Journal of Software Engineering and Applications, 2018, 11, 467-485
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

A Surfing Concurrence Transaction Model for Key-Value NoSQL Databases

Changqing Li, Jianhua Gu

School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, China
Email: li_changqing@126.com, gujh@nwpu.edu.cn

How to cite this paper: Li, C.Q. and Gu, J.H. (2018) A Surfing Concurrence Transaction Model for Key-Value NoSQL Databases. *Journal of Software Engineering and Applications*, **11**, 467-485.
https://doi.org/10.4236/jsea.2018.1110028

Received: September 24, 2018
Accepted: October 23, 2018
Published: October 26, 2018

Abstract

As more and more application systems related to big data were developed, NoSQL (Not Only SQL) database systems are becoming more and more popular. In order to add transaction features for some NoSQL database systems, many scholars have tried different techniques. Unfortunately, there is a lack of research on Redis's transaction in the existing literatures. This paper proposes a transaction model for key-value NoSQL databases including Redis to make possible allowing users to access data in the ACID (Atomicity, Consistency, Isolation and Durability) way, and this model is vividly called the surfing concurrence transaction model. The architecture, important features and implementation principle are described in detail. The key algorithms also were given in the form of pseudo program code, and the performance also was evaluated. With the proposed model, the transactions of Key-Value NoSQL databases can be performed in a lock free and MVCC (Multi-Version Concurrency Control) free manner. This is the result of further research on the related topic, which fills the gap ignored by relevant scholars in this field to make a little contribution to the further development of NoSQL technology.

Keywords

NoSQL, Big Data, Surfing Concurrence Transaction Model, Key-Value NoSQL Databases, Redis

1. Introduction

1.1. Background

In the field of database management technology, a huge change is taking place. Relational database technology has become very mature after decades of development. However, as more and more application systems related to big data

were developed, NoSQL (Not Only SQL) database systems, such as Redis [1], MongoDB [2], HBase [3], Neo4J [4], etc., are becoming more and more popular.

Typical applications of NoSQL databases involve many important areas, such as the Internet [5], mobile computation [6], tele-communications [7], bioinformatics [8], education [9], energy [10], and so on. Related researches on NoSQL databases have academic significances and practical values because NoSQL systems have been developed to support applications not well served by relational systems, often involving big data processing [11].

Major NoSQL databases can be classified generally into four categories [12] [13] [14], including key-value store, column-oriented store, document-oriented store and graph store. Representative NoSQL database products include Redis, Hbase, MongoDB, Neo4j and so on, respectively, which belong to one of the four types.

At present, although the NoSQL database systems are not perfect, but its advantages are very obvious. NoSQL databases have some inadequacies, such as not supporting SQL which is industry standard, lacking of reports and other additional features [12]. Meanwhile, the NoSQL systems generally do not provide ACID (Atomicity, Consistency, Isolation and Durability) transactional properties. By giving up ACID constraints, much higher performance and scalability can be achieved [13]. Main advantages of NoSQL are the following aspects: reading and writing data quickly, supporting mass storage, being easy to expand and low cost [12].

1.2. Motivation

Because the NoSQL database systems have weak supports for ACID features, software developers have to solve this problem at application level in scenarios where database transactions are required. This reduces portability and requires system-specific code [11]. At the same time, this increases difficulties of application developments, also reduces the efficiency of software developments.

In order to add transaction feature for some NoSQL database systems, many scholars have tried different techniques. The usual practices of these researchers were using lock technology or MVCC (Multi-Version Concurrency Control) technology.

Unfortunately, as far as the authors know, there is a lack of research on Redis's transaction in the existing literatures. Since Redis itself does not provide mechanisms of multi-version management and locking for data, more efforts must be made in order to add transaction processing mechanism to Redis.

1.3. Our Contributions

This paper proposes a transaction model for key-value NoSQL databases including Redis, and this model is vividly called the surfing concurrence transaction model. The architecture, important features and implementation principle are described in detail. The key algorithms are given in the form of pseudo pro-

gram code, and the performance also was evaluated. This is the result of further research on the related topic, which fills the gap ignored by relevant scholars in this field to make a little contribution to the further development of NoSQL technology.

Our contributions are summarized as follows:

1) This paper presents a transaction model for key-value NoSQL databases including Redis to make possible allowing users to access data in the ACID way, and this model is vividly called the surfing concurrence transaction model.

2) With the proposed model, the architecture, important features, implementation principle and the key algorithms are provided.

3) With the proposed model, the transactions of NoSQL databases can be performed in a lock-free and MVCC-free manner.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the related works. Section 3 describes the transaction model's architecture, important features and implementation principle, and the key algorithms also are given in the form of pseudo program code. Section 4 is the performance evaluation. Section 5 concludes this paper.

2. Related Works

In general, relational database management systems provide transaction support, while NoSQL database management systems can achieve higher performance and scalability by abandoning the limitations of ACID transaction features. Therefore, many scholars used different techniques in order to add transaction function to specific NoSQL database system.

Snapshot isolation technology aims to improve concurrency and consistency by maintaining different versions of data. When snapshot isolation is used in a transaction, the database server can return a committed version of the data in response to any read request. It does not acquire read lock when doing this operation, so it will not cause interference to users who are writing data. Multi-version data is a necessary condition for transaction control through snapshot isolation and can be supported by time stamps. Many scholars used this technique to implement transaction control for specific NoSQL database system. In literature [15], the author used a centralized method to implement snapshot isolation, and implemented a transaction support function of multi-row data for HBase. This method uses a lock-free commit algorithm to avoid the negative effects of distributed lock. In addition, it synchronizes partial transaction meta-data to the client, so that many queries can be performed on the client side, thus reducing the network overhead. In literatures [16] [17] [18], the authors also implemented a transaction support function of multi-row records for HBase using the method of snapshot isolation. In literature [19], the author implemented a multi-row data transaction support function for Cassandra and Hyperdex by using snapshot isolation and multi-version distributed cache technology. In literature [20], the transaction support function for MongoDB based

on snapshot isolation was built by the authors.

Some scholars used lock technology to implement transaction support for Some NoSQL databases. For instance, in literature [21], the author used Percolator to provide multi-row transactional support for BigTable. This method implements a two phase commitment protocol by designing a lightweight lock.

Of course, implementing transaction support in the database store itself also is an option. Entity group is a concept put forward in Megastore [22] which based on BigTable. An entity group is formed by multiple rows of data logically belonging to the same object data entity. The transaction operations of Megastore are entity group granularity. That is, the operation of multiple rows of data in the entity group can guarantee the ACID property. In literature [23], the author used the Paxos protocol to implement Spinnaker database which is extensible, consistent and highly available. But only the transaction function of single-row data was implemented.

At the same time, some researchers used specific transaction management protocol scheduled on client-side to achieve transaction control. In literature [24], the author implemented transaction guarantee function of multi-row data using a transaction management protocol on client-side. However, this method depends on the precision of the client clock. In literature [25] [26], the authors implemented a transaction support function of multi-row data for HBase using two-phase commit protocol. In literature [27], the author implemented a middle layer of transaction support for MongoDB database using four-phase commit protocol.

To sum up, at present, the research interests of scholars mainly focus on the transaction control of specific NoSQL databases, such as HBase, MongoDB, Cassandra, BigTable, Spinnaker, Hyperdex, and so on. At the same time, it is not difficult to find that the transaction support topics related to the NoSQL database systems of Key-Value type, such as the famous Redis and Memcached, are hardly covered in the literatures. Therefore, the results of this paper try to fill the gap in this field.

3. Surfing Concurrence Transaction Model

3.1. Model's Architecture

As shown in **Figure 1**, the surfing concurrence transaction model proposed in this paper mainly consists of six functional components, six queues and transaction objects. The six functional components include transaction client, transaction manager, transaction sorter, transaction dispatcher, transaction executor and actors. Six queues include waiting queue, waiting read queue, waiting write queue, waiting execution queue, executing queue and finished execution queue.

3.1.1. Transaction Object

The transaction object is the basic entity in this transaction model, and it consists of the following properties:

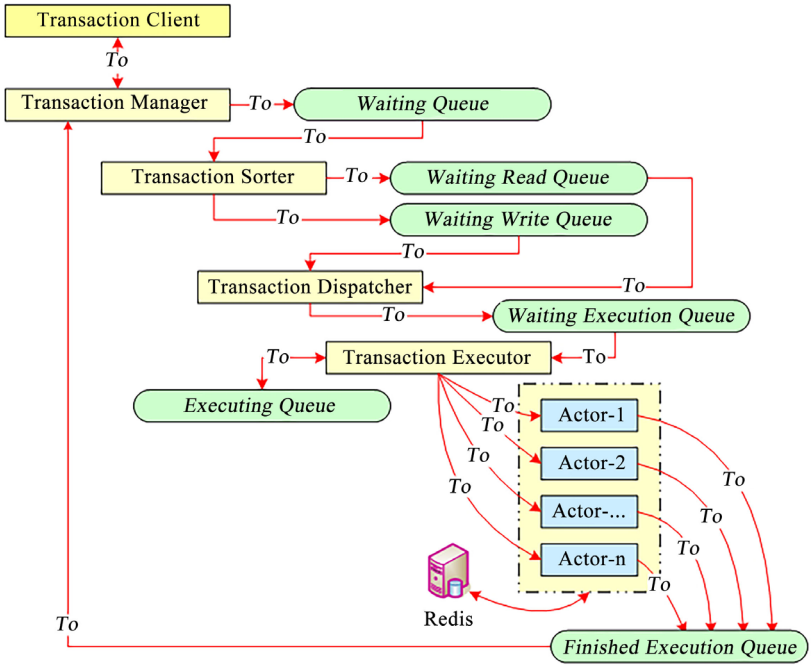

Figure 1. Model's architecture.

1) The unique identity of a transaction.
2) The list of original data operations.
3) The start time of a transaction.
4) The timeout of a transaction.
5) The list of parsed data operations.
6) Transaction type.
7) The union of keys involved in all data operations (used only when dealing with write transaction type).
8) The list of the execution results of data operations.
9) The execution result of a transaction.

When a transaction client generates a transaction object, only some properties are set, including the unique identity of a transaction, the list of original data operations, the start time of a transaction, and the timeout of a transaction. In other steps of a transaction processing, the other properties of a transaction object will be set by other functional components.

3.1.2. Transaction Client

The main functions of a transaction client are as follows:

1) Encapsulating all data operations involved in a transaction into a transaction object.
2) Sending the transaction object to the transaction manager by a network system.
3) Receiving a processed transaction object from the transaction manager.

3.1.3. Transaction Manager

The transaction manager is located on the server side of this transaction model,

and its main functions are as follows:

1) Receiving transaction objects sent from each transaction client.

2) Parsing the data operations in the current transaction object, including the command of the operation and its parameters, and saving these parsed results into the corresponding properties.

3) Storing a preprocessed transaction object into the waiting queue.

4) Monitoring the finished execution queue and returns the completed transaction object to the corresponding transaction client.

3.1.4. Transaction Sorter

The main functions of the transaction sorter is classifying the transaction objects, and moving them from the waiting queue into the waiting read queue or the waiting write queue.

1) The transaction type is determined by scanning the data operations of current transaction object, that is, if all data operations in the transaction object are read operations, then the transaction type is read transaction, otherwise, it is a write transaction.

2) For write transaction, it is necessary to calculate the union of keys involved in all data operations of a transaction object, so as to prepare for the detection on writing conflict performed by the transaction dispatcher.

3.1.5. Transaction Dispatcher

The transaction dispatcher alternately accesses the waiting read queue and the waiting write queue. When accessing the waiting write queue, some transaction objects are fetched and conflict detection is performed. Finally, the non-conflicting transaction objects are put into the waiting execution queue, and the transaction executor are notified to take away the transaction objects. The transaction dispatcher handles the waiting read queue in a similar way, but without conflict detection step. In scheduling, if there is no element in current queue, the elements in the other queue will be processed.

The following parameters are used to control the numbers of transaction objects in a batch:

1) *Nmr*: it is the maximum numbers of transaction objects that can be fetched from the waiting read queue in a batch.

2) *Nmw*: it is the maximum numbers of transaction objects that can be fetched from the waiting write queue in a batch.

By changing these two parameters, the numbers of transaction objects fetched in a batch can be adjusted. Therefore, they can be used to optimize the performance of transaction model. For example, for a scenario which has more read transaction objects, they can be set to make *Nmr* bigger than *Nmw*.

3.1.6. Transaction Executor

The main functions of the transaction executor are as follows:

1) When received notification from the transaction dispatcher, transaction executor can move all transaction objects from the waiting execution queue to

the executing queue.

2) Notifying the transaction dispatcher to prepare the transaction objects for the next batch.

3) The transaction objects are fetched from the executing queue, and they are executed concurrently by the actor threads.

4) Monitoring the execution progresses of all actor threads. When all actors finished their works, transaction executor will start the next round of work.

3.1.7. Actor

The actor is responsible for the processing of a specific transaction object, the main functions of which are as follows:

1) Performing all data operations in the transaction object. When performing a data operation, it must prewrite the log of data operation. In addition, if an exception occurs, all data operations involved in the transaction object will be rolled back.

2) After a transaction is completed, set the results of transaction execution in the transaction object and place the transaction object in the finished execution queue.

3) Notifying the transaction executor that the transaction object has been completed.

3.2. Model's Features

1) Read transaction objects and write transaction objects are not processed in the same batch, avoiding conflicts between them.

The scheduling policy of the transaction dispatcher makes each batch of transaction objects to be the same type, that is, in the same batch, either all are read transaction objects or all are write transaction objects. Therefore, the read transaction objects and the write transaction objects are not processed at the same time.

2) Pre-detecting conflicts between write transaction objects so that there are no conflicts between write transaction objects executed in the same batch.

The transaction dispatcher must pre-detect the write conflicts during the scheduling of write transaction objects, so there are no conflicts between write transaction objects in the same batch.

3) Transaction objects in the same batch are executed concurrently.

Because there are no conflicts between transaction objects in the same batch, they can be executed concurrently.

4) The maximum numbers of transaction objects which are processed in a batch can be adjusted by parameters.

Because the ratio between read transaction objects and write transaction objects can be different in many application systems, the maximum numbers of read transaction objects per batch and the maximum numbers of write transaction objects in a batch can be adjusted by the specific parameters of the model.

5) All components of this transaction model are run in a pipe-lined manner.

At a specific time point, transaction objects of the previous batch are being executed in the transaction executor, while the transaction dispatcher is also preparing the transaction objects for next batch. At the same time, the transaction sorter and the transaction manager also are busy with their tasks. Therefore, these components work in a pipe-lined manner.

3.3. Transaction Sorter Algorithm

The key idea of the transaction sorter algorithm is to discriminate between read transaction objects and write transaction objects in waiting queue, and to move the classified transaction objects into waiting read queue or waiting write queue respectively. The implementation algorithm of the transaction sorter is as shown in **Algorithm 1**, and the detailed logic descriptions are as follows:

Line 2: fetching a transaction object from the waiting queue.

Line 3: completing the determination of the transaction type:

1) If all data operations in a transaction object are read operations, then the transaction type of current transaction object is read transaction.

2) If at least one operation of writing data is included in all data operations of a transaction object, then the transaction type of current transaction object is a write transaction.

Algorithm-1: Transaction Sorter

Input: ***WQ***: *Waiting Queue.*
Output: ***WRQ***: *Waiting Read Queue*, ***WWQ***: *Waiting Write Queue.*
1: **WHILE** (*thread is not stoped*)
2: *currentTO* ← *WQ.**take***();
3: *T* ← ***calTransType***(*currentTO*);
4: **IF** (*T*='*R*') **THEN**
5: *WRQ.**put***(*currentTO*);
6: **ELSE**
7: *currentTO.AllKeys* ← ***calAllKeys***();
8: *WWQ.**put***(*currentTO*);
9: **END IF**
10: **END WHILE**

Lines 4 - 5: the transaction sorter will put it into the waiting read queue if a transaction object is read transaction type.

Lines 6 - 8: the write transaction object is processed, where line 7: computing the union of keys involved in all data operations in current transaction object; Line 8: moving the processed transaction object into the waiting write queue.

The time complexity of the transaction sorter algorithm is analyzed as follows:

Line 3: in determining the type of a transaction object, it is necessary to traverse the various data operations in the transaction object. The time complexity is $O(N)$, where N represents the numbers of data operations included by a transaction object.

Line 7: when calculating the union of keys involved in all data operations in the current transaction object, it is also necessary to traverse each data operation in the transaction object. The time complexity is $O(N)$, in which N represents

the numbers of data operations included by a transaction object.

Therefore, for a read transaction object, the time complexity is $O(N)$, and for a write transaction object, the time complexity is $O(2N)$.

3.4. Transaction Dispatcher Algorithm

The key idea of the transaction dispatcher algorithm is to take a batch of transaction objects from the waiting read queue or the waiting write queue in turn, and put it into the waiting execution queue after conflict detection. Finally, it will notify the transaction executor to take these transaction objects away. The implementation algorithm of the transaction dispatcher is as shown in **Algorithm 2**, and the details are explained as follows:

Lines 2 - 4: if the transaction executor has moved away all the transaction objects prepared by the transaction dispatcher from the waiting execution queue, it will notify the transaction dispatcher immediately to prepare the next batch of transaction objects. Therefore, the transaction dispatcher will need to wait when the notification of the transaction executor is not received; otherwise, the two batches next to each other will collide. This is to resolve the thread synchronization problem between them.

Algorithm-2: Transaction Dispatcher

Input: ***WRQ***: *Waiting Read Queue*, ***WWQ***: *Waiting Write Queue*, ***TC***: *Thread Code*, ***RT***: *Read Turn*, ***Nmr***: *Numbers of Maximum Read*, ***Nmw***: *Numbers of Maximum Write*.
Output: ***WEQ***: *Waiting Execution Queue*.

```
1: WHILE (thread is not stoped)
2:   WHILE (TC is not self)
3:     await();
4:   END WHILE
5:   IF (RT is read turn) THEN
6:     IF (WRQ is not empty) THEN
7:       r ←calBatchSizeOfRead(Nmr);
8:       FOR (i=0;i<r;i++) THEN
9:         WEQ.put(WRQ.take());
10:      END FOR
11:    END IF
12:    RT ← false;
13:  ELSE
14:    IF (WWQ is not empty) THEN
15:      w ← calBatchSizeOfWrite(Nmw);
16:      FOR (j=0;j<w;j++) THEN
17:        IF (WWQ[j] is not conflict) THEN
18:          WEQ.put(WWQ.take());
19:        ELSE
20:          break;
21:        END IF
22:      END FOR
23:    END IF
24:    RT ← true;
25:  END IF
26:  IF (WEQ is not Empty) THEN
27:    notifyTransExecutor();
28:  END IF
29: END WHILE
```

Lines 5 - 12: the transaction dispatcher handles the read transaction objects. If there is no transaction object in the current waiting read queue, the transaction dispatcher will change the processing mode directly (line 12) to process the write transaction objects. Line 7, calculating the numbers of read transaction objects that current batch can actually fetch: if the numbers of transaction objects in the current waiting read queue is larger than the parameter *Nmr*, then the numbers of transaction objects that can schedule in current batch equals the parameter *Nmr*; otherwise, All read transaction objects need to be processed in current batch. Lines 8 - 10, the transaction dispatcher transfers this batch of transaction objects from the waiting read queue into the waiting execution queue.

Lines 13 - 25: the transaction dispatcher handles the write transaction objects. If there is no transaction object in the current waiting write queue, the transaction dispatcher will change the processing mode directly (line 24) to process the read transaction objects. In line 15, calculating the numbers of write transaction objects that can actually fetch in current batch: if the numbers of transaction objects in the current waiting write queue is larger than the parameter *Nmw*, the numbers of transaction objects that can schedule in current batch equals the parameter *Nmw*, otherwise, All write transaction objects may need to be processed in current batch. In lines 16 - 22, the transaction dispatcher traverses these transaction objects to detect conflicts (line 17). If there is no conflict, the transaction object will be transferred into the waiting execution queue (line 18), otherwise, the traversal operation is stopped (line 20). The basis of conflict detection is whether the keys of data operations of these transaction objects are overlapped, and this can achieved by intersection of them. It needs to be noted that the union of all keys involved in each transaction object has been calculated in the transaction sorter algorithm.

Lines 26 - 28: after this round of scheduling, if there are transaction objects in the waiting execution queue, the transaction dispatcher will notify the transaction executor to fetch them in time.

The time complexity of the transaction dispatcher algorithm is analyzed as follows:

Lines 5 - 12: when the transaction dispatcher processes the read transaction objects, it needs to traverse the waiting read queue, and the time complexity is $O(N)$, where N represents the numbers of read transaction objects for a batch.

Lines 13 - 25: when the transaction dispatcher processes the write transaction objects, it needs to traverse the waiting write queue, and the time complexity is $O(N)$, where N represents the numbers of write transaction objects in a batch.

Since only one transaction type (read transaction or write transaction) is processed in a round of transaction scheduling, the total time complexity is $O(N)$, where N represents the numbers of transaction objects in a batch.

3.5. Transaction Executor Algorithm

The key idea of the transaction executor algorithm, as shown in **Algorithm 3**, is

to fetch all transaction objects from the waiting execution queue prepared by the transaction dispatcher and execute them concurrently. Finally, the processed transaction objects are stored into the finished execution queue. The details are explained as follows:

Lines 2 - 4: when the transaction dispatcher prepared a batch of transaction objects, it will notify the transaction executor to retrieve the transaction objects from the waiting execution queue. Therefore, the transaction executor must wait when no notification is received from the transaction dispatcher. This is to resolve the thread synchronization problem between them.

Lines 5 - 8: after receiving a notification from the transaction dispatcher, the transaction executor will transfer the transaction objects from the waiting execution queue to the executing queue, and notifies the transaction dispatcher to prepare the next batch of transaction objects. In this way, the transaction objects of the next batch can be prepared by the transaction dispatcher in parallel while the transaction objects of the current batch are being processed by the transaction executor.

Algorithm-3: Transaction Executor

Input: ***WEQ***: *Waiting Execution Queue,* ***TC***: *Thread Code,* ***EQ***: *Excecuting Queue.*
Output: ***FEQ***: *Finished Execution Queue.*

```
1: WHILE (thread is not stoped)
2:     WHILE (TC is not self)
3:         await();
4:     END WHILE
5:     FOR (i=0;i<WEQ.size;i++)
6:         EQ.put(WEQ.take());
7:     END FOR
8:     notifyTransDispatcher();
9:     CDL ← new CountDownLatch(EQ.size);
10:    FOR (j=0;j<EQ.size;j++)
11:        A ← new Actor(EQ.take(),FEQ,CDL);
12:        ThreadPool.submit(A);
13:    END FOR
14:    CDL .await();
15: END WHILE
```

Lines 9 - 14: the transaction executor executes transaction objects concurrently. In line 9, it requests a counter object firstly whose maximum numbers equals the numbers of transaction objects in this batch; in lines 10 - 13, it then fetches transaction objects from the executing queue, then these objects are given to the instantiated actors; finally, the thread pool is responsible for executing these actors, and these actors will put the finished transaction objects into the finished execution queue and subtract the counter by 1. In line 14, the transaction executor can monitor these execution progresses of individual actor thread. After this batch is completed, the execution logic will go back to lines 2 - 4 to prepare for the next batch.

The time complexity of the transaction executor algorithm is analyzed as follows:

Lines 5 - 8: the transaction executor needs to traverse the waiting execution queue and the time complexity is $O(N)$, where N is the numbers of transaction objects.

Lines 9 - 14: the transaction executor needs to traverse the executing queue, and the time complexity is $O(N)$, where N is the numbers of transaction objects.

Therefore, the time complexity of the whole logic is $O(2N)$, where N is the numbers of transaction objects.

3.6. Exception Handling Method

3.6.1. Transaction Rollback Method

During data manipulations in a transaction object, if an exception occurs, then a transaction rollback is required, that is, to undo any changes that have been made.

The transaction rollback is based on the log of the data operations, which records the following important information:

1) Checkpoint identification.

2) The identity of a transaction, the start mark and end mark of a transaction.

3) The type of data operation, including insert, delete and update.

4) Objects of data operation.

5) Data before modification: for insert, this item is empty.

6) Modified data: for deletion, this item is empty.

When a transaction rolls back, the log file will be scanned backward, all update operations of the transaction are searched, and the update operations of the transaction are reversed finally. If the data operation is an insert command, a delete operation will be performed when the data is rolled back. If the data operation is a delete command, an insert operation will be performed when the data is rolled back. If the data operation is an update command, the data will be changed into the original data when the data is rolled back.

3.6.2. Timeout Processing Method

When a transaction object fails, it may cause subsequent batches of transaction objects to be blocked. In order to solve this problem, the mechanism of timeout processing is introduced in the model. The processing methods are as follows:

1) On the transaction client, start time and timeout time are set for each transaction object. The transaction client will abort the transaction object if the result is not received when timeout happens.

2) In the transaction manager, the timeout property is detected before each transaction object is transferred to the waiting queue: if it is timed out, it is not placed into the waiting queue, and put it into the finished execution queue after setting the result of transaction execution.

3) In the transaction sorter, the timeout property is detected before each transaction object is transferred to the waiting read queue or the waiting write queue: if it is timed out, it is not placed into the waiting read queue or the waiting write queue, after setting the result of transaction execution, it will be put

into the finished execution queue.

4) In the transaction dispatcher, the timeout property is detected before each transaction object is transferred to the waiting execution queue: if it is timed out, it is not placed into the waiting execution queue, and after setting the result of transaction execution, it will be put into the finished execution queue.

5) In the transaction executor, firstly, the timeout property of each transaction object is detected before it is transferred into the executing queue: if it is timed out, it is not put into the executing queue, and after setting the result of transaction execution, it will be put it into the finished execution queue. Secondly, in the process of executing a transaction by an actor, if the transaction is timed out, the transaction will be aborted and rolled back. After setting the result of transaction execution, it will be placed into the finished execution queue.

3.6.3. Faults Detection Method

When restarted, faults detection is performed as follows:

1) Log file is scanned forward to find the nearest checkpoint mark.

2) Starting from the nearest checkpoint, the log file is scanned forward to look for transactions which have been committed before failure, and these transactions will be put into the redo queue.

3) Looking for transactions which were not completed at the time of failure, and these transactions will be put into the undo queue.

4) Transactions in the redo queue will be reprocessed, that is, the log file is scanned forward, the operations registered in the log file will be re-executed, and the results of the operations also will be written into the database.

5) Transactions in the undo queue will be revoked, that is, inverse operations will be done on all modification operations.

4. Performance Evaluation

4.1. Metric and Parameters

The numbers of concurrent transactions per second, represented by *TPS*, is used as metric of this model to evaluate the main performance, and *TPS* is obtained by calculating the reciprocal of the average processing time of a transaction.

The main parameters involved in this model are as follows:

1) *Rr*: the ratio of the numbers of read transactions to the numbers of total transactions.

2) *Rw*: the ratio of the numbers of write transactions to the numbers of total transactions.

3) *Rc*: the ratio of the numbers of conflicting transactions to the numbers of total transactions.

4) *Nmr*: The maximum numbers of read transaction objects executed in a batch.

5) *Nmw*: The maximum numbers of write transaction objects executed in a batch.

The software environment parameters are as follows:

Redis Data Store: 3.0;

JDK: 1.7, 64 bit;

OS: Red Hat Enterprise Linux 6.

The hardware environment parameters are as follows:

Computers: 2 computers for deploying respectively of Redis and prototype system. The key parameters of each computer are as follows:

CPU: Intel i5-3210M (4 cores, 2.6 GHz);

RAM: 4 G (DDR3);

Disk: 1 T, 7200 rpm;

Network card: gigabit Ethernet network card;

Network switch: 1 gigabit Ethernet switch for connecting these computers.

4.2. Change Analysis of *Rr* and *Rw* Parameters

Experiment 1: When other parameters are invariant, the changes of performance are evaluated by changing the parameter *Rr* and parameter *Rw*, as shown in Figure 2. In each scenario, 1000 transaction objects are processed concurrently.

As can be seen from Figure 2, performance improves when the ratio of parameter *Rr* to parameter *Rw* increases. The main reason is that the transaction dispatcher does not need perform conflict detection when scheduling the read transaction objects, so when the ratio of parameter *Rr* to parameter *Rw* increases, the time of conflict detection will be greatly reduced, and the overall performance will be improved.

4.3. Change Analysis of *Nmr* and *Nmw* Parameters

Experiment 2: When other parameters are invariant, the changes of performance are evaluated by changing the parameter *Nmr* and parameter *Nmw*, as shown in Figure 3. In each scenario, 1000 transaction objects are processed concurrently.

As can be seen from Figure 3, performance improves when the ratio of parameter *Nmr* to *Nmw* increases. The main reason is that in this scenario, the parameter values, *Rr* = 80% and *Rw* = 20%, means that there are more read transaction objects waiting to be processed. At the same time, when the ratio of parameter *Nmr* and parameter *Nmw* is increased, more read transaction objects can be processed in one batch, thus the overall performance is improved.

4.4. Change Analysis of *Rc* Parameter

Experiment 3: When other parameters are invariant, the changes of performance are evaluated by changing the parameter *Rc*, as shown in Figure 4. In each scenario, 1000 transaction objects are processed concurrently.

As can be seen from Figure 4, performance decreases when the parameter *Rc* increases. The main reason is that when the parameter *Rc* increases, the numbers of write transaction objects processed in the same batch will decrease, which results in lower performance. In a very extreme case, when there are conflicts

among all write transaction objects, the overall execution logic will become to "serial execution".

4.5. Delay Analysis of Transaction

Experiment 4: Performance differences between transactional mode and non-transactional mode were tested, as shown in Figure 5. In each scenario, 1000 transaction objects are processed concurrently.

As can be seen from Figure 5, there is an operational delay after using transaction mode: the average response time increases 74 microseconds (about 19%). This is mainly caused by transaction sorting and conflict detection.

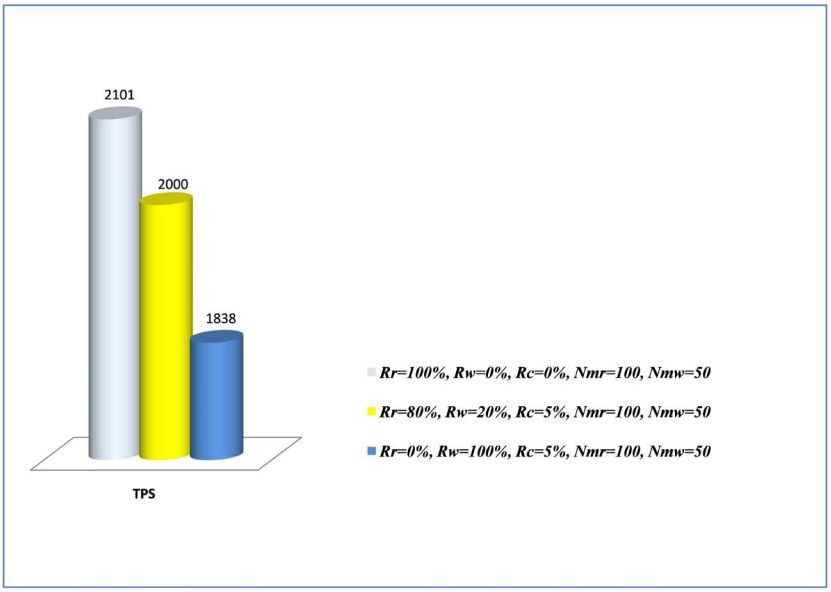

Figure 2. Change analysis of *Rr* and *Rw* parameters.

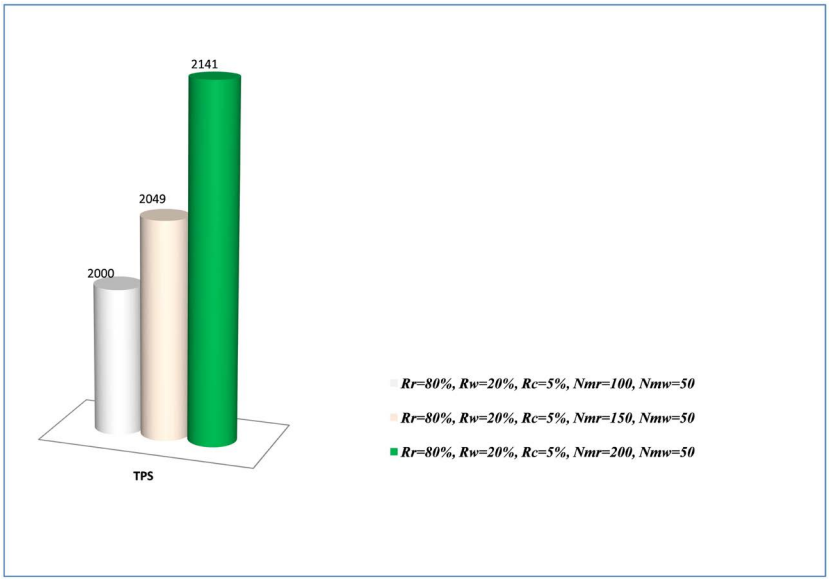

Figure 3. Change analysis of *Nmr* and *Nmw* parameters.

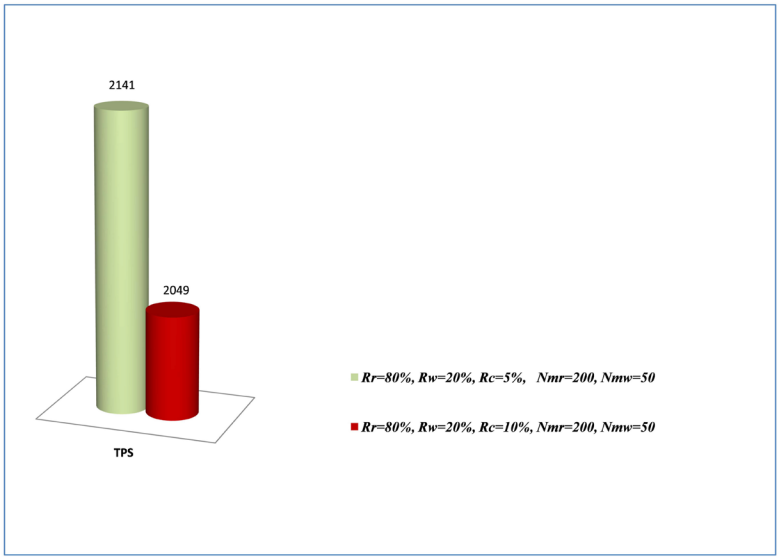

Figure 4. Change analysis of *Rc* parameter.

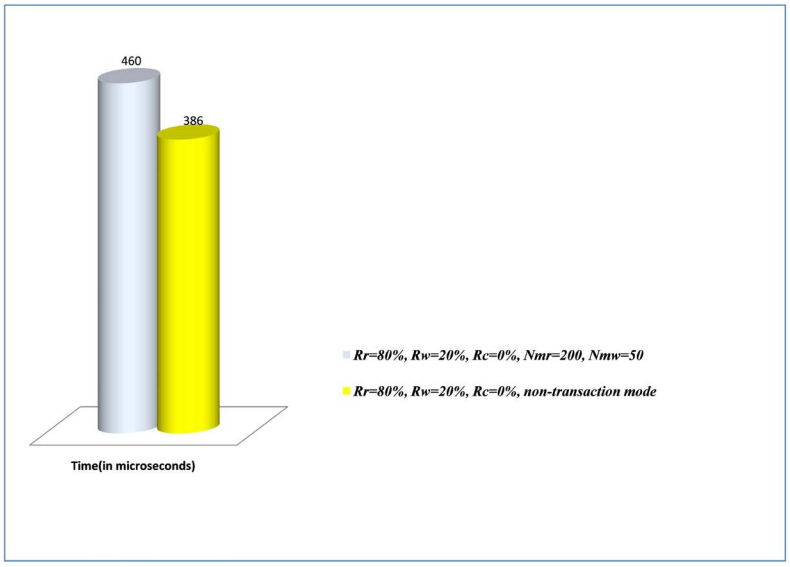

Figure 5. Delay analysis of transaction.

4.6. Evaluation Summary

1) The larger the numbers of read transaction objects in all transaction objects, the better the overall performance.

2) While the parameter *Rr* and parameter *Rw* remain unchanged, adjusting the parameter *Nmr* and parameter *Nmw* properly will help to improve the performance.

3) When the probability of conflict between write transaction objects is reduced, the overall performance is better.

5. Conclusions

The aim of this paper is to propose an effective transaction model for key-value

NoSQL databases including Redis to make possible allowing users to access data in the ACID way. The key contents were described in detail including the model's architecture, important features and implementation principle. In addition, some key algorithms were also given in the corresponding section, and these algorithms are presented in the form of pseudo program code. Finally, the performance also was evaluated. The model can effectively reduce development complexity and improve development efficiency of the software systems with transaction demand. This is the result of further research on the related topic, which fills the gap ignored by relevant scholars in this field to make a little contribution to the further development of NoSQL technology.

Future works mainly involve the optimizations of algorithms. In addition, authors also intend to support more NoSQL databases in the prototype system.

Conflicts of Interest

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

References

[1] Carlson, J.L. and Sanfilippo, S. (2013) Redis in Action. Manning Publications Co., Greenwich, CT.

[2] Chodorow, K. (2013) MongoDB: The Definitive Guide. O'Reilly Media Inc., USA.

[3] George, L. (2011) HBase: The Definitive Guide. O'Reilly Media Inc., USA.

[4] Vukotic, A., Watt, N., Abedrabbo, T., Fox, D. and Partner, J. (2014) Neo4j in Action. Manning Publications Co., Greenwich, CT.

[5] Pokorny, J. (2013) NoSQL Databases: A Step to Database Scalability in Web Environment. *International Journal of Web Information Systems*, **9**, 278-283. https://doi.org/10.1108/17440081311316398

[6] Ickert, F., Fabro, M., Almeida, E. and Scherzinger, S. (2013) NoSQL Data Model Evaluation on App Engine Datastore. *The 28th Brazilian Symposium on Databases*, 2013, 1-6.

[7] Cruz, F., Gomes, P., Rui, O. and Pereira, J. (2011) Assessing NoSQL Databases for Telecom Applications. *The* 2011 *IEEE* 13*th Conference on Commerce and Enterprise Computing*, Luxembourg, 5-7 September 2011, 267-270. https://doi.org/10.1109/CEC.2011.48

[8] Shao, B. (2015) Are NoSQL Data Stores Useful for Bioinformatics Researchers. *International Journal on Recent and Innovation Trends in Computing and Communication*, **3**, 1704-1708. https://doi.org/10.17762/ijritcc2321-8169.1503176

[9] Xiong, W., Hawley, D. and Monismith, D. (2015) NoSQL in Database Education: Incorporating Non-Relational Concepts into a Relational Database Course: Panel Discussion. *Journal of Computing Sciences in Colleges*, **30**, 151-152.

[10] Costa, C. and Santos, M.Y. (2016) Reinventing the Energy Bill in Smart Cities with NoSQL Technologies. In: Ao, S., Yang, G.-C. and Gelman, L., Eds., *Transactions on Engineering Technologies*, Springer, Singapore, 383-396. https://doi.org/10.1007/978-981-10-1088-0_29

[11] Lawrence, R. (2014) Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB. *IEEE Con*ference *on Computational*

Science and Computational Intelligence, Las Vegas, NV, 10-13 March 2014, 285-290. https://doi.org/10.1109/CSCI.2014.56

[12] Han, J., Haihong, E., Le, G. and Du, J. (2011) Survey on NoSQL Database. 6*th International Conference on Pervasive Computing and Applications*, Port Elizabeth, 26-28 October 2011, 363-366.

[13] Tbhuvan, N. and Sudheep, E.M. (2015) A Technical Insight on the New Generation Databases: NoSQL. *International Journal of Computer Applications*, **121**, 24-26. https://doi.org/10.5120/21553-4578

[14] Cattell, R. (2010) Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record*, **39**, 12-27. https://doi.org/10.1145/1978915.1978919

[15] Ferro, D., Junqueira, F., Kelly, I., Reed, B. and Yabandeh, M. (2014) Omid: Lock-Free Transactional Support for Distributed Data Stores. 2*0*14 *IEEE* 3*0th International Conference on Data Engineering*, Chicago, IL, 31 March-4 April 2014, 676-687. https://doi.org/10.1109/ICDE.2014.6816691

[16] Padhye, V. and Tripathi, A. (2015) Scalable Transaction Management with Snapshot Isolation for NoSQL Data Storage Systems. *IEEE Transactions on Services Computing*, **8**, 121-135. https://doi.org/10.1109/TSC.2013.47

[17] Ramesh, D., Jain, A.K. and Kumar, C. (2012) Implementation of Atomicity and Snapshot Isolation for Multi-Row Transactions on Column Oriented Distributed Databases Using RDBMS. 2012 *International Conference on Communications, Devices and Intelligent Systems* (*CODIS*), Kolkata, India, 28-29 December 2012, 298-301. https://doi.org/10.1109/CODIS.2012.6422197

[18] Cai, P. and Ni, L. (2012) An Approach of Multi-Row Transaction Management on HBase with Serializable Snapshot Isolation. *Proceedings of* 2012 2*nd International Conference on Computer Science and Network Technology*, Changchun, China, 29-31 December 2012, 1741-1744. https://doi.org/10.1109/ICCSNT.2012.6526257

[19] Cruz, F., Vilaça, R., Oliveira, R., Pereira, J. and Coelho, F. (2014) pH1: A Transactional Middleware for NoSQL. 2014 *IEEE* 33*rd International Symposium on Reliable Distributed Systems*, Nara, Japan, 6-9 October 2014, 115-124. https://doi.org/10.1109/SRDS.2014.23

[20] Ogunyadeka, A., Younas, M., Zhu, H. and Aldea, A. (2016) A Multi-Key Transactions Model for NoSQL Cloud Database Systems. 2016 *IEEE Second International Conference on Big Data Computing Service and Applications*, Oxford, UK, 29 March-1 April 2016, 24-27. https://doi.org/10.1109/BigDataService.2016.32

[21] Peng, D. and Dabek, F. (2010) Large-Scale Incremental Processing Using Distributed Transactions and Notifications. *Usenix Symposium on Operating Systems Design and Implementation*, Vancouver, 4-6 October 2010, 4-6.

[22] Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., *et al*. (2011) Megastore: Providing Scalable, Highly Available Storage for Interactive Services. *CIDR* 2011, *Fifth Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, 9-12 January 2011, 223-234.

[23] Rao, J., Shekita, E.J. and Tata, S. (2011) Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. *Proceedings of the Vldb Endowment*, **4**, 243-254. https://doi.org/10.14778/1938545.1938549

[24] Kanwar, R., Trivedi, P. and Singh, K. (2013) NoSQL, a Solution for Distributed Database Management System. *International Journal of Computer Applications*, **67**, 6-9. https://doi.org/10.5120/11365-6602

[25] Dharavath, R., Jain, A., Kumar, C. and Kumar, V. (2014) Accuracy of Atomic Transaction Scenario for Heterogeneous Distributed Column-Oriented Databases.

Intelligent Computing, Networking, and Informatics, **243**, 491-501.
https://doi.org/10.1007/978-81-322-1665-0_47

[26] Dharavath, R. and Kumar, C. (2015) a Scalable Generic Transaction Model Scenario for Distributed NoSQL Databases. *Journal of Systems and Software*, **101**, 43-58.
https://doi.org/10.1016/j.jss.2014.11.037

[27] Lotfy, A., Saleh, A., El-Ghareeb, H. and Ali, H. (2015) A Middle Layer Solution to Support ACID Properties for NoSQL Databases. *Journal of King Saud University—Computer and Information Sciences*, **28**, 133-145.
https://doi.org/10.1016/j.jksuci.2015.05.003