PeerJ Computer Science

Expeditious data recovery in Not Only SQL (NoSQL) databases using a log-based approach

Gayatri Shitanshu Kapadia[1] and Rustom Darash Morena[2]

[1] M.C.A. Department, Sarvajanik College of Engineering and Technology, Surat, Gujarat, India
[2] Department of Computer Science, Veer Narmad South Gujarat University, Surat, Gujarat, India

ABSTRACT

Not Only SQL (NoSQL) databases lack standardized backup and recovery procedures, making fault tolerance and rapid restoration more challenging than in traditional relational systems. Achieving efficient recovery in distributed NoSQL environments requires mechanisms that can manage operational failures while preserving availability, consistency, and performance. This study investigates existing recovery strategies for NoSQL databases and introduces a novel log-based expeditious recovery mechanism tailored for distributed architectures. Our primary contribution is a transaction evaluator and cleaner system that performs real-time inspection of operational logs, identifies failed or inconsistent transactions, and prevents their propagation from the primary node to secondary replica nodes. Unlike conventional log-shipping and replication approaches, the proposed method integrates selective log forwarding with automated transaction sanitization, thereby reducing replica divergence, data loss, and overall recovery time. We implemented the system using Node.js and MongoDB Atlas, incorporating modules for log management, collection operations, and a lightweight journaling process for secondary nodes. Experimental evaluation shows that our approach improves recovery speed and reliability compared to standard replication-based techniques. These findings demonstrate a practical and robust solution for accelerating recovery in NoSQL databases, contributing meaningful advancement to failure-recovery research in distributed data systems.

Submitted 21 April 2025
Accepted 1 December 2025
Published 17 February 2026

Corresponding author
Gayatri Shitanshu Kapadia,
gayatriskapadia@gmail.com

Academic editor
Siddhartha Bhattacharyya

Additional Information and Declarations can be found on page 15

DOI **10.7717/peerj-cs.3507**

Subjects Algorithms and Analysis of Algorithms, Computer Education, Databases, Blockchain
Keywords Journal, NoSQL databases, Recovery, Oplog, Replication, Restoration, MongoDB, Transaction, Journal, MongoDB

INTRODUCTION

Not Only SQL (NoSQL) databases lack the standardized backup and recovery procedures found in relational databases, resulting in significant challenges. Accurately reflecting data at a specific point in time is problematic, especially with large data volumes that require extensive resources for backups. Recovery processes can be further hindered by potential downtime and performance issues, as NoSQL databases prioritize availability and partition tolerance over strict consistency, complicating the achievement of a consistent state across all nodes. Additionally, the risk of stale data necessitates conflict resolution.

To overcome potential downtime and performance issues, a proposed methodology involves a transaction evaluator and cleaner system built in Node.js and integrated with MongoDB Atlas. Out of various NoSQL databases, MongoDB is a prominent NoSQL

database with 30% to 40% market share and a compound annual growth rate (CAGR) of 23.9% (Ref. *Market Share, 2024*). Hence, we selected MongoDB amongst the various NoSQL databases to implement our proposed system, which in turn can be applied to other NoSQL databases. This system uses MongoDB data structures like Journal File, Operation Log (Oplog), and Checkpoint, which play crucial roles in transaction evaluation. An oplog is a log that records all operations applied to a database. A journal is a write-ahead log (WAL) used for crash recovery. A checkpoint is a point where the database flushes recent changes from memory (cache) to disk. The Oplog, journal, and checkpoint are discussed later in this section.

The proposed system includes scripts for critical tasks such as connecting to databases, managing collections, and handling operational logs while replicating the database operation(s) of successful transactions from the primary node to secondary node(s). We have written two algorithms, one of which is a transaction evaluator and cleaner that bifurcates successful and failed transactions and saves successful transactions in a newly created oplog. Another algorithm is to minutely work on identifying the successful transactions, which is a subpart of the first algorithm. By providing newly created operation logs and incorporating journaling during various database operations at replica set levels, the proposed method seeks to ensure reliable and efficient restoration of the database in less time than the normal restore process. The proposed system achieves minimum downtime in the recovery process with no data loss.

MongoDB, which operates in a distributed environment, has the following key components, which are part of our proposed system.

Primary node

In a MongoDB replica set, the primary node acts as the reliable source of data for the set. It is the sole node in the replica set authorized to handle write operations. It's the sole node that can handle insert, update, and delete commands. The primary holds an operation log (oplog), a unique capped collection that keeps all changes to data.

Oplog file

The oplog is a special capped collection that is used to percolate all changes to the data from the primary node to secondary nodes in the MongoDB replica set. Each member of the replica set applies the operations from the oplog to maintain consistency across the set. The oplog is crucial for monitoring and recovering transactions in a distributed environment. It ensures that transactions are consistently applied across all nodes in the replica set. In the event of a failure, the oplog can help update a secondary node to match the primary, guaranteeing accurate replication of the transaction. Figure 1 presents the general format of the oplog file, highlighting key fields such as:

- ts: timestamp indicating when the operation occurred
- h: history or hash identifier
- v: oplog version
- op: type of operation (insert, update, delete, command, *etc.*)

ts:	is a Timestamp to identify operations in the oplog uniquely.
h:	is History or Hash.
v:	is a version for oplog format which is maintained for backward compatibility of MongoDB Versions.
op:	is an operation type that has following values. **"i":** insert operation **"u":** update operation **"d":** delete operation **"c":** command operation (creating a collection or index) **"n":** No-op (for heartbeat or other internal purpose)
ns:	is a namespace that indicates database and collection on which the operation is performed. The format is <databasename>.<collectionname>
o: **o2:** **fromMigrate:**	is an object which contains details of an operation. is the query document that matches the document to be updated. it is an optional field that indicates whether the chunk migration is in a sharded cluster.
wall:	it is a wall clock time which is human-readable.
ui:	is a unique index in operations such as creating or dropping collections in sharded clusters.

Figure 1 **Oplog file format.** Full-size DOI: 10.7717/peerj-cs.3507/fig-1

- ns: the namespace indicating the target database and collection
- o, o2: detailed operation objects.

Figure 2 shows sample oplog entries for insert and update operations. The examples illustrate how MongoDB records each action, allowing secondary nodes to reproduce the exact sequence of operations. These visual samples make it easier to understand how MongoDB internally represents state changes and how recovery processes interpret them.

Secondary node

In a MongoDB replica set, a secondary node copies data from the primary node by applying operations from the primary's oplog asynchronously. Secondary nodes replicate the oplog and execute the operations on their datasets, maintaining data consistency throughout the replica set.

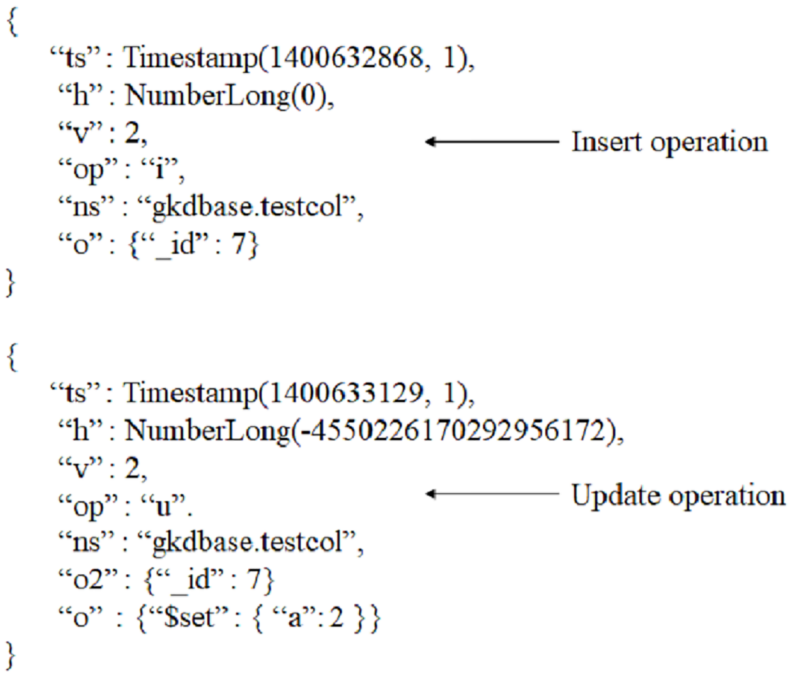

Figure 2 **Oplog example.** Full-size DOI: 10.7717/peerj-cs.3507/fig-2

Journal file

A journal file serves as a write-ahead log that MongoDB employs to keep a record of actions that alter the data. It functions as a mediator between operations in memory and permanent storage (data files on the disk). Writes to the journal file occur in memory and are periodically flushed to disk. Figure 3 illustrates the structure of the journal file, including components such as: the Header for the file identifier, MongoDB version, and checksum. All of these are useful during data integrity verification. Journal file format also contains journal sections for write operations' timestamp, database & collection name, operation type, and file offset. So, MongoDB uses this file to bring the database to a stable state by reapplying all uncommitted or incomplete transactions. The expanded visual representation in Fig. 3 clarifies how journal entries are grouped and replayed.

Data files

These files store the actual data of MongoDB collections. Each collection has a file with a ".wt" extension, which contains the BSON data for documents. During transactions, the data in these files is modified.

Checkpoint

A checkpoint is a mechanism that periodically flushes the in-memory changes made by write operations to disk. This guarantees that all operations up to a certain point are safely written and stored in the data files, providing a durability guarantee for the system.

The article further outlines a literature study, methodology, experiments, results, and conclusive insights.

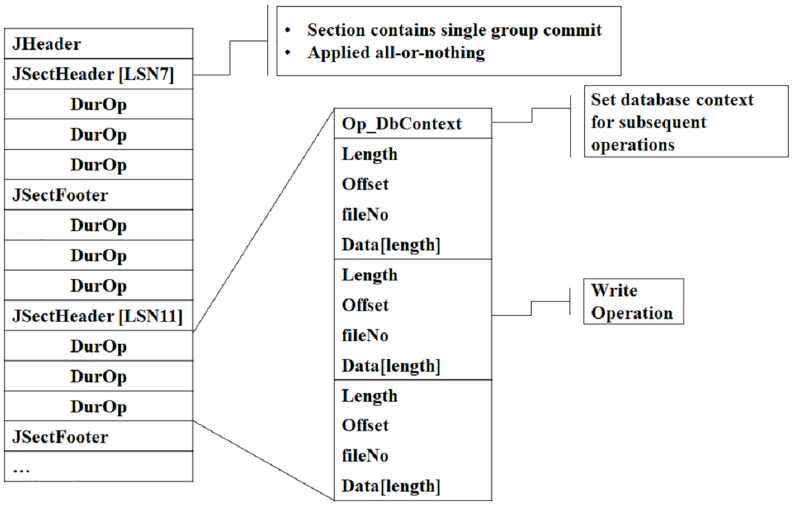

Figure 3 **Journal file format.** Full-size DOI: 10.7717/peerj-cs.3507/fig-3

LITERATURE STUDY

This section reviews the previous research to compare recovery mechanisms in NoSQL databases, fostering ideas for novel solutions that improve recovery time.

Several researchers have explored data recovery and resilience strategies in NoSQL systems, contributing valuable insights while also highlighting certain limitations in applicability and scalability. *Tsai, Hsiao & Lai (1959)* investigated multiple data recovery techniques aimed at enhancing resilience and minimizing data loss within NoSQL environments. While their work presents a diverse range of solutions, it lacks validation through real-world testing, which raises concerns about the generalizability of their approaches across heterogeneous NoSQL systems. *Sauer, Graefe & Härder (2019)* through the FineLine framework, proposed mechanisms to simplify transactional storage and recovery in the context of large datasets and complex structural changes. However, their solution appears to be highly specialized, potentially limiting its applicability beyond specific NoSQL architectures. *Luo (2012)* addressed interference issues during checkpointing in distributed systems, proposing enhancements to system consistency. Although conceptually robust, the implementation complexity in large-scale systems and insufficient discussion on integration with existing infrastructures present practical limitations. *Paithankar & Roesle (2023)* focused on pragmatic strategies for data resilience, advocating for recovery process testing and leveraging multi-cloud region deployments. Nonetheless, their strategies are primarily tailored to MongoDB Atlas, restricting broader applicability to other NoSQL platforms. *Matos & Correia (2016)* introduced an "undo" operation to bolster data integrity and consistency in MongoDB. Despite its innovation, the approach remains narrowly scoped to MongoDB with limited discussion on scalability or adaptability to other NoSQL databases. *da Silva, Holanda & Araujo (2016)* explored challenges in replication, emphasizing consistency and storage efficiency within distributed systems. However, their work lacks concrete real-world case studies and does not deeply explore the trade-offs between consistency and availability.

Wang & Tang (2012) presented foundational concepts for understanding scalable NoSQL frameworks, focusing particularly on Cassandra. While informative, the work is more introductory, offering limited technical depth concerning Cassandra's internal mechanisms.

Comparative studies by *Haughian, Osman & Knottenbelt (2016)* and *Gu et al. (2015)* analyzed replication strategies in Cassandra and MongoDB, aiding database selection decisions. Nonetheless, their evaluations are primarily qualitative and do not provide quantitative metrics to assess system performance under varied workloads. *Sung et al. (2019)* proposed durability-enhancing techniques such as snapshots and transaction logs. These approaches, though theoretically sound, may face challenges in dynamic, high-transaction environments, particularly with respect to maintaining system performance.

Tam Vo et al. (2012) through the LogBase system, addressed the demands of write-intensive workloads in cloud-based applications like financial services. While effective in such contexts, the solution may face limitations when generalized to broader application domains. *Ouyang et al. (2016)* proposed a verification framework for ensuring transactional consistency in MongoDB. While useful, its MongoDB-specific design limits applicability to other NoSQL systems. *Mansouri & Pathan (2020*, *2021)* introduced a communication-induced checkpointing algorithm that reduces the overhead associated with traditional checkpointing methods. However, the algorithm's implementation is complex and may introduce new types of operational overhead. *Ammann et al. (2002)* contributed techniques to improve recovery from malicious transactions, enhancing system security. However, the narrow focus on security-driven failures may overlook other significant failure types, such as hardware faults or network partitions. *Bao et al. (2016)* conducted an empirical analysis of persistence mechanisms using snapshots and logs. Their findings are valuable in understanding failure scenarios, although edge cases and complex failure modes remain underexplored. Finally, *Wang, Li & Lin (2007)* proposed a disaster recovery mechanism tailored for volume replication systems. While the mechanism shows promise, it lacks comprehensive evaluation across diverse failure conditions and does not fully address performance trade-offs inherent in such systems.

While studying various research articles, we have observed that there are issues relevant to data consistency across the replica sets, managing the data structures with larger datasets, transaction robustness, and recovery speed during the entire recovery process. Not much work has been done in the context of faster recovery in distributed databases. Mission-critical applications deployed on distributed databases requiring 24 × 7 uptime cannot afford even a fraction of downtime, which can lead to financial loss or reputation damage. Thus, faster recovery is vital to ensure smooth transaction processing, especially in large-scale, real-time systems.

PROPOSED SYSTEM

The proposed system has been implemented in MongoDB, a NoSQL database with a large market share across NoSQL databases. In MongoDB, when a write operation

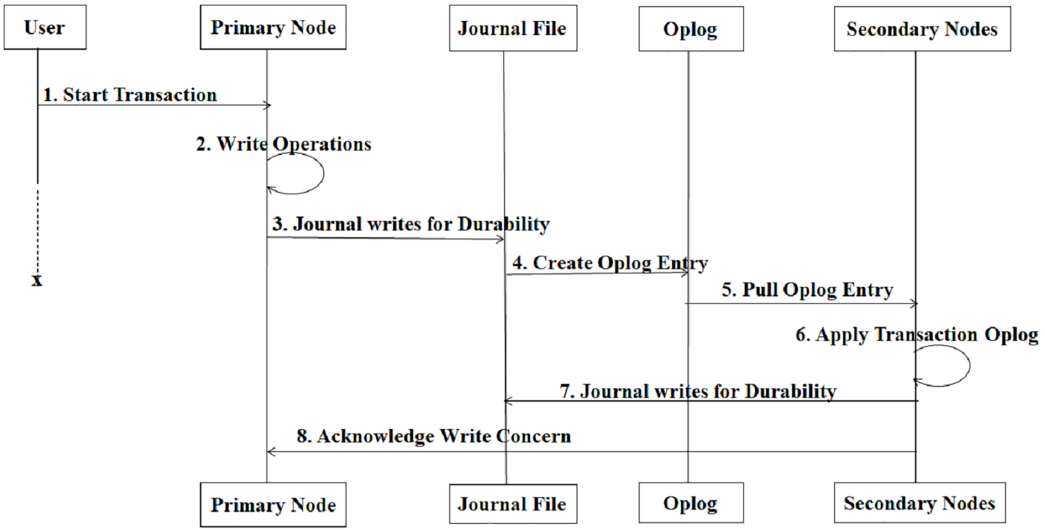

Figure 4 Sequence diagram when write operation occurs.
Full-size DOI: 10.7717/peerj-cs.3507/fig-4

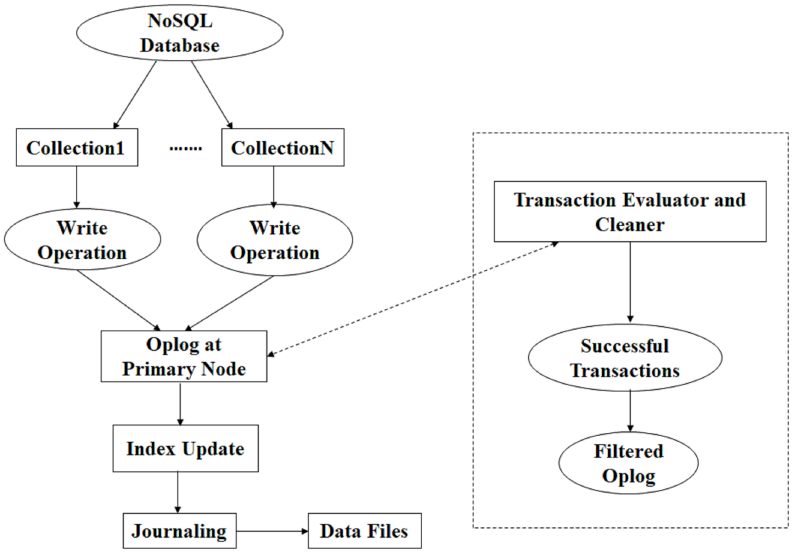

Figure 5 Proposed system architecture.
Full-size DOI: 10.7717/peerj-cs.3507/fig-5

(insert, update, or delete) occurs, the write operation is applied to an in-memory buffer (cache) first for performance reasons. Before the operation is permanently written to the data files, it is written to the journal file to ensure durability and recovery in case of failure. The data is asynchronously flushed from memory to the main data files managed by the storage engine (like WiredTiger). For replica sets, the operation is logged in the oplog (Operation Log) to replicate the changes to secondary nodes. The entire sequence is shown in below Fig. 4, which illustrates the complete flow of this process—from the initial write request to journaling, oplog creation, secondary replication, and final acknowledgment.

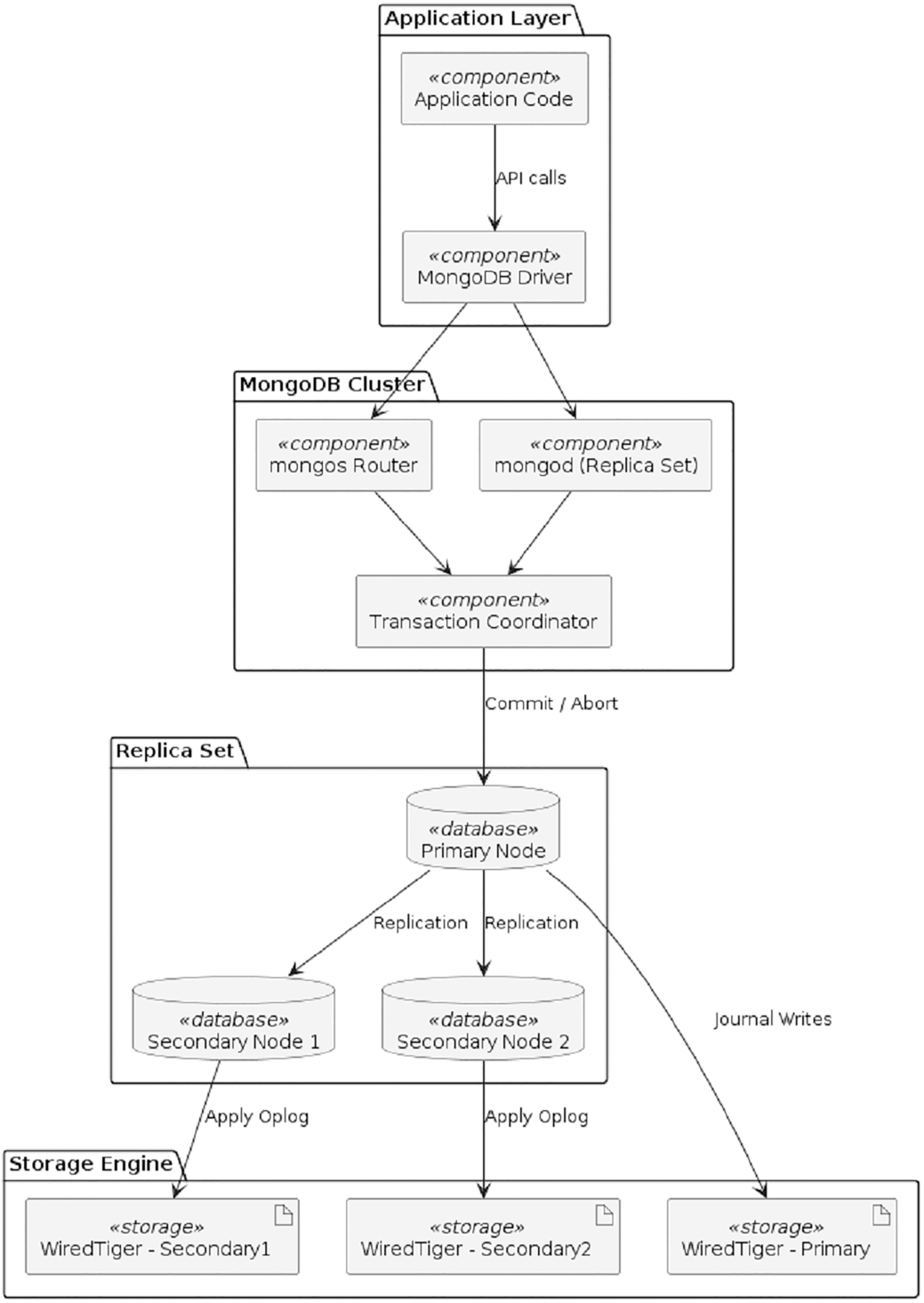

Figure 6 **Component diagram of the proposed system.** Full-size DOI: 10.7717/peerj-cs.3507/fig-6

In our proposed system, the transactions are monitored, and successful transactions are stored in a collection that is used for recovery at the time of failure. In the transaction evaluator and cleaner algorithm, the evaluator, as shown in Figs. 5 and 6.

Figure 5 presents the proposed system architecture, illustrating how write operations generated from multiple collections are processed through the primary node, recorded in the oplog, and then assessed by the transaction evaluator. The evaluator continuously scans the oplog entries to distinguish successful transactions from those that have been aborted due to concurrency conflicts, system interruptions, or incomplete operations. Only verified successful transactions are forwarded to the cleaner module, which removes redundant or invalid entries and prepares a filtered oplog. This filtered log forms a reliable source of truth for high-speed recovery because it eliminates unnecessary or faulty entries that could otherwise slow down reconstruction procedures.

Figure 6 provides the component diagram of the proposed system, offering a physical-layer view of how the modules interact within the database environment. The diagram highlights how the evaluator and cleaner components integrate with the primary node, journaling mechanism, and oplog subsystem. It also shows how the filtered oplog is persisted in a dedicated collection that can be immediately accessed during system failure. This design simplifies recovery by bypassing the need to process the full oplog, which may contain large volumes of aborted or irrelevant entries.

Together, Figs. 5 and 6 provide a detailed functional and structural representation of the proposed mechanism, clarifying how the system improves the speed and accuracy of data recovery in NoSQL databases.

Transaction evaluator and cleaner

To create filtered_oplog, which will be used to accelerate data recovery, an algorithm "to create filtered_oplog" is written to identify the successful transactions and store them in the filtered_oplog collection. The transaction evaluator and cleaner algorithm evaluates the transactions and cleans the dirty operations out of the oplog. This algorithm uses a linked list data structure to manage all the sets like transactions, sessions, database states, operations, and errors.

Transaction evaluator and cleaner algorithm, which is implemented at step 5 of algorithm to create filtered_oplog is as follows:

Algorithm: **To create filtered_oplog.**

Definitions:
Let T represent the set of all transactions.
Let S represent the set of all sessions.
Let L represent the set of all logs.
Let D represent the set of all database states.
Let E represent the set of all errors.
Let O represent the set of operations, where
O = {o1, o2,…,on}.
Let R(O) represent the set of reverse operations corresponding to O.
Step 1: Initialize and start a connection to our
MongoDB instance
$C \subseteq S \Longrightarrow$ C represents the subset of sessions
with active connections to the database.
Step 2: Start a MongoDB session for handling

(Continued)

Algorithm: (continued)

transactions
∃ s ∈ S, t ∈ T : t is associated with session s.
Step 3: Implement the transaction logic, including
logging inverse operations
∀t ∈ T, ∃ Lt ⊆ L: Log inverse operations ⟹
Lt = {start_log, o1, o2, … , commit_log}.
Perform Database Operations:
Dt ⊆ D → Dt′ = Dt ∪ O
where Dt′ is the updated database state.
Commit or Abort:
If E = ∅, Commit t; Otherwise, Abort t.
Step 4: [Record oplog to filtered_oplog]
∀ t ∈ T, ∃ Lt ⊆ L : Lt = {oplog, journal_log,
monitoring_log}.
If *t* succeeds:
Lt ∋ success_log.
If *t* aborts:
Lt ∋ error_log.
Step 5: [Filter functionality]
∀ t ∈ T, if t is aborted, read Lt for inverse ope-rations R(O).
Find oplog entries:
∃ f ⊆ Lt : f is the filtered_oplog for t.
Commit or Abort Modify:
If E = ∅, Commit R (O);
Otherwise, Abort R (O).
Step 6: [Perform undo for different operations]
∀ o ∈ O, ∃ r ∈ R (O) : r is the inverse of o where o∈ {insert, update, delete}.
Step 7: [Ensure proper testing]
∀ t ∈ T, R(O) → Dt″
where Dt″ = Dt
(validate data integrity after reversal).
Step 8: Exit

Algorithm Transaction evaluator and cleaner.

Input: O (Oplog)
Output: Successful transactions list cList (for recovery)
We are working within a universal set of oplog entries U, transactions T, and specific operation types {read, write, abort, commit}.
Step: 1 Define Sets and Initial States
O ⊆ U: The set of all entries in the oplog.
M ⊆ T: The set of transactions currently being processed.
cList, uList, wList, twList, tuList: Subsets of U, initially empty.
cList: Stores committed operations. (commit transaction list)
tuList: Stores oplog entries temporarily for read
operations. (temporary read list)
twList: Stores oplog entries temporarily for write operations. (temporary write operation list)
uList: Stores processed read operations after a commit. (read list)
wList: Stores oplog entries moved for certain write operations. (write operation list)
Step: 2 [Process Each Oplog Entry]
Iterate through o ∈ O. For each o:
Let op (o) ∈ {read, write, abort, commit} denote the operation type of o, and txn (o) ∈ T denote the transaction Ti associated with o.
Step: 3 [Check the type of operation op (o):]
If op (o) = write:
If txn (o) ∈ M:

Algorithm (continued)

```
                wList ∪ {o} → wList
              Otherwise:
                twList ∪ {o} → twList
                If op (o) = read:
                If o ∉ tuList:
                  tuList ∪ {o} → tuList
              If op (o) = abort:
                tuList\tuList → tuList
                (clear tuList)
              If op (o) = commit:
                wList ∪ twList → wList
                (move entries from twList to wList)
                twList\twList → twList (clear twList)
                uList ∪ tuList → uList
                (move entries from tuList to uList)
                tuList\tuList → tuList (clear tuList)
                cList ∪ {o} → cList (add o to cList)
              Otherwise:
                twList\twList → twList (clear twList)
    Step: 4 [Proceed to the subsequent o ∈ O]
    Step: 5 [Once all entries in O are processed]
    wList\wList → wList (clear wList)
    uList\uList → twList (clear uList)
    Step: 6 [Return committed list to store in filtered_oplog]
            return cList
            Exit
```

EXPERIMENTS AND RESULTS

While generating experimental results for transaction evaluator and cleaner in MongoDB, we have simulated transactions, monitored their status, and evaluated the performance and reliability of our data recovery mechanism. The structured approach to achieve this is described as follows:

Environment setup

MongoDB cluster with primary and secondary nodes are used for simulating recovery operations. We have worked on MongoDB Atlas (Cloud database) in conjunction with MongoDB Compass, which has also helped us to perform various data analysis, querying, and management tasks through graphical environment.

We employed MongoDB Compass, integrated with MongoDB Atlas, to simulate and analyze database recovery operations. A MongoDB Atlas cluster was provisioned by creating an Atlas account and configuring the cluster with appropriate instance specifications and regional parameters. To ensure secure connectivity, the client machine's IP address was added to the IP whitelist under the Network Access settings. For database authentication, a user was configured with administrative privileges, enabling read and write access to the cluster.

Connectivity between the application layer and the Atlas cluster was established using the official MongoDB Node.js driver. For local interaction and verification, MongoDB

Table 1 Comparison of recovery time.

Parameters	Data recovery using our methodology	Data recovery using MongoDB restore
Time taken for recovery of data	4.040100 (s.msμs)*	7.020070 (s.msμs)*

Note:
* s stands for seconds, ms stands for milliseconds (three digits), μs microseconds (three digits).

Compass was employed, using the provided connection string (URI) to interface with the remote Atlas cluster.

Recovery operations were simulated by focusing on the oplog-based replication mechanism. Specifically, we exported the oplog.rs collection from the local database, which contains a chronological record of write operations for replication purposes. Subsequently, custom scripts were developed to function as a transaction evaluator and cleaner, designed to identify and persist only those operations that were successfully committed before failure events.

Additionally, Node.js was utilized to facilitate real-time monitoring and evaluation of database activity. Leveraging MongoDB Change Streams, our implementation was capable of observing live changes within specified collections and executing callback functions in response to transactional events. This enabled precise tracking and evaluation of transaction consistency and accelerated recovery in a dynamic environment.

We have used a single MongoDB Cluster in two different user accounts to perform testing of our proposed method to recover the data and measure the time taken.

During recovery simulation, while executing our transaction evaluator and cleaner scripts using Node.js and MongoDB Atlas, we have performed experiments on 1.6 million transactions using our proposed method. These 1.6 million transactions are generated on MongoDB Compass from the databases available on the MongoDB Atlas cluster. We have generated the transactions and stored them as documents in a collection in MongoDB Compass using the insert methods. Further, these documents are used as input to filtered_oplog creation and consequently to transaction evaluator and cleaner scripts. We have also used the same 1.6 million transactions using the MongoDB restore utility. The results are as shown in Table 1, which shows the comparison of the Recovery Time taken by the proposed method and MongoDB Restore.

This experiment is exclusively executed, and the restoration process has been given high priority for execution, before recovery simulation. The recovery time difference between our methodology and MongoDB Restore varies only in microseconds every time we have executed the restoration process with the same set of transactions. Although the measured improvement of our method over MongoDB Restore appears in microsecond, this difference is significant in latency-sensitive, high-throughput systems. In environments that operate 24 × 7 and process millions of transactions per second, microsecond-level savings in recovery can reduce total downtime and accelerate recovery. This is particularly critical in applications such as financial services, e-commerce, and real-time analytics, where even minor recovery delays may result in service interruptions and revenue loss.

To confirm that the recovered data is consistent, we have applied the following data consistency accuracy formula.

$$\text{Data Consistency Accuracy} = \frac{\text{Number of Consistent Records after Recovery}}{\text{Total Documents in Dataset}} \text{ X } 100\%$$

$$\text{Data Consistency Accuracy} = \frac{1.6 \text{ million documents}}{1.6 \text{ million documents}} \text{ X } 100\% = 100\%.$$

The data consistency accuracy calculated by the above formula is 100%, which means we have total 1.6 million documents in data set and we have recovered the same number of documents *i.e.*, 1.6 million by using our proposed system. This result highlights that faster recovery is not achieved at the expense of correctness. So, our proposed system accelerates data recovery without any data loss.

At the same time, we acknowledge that our current experiments were conducted on a single cluster under controlled workloads.

The proposed system is flexible and can be made compatible with other NoSQL databases like CouchDB, Cassandra and, Redis. Our algorithms and scripts can be implemented on these NoSQL databases due to similarity in their data structures used for recovery which provide opportunities for cross-platform applicability.

In Cassandra, the data structures which are used during data recovery are Commit log (like MongoDB journaling) for durability and the memTable (like MongoDB oplog) for storing database operations. Both the Commit log and the memTable are compatible with our proposed system. These similarities make it feasible to adapt the evaluator and cleaner scripts to different NoSQL environments. These structures closely resemble MongoDB's journaling and oplog mechanisms, enabling straightforward adaptation of the proposed evaluator and cleaner scripts.

In CouchDB, write ahead log (WAL) is used for durability (like MongoDB Journaling), and append only storage design is used for all database operations. Both WAL and append only storage design are compatible with our proposed system. This design parallels MongoDB's journaling and makes CouchDB compatible with filtered log–based recovery approaches.

In Redis, append only format (like MongoDB Journaling) and Redis Database Backup files (RDB) Snapshot (like MongoDB Oplog) are used and both are compatible with our proposed system. These features make Redis suitable for adopting filtered transaction logs to accelerate recovery.

These mechanisms share functional similarities with MongoDB's journaling and oplog, suggesting that our transaction evaluator and cleaner scripts could be adapted with minimal modifications. In summary, our results demonstrate that the proposed system achieves accelerated recovery with complete data consistency. While the performance gains are expressed in microseconds, their significance is amplified in high-throughput and latency-sensitive applications where rapid recovery is essential. Furthermore, the flexibility of the approach points toward its applicability across other NoSQL platforms.

CONCLUSIONS

To achieve the research objective of accelerating data recovery and creating a robust recovery mechanism, our research work is wrapped up by emphasizing the subsequent achievements, such as monitoring and evaluating transactions by identifying successful and failed transactions using oplog and journal with the construction of a new oplog called filtered_oplog. We have developed an expeditious log-based data recovery methodology for NoSQL databases using filtered_oplog. We have validated the recovered data accuracy by applying data recovery accuracy formulae with verification and confirmation that no loss of data after implementing our methodology. Moreover, the additional overhead is very little compared to the time saved during the recovery process.

Our proposed method works with write-ahead log-like data structures without changing it. Additionally, our method is better than existing methods because it improves the recovery time. Although at a moment, the recovery time of our method minutely differs from that of the existing recovery utility. However, with the larger dataset, our method would have a bigger difference. While existing research explores a wide range of strategies for data recovery in NoSQL systems, such as log-based recovery, checkpointing, undo operations, and resilience practices in platforms like MongoDB Atlas, many of these approaches either focus on system-wide techniques or introduce added complexity and resource overhead. In contrast, the proposed expeditious log-based data recovery methodology introduces a lightweight and precise solution by constructing a filtered_oplog derived from MongoDB's native oplog and journal files. This method not only differentiates between successful and failed transactions, but also ensures accurate data recovery without incurring additional system overhead. Unlike previous works, the proposed approach was quantitatively validated for accuracy, demonstrating complete data recovery and improved efficiency, thereby offering a more practical and resource-efficient solution tailored specifically for NoSQL environments.

The novelty of the proposed method lies in its ability to operate directly on existing write ahead log–like (WAL) structures without requiring any modification to the native logging mechanism of the database. Unlike traditional utility-based recovery approaches, which exhibit near-constant performance regardless of scale, our method shows increasing efficiency gains as dataset volume grows. While the difference in recovery time is modest for smaller datasets, the proposed approach shows progressively larger performance improvements with real-world, large-scale data.

Our proposed system is closely associated with the work of *Matos & Correia (2016)* and *Ammann et al. (2002)*. *Matos & Correia (2016)* tool requires database downtime and the recovery process is slowed down with high-version of documents. Whereas our proposed system works with real time database, no downtime is required and the recovery process is expedited. The work of *Ammann et al. (2002)* presumes that the malicious transactions are pre-identified by a security mechanism, not by the recovery process. So, it is dependent on the security mechanism for the recovery process. However, the proposed work identifies such transactions well in advance to accelerate the recovery process.

Looking ahead, the methodology can be generalized to work across diverse NoSQL platforms by adopting universal log formats capable of accommodating multiple data models. This extension would support cross-platform resilience strategies and allow seamless integration with heterogeneous systems. Moreover, incorporating blockchain-backed logging could further enhance data integrity and ensure immutable, tamper-proof recovery records.

Overall, the enhanced analysis, precision filtering, and real-time operability introduced in this work constitute a substantial and novel contribution to the domain of NoSQL data recovery.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Gayatri Shitanshu Kapadia conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.
- Rustom Darash Morena analyzed the data, authored or reviewed drafts of the article, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

The code is available in the Supplemental Files.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.3507#supplemental-information.

REFERENCES

Ammann P, Jajodia S, Senior Member, IEEE, Liu P, Member, IEEE. 2002. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering* **14(5)**:1167 DOI 10.1109/TKDE.2002.1033782.

Bao X, Liu L, Xiao N, Lu Y, Cao W. 2016. Persistence and recovery for in-memory NoSQL services: a measurement study. In: *2016 2016 IEEE International Conference on Web Services (ICWS)*. San Francisco, CA, USA, 530–537 DOI 10.1109/ICWS.2016.74.

da Silva GHG, Holanda M, Araujo A. 2016. Data replication policy in a cloud computing environment. In: *2016 11th Iberian Conference on Information Systems and Technologies (CISTI)*. Piscataway: IEEE, 1–6 DOI 10.1109/CISTI.2016.7521383.

Gu Y, Wang X, Shen S, Ji S, Wang J. 2015. Analysis of data replication mechanism in NoSQL database MongoDB. In: *2015 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*. Piscataway: IEEE, 66–67 DOI 10.1109/ICCE-TW.2015.7217033.

Haughian G, Osman R, Knottenbelt WJ. 2016. Benchmarking replication in cassandra and MongoDB NoSQL datastores. *International Conference on Database and Expert Systems Applications* **9828(8)**:152–166 DOI 10.1007/978-3-319-44406-2_12.

Luo C. 2012. Interference-free checkpointing algorithm for distributed database system. In: *2012 International Conference on Industrial Control and Electronics Engineering* DOI 10.1109/ICICEE.2012.65.

Mansouri H, Pathan Al-SK. 2020. A resilient hierarchical checkpointing algorithm for distributed systems running on cluster federation. In: Thampi SM, Martinez Perez G, Ko R, Rawat D, eds. *Security in Computing and Communications. SSCC 2019. Communications in Computer and Information Science.* Vol. 1208. Singapore: Springer Nature Singapore Pte Ltd, 99–110 DOI 10.1007/978-981-15-4825-3_8.

Mansouri H, Pathan Al-SK. 2021. A communication-induced checkpointing algorithm for consistent-transaction in distributed database systems. In: Thampi SM, Wang G, Rawat DB, Ko R, Fan CI, eds. *Security in Computing and Communications. SSCC 2020. Communications in Computer and Information Science.* Vol. 1364. Singapore: Springer Nature Singapore Pte Ltd, 21–32 DOI 10.1007/978-981-16-0422-5_2.

Market Share. 2024. Global NoSQL Market to Reach USD 81.9 Billion by 2033, Accelerated by Rising Number of E-Commerce Platforms. *Available at https://www.imarcgroup.com/global-nosql-market* (accessed 7 July 2025).

Matos D, Correia M. 2016. NoSQL Undo: recovering NoSQL databases by undoing operations. In: *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), 31 October 2016–02 November 2016.* DOI 10.1109/NCA.2016.7778616.

Ouyang H, Wei H, Huang Yu, Li H, Pan A. 2016. Verifying transactional consistency of MongoDB. In: *42nd Conference on Very Important Topics (CVIT 2016).*

Paithankar D, Roesle E. 2023. Data resilience strategy with MongoDB atlas. *Available at https://www.mongodb.com/resources/products/capabilities/data-resilience-strategy-with-mongodb-atlas* (accessed 21 January 2026).

Sauer C, Graefe G, Härder T. 2019. FineLine: log-structured transactional storage and recovery. *Proceedings of the VLDB Endowment* **11(13)**:2249–2262 DOI 10.14778/3275366.3284969.

Sung H, Jin M, Shin M, Roh H, Choi W, Park S. 2019. LESS: logging exploiting SnapShot. In: *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*. Kyoto, Japan, 1–4 DOI 10.1109/BIGCOMP.2019.8679377.

Tam Vo HT, Wang S, Agrawal D, Chen G, Ooi BC. 2012. LogBase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment* **5(10)**:1004–1015 DOI 10.14778/2336664.2336673.

Tsai CP, Hsiao HC, Lai YC. 1959. The data recovery service in NoSQL. In: *2022 IEEE International Conference on Big Data (Big Data)*. Vol. 2022, Osaka, Japan, 2394–2401 DOI 10.1109/BigData55660.2022.10020447.

Wang Y, Li Z, Lin W. 2007. A fast disaster recovery mechanism for volume replication systems. In: *HPCC'07: Proceedings of the Third International Conference on High Performance Computing and Communications*. Berlin Heidelberg: Springer-Verlag Berlin Heidelberg, 732–743.

Wang G, Tang J. 2012. The NoSQL principles and basic application of cassandra model. In: *2012 International Conference on Computer Science and Service System*, 1332–1335 DOI 10.1109/CSSS.2012.336.