A Lump-sum Update Method as Transaction in MongoDB

Tsukasa Kudo†, Masahiko Ishino‡, Kenji Saotome*, and Nobuhiro Kataoka**

†Faculty of Comprehensive Informatics, Shizuoka Institute of Science and Technology, Japan
‡Faculty of Information and Communications, Bunkyo University, Japan
* Hosei Business School of Innovation Management, Japan
** Interprise Laboratory, Japan
kudo@cs.sist.ac.jp

Abstract - Along with the progress of the cloud computing, it became necessary to deal with various and large quantity data in the distributed database environments. So, the various NoSQL databases have been proposed and put to practical use. However, as for the NoSQL databases, since it supports the distributed environment, the integrity of the database update is basically guaranteed only by the object unit. Therefore, there are serious restrictions to update the plural objects as a transaction. On the other hand, it is often necessary to perform the lump-sum or long-time update as a transaction in business systems. In this paper, we propose a method to update plural objects in a lump-sum as a transaction in MongoDB, which is a kind of the NoSQL database. Furthermore, through the evaluations by a prototype, we confirmed that the lump-sum update can be executed as a transaction without the latency of the concurrent object unit update.

Keywords: database, NoSQL database, MongoDB, transaction processing, batch processing, concurrency control

1 INTRODUCTION

Nowadays, a large amount of data has been published, and it is utilized in various fields as big data. As a feature of big data, Volume (huge amount), Velocity (speed), Variety (wide diversity) have been pointed [8]. For example, large amounts of data, such as in the online shops and the video sharing sites, must be accessed efficiently by worldwide users, though it has more complex data structures than the conventional relational databases, including images and videos as well as texts.

To cope with this situation, various kinds of NoSQL (Not Only SQL) databases has been proposed and put to practical use [12]. As for the NoSQL database, to achieve the above-mentioned feature for the above problems, it is composed as the distributed database having a large number of servers. That is, it ensures the efficiency and reliability by redundancy such as replication and so on. Also, for example, MongoDB, which is a kind of the NoSQL database, is the document-oriented database and its structure is not defined by the schema. So, it is possible to add necessary attributes to its each data at any time and to manipulate various kinds of data flexibly [1].

On the other hand, unlike the relational database management system (hereinafter, "RDBMS"), it is not guaranteed to maintain the ACID property of the transaction processing in the case of the plural data manipulation. That is, it is generally maintained only on the individual update units called the atomic object. Also, as for the distributed environment, only the eventual consistency is guaranteed, that is, its consistency is not maintained until the completion of all the data manipulation including such as the synchronization of the replication [13]. They cause the serious restrictions on the data manipulation, for example, the intermediate result of the update having no consistency can be queried.

Here, even as for the RDBMS, there is a problem about maintaining the ACID properties in the case to update a large amount of data associated mutually in a lump-sum. That is, since the lock method is used to perform above-mentioned data manipulation concurrently with the other update, it causes the long latency of the latter update. For this problem, we have proposed the temporal update method using the transaction time database that manages the history of the time series of the data, and shown that it is possible to maintain the ACID properties without this long latency even in the above-mentioned case [6]. In this method, each update result of plural transactions is saved, and only the valid results are queried after the update completion. So, it is expected that we can update the plural data in MongoDB as the single transaction by the method of applying the temporal update method.

In this paper, firstly, we propose an update method for MongoDB, which utilizes the concept of the temporal update. And, our goal in this paper is to show that the efficient lump-sum update maintaining the ACID properties can be realized even in the above-mentioned case. In other words, by this method, we can update the plural data in a lump-sum as the single transaction, which was difficult to be executed by the conventional method of MongoDB.

The remainder of this paper is organized as follows. In Section 2, we show the problem of MongoDB about the concurrency control, and the abstract of the temporal update. In Section 3, we propose the lump-sum update method for MongoDB. In Section 4, we show the implementation and evaluations of this method, and show the considerations about this evaluation results in Section 5.

2 CONCURRENCY CONTROL OF MONGODB AND TEMPORAL UPDATE

2.1 Target Case of Data Update Process

Currently, many databases of mission-critical business systems are built by the RDBMS, and a lump-sum updates of a large amount of data spanning a long period of time are often performed. For example, in the banking systems, there

is a large amount of account transfer business, which is entrusted by the card companies and so on. Meanwhile, users update the database immediately for their deposits and withdrawals by the ATM. These processes are not only performed simultaneously by many users, but also the users in the latter case are sensitive to the delay in the response of the system. Furthermore, at present, these processes are provided as the nonstop services. That is, the both of these processing have to be executed concurrently. However, since the lock method is generally used to maintain the ACID properties of the transactions, there is the problem that the users are often kept waiting for a long while by the update of large amount of data.

Here, the ACID properties are the properties that the transactions should maintain, and it is composed by the following four properties [2].

Atomicity Transactions execute all or nothing.

Consistency Transactions transform a correct state of the database into another correct state.

Isolation Transactions are isolated from one another.

Durability Once a transaction commits, its updates persist in the database, even if there is a subsequent system crash.

For the above-mentioned problem due to the lock method, the mini-batch is used widely to shorten the wait time of users, by which the process for a large amount of data update is divided into several update processes and they are performed one after another. So, the long wait does not occur.

Meanwhile, we showed that the ACID properties cannot be maintained by the mini-batch in the case where the data associated mutually. Furthermore, we proposed the temporal update method to update the data with maintaining the ACID properties even in this case [5]. On the other hand, as well as the mini-batch, the ACID properties of the transactions are maintained only on the update of the atomic object in MongoDB. So, in the case of updating plural data in a lump-sum, it has been pointed out that the same problems as the mini-batch occurs [1].

That is, the aim of this paper is to propose the update method for MongoDB, in which we apply the concept of the temporal update method, to update plural data in a lump-sum with maintaining the ACID property. In this paper, "online entry" represents the update on the atomic object, in which the ACID property is maintained in MongoDB; "batch update" represents the update composed of plural data manipulation in a lump-sum, in which the ACID property is not maintained to the contrary. Incidentally, as for the relational databases, the former corresponds to the update of the single transaction; the latter corresponds to the update composed of the plural transactions such as the mini-batch.

In this section, we show these related works below: first, the overview of MongoDB, and the issue intended by this paper; second, the temporal update method for the relational databases.

```
{ "_id" : 1, "name" : { "first" : "Tsukasa", "last" : "Kudo" },
"address" : "Hukuroi-shi, Shizuoka"}
```

Figure 1: Composition of MongoDB document.

2.2 Overview of MongoDB and Issue in Concurrency Control

MongoDB is a kind of document-oriented NoSQL database, which data is the documents expressed by JSON (JavaScript Object Notation) format shown in Fig. 1 [15]. The document is composed of the fields. For example, in this figure, {"_id": 1} is a field, of which identifier is "_id" and value is 1. Here, "_id" corresponds to the primary key of the relational database. And, the field is able to have a nested structure. For example, the name field (name) in the figure is composed of the following fields: the first name field (first) and the last name field (last). Since the document structure of MongoDB is not defined by schema, any necessary fields can be added to any document at any time. So, each document is able to have different fields except "_id". Furthermore, since it is possible to store the various kinds of objects, such as images and videos to its fields, it can handle a variety of data compared to the RDBMS. Here, the set of documents is called the collection. So, the collection and document correspond to the table and record in the relational database, though it is not strict.

As for the data manipulations, the following CRUD operations are provided as well as the RDBMS: insert, find (corresponding to select), update and remove (corresponding to delete). Furthermore, since findAndModify command is also provided to execute both of the query and update exclusively, they can be executed as the atomic operation. That is, the update of the atomic object such as the single document can be performed as the single transaction.

However, unlike SQL of the RDBMS, it does not provide the command to update the plural documents as the single transaction. That is, there is a problem that the ACID properties of the transaction, especially the isolation and atomicity, cannot be maintained in the case where the plural documents are updated in a lump-sum.

For this issue, two phase commit protocol is shown [10]. In this method, for example, in the case of performing the account transfer from the account X to the account Y of a bank, its processing ID is saved in the document of the account transfer management collection, which has the status about this processing: initial, pending, applied and done. These accounts are updated one after another in pending; meanwhile, this processing ID is saved to the documents of these accounts, and the updating accounts can be managed. Then, the status transit to applied. In the case of successful completion, this processing ID is deleted from the documents to exit. And, in the case of abnormal termination, the compensation transaction is performed to cancel the updates of the accounts and recover to a consistent state [10].

It is considered that this method is same as the saga for the RDBMS [9], which executes a mass update sequentially as divided plural update set. And, in the case of failure, the compensation transactions are executed to recover the data.

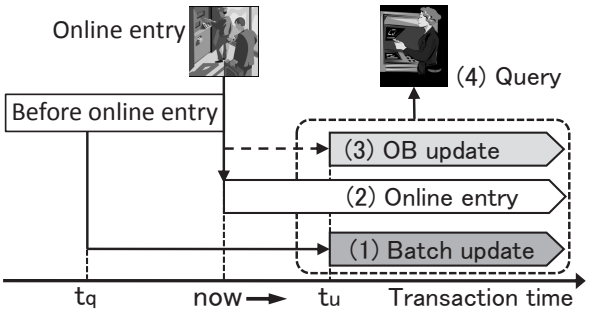

Figure 2: Data manipulation by temporal update.

On the other hand, it has been shown that the ACID properties of the transaction is not maintained in the case of the concurrent execution with the other transactions [3]. For example, in the case where the failure occurs in the transaction after the account X was updated and the result was queried by the other transaction, the former transaction must be canceled. However, since the result of this transaction has been already queried by other transactions, it causes the problem in the actual system operations, such as the cascading aborts. This is due to the fact that the entire update cannot be processed as the single transaction in the same as the mini-batch.

2.3 Temporal Update Method for Relational Database

In Section 2.1, we mentioned that we proposed the temporal update method for the problem as for the mini-batch in the RDBMS. In Fig. 2, we show the data change of the time series about the transaction time in this method [7]. The concept of the temporal update method is the following: all the update results of each kind of transactions are saved, in which the online entries and batch update are included; and only the valid data is queried [5], [6].

This method utilizes the concept of the transaction time database, which is a kind of temporal database that manages the time history of its data [14]. And, the transaction time expresses when some fact existed in the database, so its relation is expressed by $R(K, T_a, T_d, A)$. Here, K shows the primary key attribute of the data of above-mentioned fact; T_a shows the transaction time when the data was inserted into the database; T_d shows the transaction time when it was deleted from the database; A shows the other attribute. In other words, though the data is deleted logically from the database by setting the deletion time to T_d, it remains physically in the database. So, all the CRUD operations on the data of the database can be managed as a history of the time series. Here, until the data is deleted, the value now is set to T_d, which indicates the current time [16].

The feature of this method is that we avoid the conflicts between the batch updates and online entries by expanding the concept of the transaction time into the future. That is, as for the batch update, the data at the past time t_q is queried, and the processing result is stored at the future time t_u. On the other hand, as for the online entries, the data at the current time now is manipulated. Thus, the conflict between the both processing can be avoided without using the long locks.

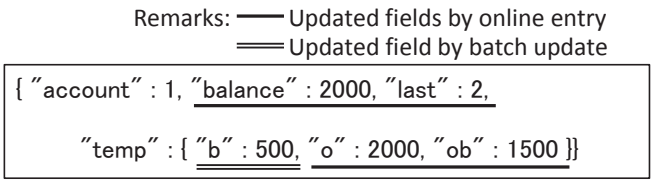

Figure 3: Data structure of document in proposal method.

Here, the batch update processing must be applied to the result of the online entries performed between the time t_q and t_u. So, the batch update processing is applied individually to the results of the online entries, and these processing results are stored into the database as the OB update in Fig. 2.

As a result, three kinds of update result data is stored at the time t_u as shown in Fig. 2: (1) the batch update, (2) the online entry and (3) the OB update. Therefore, the valid data has to be queried by the query processing, which is shown by (4) query. It is achieved by querying these data in the following order of priority: the OB update, online entry and batch update. To be concrete, in the case where both of the batch update and online entry are executed, the OB update result is queried; in the case of only the batch update, its result is queried; in the case of only the online entry, its result is queried; Therefore, we can query the same update result as in the case where the batch update is executed on the online entry results at the time t_u, without the long latency of the online entry by the lock method.

Moreover, to apply this method to the distributed database environment, we improved it not to have to determine the completion time t_u beforehand [7]. It was implemented by the view, which has the feature that the above-mentioned valid data is changed at the batch update completion time t_u.

3 PROPOSAL OF LUMP-SUM UPDATE METHOD FOR MONGODB

In this section, we propose a lump-sum update method for MongoDB, which is based on the concept of the temporal update method mentioned in Section 2.3. First of all, since there is the restriction about the transaction processing in MongDB as mentioned in Section 2.2, we have to adopt the method to fit the characteristics of MongoDB.

Concretely, since the plural document cannot be updated as a transaction in MongoDB, the following modification is necessary. First, all the update result must be stored in the single document: the online entry, batch update and OB update. So, the data structure must be modified. Second, for the same reason, the transaction time database cannot be constructed. So, the update process must be managed according to the processing stage, instead of the transaction time.

3.1 Data Structure of Document in Proposed Method

If we applied the temporal update intended for the RDBMS to the lump-sum update in MongoDB, multiple documents of the update result would be created for one fact of the real world: by the online entry, batch update and OB update. It

means, for example, in order to execute the OB update with the online entry, the two documents represented by (2) and (3) in Fig. 2 must be manipulated as one transaction. However, as mentioned above, plural documents cannot be updated as one transaction in MongoDB. So, if we applied this method to MongoDB just as for the RDBMS, the problem would occur: the consistency among data is not maintained. Meanwhile, since the document structure of MongoDB is not defined by the schema, its fields can be added flexibly. And, more importantly, the data manipulations on the different fields of the same document do not conflict.

For this reason, in this study, we propose the following lump-sum update method for MongoDB as shown in Fig. 3. In this method, all the results of the online entry, batch update and OB update are saved in the same document, and each field is indicated by "o", "b" and "OB". And, the valid field is queried in the same way as the temporal update for the RDBMS.

Here, as shown in Section 2.1, we defined the online entry as the process updating each document individually. And, it can be executed by the transaction feature of MongoDB. Furthermore, OB update, which accompanies with the online entry, updates the same data as the online entry. So, it can be executed by the transaction feature, though both of the OB update and online entry cannot be executed by the transaction feature in a lump-sum. On the contrary, since the batch update updates plural documents in a lump-sum, it cannot be executed by the transaction feature as the whole process.

For example, Fig. 3 shows the document of the balance of the deposit account: the account number (account), balance (balance) and update number (last). In addition, in the following, we omit to write "_id" of documents. Here, "last" corresponds to the time stamp, and it is increased by one for each update of the balance. In addition, it is used for the optimistic concurrency control as described later. Also, "temp" is the temporal field that is added temporarily during the execution of the batch update, and the update results are inserted to the corresponding field with the processing classification: the field of identifier "b" is for the batch update; "o" is for the online entry; "OB" is for the OB update. In addition, "balance" and "last" fields are also updated by the online entry.

As shown in Fig. 3 by the underline and double underline, the online entry, including the OB update, updates the different fields from the batch update field. So, there is no conflict between these update processing. Then, the valid update result can be queried similarly to the temporal update method, that is, by querying the update result data with the following priority: the OB update, online entry and batch update.

In summary, as for the temporal update method in RDBMS, we avoid the conflicts between the batch update and online entry by storing the each update results to the different records. In this method for MongoDB, we avoid this conflicts by storing each update result to the different fields.

3.2 Transition of Processing Stage in Proposed Method

In MongoDB, since plural documents cannot be updated as the single transaction, the transaction time database also cannot be composed. For example, in Fig. 2, two data is manipulated to update one fact at time now: the data "Before online entry" is logically deleted, and the update result "(2) Online entry" is added. In other words, the transaction time database needs to manipulate two data as a single transaction.

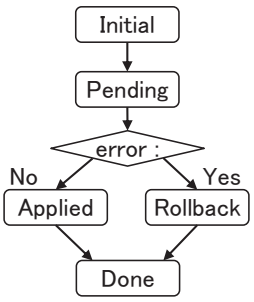

Figure 4: Processing stage transition in update

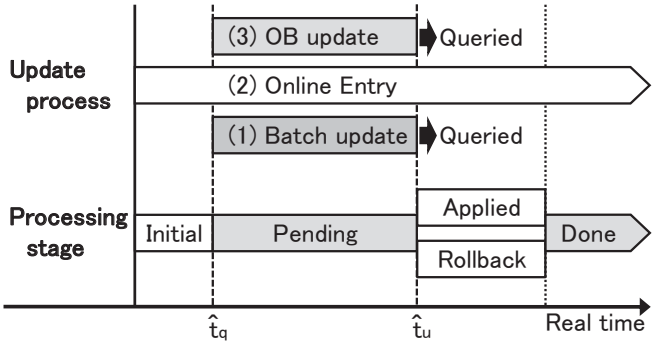

Figure 5: Correspondence between update process and processing stage

On the other hand, as for the temporal update in RDBMS, as shown in Fig. 2, since the batch update starts at the time t_q and completes at t_u, the OB update must be executed during this time period, which accompanies with the online entry. Furthermore, the batch update is performed to the data of the transaction time t_q. That is, these control is required for the proposed method, too.

To address this issue, we define the following four processing stage like the two phase commit protocol in MongoDB, which was shown in Section 2.2, as shown in Fig. 4. And, the update processing is performed with transitioning among them sequentially: "initial" shows that the stage is before batch update; "pending" shows it is during the batch update; "applied" shows batch update has completed, and the data of temporal field is being reflected to the regular field; "done" shows all the processing has completed. Incidentally, in the case where the failures occur in the batch update processing, the processing stage transitions from "pending" to "rollback". In the "rollback" stage, the batch update results are canceled, and the processing stage transitions to "done".

We show the correspondence between the update process and processing stage as for the "real" time in Fig. 5. Hereinafter, we use real time $\hat{t_q}$ corresponds to t_q in Fig. 2, and $\hat{t_u}$ corresponds to t_u. At the transition time $\hat{t_q}$ from "initial" to "pending", the batch update and OB update start. And, at $\hat{t_q}$ the time from "pending" to "applied", the both complete.

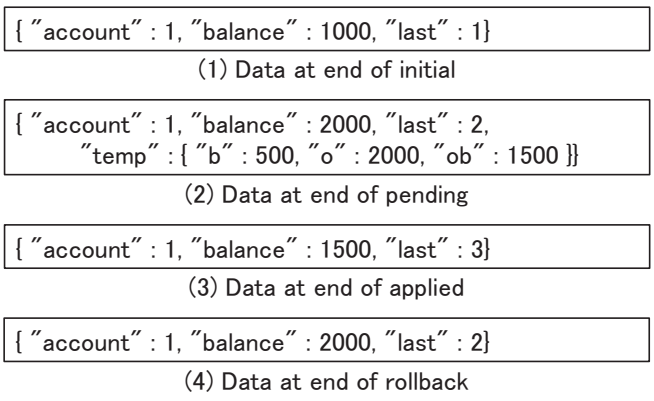

Figure 6: Data change in update

Here, as mentioned above, since MongoDB is not the transaction time database, the batch update cannot query the data history at the time $\hat{t_q}$. So, in the case where the target data is updated by the online entry before the batch update, this batch update must perform on this update result. However, the result of the OB update, which accompanies with the online entry, reflects both of the batch update and online entry results, and finally this is queried based on the query priority. So, the query result of the proposed method is same as the result of the temporal update method.

Here, the query data as of the proposed method is decided at the time $\hat{t_u}$, and the query results do not change after $\hat{t_u}$. That is, as shown in Fig. 5, the batch update and OB update result can be also queried corresponding on the query priority. The transition from "applied" and "rollback" to "done" means only the delete completion of the unnecessary intermediate results.

Figure 6 shows the data at the end of each processing stage. (1) shows the data at the end of "initial". Since it is prior to batch updates, "temp" field does not exist. Also, (2) shows "pending". Since the batch update has completed, the data has been set to temp field. In the case of this figure, the batch update debited 500 from the account. Meanwhile, the online entry deposited 1000 to the same account, and OB update debited 500 from this result. Then, all the results were stored in the temp field. At this time, "balance" and "last" fields have been also updated by the online entry. Incidentally, since the value is set only to the fields corresponding to the executed updates, all the fields of temp field are not always set.

While the processing stage is "applied", the valid data is queried by the online entry transactions. In the case of this figure, the balance of 1500 in "ob" field is queried. Furthermore, in this stage, the valid data is reflected into balance field, then temp field is deleted. This is the processing for the next batch update. At the end of this stage, each field has the value shown in (3), and 1500 is set to balance field, which is the result of the OB update. In this way, the query results of the online entry do not change through this stage.

On the other hand, in the case where the batch update processing fails, the processing stage transitions to "rollback". In this stage, only balance field is queried by the online entry continually; temp field is ignored. And, temp field is deleted without affecting the online entry. So, when the rollback has completed, balance field is not changed and this document become the state shown in (4).

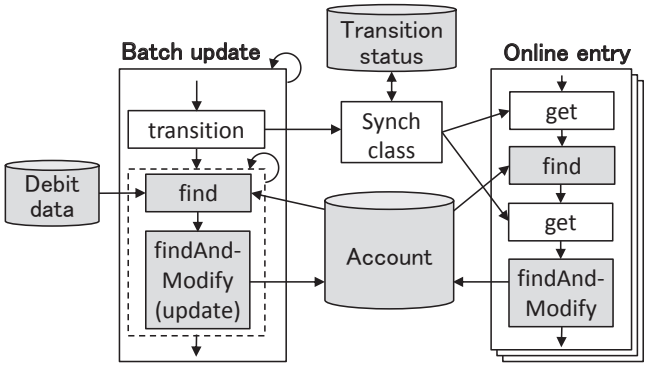

Figure 7: Software structure of prototype

In this way, the processing stage transitions to "done", and we get the result (3) in the case of successful completion; we get (4) in the case of abnormal termination.

4 IMPLEMENTATION AND EVALUATIONS

4.1 Implementation of Prototype

To evaluate the proposed method, we constructed a prototype intending to manipulate the deposit accounts of the banking system. We use MongoDB Ver. 2.6.7 for the database; Java Ver. 1.6 for the programming language; MongoDB Java Driver Ver. 2.13 to access MongoDB from Java [11]. In addition, OS is Windows 7 (64bit). Figure 7 shows its software construction. The batch update and online entry programs are implemented by Thread class of Java to execute the both concurrently. Each program executes the following processes as shown in this figure: it query the data of the deposit account from the database (find); then, it updates the data of the database (findAndModify, update).

The batch update program executes the processing to debit from the deposit account collection (Account) in a lump-sum, based on the account and amount information stored in the debit data collection (Debit data). As shown in Fig. 4, the processing stage transitions from "initial" to "pending". This process is executed at the first (transition) of the batch update program (Batch update), then the batch update is executed. After its completion, the processing stage transitions to "applied", and the data in temp field is reflected into balance field. Incidentally, in the case of abnormal termination, it transitions to "rollback". Finally the processing stage transitions to "done". The information of the processing stage is stored in the transition status collection (Transition status), and it is accessed through "Synch class" by the batch update and online entry programs.

Meanwhile, the online entry executes the processing to deposit to each deposit account individually. As for this prototype, it was configured to perform deposits of certain amount of money from the plural terminals concurrently. Here, the online entry has to be accompanied by the OB update during the processing stage of the batch update is "applied". So,

its program was configured to query the processing stage by Synch class (get). And, to query this data efficiently by the program without accessing the database, it is saved in the instance of Sync class, However, in the case where the processing stage transitions from one stage to the next stage during the online entry executing, there is the possibility of the incorrect OB update execution. In other words, in the case where the transition occurs between the "find" and "findAndModify" in Fig. 7, there may be the unnecessary OB update execution or the lack of it.

For this issue, We implemented Synch class using Synchronized keyword of Java, by which only one program can call it at the same time by the synchronization control. Then, we configured the online entry program to query the processing stage before not only "find" but also "findAndModify" as shown by the "get" in Fig. 7. And, in the case where the transition occurs between them, the online entry program performs a retry. Furthermore, in order to prevent the transition between "get" prior to findAndModify and the completion of findAndModify, we configured Synch class to wait a certain time before transition, which is requested by the batch update program. That is, the executing update of the online entry program can be completed before the transition by this way. Incidentally, while the processing stage is "applied", not only "balance" field but also the valid field has to be queried from this document. We implemented a class to manipulate the fields, and these manipulations were implemented by using the method of this class.

Since the online entries are executed from plural terminals concurrently, it is necessary to execute the concurrency control. So, we implemented the optimistic concurrency control by "last" field (update number) using findAndModify command, which is a method to perform the query and update of a document at the same time exclusively as mentioned in Section 2.2. And, in the case where the query condition matches to no data, the update is not performed and $null$ is returned as the query result. Therefore, we set the query condition of findAndModify command {"account":account number, "last": read updated number by "find" }, that is, the value of "last" is the result queried by find command just before. As a result, in the case where the target document was updated by the other program after the execution of this "find" command, no data matches this condition. And, in this case, the online entry program has to retry these processes from the beginning.

Table 1 shows the target fields at each processing stage, which is queried and updated. As for the batch update, it is not executed when the processing stage is "initial" or "done"; it updates the different fields from the online entry when the processing stage is "pending" or "rollback". So, there is no conflict between the batch update and online entry. However, when the processing stage is "applied", the both update the same fields: "balance", "last" and "temp". That is, there is the conflict between them. Therefore, as for the batch update program, we also implemented the optimistic concurrency control using findAndModify command similarly to the online entry program. Incidentally, the batch update program queries "balance" when the processing stage is "pending", which is updated by the online entry program at the same time. That

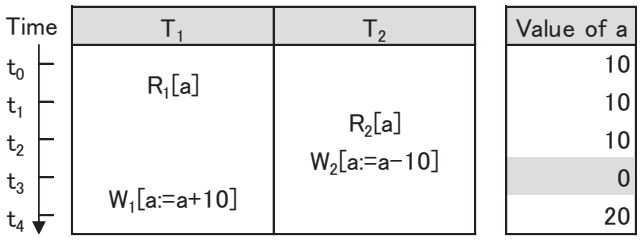

Figure 8: Lost update example of transaction

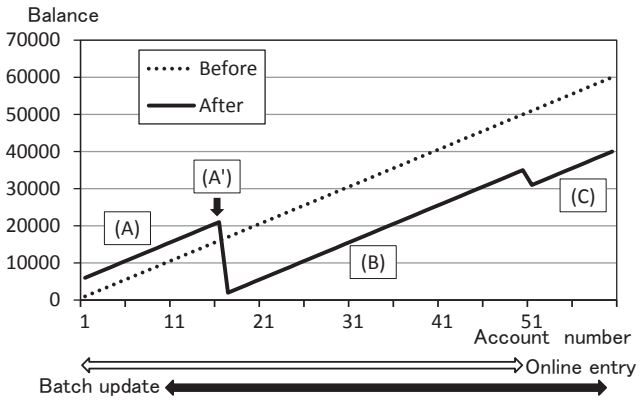

Figure 9: Result of case of successful completion

is, there is conflict between them. However, as we already mentioned in Section 3, the case where the online entry is executed, the OB update result becomes valid, which is created based on the execution result of the former. In other words, the batch update result is not used. Therefore, the concurrency control for this query is not required.

4.2 Evaluations of Concurrency Control

The proposal method does not lock the target documents through the duration of the batch update. That is, similar to the relational database, it has to be confirmed the inconsistencies by the concurrent execution of transactions do not occur. So, we performed the following three kinds of experiments to evaluate the concurrency control between the batch updates and online entry, using the prototype shown in Fig. 7.

First, we performed the experiment in the case of successful completion of the batch update. The purpose of this experiment is to confirm that there is no lost update occurred by the illegal interface between the batch updates and online entries. Figure 8 shows the example of the lost update, in which the time series manipulations on the data a are executed by the transaction T_1 and T_2: R_i indicates the query; W_i indicates the update. And, the column "Value of a" shows the value of a in the time series. As for the value of a, T_1 queries it by R_1 and updates by W_1; meanwhile, T_2 updates it by W_2. So, since the update result of T_2 is overwritten by T_1, it is lost. That is, the lost update has occurred.

In this experiment, as shown in Fig. 9, the number of the target deposit account is 60. And, its balance data is set prior to the experiment, which is calculated by the following equation as shown by the broken line.

$$balance = account\ number \times 1000 \qquad (1)$$

Table 1: Read and write fields in each processing stage

Processing stage	Batch update		Online entry	
	find	findAndModify	find	findAndModify
Initial	—	—	balance, last	balance, last
Pending	balance	temp.b	balance, last	balance, last, temp.o, temp.ob
Applied	last, temp	balance, last, temp (delete)	balance, last, temp	balance, last, temp (delete)
Rollback	—	temp (delete)	balance, last	balance, last
Done	—	—	balance, last	balance, last

{ "account" : 15, "balance" : 18000, "last" : 3 }
{ "account" : 16, "balance" : 19000, "last" : 3 }
{ "account" : 17, "balance" : 20000, "last" : 3,
"temp" : { "o" : 20000, "ob" : 0 } }
{ "account" : 18, "balance" : 21000, "last" : 3,
"temp" : { "o" : 21000, "ob" : 1000 } }
{ "account" : 19, "balance" : 22000, "last" : 3,
"temp" : { "b" : 0, "o" : 22000, "ob" : 2000 } }
{ "account" : 20, "balance" : 23000, "last" : 3,
"temp" : { "b" : 1000, "o" : 23000, "ob" : 3000 } }

Figure 10: Data at end of pending

Here, the horizontal axis shows the account number of the deposit account; the vertical axis shows its balance. Then, the batch update program debits 20000 from the deposit accounts which account number is between 11 and 60. Here, the account, which balance is less than 20000 at this debiting time, is excluded from this processing. In this experiment, since the batch update is successfully completed, the processing stage shown in Table 1 transitions from "pending" to "applied".

Meanwhile, the online entries are executed from five terminals concurrently, and each entry deposits 1000 to the deposit account which account number is between 1 to 50. Here, in order to avoid the conflict among the online entries, the different first update account number is assigned to each terminal: 1, 11, 21, 31, 41. Then, Each terminal updates the deposit account one after another. Here, after the program has processed the account which account number is 50, it processes the account which account number is 1. In this way, 50 deposit accounts are updated from each terminal.

The solid line in Fig. 9 shows this experimental result. The account indicated by (A) is not the target of the batch update or its balance was less than 20000. So, the batch update did not debit from it, and only the online entries deposited 5000. The account indicated by (B) is the target of the batch update, and the batch update or OB update debited 20000. Also, the online entries deposited 5000. So, the balance became 15000 reduction. The account indicated by (C) is not the target of the online entry, and only the batch update debited 20000. That is, even in the case where the batch update was executed concurrently with the online entry, no lost update occurs in both of the processes. As a result, we got the consistent update result.

Second, to investigate the change point from (A) to (B) in

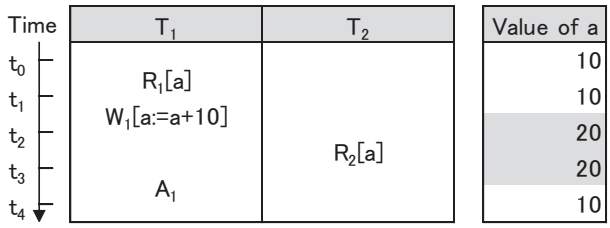

Figure 11: Dirty read example of transaction

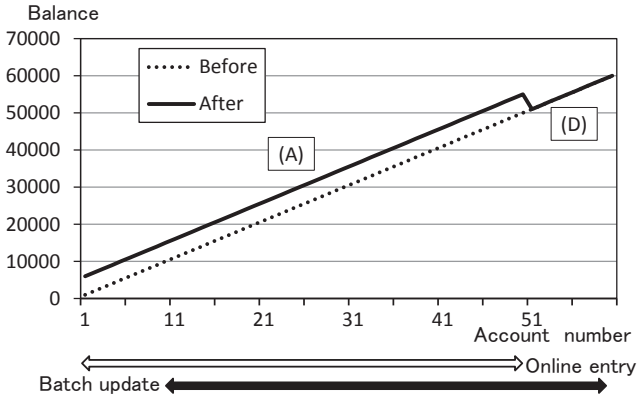

Figure 12: Result of rollback of batch update

Fig. 9, which is shown by (A'), we performed the experiment, in which both of the online entry and batch update were interrupted when the processing stage transitioned to "applied". Incidentally, the other experimental environment is the same as the first experiment. Figure 10 shows the query results of the deposit account data at the end of the experiment, which is in the vicinity of (A').

Since the intermediate results of "temp" field at this time remained, the following data was queried. As for the account number 15 and 16, since the balance was less than 20000, neither of the batch update and OB update were performed. As for 17 and 18, since the balance was less than 20000 when the batch update was performed, the batch update was not performed. However, since the online entries deposit after this time, the balance exceeded 20000 and the OB update was performed. Lastly, as for 19 and 20, both of the batch update and OB update were performed. Incidentally, in the first experiment result, since the online entries were continued even after the transition to "applied", every balance of deposit accounts are grater than 0.

Third, we experimented the case of the abnormal termina-

tion of the batch update, and the processing stage was transitioned from "pending" to "rollback", which is shown in Table 1. The purpose of this experiment is to confirm that the dirty read of the online entry does not occur, even in the case of the abort and rollback of the batch update. Figure 11 shows the example of dirty read, and the representations are same as Fig. 8, and A_1 shows the abort of T_1. Though T_1 was aborted after updating a, T_2 had already queried this updated data. That is, since T_2 was executed using the data, which did not actually exist, the consistency of its result was not maintained.

We show the result of the third experiment in Fig. 12. Similar to Fig. 9, the broken line shows the balance data at the beginning of this experiment; the solid line shows the data at the end of this experiment. The former is the same as in the first experiment. In this experiment, the processing stage transitioned from "pending " to "rollback" to execute the rollback of the batch update, then "temp" field was deleted. As a result, as for the balance data in the range of (A), only the deposit of 5000 was executed by the online entries. On the contrary, the balance data does not change in the range of (D), which is outside of the online entry. Therefore, there was no dirty read of the online entries, and the consistency of the result was maintained. Therefore, the batch update could be canceled without affecting the online entry.

5 CONSIDERATIONS

First, we consider whether the ACID properties of the transaction are maintained by the proposal method. As for the atomicity, the batch update completes as either of the following state: its update results are queried after the processing stage transits to "applied" as shown in Fig. 9; it is canceled in the stage of "rollback" without affecting the online entry as shown in Fig. 12. Therefore, the consistency was also maintained, that is, the collection transitions from a consistent state to another consistent state.

Next, as for the isolation, the batch update updates the different fields from the online entry, and the intermediate results of each processing are not queried by the other when the processing stage is "pending" as shown in Fig. 3. Furthermore, in the both case of the successful completion and rollback, the batch update could be executed without affecting the online entries. So, the isolation is maintained. Lastly, as for the durability, the integration processing of the online entry and batch update results is executed when the processing stage is "applied". However, since the update results have been already reflected into the database, the durability is maintained by database management system of MongoDB. And, the query results do not change even if this process is interrupted.

Thus, the ACID properties of the transaction can be maintained by this method in MongoDB, even if the batch update is executed concurrently with the online entries. As shown in Section 3, this batch update corresponds to updating plural documents in a lump-sum. In other words, the update of plural documents in MongoDB can be executed as a transaction concurrently with the update of individual document.

Second, we consider the efficiency, which is the latency on the online entries by the batch update. As shown in Table 1, the batch update and online entry update the different fields from each other while the processing state is not "applied". So, the latency does not occur by the concurrency control for the conflict. On the other hand, since the both update the same fields, "balance" and "last", while the processing stage is "applied", the concurrency control is needed for these conflicts. Here, the concurrency control is executed by the optimistic concurrency control for not only the online entries but also each individual update of the batch update. Therefore, it can be executed without a long latency, such as waiting for the completion of the entire batch update.

In addition, when the processing stage transitions to "applied", the batch update itself was already completed and the online entry is executed using a valid data reflecting the batch and OB update results. So, for example, in the case where the batch update is not executed often, its processing in "applied" can be wait to execute until the frequency of the online entry becomes less comparatively. Incidentally, in this experiment, the delays of certain period of time were put in Sync class in order to prevent the transition of the processing stage during the updating of the online entry, by a simple way. And, this causes the latency of the other online entries. As for this issue, we consider it can be shortened by the immediate transition after the completion of this online entry.

Third, through our experiments, we found that the different concurrency control method from the relational database can be applied to MongoDB, which is a kind of document oriented NoSQL database. As for the relational database, the row lock method is generally used, and a key-range lock method supports the concurrency control for the wide range of rows of tables [3]. On the other hand, it is controlled as an update of the entire row even when a part of the row is updated. As for MongoDB, in contrast, although there is a restriction of the concurrency control to update plural documents collectively, there is no conflict among the update of the different fields. This shows, while the long-lived transaction is updating the large-capacity field such as videos, the other fields can be updated by the other transactions concurrently. In other words, though there is little necessity for the concurrency control in the NoSQL database now, we consider the different update model from the relational database will be necessary according with the spread of its application fields.

Lastly, the update of plural documents in a lump-sum is often executed in the actual business system operations. In particular, the case of the experiment in this paper is taken as a typical example [3]. On the other hand, it had been the problem in the MongoDB and other NoSQL databases. With the expansion of the application fields of the NoSQL databases, it is considered that the request for the data manipulation like this shall increase, which the lump-sum update is executed as the single transaction as well as the relational databases. So, we consider that this method is valid for such a data manipulation.

6 CONCLUSION

Recent years, the utilizations of the NoSQL databases are spreading. However, there is the problem that the plural objects cannot be updated with maintaining the ACID properties of transactions. In this paper, we proposed the lump-sum

update method for MongoDB, which is a kind of NoSQL database. This method is based on the concept of the temporal update method to execute the batch update as the single transaction in the relational databases. Concretely, in this method, the results of the following update are stored in temporal fields of the document and only the valid data is queried: the batch updates; the online entry; the OB update, which is applied the batch update individually to the online entry result.

And, we showed that the plural documents can be updated as the single transaction in MongoDB by this method, even while the documents are being updated concurrently by the other transactions, that is, the online entries. Furthermore, we confirmed that this method achieves the above-mentioned function through the experiments using a prototype, which intended the deposit account.

Meanwhile, in the actual business systems, large number of transactions which update the plural data are executed concurrently. So, the future study will be focused on the concurrency control of such a update in the NoSQL databases.

ACKNOWLEDGEMNTS

This work was supported by JSPS KAKENHI Grant Number 15K00161. Also, the motivation of this study is an extension of our method to the NoSQL database, which has aimed to execute the batch update as a transaction in the relational databases and has been registered as a patent [4]. We would like to appreciate the members of Mitsubishi Electric Information Systems Corp. who supported to get this patent.

REFERENCES

[1] K. Banker, MongoDB in Action, Manning Pubns Co. (2011).
[2] C.J. Date, An Introduction to Database Systems, Addison-Wesley (2003).
[3] J. Gray, A. Reuter, Transaction Processing: Concept and Techniques, San Francisco: Morgan Kaufmann (1992).
[4] Mitsubishi Electric Information Systems Corp., T. Kudo, Database System, Japan patent 4396988 (2009).
[5] T. Kudo, Y. Takeda, M. Ishino, K. Saotome, and N. Kataoka, "Evaluation of Lump-sum Update Methods for Nonstop Service System," Int. J. of Informatics Society, Vol. 5, No. 1, pp. 21–28 (2013).
[6] T. Kudo, Y. Takeda, M. Ishino, K. Saotome, and N. Kataoka, "An implementation of concurrency control between batch update and online entries," Procedia Computer Science, Vol. 35, pp. 1625–1634 (2014).
[7] T. Kudo, Y. Takeda, M. Ishino, K. Saotome, and N. Kataoka, "Application of a Lump-sum Update Method to Distributed Database," Int. J. of Informatics Society, Vol. 6, No. 1, pp. 11–18 (2014).
[8] D. Laney, "3D Data Management: Controlling Data Volume, Velocity and Variety," META Group (2012) http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.
[9] H. Garcia-Molina, and K. Salem, "SAGAS," Proc. the 1987 ACM SIGMOD Int. Conf. on Management of data, pp. 249–259 (1987).
[10] MongoDB Inc., The MongoDB 3.0 Manual, http://docs.mongodb.org/manual/.
[11] MongoDB Inc., MongoDB API Documentation for Java, http://api.mongodb.org/java/.
[12] E. Redmond, and J.R. Wilson, Seven Databases in Seven Weeks: A guide to Modern Databases and The NoSQL Movement, Pragmatic Bookshelf (2012).
[13] P.J. Sadalage, and M. Fowler, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Addison-Wesley Professional (2012).
[14] R. Snodgrass and I. Ahn, "Temporal Databases," IEEE COMPUTER, Vol. 19, No. 9, pp. 35–42 (1986).
[15] S.S. Sriparasa, JavaScript and JESON Essentials, Packt Pub. Ltd. (2013).
[16] B. Stantic, J. Thornton and A. Sattar, "A Novel Approach to Model NOW in Temporal Databases," Proc. 10th Int. Symposium on Temporal Representation and Reasoning and Fourth Int. Conf. on Temporal Logic, pp. 174–180 (2003).

(Received September 29, 2015)
(Revised December 8, 2015)

Tsukasa Kudo received the M.E. from Hokkaido University in 1980 and the Dr.Eng. in industrial science and engineering from Shizuoka University, Japan in 2008. In 1980, he joined Mitsubishi Electric Corp. He was a researcher of parallel computer architecture, an engineer of application packaged software and business information systems. Since 2010, he is a professor of Shizuoka Institute of Science and Technology. Now, his research interests include database application and software engineering. He is a member of IEIEC and Information Processing Society of Japan.

Masahiko Ishino received the master's degree in science and technology from Keio University in 1979 and received the Ph.D. degree in industrial science and engineering from graduate school of Science and technology of Shizuoka University, Japan in 2007. In 1979, he joined Mitsubishi Electric Corp. From 2009 to 2014, he was a professor of Fukui University of Technology. Since 2014, he belong to Bunkyo University. Now, His research interests include Management Information Systems, Ubiquitous Systems, Application Systems of Data-mining, and Information Security Systems. He is a member of Information Processing Society of Japan, Japan Industrial Management Association and Japan Society for Management Information.

Kenji Saotome received the B.E. from Osaka University, Japan in 1979, and the Dr.Eng. in Information Engineering from Shizuoka University, Japan in 2008. From 1979 to 2007, he was with Mitsubishi Electric Corp., Japan. Since 2004, he has been a professor of Hosei business school of innovation management. His current research areas include LDAP directory applications and single sign-on system. He is a member of the Information Processing Society of Japan.

Nobuhiro Kataoka received the master's degree in electronics from Osaka University, Japan in 1968 and the Ph.D. in information science from Tohoku University, Japan in 2000. From 1968 to 2000, he was with Mitsubishi Electric Corp. From 2000 to 2008, he was a professor of Tokai University in Japan. He is currently the president of Interprise Laboratory. His research interests include business model and modeling of information systems. He is a fellow of IEIEC.