Durablity) properties, meaning atomicity, consistency, isolation, and durability, rather they give BASE (Basically Available, Soft state, Eventually Consistent) properties that implies basically available, soft state and eventually consistent[18]. The system can attain higher performance and scalability by giving up ACID limitations[5]. Most of the systems now become eventually consistent that means changes are eventually circulated to all nodes, but many of them provide some degree of consistency[19]. In this paper we will be exploring how different NoSQL databases handle concurrency control technique to preserve consistency.

3.1. MongoDB

MongoDB is a document-oriented NoSQL database system. It is a schema free, open-source and cross-platform database. MongoDB has its own rich ad-hoc query language. MongoDB allows different users to read and modify data concurrently. Pessimistic locking is used to protect the consistency[20]. This technique ensures atomicity of all writes to a single document. Therefore, the customers never observe the inconsistent information.

Multi-granularity locking allows locks on global database or collection level. The root nodes are locked using intent lock for achieving granularity. The specific database can concurrently apply lock both in Intended Share (IS) and Intent Exclusive (IX) mode. To apply write lock on collection, database and the global, the Intent exclusive (IX) lock mode will be applied.

For shared access of database or collection MongoDB uses reader-writer locks. These locks are in serialized order. MongoDB locks are write-greedy, that implies that the write has preference over read[21] In case when both read and write are trying for a lock, MongoDB grants the lock to the write. MongoDB allows different storage engines to execute a particular concurrency control technique. The storage engine MMAPv1 applies locks on whole collections and not individual documents while WiredTiger do optimistic concurrency controlling in the purview of collection level like at the documentation level. The conflicts will be resolved by abort and retrying one of the operations involved. By default, WiredTiger provides optimistic concurrency control and apply intent locking at the universal level, database and collection levels[22].

WiredTiger also utilizes Multiversion Concurrency Control (MVCC)[15]. The instance of beginning of any activity, WiredTiger provides at-that-moment snapshot of the data. This snapshot gives a consistent picture of the data captured in-memory. The WiredTiger composes all the data in a snapshot to disk over all data documents in a consistent manner. In the data records these data go about as a checkpoint[22].

The checkpoints are recovery points. The data records are consistent up to the last checkpoint. WiredTiger make checkpoints at intervals of 60 s. The checkpoint is legitimate until the compose of another checkpoint. While writing another checkpoint if MongoDB terminates or goes up against an error and going to restart, MongoDB can recoup from the last substantial checkpoint. The new checkpoint opens up and stable when

WiredTiger's metadata table atomically refreshed with new checkpoint. When the new checkpoint is available, WiredTiger liberates pages from the old checkpoints[22].

Aside from that, further concurrency may be obtained by sharding with allocating collections over multiple mongod instances[22]. Sharding allows shared servers (mongos processes) to execute a great number of processes simultaneously to the several downstream mongod instances. In some circumstances, reading and written operations in MongoDB can result in yield locks. When it comes to handling transactions, such as query handling, updating, and deleting, MongoDB can also yield locks between single documents. These write operations can also be altered in multiple documents[22].

The different types of consistencies are proposed by MongoDB. The consistency can be selected in MongoDB. The consistency levels are visible to users in MongoDB replica sets. MongoDB gives three types of consistency; majority, locally and linearizable consistency. The parameters or levels of any read or write operation are readConcern and writeConcern consistency. Default consistency is local that will return the current node's most-recent version. The operation is first committed locally in the commit transaction. Once it is written on the database and copied to the secondaries this will be circulated to enough nodes then the operation is said to be majority committed. It means that the changes are sustainable and permanent in the duplicate nodes. A numeric value or majority specifies writeConcern. A write process at w:1 by the user may get the confirmation on getting locally committed. A user will be acknowledging write operations with w:N when is receives at least N-nodes from the replica set. Until the write operation is majority committed, a write will not accept acknowledgement with w: majority.

At the execution of the query for a read process with readConcern at local level, the state of a replica set will be reflected by the data returned. Although the data returned with majority committed in the replica set gives the latest data of the specific node. It reads locally committed data. The reads through readConcern at majority level will return majority committed data[23].

The best consistency provided by the linearizable readConcern with w: majority write operations. Reads with readConcern level linearizable will provide the recent majority write that committed just before the beginning of read operation[24]. MongoDB also delivers available and snapshot read concern levels for causally consistent reads.

3.2. Cassandra

It is column based open-source NoSQL database. Here multiple data centres support large structured, semi-structured, and unstructured data. Cassandra provides scalability and high availability with high performance.

Facebook, a popular social networking website used by millions to link with friends and relatives, uses Cassandra for database management. Cassandra does not provide ACID properties of transactions. Cassandra is an AP system that means it delivers high availability and partition tolerance through eventual consistency.

The insert or update columns within a single row is a one writing operation. The transactions grouping into multiple rows is not supported by Cassandra. When there is successful write on one replica but there are failures on other replicas, Cassandra does not roll back any operation. Cassandra will inform this failure to the customer and can persist the write to a replica.

The latest update to a column is defined by timestamps[8]. This timestamp is provided by the client operation. When multiple users work simultaneously updating the single column in a row concurrently the final timestamp is taken. The most recent update is always eventually persistent. When Cassandra wants to perform an atomic update of rows within mem-tables (the inside memory structure where buffering of all writes is executed), for simultaneous reads and writes it implements Optimistic Concurrency Control[9]. In case

of higher conflicts, a solo partition uses per-tuple Pessimistic Concurrency Control is used with high abort rates. It provides both Optimistic Concurrency Control including Pessimistic Concurrency Control.

Cassandra provides both eventual and tunable consistencies with atomic, isolated, and durable transactions. This allows the user to select between strong or eventual consistency. Majorly, consistencies are of two types, immediate and eventual consistency.

Immediate consistency holds the identical data on all replica nodes at same time. Linearizable consistency is serializable isolation level for lightweight transactions[24]. Cassandra does not provide traditional locking for concurrent transactions. The Paxos protocol implemented with linearizable consistency that guarantees that there will be serializable transaction isolation level[25].

Read and write consistencies are controlled by eventual consistency. This allows different data on replica nodes, but latest version of the partition data will be provided for the query raised. Cassandra extends eventual consistency with tunable consistency.

For each operation, the consistency level can be tuned. Consistency can vary according to the client application. Cassandra either can be a CP or an AP system depending upon the application requirements. In tunable consistency, level can be set for each read and write request. Below are the different levels of consistency that can be set to achieve the data consistency in the database.

- All-writes/reads takes place in all replica nodes in the cluster have a commit log and a memory table.
- Any-writes/reads should write to minimum one node.
- One-writes/reads takes place in at least one replica' nodes commit log and memory table.
- Two-writes/reads occurs in at least two nodes' commit log and memory table.
- Three-writes/reads takes place in at least three nodes' commit log and memory table.
- LOCAL_ONE-writes/reads should be sent to and accepted by minimum one replica node in the local data centre.
- EACH_QUORUM-writes/reads a quorum of replica nodes in each data centre should write on commit log and memory table.
- Quorum-writes/reads should take place in commit log and memory table on a node's quorum in every data centres.
- LOCAL_QUORUM-writes/reads should take place in a quorum of replica nodes in the identical data centre as the coordinator node commit log and memory table.

3.3. DynamoDB

Amazon DynamoDB is a fast access key value document database. It is used as storage for large number of main services of Amazon's e-commerce platform. More than 20 million requests every second are supported by DynamoDB, and the system manages over 10 trillion requests per day.

DynamoDB has been found ensuring the desired levels of availability and performance. It is rugged enough to tackle failures of server, data centres and network partitions[26].

DynamoDB is provided by Amazon in various AWS regions throughout the globe. Here each of the AWS regions are autonomous and isolated from other regions. These regions constitute of several unique locations known as availability zones. These availability zones individually are unaffected from the failures of another zone(s)[26]. By default, the transaction is enabled in every single-region's DynamoDB tables and is disabled on global tables. We can enable transactions on global tables. The replication across regions is asynchronous and eventually consistent. Concurrent writes to the same item in different regions may not be serially isolated. Items do not lock during the transaction. If an element is updated outside of the transaction while it is still running, the transaction is terminated and an exception is issued.

This locking can be done by two different methods acquireLock or tryAcquireLock. The tryAcquireLock will return Optional.absent() if there is no lock granted and acquireLock will throw a LockNotGrantedException[27].

DynamoDB uses optimistic locking technique and prevents overwriting of the database. It also supports the object persistence and guarantees that before the item is updated or deleted, the item version is same in both application and on the server side.

Here each item is versioned using an attribute. The version of the item is recorded by the application when the transaction is retrieving that item from a table. The item can be updated unless the server's software version is same. When version mismatch is encountered, it means that some other user has altered the item and hence this effort is aborted as you are trying to update an older version of the item. In this case the application aborts and it will restart again.

In DynamoDB a user can choose consistency. Read can be eventually or strongly consistent. Eventually consistent reads option used to achieve maximize the read throughput. Consistency reached across every copy of data in seconds. The frequent read after a short time must get the updated data. The updated replicas send the update messages to all other replicas and eventually all the replicas will be consistent[28].

A strong consistency used to read and return an outcome that mirrors all the writes that was gotten as an effective reaction prior to the read. The latest updated data item is consistently restored through a strong consistent receive via the local secondary index. Within same global table, updates in any of the duplicate table is allowed to copy in to the various replicas. A global table contains the recently updated data and propagates it to all other replica tables instantly. In any global table, similar data is stored in each of the replica tables.

DynamoDB ensures that the information stored in a secondary index is eventually consistent, while local secondary indexes are strongly consistent partitions[26].

DynamoDB's problem is that it does not provide strong consistent reads within a region. A strong consistent reads and writes can be done within the same region. When write operation performed in a region and tried to read from different region, this read response can get old data and will not provide updated writes in the other region, therefore eventually consistency is given by DynamoDB.

In case where an application updates a similar data in various regions at the same time, inconsistency can emerge. DynamoDB solve this by giving last writer wins technique between simultaneous updates. Here every replica will approve on the last modification and attend a situation where each of them is having identical data[26].

The summary of the characteristics of each NoSQL database discussed in this paper is prepared while focusing on how they handle concurrency control techniques to preserve consistency and key characteristics are mentioned in the following **Table 3**.

Table 3. Summary of the characteristics on how they handle concurrency control techniques to preserve consistency.

Characteristic	**MongoDB**	**Cassandra**	**DynamoDB**
Database type	Document-oriented	Column-based	Key-value document
Consistency model	Mostly eventual consistency, with options for different levels of consistency.	Eventual and tunable consistency	Eventual and strong consistency within regions
Concurrency control	Pessimistic and optimistic locking, multi-granularity locking, reader-writer locks, MVCC (WiredTiger), sharding for increased concurrency.	Optimistic Concurrency Control (OCC), Pessimistic Concurrency Control, Paxos protocol for linearizable consistency, eventual consistency with tunable consistency levels	Optimistic locking, versioning of items, last writer wins technique.

Table 3. (*Continued*).

Characteristic	MongoDB	Cassandra	DynamoDB
Locking mechanism	Pessimistic locking, Intent locks, read-writer locks.	OCC, pessimistic locking for high-conflict situations.	Optimistic locking, versioning of items
Consistency levels	Majority, locally, linearizable, available, snapshot, tunable consistency levels.	Immediate consistency, linearizable consistency (Paxos protocol), eventual consistency with tunable consistency levels	Strong consistency within regions, eventual consistency across regions.
Transaction support	Yes, with different consistency levels.	Yes, eventual and tunable consistency.	Yes, eventual and strong consistency within regions.
Handling of concurrent writes	Write-greedy locks, conflicts resolved by abort and retry, Multiversion Concurrency Control (MVCC).	Optimistic locking, last writer wins, Paxos protocol for linearizable consistency.	Versioning of items, last writer wins technique.
Handling of concurrent reads	Serialized order, read-greedy locks.	Eventual consistency, strong consistency within regions.	Eventual consistency within regions.
Multi-region consistency	Limited, eventually consistent.	Eventual consistency, strong consistency within regions.	Strong consistency within regions, eventual consistency across regions.

4. Literature review

Grolinger et al.[5] have given NoSQL and NewSQL arrangements which provides a view in the field, offering help to user to pick the fitting data store, and distinguishing difficulties and openings in the field. Precisely, the most distinctive arrangements looked at concentrating on data models, analytics, scalability, and protection capabilities They also talked about the capacity to expand read and write requests, as well as segmentation, mirroring, stability, and transaction management. Lamport[29] has given the Paxos algorithm to implement a fault-tolerant distributed system that is basically a consensus algorithm—the "synod" algorithm. He solved the consensus problem with the Paxos algorithm.

Seth Gilbert et al.[4] described that delivery of atomic, consistent data is not possible while having partitions in the network. Any two from three characteristics that is consistency, availability, and partition tolerance can be obtained. It is not possible to offer consistent data and granting old data to be returned during failure of transaction.

According to Khan et al.[30] the SQL database can be chosen if it places a precedence on data standardization and consistency. NoSQL are preferred to a business with great amount of unstructured data and data availability is a high requirement. A relational database can be used in place of NoSQL database for the aggregation of small datasets.

Karamolegkos et al.[1] proposed EverAnalyzer, a self-adjustable big data management platform which is enable to collect metadata and recommends the optimum framework for the data processing or analytical jobs.

K.p. Eswaran et al.[6] provided a very basic database schema and talked about session, reliability, and locks. Authors have claimed that consistency is required between transactions, and if all transactions are well-formed and two-phase then any permissible schedule is consistent.

H.T. Kung, et al.[9] have proposed an upbeat protocol. Only after the verification process does the commitment gets executes. The method is hopeful, assuming that conflicts between transactions will be rare.

Papadimitriou et al.[10] proposed different methods of concurrency control using multiple versions. Enhanced multiversion concurrency control presented with efficient and necessary situations for an implementation to be l-SR concurrency control and extended concurrency control theory. A graphic structure was also given by them as "Multiversion Serialization Graphs" (MVSGs) and theory of three algorithms of Multiversion Concurrency Control. Here one algorithm considers the time stamps, the second takes up locking, and the third add the both locking and timestamps.

The foundation, basic features, and data models of NoSQL were covered by Han et al.[31] They also talked about NoSQL databases and how they relate to the CAP theory.

Abramova et al.[3] discussed NoSQL databases, different characteristics and their operations. They also compare and evaluate MongoDB and Cassandra and given the conclusion that if data magnitude increased, MongoDB decreases the performance and star performing poorly by giving absurd results, while Cassandra has shown better results.

5. Discussion

Comparative analysis of NoSQL databases: A closer look at consistency, availability, and atomicity.

In the realm of modern data management, NoSQL databases have revolutionized how organizations handle diverse data requirements. This review delves into the intricate world of various NoSQL databases, analyzing their strengths and limitations with a particular focus on the CAP theorem's tenets of consistency, availability, and partition tolerance[32].

CAP theorem and consistency vs. availability:

The CAP theorem suggests that a distributed system can only achieve two of the three fundamental properties: consistency, availability, and partition tolerance. When network partitioning occurs, systems have to choose between maintaining consistency and ensuring availability. It's important to acknowledge that striving for both consistency and availability simultaneously is often not feasible due to the inherent trade-offs within distributed systems. The synchronous replication usually ensures strong consistency while asynchronous replication leads to eventual consistency. NoSQL cannot provide consistency and high availability together. Distributed system is able to offer only two from the three needed features of the CAP theorem—consistency (C), availability (A), and partition tolerance (P)[4].

MongoDB's consistency and atomicity:

When network partition maintains consistency, they compromise on availability.

MongoDB, a leading document-oriented database, excels at handling semi-structured data. It provides strong consistency within a single document through atomic operations. Atomicity extends to subdocuments within documents, but operations spanning multiple documents or collections lack atomicity. MongoDB ensures complete isolation. MongoDB's default configuration ensures strong consistency on the primary server, while secondary nodes may exhibit eventual consistency. However, MongoDB's prohibition of reads from secondary servers due to inconsistent data can affect system performance during synchronous replication. The performance of the system is suffered during the synchronous replication of data store. Wherever throughput has been given more importance over durability, this consistency level is used. Majority consistency is chosen where latency is given lower importance than safety. While local consistency used for reading the latest data. MongoDB provides optimistic and multiversion concurrency control techniques. MongoDB's storage engine MMAPv1 locks whole collections instead of single documents while WiredTiger locks at the document level.

Apache Cassandra's consistency and BASE model:

In Apache Cassandra atomicity is implemented at the levels where rows are in prominent functions also taken as partition level. Insertion or updates of columns inside a row is considered as one write operation. Multiple row updates system is not supported by Cassandra. It also supports full row-level isolation. All replica writes are saved in both inside memory and within commit log offers durability to transaction. Apache Cassandra, a column-family database, introduces the BASE model, emphasizing basically available, soft-state, eventually consistent. By default, Cassandra gives eventual consistency but also permits tune consistency and availability. It follows a hybrid approach of Optimistic Concurrency Control (OCC) and Pessimistic Concurrency Control (PCC). Cassandra's architecture avoids a master node, focusing on equal nodes within a

cluster, and employs quorum reads/writes and lightweight transactions for managing consistency[33]. The quorum is the smallest number of copies required to reply to a read/write request. Cassandra always considers eventual consistency, but it can also give strong consistency by selecting (read quorum + write quorum) more replicas than the number of replicas available. Unlike MongoDB, Cassandra uses a masterless "ring" architecture[34]. This can give numerous advantages over master-slave architecture[35]. All nodes are equal in a cluster. A majority of nodes utilize to attain quorum. MongoDB suffers with issues such as memory hog because the scaling of databases.

DynamoDB's consistency and availability:

Amazon DynamoDB adopts the BASE approach and focuses on being basically available, soft-state, and eventually consistent. DynamoDB sacrifices atomicity for scalability, with units achieving atomicity but not across multiple items. It supports eventual and consistent reads, and while strongly consistent reads come with some trade-offs, elective parameters can enhance consistency. However, DynamoDB's limitations become evident in multi-region applications, where strongly consistent reads face network delays and partitions.

DynamoDB may apply a set of commands to assure that either all or none of the commands executed. DynamoDB does not deliver isolation. Still isolation levels can be implemented. Synchronous replicates data take place within an AWS region which offers a high uptime and durability. DynamoDB uses optimistic locking to protect database from being overwritten by the other writes. Both eventual and consistent reads are supported by DynamoDB. It is eventually consistent by default. Elective parameters can be used to create a request that is highly consistent. In the situation of eventually consistent reads, a request made just after a writing operation might not be able to obtain the most recent update. The strongly consistent reads request will get the latest data of successful write operation. DynamoDB is an available and partition-tolerant (AP) database which provides eventual consistency. In DynamoDB strongly consistent reads are not highly available with network delays and partitions. These failures mostly happen in multi-region/global application working on public clouds. This can be overcome by limiting strongly consistent reads only to a single region. Therefore, DynamoDB is unsuitable for utmost multi-region applications and it is not a reliable solution for single-region application too.

Comparative assessment:

NoSQL databases exhibit distinct strengths and limitations with respect to consistency, availability, and atomicity. MongoDB's strong consistency within documents and subdocuments makes it suitable for semi-structured data, while Apache Cassandra's BASE approach provides balanced availability and consistency at the cost of atomicity. DynamoDB's eventual consistency can be enhanced through elective parameters, but it struggles with strong consistency in multi-region deployments.

Final remarks:

The choice of a NoSQL database hinges on the specific requirements of the application. MongoDB shines in scenarios where strong consistency within documents is pivotal. Apache Cassandra excels when balancing availability and eventual consistency is paramount. DynamoDB, while offering scalability and elective consistency, might be less suitable for multi-region applications. As the landscape of computer science evolves, understanding the nuanced trade-offs between these databases empowers developers and researchers to make informed decisions aligned with the goals of their projects. The quest for the ideal NoSQL database lies in striking a harmonious balance between the key components of the CAP theorem—consistency, availability, and partition tolerance—in accordance with the unique demands of the application at hand.

Figure 1 depicts that MongoDB gives strong consistency and partitioning with poor availability while Cassandra and DynamoDB is giving better availability and partitioning with eventual consistency. **Figure 2** shows market shared by NoSQL databases. **Table 4** describes comparative study of CAP (consistency,

availability and partitioning) as well as ACID (atomicity, consistency, isolation and durability) properties of NoSQL databases.

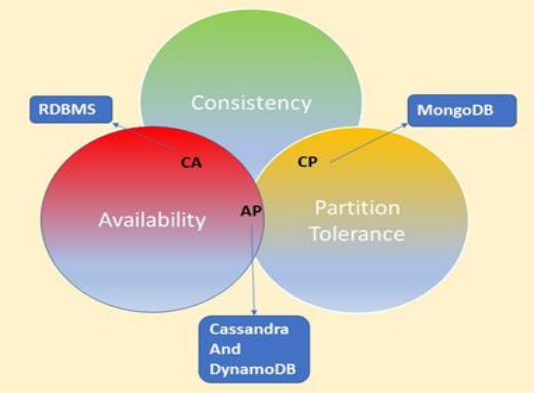
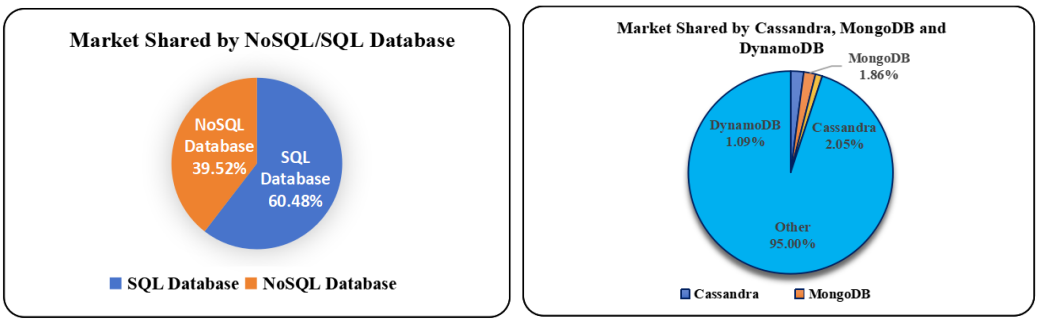


Figure 1. Eric A. Brewer's CAP theorem.



(a) **(b)**

Figure 2. Market shared by NoSQL/SQL database.

Figure 2a shows the market shared by SQL and NoSQL database and **Figure 2b** further gives information about total market shared by MongoDB, Cassandra and DynamoDB.

Table 4. Comparative study between MongoDB, Cassandra and DynamoDB.

NoSQL	MongoDB	Cassandra	DynamoDB
Type	Document store database	Column family databases.	Column family databases.
Availability	Sharding supports higher availability	Very high availability, emphasis on availability described by CAP theorem.	Very high availability.
Consistency	Tunable consistency. Write interest and partiality of read constraints can be defined.	Tunable consistency. Read and write consistency levels can be defined.	Eventually consistent asynchronous replication.
Partitioning	Sharding helps in partitioning range and it is hash based. The built-in feature is auto-sharding. Readers-writer locks are configurable. For strong consistency two methods can be set 1) linking to read only from primary 2) write concern limitation to "replica acknowledged".	Helps in partitioning (random partitioner, byte order partitioner). This will be configurable, depends upon quorum read request and quorum write requests.	Configurable.
Concurrency control	Readers-writer locks.	Client gives timestamps to define the latest update to a column and it is always accepted and it will be persistent.	Optimistic concurrency control used by increment the version numbers, and can also use pessimistic concurrency control.

Table 4. (*Continued*).

NoSQL	MongoDB	Cassandra	DynamoDB
Atomicity	Single document level availability.	Atomicity supported at row-level.	Atomicity on single item.
Isolation	Complete isolation.	Row-level isolation.	Isolation absents by default.
Durability	Durable but it can be further configured.	Stored in memory and commit log. Replication on nodes.	Durability option present.
Customers	Cisco, Adobe, Facebook, SAP, Google, UPS, eBay, BOSCH, PayPal, Forbes, etc.	Instagram, GoDaddy, Hulu, Intuit, Netflix, Reddit, The Weather Channel, eBay, GitHub, Constant Contact, CERN, Comcast, etc.	Netflix, Snapchat, The New York Times, HTC, Samsung, Amazon, Electronic Arts, AdRoll, Dropcam, Twitch, Clubhouse, Shazam, etc.

6. Research gap

Generally, all the NoSQL provides user to choose the consistency as per requirements from eventual or strong. The schema flexibility, data-types and the scalability of NoSQL benefits the applications. However, these NoSQL also provides lesser transaction support which prevent market to shift to these technologies.

- There should not be any compromise for consistency, user should always get committed data.
- Eventual consistent data can be from some write that can be abort later. Eventual consistency offers stale data.
- MongoDB provides strong concurrency control and high consistency; therefore, it may give less throughput. Locking protocols suffers from deadlocks. The need of new concurrency control technique is there for fast accessing of data.
- Cassandra's eventual consistency is not up to date, it is compromising with consistency of data. Consistency can improve with new concurrency control technique.
- In DynamoDB if a transaction gets aborted, it will restart and take a new timestamp. In cyclic restarting a transaction will repetitively start again and abort with no ending. The cascading rollbacks also degrade system's performance.
- There is always a risk of the redoing of work with optimistic concurrency control techniques. The performance degradation occurs due to rollback of the transaction when a conflict occurs. The main disadvantage with optimistic approach is starvation.
- There should a better concurrency control technique for AWS applications of Amazon database.

7. Conclusion

In conclusion, this comprehensive analysis sheds light on the intricate dynamics within the realm of NoSQL databases, specifically focusing on the pivotal choice between strong consistency models and eventual consistency models. This exploration underscores the paramount importance of tailoring database selection to the unique demands of specific applications, offering an invaluable guide for practitioners and researchers alike. The end user can be benefited from the given study that he can choose any NoSQL according to the need of his work.

The study delves into the realm of Optimistic Concurrency Control, unveiling its underlying premise that conflicts arising from concurrent operations are rare occurrences. By deferring conflict resolution until the conclusion of operations, this approach showcases its adaptability and efficacy in addressing potential contention.

In the context of practical application, Cassandra emerges as an indispensable asset for applications necessitating heightened availability and expeditious write operations. On the other end of the spectrum, MongoDB assumes a prominent role in scenarios mandating efficient document search, storage capabilities, and intricate aggregation functions. The juxtaposition of these database systems highlights MongoDB's unique

ability to offer both strong consistency and partition tolerance, while acknowledging the trade-offs inherent to Cassandra and dynamo—assuring availability without an analogous commitment to consistency.

The discourse surrounding eventual consistency underscores a common ground amongst these databases. While transient periods of incongruity might arise, the system diligently converges towards consistency in due course. This nuanced perspective augments our understanding of the pragmatic implications of these systems.

The multifaceted considerations elucidated in this research equip practitioners with a refined framework for evaluating NoSQL databases within the broader landscape of distributed systems. By acknowledging the intricate interplay between consistency models, concurrency control strategies, and the distinctive attributes of each database, stakeholders are empowered to make judicious decisions, aligning technological choices with the imperatives of their respective applications. This research thus offers a significant contribution to the scholarly discourse, fostering deeper insights and facilitating more informed decision-making in the ever-evolving domain of NoSQL databases and distributed computing.

Author contributions

Conceptualization, SK and RDM; methodology, SK; software, SK; validation, SK and RDM; formal analysis, RDM; investigation SK; resources, SK; data curation, SK; writing—original draft preparation, SK; writing—review and editing, SK and RDM; visualization, SK; supervision, RDM; project administration, SK; funding acquisition, SK. All authors have read and agreed to the published version of the manuscript.

Conflict of interest

The authors declare no conflict of interest.

References

1. Karamolegkos P, Mavrogiorgou A, Kiourtis A, Kyriazis D. EverAnalyzer: A self-adjustable big data management platform exploiting the Hadoop ecosystem. Information 2023; 14(2): 93. doi: 10.3390/info14020093
2. Poorvadevi R, Rajalakshmi S. Preventive signature model for secure cloud deployment through fuzzy data array computation. ICTACT Journal on Data Science and Machine Learning 2017; 7(2): 1402–1407. doi: 10.21917/ijsc.2017.0194
3. Abramova V, Bernardino J. NoSQL databases: MongoDB vs Cassandra. In: Proceedings of the Conference C3S2E; 10–12 July 2013; Porto, Portugal. pp. 14–22.
4. Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. ACM SIGACT News 2002; 33(2): 51–59. doi: 10.1145/564585.564601
5. Grolinger K, Higashino WA, Tiwari A, Capretz MA. Data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing: Advance System and Applications 2013; 2(1): 2–24. doi: 10.1186/2192-113x-2-22
6. Eswaran KP, Gray JN, Lorie RA, Traiger IL. The notions of consistency and predicate locks in a database system. Communications of the ACM 1976; 19(11): 624–633. doi: 10.1145/360363.360369
7. Kanungo S, Morena RD. Comparison of concurrency control and deadlock handing in different OODBMS. International Journal of Engineering Research and Technology 2016; V5(5): 492–498. doi: 10.17577/ijertv5is050615
8. Reed DP. Implementing atomic actions on decentralized data. ACM Transactions on Computer Systems 1983; 1(1): 3–23. doi: 10.1145/357353.357355
9. Kung HT, Robinson JT. On optimistic methods for concurrency control. ACM Transactions on Database Systems 1981; 6(2): 213–226. doi: 10.1145/319566.319567
10. Papadimitriou CH, Kanellakis PC. On concurrency control by multiple versions. ACM Transactions on Database Systems 1984; 9(1): 89–99. doi: 10.1145/348.318588
11. Bernstein PA, Goodman N. Multiversion concurrency control-theory and algorithms. ACM Transactions on Database Systems 1983; 8(4): 465–483. doi: 10.1145/319996.319998
12. Kanungo S, Morena RD. Analysis and comparison of concurrency control techniques. International Journal of Advanced Research in Computer and Communication Engineering 2015; 4(3): 245–251. doi: 10.17148/ijarcce.2015.4360
13. Kanungo S, Morena RD. Evaluation of multiversion concurrency control algorithms. International Journal of Research in Electronics and Computer Engineering 2018; 6(3): 807–813.

14. Kanungo S, Morena RD. Effective correctness criteria for serializability in multiversion concurrency control technique. International Journal of Innovative Technology and Exploring Engineering (IJITEE) 2019; 8(12): 1674–1653. doi: 10.35940/ijitee.l3162.1081219
15. Kanungo S, Morena RD. Issues with concurrency control techniques. International Journal of Electrical Electronics & Computer Science Engineering 2017; 1–6.
16. Lee SY, Liou RL. A multi-granularity locking model for concurrency control in object-oriented database systems. IEEE Transactions on Knowledge and Data Engineering 1996; 8(1): 144–156. doi: 10.1109/69.485643
17. Lamport L. Paxos made simple. ACM SIGACT News 2001; 32(4): 51–58.
18. Cattell R. Scalable SQL and NoSQL data stores. SIGMOD Record 2011; 39(4): 12–27. doi: 10.1145/1978915.1978919
19. Lotfy AE, Saleh AI, El-Ghareeb HA, Ali HA. A middle layer solution to support ACID properties for NoSQL databases. Journal of King Saud University—Computer and Information Sciences 2016; 28(1): 133–145. doi: 10.1016/j.jksuci.2015.05.003
20. Kudo T, Ishino M, Saotome K, Kataoka N. A proposal of transaction processing method for MongoDB. Procedia Computer Science 2016; 96: 801–810. doi: 10.1016/j.procs.2016.08.251
21. Schultz W, Avitabile T, Cabral A. Tunable consistency in MongoDB. Proceedings of the VLDB Endowment 2019; 12(12): 2072–2082. doi: 10.14778/3352063.3352125
22. Concurrency. Available online: https://www.mongodb.com/docs/manual/faq/concurrency/ (accessed on 24 August 2023).
23. Seguin K. The little MongoDB Book. Openmymind.net; 2011.
24. Brewer EA. Towards robust distributed systems. In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing; 16–19 July 2000; Portland, Oregon, USA. pp. 1–7.
25. About Transactions and Concurrency control (2020) from Datastax. Available online: http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_ltwt_transaction_c.html (accessed on 24 August 2023).
26. Amazon DynamoDB, Developer Guide (2012). API Version 2012-08-10. Available online: https://s3.cn-north-1.amazonaws.com.cn/aws-dam-prod/china/pdf/dynamodb-dg.pdf (accessed on 24 August 2023).
27. Amazon DynamoDB transactions: How it works. Available online: https://docs.aws.Amazon.com/amazondynamodb/latest/developerguide/transaction-apis.html (accessed on 24 August 2023).
28. Building distributed locks with the DynamoDB lock client. Available online: https://aws.amazon.com/blogs/database/building-distributed-locks-with-the-dynamodb-lock-client/ (accessed on 24 August 2023).
29. Gray J, Lamport L. Consensus on transaction commit. ACM Transactions on Database Systems 2004; 31(1): 133–160. doi: 10.1145/1132863.1132867
30. Khan W, Kumar T, Zhang C, Raj K, et al. SQL and NoSQL database software architecture performance analysis and assessments—A systematic literature review. Big Data and Cognitive Computing 2023; 7(2): 97. doi: 10.3390/bdcc7020097
31. Han J, Haihong E, Le G, Du J. Survey on NoSQL database. In: Proceedings of the 6th International Conference on Pervasive Computing and Applications; 26–28 October 2011; Port Elizabeth. pp. 363–366.
32. Chen JK, Lee WZ. An Introduction of NoSQL databases based on their categories and application industries. Algorithms 2019; 12(5): 106. doi: 10.3390/a12050106
33. Apache Cassandra documentation v4.0-beta3. Available online: https://cassandra.apache.org/doc/ (accessed on 24 August 2023).
34. How are consistent read and write operations handled? Available online: https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/dml/dmlAboutDataConsistency.html (accessed on 24 August 2023).
35. Tunable consistency from Datastax. Available online: How Cassandra Balances Consistency & Performance | DataStax (accessed on 24 August 2023).