J Inf Process Syst, Vol.13, No.6, pp.1575~1584, December 2017
https://doi.org/10.3745/JIPS.04.0052

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Transaction Processing Method for NoSQL Based Column

Jeong-Joon Kim*

Abstract
As interest in big data has increased recently, NoSQL, a solution for storing and processing big data, is getting attention. NoSQL supports high speed, high availability, and high scalability, but is limited in areas where data integrity is important because it does not support multiple row transactions. To overcome these drawbacks, many studies are underway to support multiple row transactions in NoSQL. However, existing studies have a disadvantage that the number of transactions that can be processed per unit of time is low and performance is degraded. Therefore, in this paper, we design and implement a multi-row transaction system for data integrity in big data environment based on HBase, a column-based NoSQL which is widely used recently. The multi-row transaction system efficiently performs multi-row transactions by adding columns to manage transaction information for every user table. In addition, it controls the execution, collision, and recovery of multiple row transactions through the transaction manager, and it communicates with HBase through the communication manager so that it can exchange information necessary for multiple row transactions. Finally, we performed a comparative performance evaluation with HAcid and Haeinsa, and verified the superiority of the multirow transaction system developed in this paper.

Keywords
Bigdata, HBase, Multi-Row Transaction, NoSQL

1. Introduction

Data sets beyond the scale that can be handled by existing data management tools are being defined as an issue of big data [1,2]. Since simultaneous input and various attributes of big data are limited in terms of technology or cost to process in existing relational database, various kinds of NoSQL [3] such as column based database and document based database was developed.

While NoSQL supports high speed and high availability when processing large and varied large data, it does not support multiple row transactions, which cannot guarantee data integrity. Therefore, recent research on multirow transaction in NoSQL has been actively carried out [4,5]. However, existing researches have a disadvantage that the number of transactions that can be processed per unit time is low and the performance is degraded.

Therefore, in this paper, we design and implement a multi - row transaction system on Hadoop HBase, which has been recently studied in NoSQL. This system adds a transaction column to all user

Manuscript received April 6, 2017; first revision June 5, 2017; accepted June 17, 2017.
Corresponding Author: Jeong-Joon Kim (jjkim@kpu.ac.kr)
* Dept. of Computer Science & Engineering, Korea Polytechnic University, Siheung, Korea (jjkim@kpu.ac.kr)

tables of HBase and implements transaction manager to control transaction execution, so that it can process high transaction per unit time compared with existing system. In addition, we implemented a communication manager for efficient data exchange with HBase.

Finally, we compared the performance of two systems that implemented multiple row transactions on HBase in a similar way to this paper, and verified the superiority of the multi-row transaction system developed in this paper.

In this paper, we add a column that manages transaction information to all user tables of HBase, so there is no load on a specific region server, and it is executed through transaction manager composed of transaction execution management module, transaction conflict management module and transaction recovery management module, Collision, and recovery information, thereby improving the concurrency of multiple row transactions.

2. Related Research

2.1 HBase

HBase [6] is Google's BigTable's open source Clone project and is a kind of NoSQL as a column-based distributed database that hooks into Hadoop.

HBase does not support transactions by default, but you can set a lock on a single row to enable transactions if you want. However, since it does not support transactions for multiple rows, it is difficult to use in areas where data integrity is important [7].

2.2 Multi-Row Transaction

A multi-row transaction means that individual operations for two or more rows are grouped into a single unit of work and processed as a single unit of operation. However, existing NoSQL does not support multi-row transactions. However, in the field where data integrity is considered to be important, there is a steady progress in recent multi-row transactions to use NoSQL.

HAcid [5] is a multi-row transaction system implemented in HBase, and it creates a table that manages transaction information in one region server, so that all multi-row transactions Manage information. In addition, a column managing transaction information is added to all user tables. When a transaction is committed, a multi row transaction is implemented by comparing collisions between a table in a region server and a column in a user table. Since this method manages transaction information in one region server, load is concentrated on the region server, and there is a limitation in performance improvement because there are too many transaction information to be managed.

Haeinsa [4] uses a method to manage information of multiple row transactions by adding a column that manages transaction information to all user tables. This method is easy to expand the cluster because it does not load on a specific region server, and it can perform better than HAcid because there is not much transaction information to manage. However, since the recovery information is managed together in the column, There is a drawback that the concurrency of a multi-row transaction is reduced.

3. System Design and Implementation

3.1 System Design

The multi-row transaction system for HBase environment is designed to create transaction columns in all user tables of HBase in order to manage transaction information, transaction manager to manage transaction execution, conflict, recovery, and communication manager. Fig. 1 shows the overall system architecture.

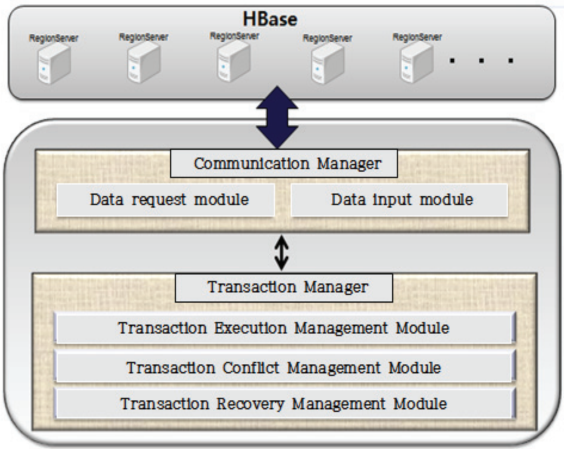

Fig. 1. Overall system structure.

The communication manager is composed of a data request module and a data input module and is responsible for providing all the necessary data to the transaction manager when a transaction is executed, collided, or restored through communication with HBase.

The transaction manager consists of a transaction execution management module, a transaction conflict management module, and a transaction recovery management module. The transaction manager manages requests in connection with a communication manager when a transaction is executed, collided, or restored.

In addition, every user table in HBase has a column that manages transaction information named Transaction and stores transaction information needed to manage execution, collision, and recovery of multiple row transactions. The transaction column is added to the HBase user table and stores the lock status and commit timestamp (CommitTimestamp). Table 1 describes the information managed by the transaction column.

Table 1. Information managed by transactional columns

Information	Explanation
Lock state	
STABLE	The transaction is accessible to the row.
PREWRITTEN	The transaction has already been accessed in the row. Other transactions cannot be accessed.
CommitTimestamp	This is the timestamp at the time the last transaction accessed in the row commits.

As shown in Table 1, the transaction column can have two lock states: STABLE and PREWRITTEN. If the transaction does not access the row, it remains in the STABLE state. If the transaction including the write operation is accessed, the transaction state changes to the PREWRITTEN state. The PREWRITTEN state cannot be accessed by other transactions, and when all operations on the row are complete, the state returns to the STABLE state. It also stores the timestamp at the time the commit of the transaction is completed. Read-only transactions are accessible to the row regardless of the lock state.

3.2 Communication Manager

A data request module, and a data input module, is responsible for requesting HBase for transaction execution, collision, and recovery between the transaction manager and HBase and sending the data to the transaction manager.

When the transaction execution management module sends information necessary for transaction execution, the data request module requests the corresponding data using the Get function to HBase. Fig. 2 shows the data request structure using the Get function.

```
Get(Table name, row key, column name, transaction column)
```

Fig. 2. Data request structure using Get function.

When the data request module requests data to HBase with the structure shown in Fig. 2, the column value satisfying the condition, the lock state of the transaction column, and the commit time stamp are returned. If the state of the transactional column is PREWRITTEN, it waits until it becomes STABLE and returns the value. The data entry module enters HBase's contextual data when the transaction is committed, when the transaction conflicts, and when the transaction is restored. First, when the transaction is committed, the transaction execution management module receives the information to be input to HBase and inputs it to HBase using CheckAndPut function. Fig. 3 shows the data input structure using the CheckAndPut function.

```
CheckAndPut(Table name, row key,
{Transaction column: (current lock status, current commit timestamp)},
{{Transaction column: (lock state to change, commit timestamp to change)},
Column name: value})
```

Fig. 3. Data input structure using CheckAndPut function.

Finally, we evaluated the performance of the multi - row transaction system developed in this paper with HAcid and Haeinsa. In the performance of read-only transactions, performance was better than that of HAcid, and performance was slightly lower than that of Haeinsa. Update transaction performance is better than HAcid and Haeinsa.

```
RollBack(Table name, row key, {Transaction column: (lock state before change,
commit timestamp before change)}, column name: value)
```

Fig. 4. Data input structure using RollBack function.

In Fig. 4, enter the lock status of the transaction column, the commit timestamp, and the value to put in the desired column. Finally, when the system is restarted after abnormally terminated, the transaction recovery information is received from the transaction recovery management module to recover the transaction and the transaction is continued in the same manner as in Fig. 3.

3.3 Transaction Manager

The transaction manager is responsible for controlling the execution of transactions using each module during transaction execution, collision, and recovery.

The transaction execution management module manages transaction execution information composed of commands such as Start(), Read(), Write(), and Commit(). Table 2 shows transaction execution information.

Table 2. Transaction execution information

Command	Explanation
Start()	The start of the transaction is reported, and the operation after the Star() is executed is included in one transaction until the Commit() command is issued.
Read(Table name, row key, column name)	It is used when reading the desired value in a row of a specific table.
Write(Table name, row key, column name, operation)	Used to record the desired operation result in a row of a specific table.
Commit()	Informs the end of the transaction, and applies all the computed values to the HBase through the data input module.

As shown in Table 2, when the Commit () command is executed, the data input module inputs data using the CheckAndPut function as shown in Fig. 3.

The following shows an example of using transaction execution information. From the Employee table, it is an example of forwarding 1,000,000 won from the Employee table to a row Pay whose key is A, while paying for rows whose row key is B.

```
1: Start();
2: PayA = Read(Employee, B, Pay);
3: Write(Employee, B, Pay, PayA - 1000000);
4: PayB = Read(Employee, A, Pay);
5: Write(Employee, A, Pay, PayB + 1000000);
6: Commit();
```

(1 line) When Start() is executed, a transaction is started, and operations up to Commit() and before are included in one transaction. (2 to 3 lines) Read B's current Pay via Read() and transfer to A via Write() and pull out 1,000,000 won. At this time, after reading the current value of P held by B using the Get() function of the data request module described above, the operation is executed. (4 to 5 lines) Read the current Pay of A via Read() and execute the operation of adding 1,000,000 won sent from B via Write(). At this time, similarly, using the Get() function of the data request module described earlier, after reading the current value of Pay held by A, the operation is executed. (6 lines) When Commit() is executed, and applies the calculation result to HBase using the CheckAndPut() function of the data transfer module described above.

The transaction conflict management module is responsible for resolving conflicts by determining whether to roll back when a transaction conflict occurs. When a collision occurs, a transaction that is late in the row is put in a wait state. If the state of the row is changed to STABLE, then the transaction is continued. If the commit timestamp is changed at this time, the commit timestamp condition of the CheckAndPut function is not satisfied and the transaction is rolled back. At this time, the transaction conflict management module sends rollback information to the data input module to rollback the transaction.

The following shows the operation procedure of the transaction conflict management module at the time of conflict of transactions.

```
1: IF state is PREWRITTEN
2: THEN WAIT
3: IF state is STABLE
4: AND IF CommitTimestamp is Unchanged
5: ROLL FORWARD
6: ELSE
7: ROLL BACK
```

(1 line) If the state of the approaching row is PREWRITTEN (2 line), the transaction accessed later will fall into the standby state. (3 line) Wait for the state of the row to change to STABLE. (4 line) If the timestamp of the commit has not been changed. (5 line) Continue the transaction. (6 line) But if the timestamp of the commit of the row changed to STABLE was changed (7 line) roll back the transaction.

Fig. 5 shows the structure of the rollback data that the transaction conflict management module sends to the data input module to roll back the transaction.

`RollBack`(Table name, row key, {Transaction column: (lock state before change, commit timestamp before change)}, column name: value)

Fig. 5. Transaction rollback data structure.

The data input module receiving the rollback information shown in Fig. 5 completes the rollback of the transaction by using the corresponding data as the argument value of the RollBack function. When the rollback is complete, it returns to the low state before the transaction occurred. The transaction recovery management module manages the transaction recovery data so that transactions for which the system has been abnormally terminated and the operation has been interrupted start work again after the system is restarted. The transaction recovery management module updates the changes every time the lock status of the transaction column changes, allowing the transaction to know what the transaction was when the system terminated abnormally. If PREWRITTEN of rows T1 through B has ended and the system terminated abnormally before PREWRITTEN in row A, the transaction recovery management module sends the value of the argument entering the CheckAndPut function to the data input module as recovery information.

Fig. 6 shows the structure of the recovery data that the transaction recovery management module sends to the data input module to recover the transaction.

```
Table name, row key,
{Transaction column: (current lock status, current commit timestamp)},
{Transaction column: (changed lock status, commit timestamp to change)}, column
name: value}
```

Fig. 6. Transaction recovery data structure.

The data input module receiving the recovery data as shown in Fig. 6 completes the transaction recovery by using the corresponding data as the argument value of CheckAndPut function.

4. Performance Evaluation

4.1 Performance Evaluation Environment

For performance evaluation, Ubuntu 14.04.2 LTS 64bit was used as the operating system. We also used Hadoop version 1.2.1 and HBase 0.94.3 version. Eclipse JUNO version was used as development tool. HBaseMaster consists of Intel i7 CPU and 8G RAM, and Hadoop Namenode consists of Intel i5 CPU and 4G RAM. In the remaining 14 clusters, HBase's regional server and Hadoop Datanode were installed and conFigd with Intel Intel i5 CPU and 2G RAM.

4.2 Performance Comparison Evaluation

4.2.1 Read-only transaction performance comparison evaluation

Read-only transaction performance comparison evaluation was performed by measuring the transaction per second (TPS) after 100,000 times of arbitrary read-only transactions in the multi-row transaction system, HAcid and Haeinsa developed in this paper. Fig. 7 shows the result of the read-only transaction performance comparison evaluation. The system developed in this paper is denoted as ‘Mine’.

As shown in Fig. 7, Mine has an average performance of 300% better than HAcid in terms of read-only transaction performance. Since HAcid does not improve performance in a certain number of clusters or more, performance increases as the number of clusters increases. On the other hand, Mine has an average performance of 5% lower than that of Haeinsa, because Haeinsa uses a method to increase the execution speed instead of applying consistency lower than Mine when performing a read-only transaction.

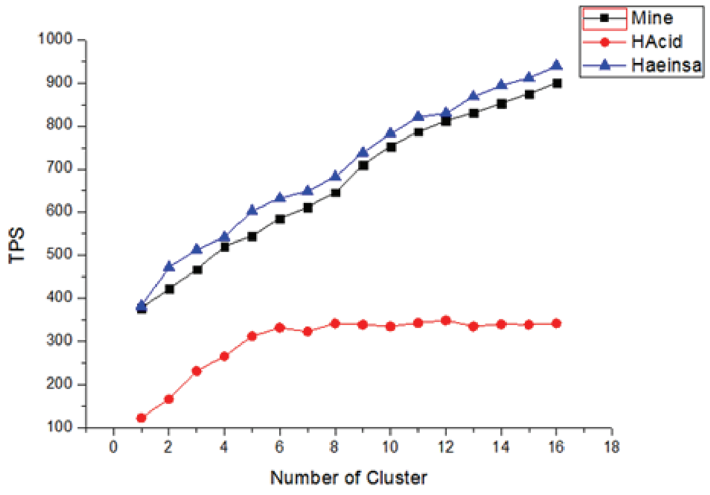

Fig. 7. Performance evaluation of read-only transactions.

4.2.2 Update transaction performance comparison assessment

Update Transaction performance comparison evaluation is performed by performing a random update transaction 100,000 times in each of the multi-row transaction system, HAcid, and Haeinsa developed in this paper, and then comparing the measured TPS. Fig. 8 shows the update transaction performance comparison evaluation result. The system developed in this paper is denoted as 'Mine'.

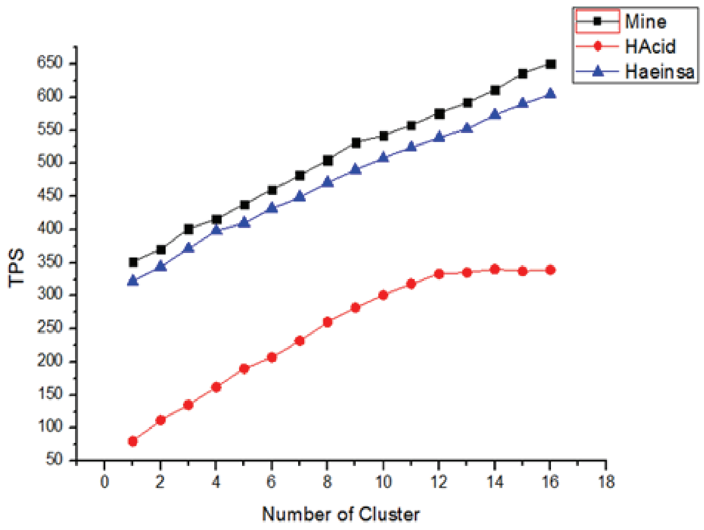

Fig. 8. Update transaction comparison performance evaluation.

As shown in Fig. 8, when comparing update transaction performance, Mine shows 440% better performance than HAcid. Since HAcid does not improve performance in a certain number of clusters or more, performance increases as the number of clusters increases. In addition, Mine shows an average of 8% better performance than Haeinsa. This is because the recovery manager manages recovery data separately, which reduces the steps required to execute multiple row transactions compared to Haeinsa.

In the read-only transaction performance evaluation, Mine showed a performance superior to HAcid, which is somewhat lower than Haeinsa, but instead of having Haeinsa achieve high performance in executing read-only transactions, Department abandoned to use a method. In the execution performance of the Update transaction, Mine showed better performance than both HAcid and Haeinsa, and the environment with higher ratio of Write operation showed better performance.

5. Conclusion

In this paper, we design and implement a multi-row transaction system for data integrity in a large data environment that requires multi-row transaction processing based on column-based distributed database HBase, which is one of the most widely used NoSQL. Existing researches have disadvantages in that there is a limit to performance improvement by managing too much transaction information in one region server, or the concurrency of multiple row transactions is reduced by managing transaction recovery information in all user columns together.

In this paper, we add a column that manages transaction information to all user tables of HBase, so there is no load on a specific region server, and it is executed through transaction manager composed of transaction execution management module, transaction conflict management module and transaction recovery management module, Collision, and recovery information, thereby improving the concurrency of multiple row transactions. In addition, through the communication manager composed of the data request module and the data input module, the data exchange between the HBase and the transaction manager is performed smoothly.

Finally, we evaluated the performance of the multi-row transaction system developed in this paper with HAcid and Haeinsa. In the performance of read-only transactions, performance was better than that of HAcid, and performance was slightly lower than that of Haeinsa. Update transaction performance is better than HAcid and Haeinsa.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2017R1A2B4011243).

References

[1] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35-40, 2010.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, NV, 2010, pp. 1-10.

[3] R. P. Padhy, M. R. Patra, and S. C. Satapathy, “RDBMS to NoSQL: reviewing some next-generation non-relational database’s,” *International Journal of Advanced Engineering Science and Technologies*, vol. 11, no. 1, pp. 15-30, 2011.

[4] GitHub Haeinsa [Online]. Available: http://github.com/VCNC/haeinsa.

[5] A. Medeiros, “HAcid: a lightweight transaction system for HBase,” M.S. dissertation, Aalto University, Helsinki, Finland, 2012.

[6] L. George, *HBase: The Definitive Guide*. Sebastopol, CA: O’Reilly Media, 2011.

[7] Apache HBase [Online]. Available: http://hbase.apache.org.

Jeong-Joon Kim http://orcid.org/0000-0002-0125-1907

He received his B.S. and M.S. in Computer Science at Konkuk University in 2003 and 2005, respectively. In 2010, he received his Ph.D. in at Konkuk University. He is currently a professor at the Department of Computer Science at Korea Polytechnic University. His research interests include Database Systems, BigData, Semantic Web, Geographic Information Systems (GIS) and Ubiquitous Sensor Network (USN), etc.