Implementation and Evaluations of Recovery Method for Batch Update

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 - For the operation of the nonstop service systems, some methods have been put to practical use to perform the batch update concurrently with the online entries. However, the whole batch update cannot be executed as a transaction by the conventional methods. So, in the case where a transaction failure occurs in the batch update, there is the problem that the rollback of the update cannot be executed with maintaining the isolation from the online entries. For this problem, we have proposed the temporal update method, by which the batch update can be executed as a transaction. In this study, we show the consistency of the batch update result can be checked before the commit by this method, even in the case of the concurrent execution with the online entries. Furthermore, we show the following: the recovery of the transaction failure by this method can be executed without affecting to the online entries; it is more efficient than the conventional methods.

Keywords: database, batch processing, transaction, recovery, business system, nonstop service.

1 INTRODUCTION

In the actual business systems, databases are indispensable, and it is generally updated by two methods. First is the lump sum update by a great deal of data (here in after, "batch update"), that is the batch processing [5]. It is used widely in various fields: the settlement of account in the accounting systems, a great deal of account transfer for entrust company in the banking systems, and so on. Here, since its target data are a great many, the impact of the failure affects the extensive range of business. So, various kinds of mechanisms are introduced to maintain the safety of the system operations. For example, the temporary process is executed prior to the definite process. In the former, various kinds of confirmations are executed beforehand: validity of the input data, the consistency of the processing results, and so on. After this confirmations, the definite process is executed to update the business data of the database.

On the other hand, users enter their data from the many online terminals concurrently: in the accounting systems, the sales information is entered from the POS (Point of Sales) terminals; in the banking systems, the deposit and withdrawal data are entered from the ATMs (Automatic Teller Machines). We call this entry "online entry." As for the online entry, small amount of data are entered at each time, and they are reflected into the database immediately. And, their concurrent execution is controlled by the transaction processing [5], [13] of the DBMS (Database Management System). Here, in the present time, non-stop service systems are used widely: such as the internet shops, ATMs, and so on. So, both of the above-mentioned two methods have to update the same table of the database concurrently.

As for the online entry, since it is executed as a transaction, its consistency is maintained. Moreover, the concurrency control is performed by the DBMS using the lock method to execute the many transactions as a serializable schedule. So, transactions are executed without affecting each other based on the isolation. And, in the case of transaction failure, its rollback is executed to cancel the entry without affecting the other entries based on the atomicity and isolation. On the other hand, if the lock method is used by the batch update for a great deals of data, then the conflicting online entries are made to wait for a long while. That is, though the batch update can be executed as a transaction, it causes the problem of the long latencies of the online entries.

For this problem, the mini-batch is used widely to shorten the latency. It splits the batch update to the short time transactions, and executes them sequentially [5]. However, the atomicity and isolation of the ACID properties cannot be maintained as for the whole mini-batch. That is, since the updated data are committed one after another, the rollback of the whole target data cannot be executed even in the case of the failure. That is, the update must be cancelled by the restore of the backup of the pre-update data or by the compensating transactions. However, in the case where it is executed concurrently with the online entries, they use the updated data immediately. So, it is difficult to cancel their result, and these methods are not practical. As a result, there is the problem that the mini-batch must complete all the update by removing the cause of the failure.

This means that the above-mentioned safety of system operations cannot be maintained. In particular, in the actual system operations, there are faults due to not only the program error but also the data quality, operation error, and so on. Therefore, there is the problem that the complex or atypical batch update process must be often executed by stopping the online entries to separate the both updates. For this problem, we have proposed the temporal update method, which utilizes the data history of the time series, and shown that the batch update can be executed as a transaction without long latency of the online entries by this method [9]. However, we have not evaluated this method in the case of failure yet.

So, our goal in this paper is to evaluate the recovery function of the temporal update for the transaction failure assum-

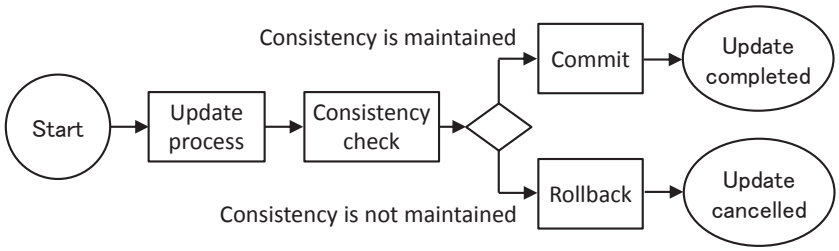

Figure 1: State transition of update transaction.

ing the actual business system operations. First, we show the requirements for the recovery of the batch update failure due to the actual business system operations. Here, we assume the batch updates are executed concurrently with the online entries in these system operations. Next, based on these requirements, we perform the experiment about this recovery method by implementing the temporal update prototype.

The remainder of this paper is organized as follows. We show the batch update model and related works in Section 2, and show the requirement about recovery functions of the temporal update in Section 3. We show the implementation and evaluations of these functions in Section 4, discuss on the evaluation results in Section 5, and conclude in Section 6.

2 RELATED WORKS

2.1 Batch Update Operations and Methods

To maintain the consistency of the database, various kinds of functions are implemented in the actual business systems. As for the update process of the online entry mentioned in Section 1, it is executed as a transaction, and the consistency of the update result is checked before its commit. And, in the case where the consistency is not maintained, the rollback is executed to cancel the update as shown in Fig. 1. For example, in the banking systems, if the updated result of the account transfer becomes minus, it cannot be executed. So, the rollback of the transaction is executed, and the process is cancelled. On the other hand, in the case where the consistency is maintained, the commit is executed.

As for the batch update, in the case of being executed in the different time period from the online entries such as the night batch, the operations like this can be performed. We show the dataflow of the batch update process in Fig. 2. Since the batch update processes a great deal of data in a lump, commits are usually executed on the way based on the resource constraint. So, the backup of the target table is performed prior to the update. Then, in the case where the consistency of the update result is not maintained, the target table is recovered by the backup data. That is, the update is cancelled. On the other hand, in the case where the consistency of the result is maintained, the update completes and the target table is used by the business such as online entries.

Here, there are other recovery methods, for example, the differential backup function of the DBMS, compensating transaction. However, the recovery by the former cancels the update of the other transactions executed simultaneously, and the latter needs the extra program. So, the recovery of the batch update is generally executed by above-mentioned process.

Thus, even to execute the batch updates safely, the ACID properties of the transaction must be maintained. That is, as for atomicity, the state of database has to transit to a state of either: the commit state of the update in the case of successful completion; the state of the update cancelled by the rollback in the case of failure. As for the consistency, the update result must be checked to satisfy the various constraints before its completion. Then, the other transactions can access the table with the consistency. As for the isolation, the target table of the update must not be accessed until the completion to avoid the influences on the other processing. In addition, the durability is maintained by the function of the DBMS as well as the online entry transaction.

This process is implemented by the lock method for the target table or data during the batch update. Though the ACID properties of the batch update are able to be maintained by this method, there is a problem that the online entries to access the target data must wait for a long while. However, as for the non-stop service systems, since the online entries are always performed, it is impossible to separate the processing time zone between the batch update and them. As a result, there is a problem that it is difficult to execute the batch updates with maintaining the ACID properties. So, as for the concurrency control between the batch update and online entries, some methods have been put into practical use [13],[14]. However, as for the conventional methods, there are some problems to apply it to the non-stop service systems.

First, the mini-batch splits the batch update to the short time transactions and executes them sequentially to shorten the time to lock each data. However, as for this method, there is the problem that the ACID properties cannot be maintained as the whole mini-batch update as mentioned in Section 1. Next, as for the timestamp ordering, a unique timestamp is assigned to each transaction [2],[3]. And, if the transaction accesses the data updated by the larger timestamp transaction, it is aborted. However, since the batch update takes a long time, it has to be abort in most cases. Moreover, as for the multiversion concurrency control based on the snapshot isolation level, it also uses the above-mentioned method [1], [10]. So, there is the same problem.

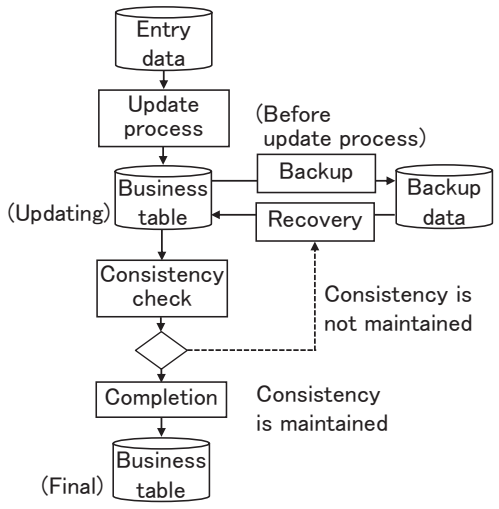

Figure 2: Data flow of batch update process.

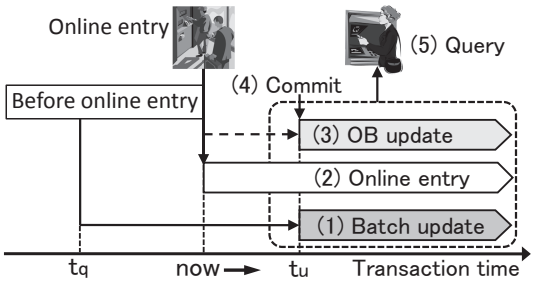

Figure 3: Outline of temporal update method.

2.2 Temporal Update Method

To solve this problem, we have proposed the temporal update method [9]. In this method, we use the extended transaction time. Here, the conventional transaction time expresses when a fact existed in the database [11], [4]. For example, if a fact existed between the time T_a and T_d, its transaction time period T is expressed by $T = \{T_a, T_d\}$. That is, the fact is entered into the database at time T_a, and deleted at time T_d logically to remain the data history. As long as the data has not been deleted yet, the instance of T_d is expressed by now, which is the current time to query the database [12]. And, we extend this time to the future.

In Fig. 3, we show the outline of this method. In this method, the batch update queries the data of the past time t_q, and inserts the update result in the future time t_u as shown by (1) in this figure. On the other hand, the online entry accesses the data of the current time now as shown by (2). So, it does not conflict with the batch update. Here, even for the online entry results during the batch update, it is also necessary to perform the batch update. Thus, the online entry result is updated similarly to the batch update separately as shown by (3). We call this update "online batch update" (hereinafter, "OB update"). Then, by the commit of the batch update as shown by (4), the batch and OB update results become to be queried by the other processing.

However, since plural data exist after the commit, valid data

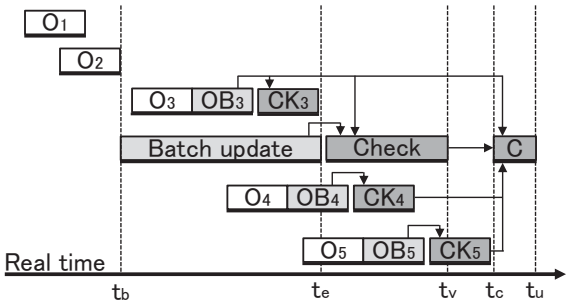

Figure 4: Execution timing of transactions.

have to be queried as shown by (5). The target data are selected as follows: firstly, the latest update data are selected for each primary key, and they can be identified by the abovementioned time attribute T_a; secondly, if there are still plural data, the target data are determined in the order of the OB update, online entry and batch update. In other words, the online entry has higher priority to the batch update. And, the OB update, which reflects the batch update on the online entry result, has the highest priority.

We have shown the following in our previous studies of the temporal update method [6],[7]. Firstly, we can execute the batch update about the data related to each other concurrently with the online entry, though the both are executed separately by using the conventional methods. Secondly, we can execute the batch update more efficiently than the mini-batch, which is the conventional method to update the data unrelated each other in a lump, in both of the centralized and distributed database environment. Moreover, we have shown that we can apply this method even if the completion time cannot be predicted at the batch update starting [8]. On the other hand, it is necessary to consider the following to apply this method: the implementations of the OB update program, the transaction time and so on; in the case where the online entry frequency is high, its efficiency declines.

However, these previous studies assume only the case where the temporal update completed successfully. And, the following studies as for the transaction failure have not performed yet: the study on confirmation of the consistency of the temporal update results before the commit; the study on rollback of the transaction in the case where some anomaly is detected as shown in Fig. 1.

3 REQUIREMENT FOR CONSISTENCY MAINTENANCE

In this section, we explore the requirements to maintain the consistency of the result of the temporal update, in the case where it is executed concurrently with the online entries. We show an execution timing example of transactions related to the temporal update in Fig. 4. In this figure, O_i shows the online entry; OB_i shows the OB update; C shows the commit of the batch and OB update. OB_i is executed following to O_i during the batch update.

Since the batch update processes a great deal of data, it is often composed by plural transactions due to the resource re-

strictions. On the other hand, at the batch update completion time t_e, from the viewpoint of the transaction processing of DBMS, the whole batch update and the completed OB update results must not be queried by the other transactions until the completion of the commit C. In other words, as for the example in this figure, the transaction of the batch update and OB_3 are prepared to commit as the temporal update. We call this status "pre-commit." And, with the passage of time, the other OB updates complete one after another such as OB_4, OB_5, and they are committed in a lump with the batch update at the time t_c.

Here, the consistency of the batch and OB update results have to be checked before the commit as shown in Fig. 2. And, if their consistencies are not maintained, the rollback of these updates must be done. However, since the OB updates are performed until the commit C, their results have to be also checked individually by CK_i as shown in this figure. And, if the consistency of OB_i is not maintained, both of its rollback and the corresponding online entry O_i's rollback are done.

Thus, the requirement to maintain the ACID properties in the temporal update is as following. First, as for the consistency, it has to be checked prior to the commit as shown in Fig. 4. Here, the consistency has to be checked by both of the DBMS functions and the business logics. The latter indicates that these update results have to be queried from only the batch update application program prior to the commit. In other words, prior to the commit of the temporal update, its results and the committed online entries have to be queried to verify the consistency of the database.

On the other hand, as for the isolation, the batch update and OB update results must not be queried by the online entry application program until the commit. In addition, even if rollback due to the failure is not completed until the commit time t_u that was scheduled, it must not be queried similarly. And, as for atomicity, both rollbacks of the batch update and the OB update have to be executed in the case of failure. That is, the database transitions to the state where only the online entries were performed. Moreover, in the actual system operations, if the batch update was aborted due to the failure, it is necessary to be rerun by removing the cause of the failure immediately. So, the rollback also has to be executed efficiently.

In addition, since the OB update accompanies the online entry, it is not executed if the latter is failed. Conversely, when a failure is detected in the OB update or batch update, it is necessary to execute their rollbacks in each case. First, in the case where a transaction failure or consistency anomaly is detected during the OB update, both rollbacks of it and the corresponding online entry must be executed to prevent the anomaly between them. That is, the online entry and OB update have to be executed as a single transaction. Second, in the case where some consistency anomaly is detected by the check of the batch update, the commit of only the online entry can be executed. So, the batch update and all the corresponding OB updates must be cancelled by the rollback.

For example, we show the data manipulation of the residence indication. In this business, all the address is changed to be easy to understand, and it is performed in the whole target district at the same. Here, the data of one household

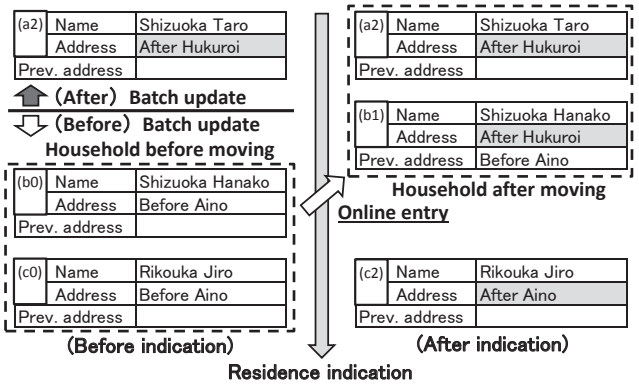

Figure 5: Update example of mini-batch.

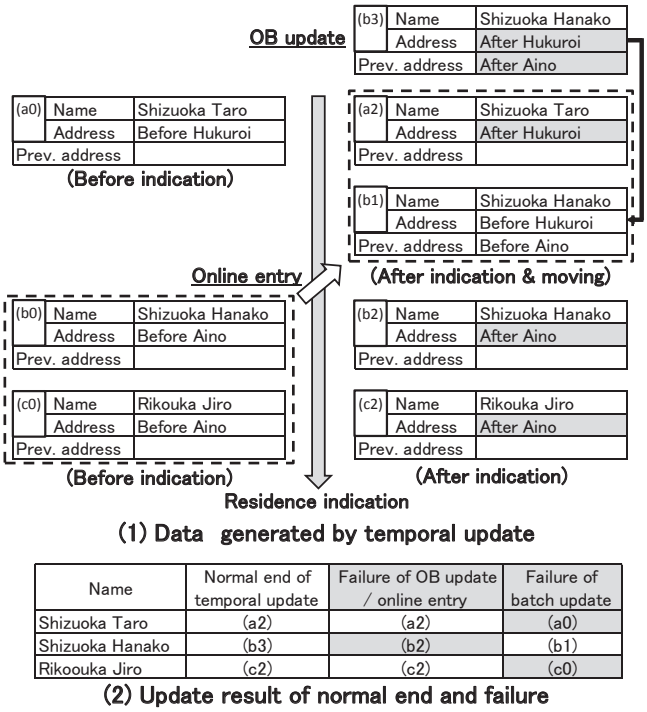

Figure 6: Update example of temporal update.

must have the same current address, and a resident after moving has the previous address, which must be consistent with the current address as timing. So, since the data are related with each other, it is necessary to process the whole batch updates as a transaction. Therefore, it cannot be executed by the mini-batch concurrently with the online entries. We show this example in Fig. 5. Here, "Address" shows the current address, and "Prev. Address" shows the previous address in it. In this case, household addresses are changed by the mini-batch update. Concurrently, by the online entry, a member of a household (b0) that has not changed yet moves to another household that has already changed, and its result is shown by (b1) in this figure. Here, the resident card has the previous address, and it does not reflect the residence indication. However, the current address reflects it, though it is same timing with the previous address. Therefore, an anomaly between the current address and previous address occurs.

On the other hand, we show the case of executing the same processing by the temporal update method in (1) of Fig. 6. As for this method, its batch update results are not queried until the commit. So, as shown by (b1) in this figure, both of the current address and the previous address of the online entry result are based on the data before the residence indication. Similarly, the following data are generated: the batch update result (b2), which does not reflect the online entry; the OB update result (b3), which reflects the residence indication on the online entry result. As shown in Section 2.2, since the OB update result (b3) has the highest priority, the moving result reflecting the residence indication is queried.

And, in the case where the transaction failure occurs in this processing, the following recovery is executed as shown in (2) of Fig. 6. First, if the OB update fails, then the current address shown by (b1) remains. So, it causes anomaly among (b1) and (a2), that is, the residents in the same household have different current addresses. Therefore, both of the OB update and the online entry must be cancelled by the rollback. As a result, only the residence indication result (b2) remains, which maintains the consistency. Second, in the case where the online entry fails, its result is same as this. Third, if the batch update fails by some reason, the batch update and OB update are cancelled. They are (a2), (b2), (c2) and (b3). So, the online entry result (b1) remains instead of the original data (b0). That is, only the moving result remains.

As shown above, the requirement to maintain the consistency of the temporal update result is the following. First, OB update has to be executed in the single transaction with the corresponding online entry. Second, prior to the batch and OB update commit, the consistency of their result have to be checked. Third, if the anomaly is detected, the rollbacks of the batch and OB update have to be executed without affecting to the online entries. Fourth, this rollback has to be executed efficiently.

4 IMPLEMENTATION AND EVALUATIONS

We perform the experiment by the prototype of the bank account system, because it is a simple business system. And, verify that the requirements mentioned in Section 3 can be realized by the temporal update method.

4.1 Implementation of Temporal Update

In Fig. 7, we show the program composition of the temporal update to realize the data manipulation shown in Fig. 4. As for the online entry transaction, in the case where the online entry O_i conflicts the batch update, the OB update OB_i and its consistency check CK_i are executed. In addition, in the case where some anomaly is detected, the rollback of this transaction is executed. On the other hand, as for the batch update, its status becomes pre-commit shown in Section 3, when its execution and the commit of DBMS has completed. Next, its consistency check is executed. Then, it waits the completion of the running online entry transactions that update the target data of the batch update, and executes the commit.

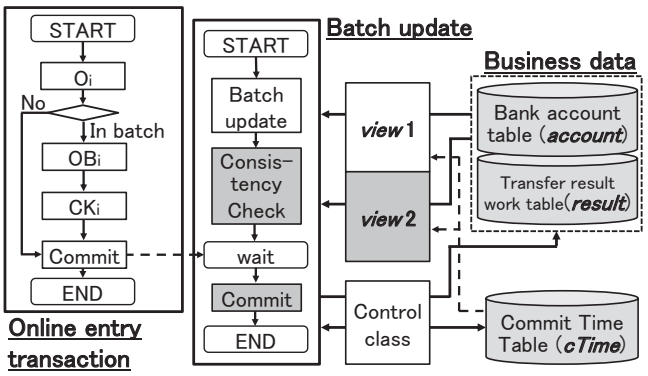

Figure 7: Program composition of temporal update.

As for the table of database, we use the following three tables. Bank account table (hereinafter, "$account$") stores the deposit balance of the account, and Commit time table (hereinafter, "$cTime$") stores the data to manage the temporal update. In addition, Transfer result work table (hereinafter, "$result$") stores the transfer amount and the result of account transfer, which is performed by the batch and OB update.

Here, the data of $account$ are queried via following two views [3] as shown in Fig. 7, and $cTime$ is used for these views. First, "$view1$" is used by the online entry transactions, and the data of the batch update and OB update are not queried until the commit of the batch update. Of course, the transaction can query the OB update data made by itself. Second, "$view2$" is used by the batch update, and has the same function as $view1$. Also, the batch update program can query the batch and OB update data even before they are committed.

In Fig. 8, we show the table composition and its query result via views. Firstly, as for $account$, if we express its relation scheme by R_a, then its attributes are expressed by the following. In addition, T_a and T_d are the transaction time attributes mentioned in Section 2.2, and they are shown in the form of date for the simplicity.

$$R_a(K, T_a, T_d, P, D, A) \tag{1}$$

- K**:** primary key attributes. It is the primary key of the projection (K, A), which is the attributes for business.
- T_a**:** addition time of the data. Here, as for the batch and OB update, it is the start time of the temporal update and, "@" is set for the first place as shown at (c) of Fig. 8. It is for the case where the completion time of the batch update cannot be predicted.
- T_d**:** deletion time of the data.
- P**:** process class. This shows the process that updated this data: the OB update, the online entry, and the batch update. The corresponding value set is expressed by $\{\mathrm{P_{ob}}, \mathrm{P_o}, \mathrm{P_b}\}$. Here, we make $\mathrm{P_{ob}} > \mathrm{P_o} > \mathrm{P_b}$.
- D**:** deletion flag. This shows whether the queried data is the target of the query. So, it has the logical value $\{true, false\}$. And, if $D = false$ then the data does not be queried. It is used by the OB update to hide

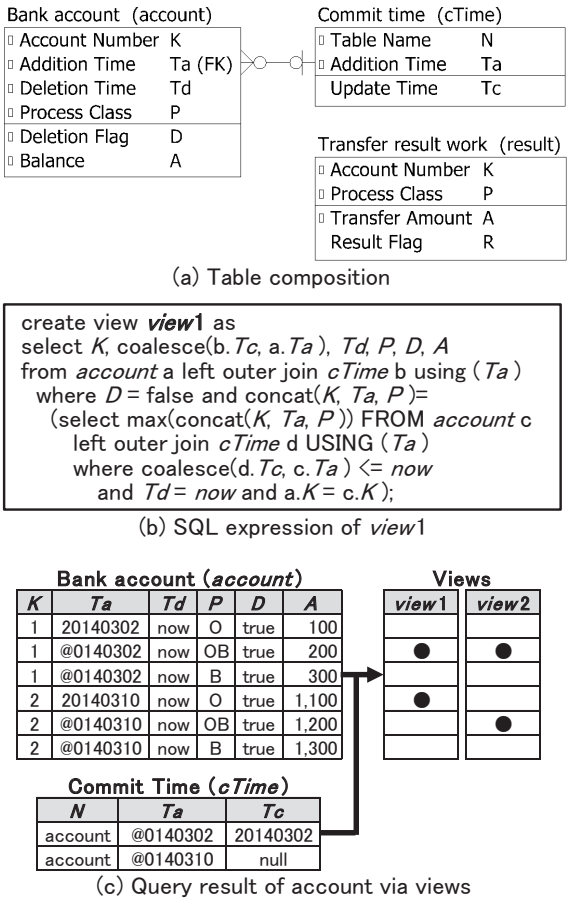

Figure 8: Data and views of temporal update.

the corresponding batch update result, which data was deleted by the online entry.

- A**:** the other attribute. As for the Fig. 8, it shows the balance of the bank account.

Next, as for $cTime$, if we express its relation scheme by R_c, then its attributes are expressed by the following.

$$R_c(N, T_a, T_c) \tag{2}$$

- N**:** table name updated by the target batch update.
- T_a**:** addition time of the target batch update and OB update data.
- T_c**:** commit time of the target batch update. Until the completion of the commit C, it is set to "null."

Here, $cTime$ controls the query results of views. As for $view1$, by the commit of the batch update, the time of this commit is set to T_c. So, if $T_c \neq null$, then the valid data are queried as follows. We show its SQL expression in (b) of Fig. 8. First, as for the data $P = \mathrm{P_b}$ (batch update result) or $P = \mathrm{P_{ob}}$ (OB update result) in $account$, if $account.T_a = cTime.T_a$ then the value of $account.T_a$ is replaces by the one of corresponding $cTime.T_c$. Next, as for each K, the data having the largest P from among the data having the latest T_a are queried. Incidentally, "now" is the current time mentioned in Section 2.2.

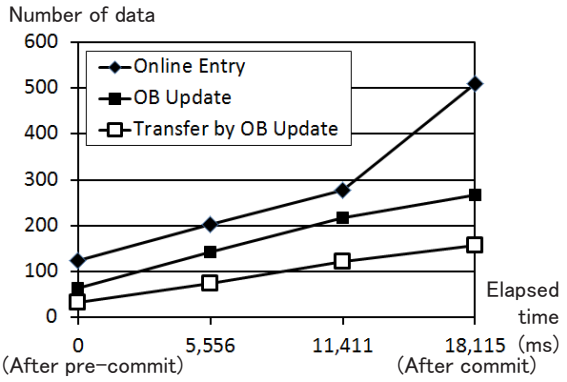

Figure 9: Transition of number of data via $view2$.

Similarly, we define $view2$. As for it, the data are queried for each K, which has the latest T_a then the largest P. Here, since "@" is larger value than the number, the batch and OB update result has the latest T_a. For example, if the online entry is executed concurrently with the batch update, OB update result is queried.

We show an example of the query result via views in (c) of Fig. 8. As for the data $K = 1$, since its batch update is committed, the query results are same in the both of two views. As for uncommitted data $K = 2$, the online entry result being executed concurrently with the batch update is queried by $view1$; its OB update result is queried by $view2$.

In addition, $result$ is a table about the interface with the company which entrusted the account transfer: the account number, transfer amount of each account is indicated by the company; the result flag is set by the system to each account data corresponding to the successful or failure, about the account transfer. Since this table is used as a work file, the data history is not necessary. However, the result flags are set by both of the batch and OB update processes. Here, the execution order of them is undecided. So, we added the process class P to this table to store both of the results, and select the data in the same way as $account$.

4.2 Experimentations and Evaluations

We performed this experiment by the stand alone PC environment: CPU was Xeon CPU E5-1620 3.60 GHz with 8 GB memory, and its OS was Windows 7 (64 bit); DBMS was 5.6.17 version of MySQL; the transaction control was performed by its database engine InnoDB; the concurrent execution was implemented by Thread class of Java.

4.2.1 Consistency Check before Commit

To evaluate the requirement mentioned in Section 3, we performed experiments using the tables shown in Fig. 8. Prior to each experiment, we set 10 thousand data to the $account$, which bank balance was one million; set 9 thousand data of transfer amount A to $result$ to perform the bank transfer, and their result flag R (transfer result) was set to "null." Here, to examine the account transfer about both of the success and failure cases, we calculated the value of A by the following

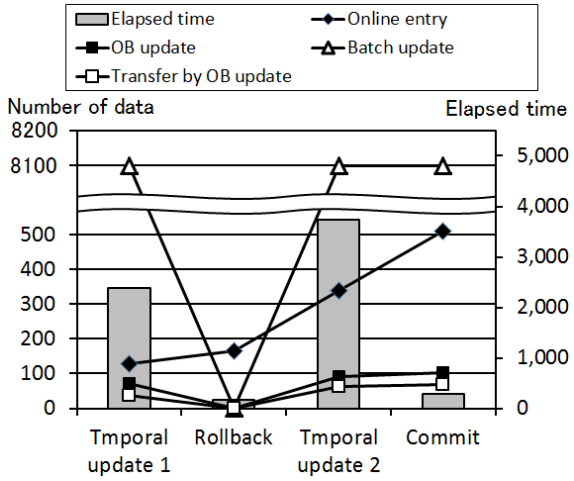

Figure 10: Transition of number of data about rollback.

function of K.

$$A = [111 - (K \ mod \ 110) + 1] \times 10^4$$

Concurrently, the online entries were executed from 5 terminals, by which 50 % of the account balance of each bank account was transferred to its corresponding bank account. Each was executed in the interval of about 0.3 sec. And, if the online entry conflicted with the batch update, then its OB update was executed.

To examine whether the consistency of the batch and OB update can be checked before the commit, we experimented to query the data via $view2$. In Fig. 9, we show the experimental results from the pre-commit until the commit completion in the temporal update. In this figure, "Online Entry" shows the change of the number of the data in $account$ updated by the online entries with the elapsed time; "Transfer by OB Update" shows the same number updated by the OB update. Next, "OB Update" shows the change of the number of the data in $result$ inserted by the OB update. Here, even though the bank transfer is failed due to the lack of the account balance, the result is inserted to $result$. But, $account$ is not updated. So, the numbers of the latter two is different. In addition, though not shown in this figure, the unchanging number of the data updated by the batch update could be queried, because the pre-commit of the batch update process had already completed.

As shown in this figure, we could query the data of each query time, including the pre-committed batch and OB update data. Thus, using $view2$, we could check the consistency of the database any time before their commit.

Similarly, we examined the case of the rollback, and in this case, the experimental process to manipulate the data was as follows. In addition, the above-mentioned online entries were executed over this experiment, and the data of target tables were queried after the each stage. Firstly, the temporal update and its pre-commit were executed; secondly, the rollback of the batch and OB update was executed; thirdly, the temporal update was rerun for the data of this time, and its pre-commit was executed; finally, the commit of the temporal update was executed.

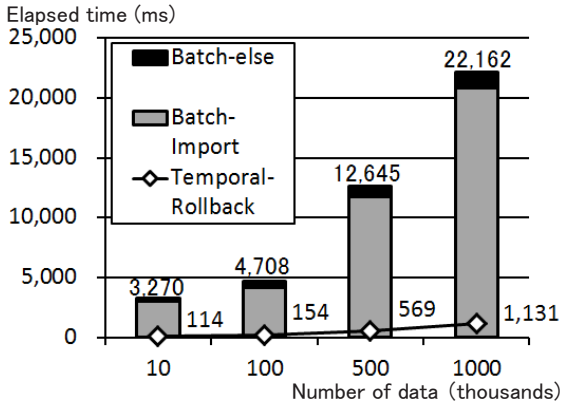

Figure 11: Comparison of elapsed time for recovery.

We show the experimental result in Fig. 10 similar to Fig. 9. Here the elapsed time of the each stage is shown by the bar graph and right axis, instead of the total elapsed time from the beginning. Fig. 10 shows, in addition to the result shown in Fig. 9, the number of the batch and OB update result becomes to 0 after the rollback. That is, the status, in which the batch and OB update were cancelled at this stage, could be queried. On the other hand, as for the requirement about the affecting to the online entries, since this rollback did not affect to the online entries, the number of the online entry data was constantly increased.

4.2.2 Efficiency of Rollback

To evaluate the efficiency of the rollback in the temporal update, we compared its elapsed time with the time for the recovery in the conventional batch and mini-batch update.

Here, as for the temporal update, its rollback can be executed by deleting the data from both of the business table and Commit time table $cTime$: as for the former, the target data set X of $account$ is expressed as following.

$$X = \{x \in R_a | (x[P] = \mathrm{P_b} \cup x[P] = \mathrm{P_{ob}}) \cap T_a = \mathrm{@time}\}$$

Here, $x[P]$ shows the value of attribute P in x; similarly, "@time" shows the value of T_a of this temporal update, which is "@0140310" in Fig. 8. Similar to this, as for $cTime$, the target data set Y is expressed as follows.

$$Y = \{y \in R_c | y[N] = \mathrm{account} \cap T_a = \mathrm{@time}\}$$

On the other hand, the recovery for the batch and mini-batch update has to be executed by the following process as shown in Fig. 2: firstly, the backup of the target data is always executed before the update; next, in the case of failure, the data of the target table are cleared, and the target table is restored by the backup.

In Fig. 11, we show the experimental result for the above-mentioned recovery in the four cases of the number of data in $account$. Here, "Batch-else" shows the total elapsed time of the backup and clear, and in this experiment we executed the clear by the "truncate" statement of SQL; "Batch-import" shows the elapsed time for the restore by the backup data, which is executed by the "import" statement of MySQL.

As shown in this figure, as for the both methods, as the data increase, the elapsed time is longer. However, the rollback of the temporal update could be executed so efficiently as compared with the method by the backup and restore. For example, in the case of the maximum number (1000 thousands), the elapsed time of the former was about 1/20 of the latter. Furthermore, most of the elapsed time of the recovery by the backup was spent for the restore.

5 DISCUSSIONS

Through the implementation and experiments, we confirmed that the requirements mentioned in Section 3 can be meet by the temporal update. First, as shown in Fig. 7, both of the online entry and OB update could be composed as a single transaction. Second, as shown in Fig. 9, the batch and OB update results could be queried before their commit. That is, the consistency about them could be checked before the commit. Third, as shown in Fig. 10, the rollback of them could be executed without effect on the online entry result. That is, if anomaly is detected by the above-mentioned check, the update is cancelled by the rollback. Finally, as shown in Fig. 11, this rollback is very efficient comparing with the conventional recovery method: about 20 times in the experimental case. Moreover, as shown in Fig. 10, we performed the verification of the case of rerun. In the actual system operations, if the anomaly is detected, the cause has to be removed and the job has to be rerun to complete the business. We think the result of this verification shows that this method is useful for the actual system operations.

Here, for the reason of the efficiency of the rollback, it can be pointed out that the rollback of the temporal update can be performed by a simple delete command as shown in Subsection 4.2.2. That is, in the temporal update, the data histories about the transaction time are managed. So, as shown in Fig. 8, since the batch and OB update results are stored in the table as unrelated records to the online entry results, they are classified by only the attribute of Process class P. As a result, the target data of the rollback can be deleted efficiently.

In the actual system operations, various kinds of failures are detected in the batch update and often it has to be rerun. So, we consider this efficient rollback is useful to shorten the turnaround of the batch update. Moreover, as shown in Fig. 10, we performed the verification of the case of rerun. In the actual system operations, if the anomaly is detected, the cause has to be removed and the job has to be rerun to complete the business.

Also, for example, even in the case where the batch update can be executed concurrently by using the mini-batch, the online entries are often stopped for the safety. As a reason for this, it can be pointed out that above-mentioned various failures, such as manipulation errors, dead-locks and so on, become threats of system operations, which may cause the error of the online entry result and to disturb the online entry. From the viewpoint of the safety of the batch update operations, the updated results should be separated from the online entry results; the consistency of the updated results should be checked before their commit. And, as above-mentioned, these requirements can be realized by the temporal update. Therefore, we think that this method is useful for the actual system operations.

As for the implementation of the temporal update, some attributes have to be added to the target and related tables: such as the transaction time, process class, and deletion flag. Also, some functions have to be composed: the views to query these tables; the OB update for the online transaction. On the other hand, the temporal update can be composed without considering the online entry. That is, for example, unlike the mini-batch, it is not necessary to split the update into the short time transactions, or to implement the recovery methods such as the compensating transactions. So, we consider that the temporal update is useful in the case where the implementation like this is necessary in other methods.

Lastly, we would like to discuss about the advantages and disadvantages of this method comparing with the conventional method. As for the advantages, firstly, this method can execute the whole batch update as a transaction concurrently with the online entries. That is, in the conventional method, there are the following problems about their concurrent executions: the batch update with table lock can be executed as a transaction, but the online entries must wait for its completion; the mini-batch can be executed concurrently with the online entries, but it is not a transaction as the whole processing. So, the latter cannot do its rollback and cannot maintain the isolation. And, the data have to be recovered by the backup data in the case of failure as for the both. Therefore, this method is useful for the following batch update: the batch update on the data related with each other, as shown in Fig. 5; the batch update which recovery for the failure has to be performed in a short time. On the other hand, as for the disadvantage, above-mentioned functions must be implemented for this method as shown in Fig. 7. So, the application of this method should be decided based on this trade off.

6 CONCLUSIONS

At the present time, since most of the business systems provide the nonstop services, the batch update has to be executed concurrently with the online entries. However, in such the environment, since it cannot be executed by the conventional methods as a transaction, some problems remain. So, we have proposed the temporal update method, and shown it can be executed as a transaction. However, to apply this method to the actual business systems, it has to equip the functions for the failures.

In this paper, we analyzed the batch update operations in the actual business systems, and showed the requirement to execute the temporal update safely during the online entries. Moreover, through the experiment by the prototype, we confirmed that the temporal update satisfy these requirements. Especially, we find that the rollback can be executed so efficiently comparing with the conventional methods. Therefore, we can extract conclusions that this method is useful in the actual system operations.

Future studies will focus on the investigation of the application fields of this method, and its application evaluations.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 24500132.

REFERENCES

[1] H. Berenson et al., "A Critique of ANSI SQL Isolation Levels," Proc. ACM SIGMOD 95, pp. 1-10 (1995).
[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley (1987).
[3] T.M. Connlly, and C.E. Begg, "Database Systems: A Practical Approach to Design," Implementation and Management, Addison-Wesley (2009).
[4] N. Edelweiss, P.N. Hübler, M.M. Moro, and G. Demartini, "A Temporal Database Management System Implemented on top of a Conventional Database," Proc. XX International Conference of the Chilean Computer Science Society, pp. 58–67 (2000).
[5] J. Gray, and A. Reuter, "Transaction Processing: Concept and Techniques," San Francisco: Morgan Kaufmann (1992).
[6] 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).
[7] T. Kudo, Y. Takeda, M. Ishino, K. Saotome, and N. Kataoka, "A Mass Data Update Method in Distributed Systems," 17th Int. Conf. in Knowledge Based and Intelligent Information and Engineering Systems - KES2013, Procedia Computer Science, Vol. 22, pp. 502–511(2013).
[8] T. Kudo, Y. Takeda, M. Ishino, K. Saotome, and N. Kataoka, "Application of a Lump-sum Update Method to Distributed Database," Proc. of Int. Workshop on Informatics (IWIN2013), pp. 49–56 (2013).
[9] T. Kudo, Y. Takeda, M. Ishino, K. Saotome, and N. Kataoka, "A batch Update Method of Database for Mass Data during Online Entry," Procs. 16th Int. Conf. on Knowledge-Based and Intelligent Information & Engineering Systems – KES 2012, pp. 1807–1816 (2012).
[10] A. Silberschatz, H.F. Korth, and S. Sudarshan, "Database System Concepts," McGraw-Hill Education (2010).
[11] R. Snodgrass, and I. Ahn, "Temporal Databases," IEEE COMPUTER, Vol. 19, No. 9, pp. 35–42 (1986).
[12] B. Stantic, J. Thornton, and A. Sattar, "A Novel Approach to Model NOW in Temporal Databases," Procs. 10th Int. Symposium on Temporal Representation and Reasoning and Fourth Int. Conf. on Temporal Logic, pp. 174–180 (2003).
[13] T. Wang, J. Vonk, B. Kratz, and P. Grefen, "A survey on the history of transaction management: from flat to grid transactions," Distributed and Parallel Databases, Vol. 23, Issue 3, pp. 235–270 (2008).
[14] D.S. Yadav, R. Agrawal, D.S. Chauhan, R.C. Saraswat, and A.K. Majumdar, "Modelling long duration transactions with time constraints in active database," Procs. the Int. Conf. on Information Technology: Coding and Computing (ITOC' 04), Vol. 1, pp. 497–501 (2004).

(Received November 15, 2014)

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, Information Processing Society of Japan and The Society of Project Management.

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.