Application of a Lump-sum Update Method to Distributed Database

Tsukasa Kudo[†], Yui Takeda[‡], Masahiko Ishino*, Kenji Saotome**, and Nobuhiro Kataoka***

[†]Faculty of Comprehensive Informatics, Shizuoka Institute of Science and Technology, Japan
[‡]Mitsubishi Electric Information Systems Corporation, Japan
* Department of Management Information Science, Fukui University of Technology, Japan
** Hosei Business School of Innovation Management, Japan
*** Interprise Laboratory, Japan
kudo@cs.sist.ac.jp

Abstract - At the present time, with spread of the internet business, many business systems have become to be built as distributed systems. Accordingly, the database is also dispersed to plural business systems as the distributed database. By the way, in the actual business systems, a lump-sum update of a great deal of data has often to be performed concurrently with the online transactions. However, in this case, there is a problem of efficiency in the conventional update methods, which update databases by the chain of many divided transactions. For this problem, we have shown the efficient lump-sum update method for centralized business systems, which use the transaction time. However, as shown by the distributed transactions, some transaction features for the distributed database are different from the centralized database. Therefore, in this paper, first we show the problems on applying this method to the distributed databases. Secondly, we propose their measures. Moreover, through the evaluations using a prototype, we confirmed that our measures are valid and this method can be applied to the distributed databases.

Keywords: database, distributed database, distributed transaction, business system, nonstop service.

1 INTRODUCTION

With spread of the internet business and decentralized technology, many business systems have become to be built as distributed systems at the present time. That is, each system performs its business by mutual cooperation with other systems [7], [11]. For example, in corporations, business systems are built at each branch office, and the business of this office is performed using its system. On the other hand, each business system is operated as a part of the distributed system, since each system accesses the data of the other systems when it is necessary. In this way, the distributed business system as the whole corporation is composed. And, it is possible to reduce the communication cost and perform the suitable system operation for the each office. On the one hand, as for the online services on a wide range of Internet such as net shops, non-stop services have become common because of the convenience of customers, the globalization and so on. So, it is difficult to stop the online service for the particular business.

However, in such a business system, a great deal of data often has to be updated in a lump-sum. So, formerly, it was executed as a night batch to avoid the time zone of the online service. However, because of the spread of nonstop service, it has to be executed concurrently with the online service at present. Therefore, some methods were put to practical use to execute it concurrently with the user entry of online service (hereinafter “online entry”). Here, these conventional methods divide the updating of a great deal of data into short time transactions, and then execute and commit them one after another. Therefore, though its impact on the online entry is small, there are problems: the intermediate results of the updating can be queried by the other transactions, that is, the isolation cannot be maintained; its processing efficiency declines because of the increase of its commit number.

For these problems, we had proposed an updating method that utilizes the records about the transaction time[2]. Moreover, we had shown the following evaluation result as for the updating of a great deal of data in the centralized systems: it executes the update more efficiently than the conventional method with maintaining the ACID property[3]. Since the transaction time is a kind of time of the temporal database, hereinafter we call it “temporal update” method. However, in order to apply this method to a distributed database, it is necessary to measure against not only the distributed transaction but also the various problems of the distributed environment.

Here, the temporal update has the characteristic of its process as follows: the completion time of updating is set beforehand; as for the commit of this method, its execution control has to be performed for the serialization between the online entries. However, there are problems: the dispersion in the update time is often increased by the efficiency of the network and cooperative business system environments; the synchronization between plural servers often causes the decline of efficiency. Therefore, in this paper, we propose a method for applying the temporal update method to the distributed databases. Moreover, we show the following experimental results of the prototype: the problems can be solved by the proposal method; since the implementation of the destination server of data transfer is easy, this method is valid for a data distribution system having many destination servers.

The remainder of this paper is organized as follows. Section 2 shows the related works about the update transaction; an overview of the temporal update method; the problem about applying it to the distributed databases. In Section 3, we propose the method for this problem, and show its implementation in Section 4. In Section 5, we show the experimental results of the prototype, and show our considerations in Section 6.

2 PROBLEM OF DATABASE UPDATE IN A LUMP-SUM

2.1 Related works

As for the update of the database, it is necessary that the transaction maintains the ACID properties: atomicity, consistency, isolation, and durability[1]. Here, since a lot of users use the online entry concurrently, a lot of corresponding transactions access the database concurrently. Therefore, database updates are serialized by locking the data to be updated by each transaction, and we can obtain a result as if transactions were executed sequentially. On the other hand, in the actual business systems, it is necessary to update a great deal of data in a lump-sum. For example, in the banking systems, the ATM is provided as the online service, and a lot of users perform online entry at the same time. On the other hand, a great deal of account transfer that is entrusted from the credit card company and so on is executed as the lump-sum update.

Since the update time of such a lump-sum is often so long, there is the problem that it can't lock the whole target data to serialize between the online entries. It makes the online entries wait for a long while. So, some methods were put to practical use to execute it concurrently with the online entries. In the mini-batch, a great deal of data is divided into small units, and they are updated and committed individually to shorten each update time. That is, the lump-sum update is performed by a set of short transactions. Also, in sagas, they compose a sequence of transactions and are performed one after another. It has a configuration to recover by executing the compensating transaction corresponding to each transaction in the case of fault [1], [10]. But, in these methods, the update and commit are repeated alternately many times. And, it makes the problems: the intermediate results of the updating can be queried by the other transactions; the efficiency declines because of the increase of commit number.

Here, in the distributed databases, the transaction needs not only the update and commit feature as for the individual database but also the feature for simultaneous update of multiple databases. So, the distributed transaction feature was put to practical use, in which the concurrency control across multiple databases is performed by two-phase commit and so on [6]. In this way, as for the distributed database, the different transaction feature from the centralized database has to be introduced.

By the way, the temporal database was proposed from the viewpoint of the record management about the time [8]. The transaction time is one of the times of this database: the time that a fact is valid in the database, which is expressed by the half-open interval $[T_a, T_d)$. Here, T_a shows the addition time that the data of the fact was added to the database; T_d shows the deletion time that it was deleted from the database. Even in the case of data deletion, data is deleted only logically by setting the deletion time, so the data record is left. Incidentally, if the data wasn't deleted, the value of attribute T_d is expressed by now [9]. It shows the current time and changes with passage of time. The relation R_t of the table having the transaction time is expressed below.

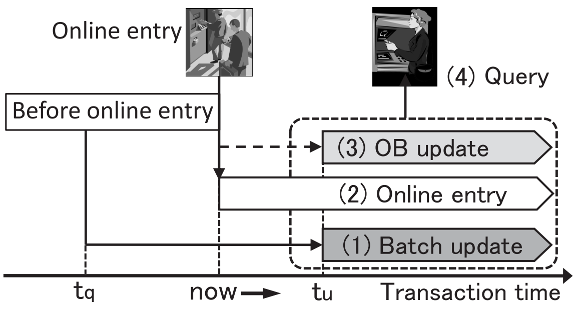


Figure 1: Change of data by temporal update method.

$$R_t(K, T_a, T_d, A) \tag{1}$$

Here, T_a and T_d are the above-mentioned transaction time. Let $q[T_a]$ be the value of attribute T_a of tuple q. Then the data set of the snapshot of R_t at the designated time t is expressed by the following $Q(t)$.

$$Q(t) = \{q | q \in R_t; q[T_a] \le t \wedge t < q[T_d]\} \tag{2}$$

Here, K is the primary key attribute of the snapshot; A is the other attribute. The transaction time is not exposed to the users. Since the update of the online entry is performed at the current time $t = now$, the data of Equation (2) at any past time can be queried without the conflict with the online entry.

For a data update method, we can use the optimistic concurrency control [4] by utilizing the transaction time. The lump-sum updates are often performed by the following procedure: reading the target data; generating the update data from it; updating the database. In this method, the transaction confirms the transaction time of the target data again at the update timing. And, if it was not changed from the read timing, it shows the data was not updated by another transaction. However, the transaction needs to lock the data between this confirmation and the commit.

For another update method which utilizing a time, there is the timestamp-ordering concurrency control. It uses the time stamps (start time) of transactions $\{T_1, T_2, ..., T_n\}$, which compose the ordered set. The time stamp is stored in data when a transaction accesses the data, and the order of transactions that access to each data is maintained by it [4]. However, when one transaction is updating a data, the other transactions that update the same data have to wait until the commit.

Thus, these methods intend for short time transactions. For example, since the lump-sum update takes a long while, the other transactions are also waited for a long while. That is, it disturbs the non-stop services. Moreover, we can't find the lump-sum update method for a great deal of data with maintaining the ACID property.

2.2 Temporal Update Method

As for the centralized database, we proposed the temporal update method that utilizes the transaction time to update a great deal of data in a lump-sum with maintaining the ACID property of the transaction [2]. Figure 1 shows the data change with the transaction time in the case of the database

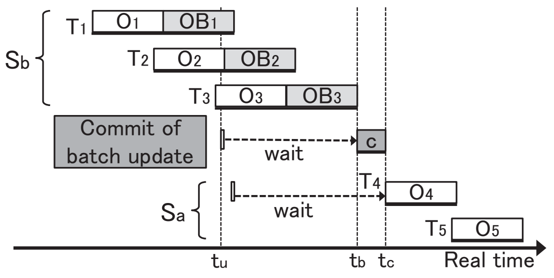


Figure 2: Serialization of batch update and online entry.

update by this method. In this method, the lump-sum update is executed by both the batch update (1) and OB update (the online batch update) (3). Here, the batch update (1) corresponds to the usual lump-sum update. It updates the data at the past time t_q, and stores the update results at the future time t_u that was "beforehand" established. Incidentally, though the online entry (2) is executed even during the batch update (1), it updates the data at the current time now. So, the competition between (1) and (2) can be avoided. On the other hand, since the batch update also has to update the data changed by the online entry, the OB update performs the process corresponding to it individually.

As a result, at time tu, three kinds of results are stored: the batch update, the online entry and the OB update. Therefore, we query only the high-priority data by the following order of priority, and we can get the query result as if the batch update and online entry were performed in a series.

(A) First, for each value of the primary key K of Equation (1), we select the data that was updated at the latest time. Here, we use t_q to the updated time as for only the batch update.

(B) Second, if there are plural data having the same key value, we select the data by the following priority of update process: the OB update, the online entry, the batch update.

In the case of Fig. 1, since both results of the OB update and online entry were updated at the latest time, the OB update result is queried based on above-mentioned (B). Also, if there is no online entry as for the case of Fig. 1, there are the online entry data entered before t_q ("Before online entry" in the Figure) , and the result of the batch update. So, the batch update result is queried based on above-mentioned (A). That is, in this method, all the updated data are stored, and only the valid data is queried. As a result, we can perform the lump-sum update and the online entry concurrently, without their competition.

However, at the end of a batch update, the control for the serialization between it and the online entry is necessary. That is, the online entry, which is begun before time t_u, is accompanied by the OB update to reflect the batch update. On the other hand, another one begun after t_u has to be performed using the result of the batch update. We show this in Fig. 2. Therefore, in this method, the commit of the batch update is executed after the online entries (S_b) that are begun before time t_u; the other online entries (S_a), which are begun after t_u, are waited until this commit.

2.3 Problem about Application to Distributed Database

As for the temporal update method, we had been assuming the centralized database and the controlled operation. That is, as mentioned in Section 2.2, we assumed that the batch update time t_u can be established beforehand; the individual online entry completes in a short time. In the actual systems, the former feature is often used in the lump-sum update executing at the designated time: the bank transfer at the designated time; the change of organization data at the designated date; and so on. However, as for the case where we apply this method to the distributed database, it has to be executed in the various environments. So, the following problems occur.

First, the dispersion of the batch update time is very larger than the centralized database. Though the batch update updates plural databases at the same time, the individual environment in this distributed system is varied: the traffic on the network; the load and performance of each server. So, the prediction of the time t_u is extremely difficult. Therefore, in the case that the prediction time is earlier than the actual elapsed time, the batch update doesn't complete by t_u. So, it is aborted, and it has to be re-run. On the contrary, if it is later, the unnecessary OB update must be continued after the batch update completion. As a result, there is a problem that the efficiency as the whole system declines.

Second, the problem that the online entry wait of one server spreads to the other servers occurs. As for the distributed database, the related online entries executing in all the servers have to wait the completion of the commit of the batch update until t_c as shown in Fig. 2. However, since the systems were built individually at each branch office, there may be the online entries having a long time transaction. Therefore, there is a problem about the delay of the commit of the batch update, which is caused by some online entry. And, it delays the online entries in the all related servers.

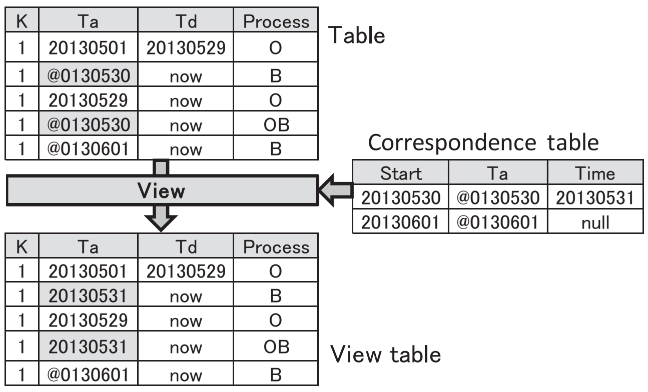
3 PROPOSAL METHOD FOR DISTRIBUTED DATABASE

To solve the problem for applying the temporal update to the distributed database, we propose the following two methods.

3.1 Setting Method of Dynamic Batch Update Completion Time

For the problem about the elapsed time of the batch update, we propose the method to perform its commit immediately after the batch update. Here, as shown in "Table" of Fig. 3, since the predicted completion time is set to every updated data by the batch update and OB update, it takes time to update these time again. Therefore, in this method, we use a view table to change these times when these data are queried.

Figure 3 shows the overview of this method. Here, time is shown as date: year, month and day. The predicted completion time of the batch update is set as the temporary time, and its first digit is replaced by "@" as an example. In the case



Table

K	Ta	Td	Process
1	20130501	20130529	O
1	@0130530	now	B
1	20130529	now	O
1	@0130530	now	OB
1	@0130601	now	B

Correspondence table

Start	Ta	Time
20130530	@0130530	20130531
20130601	@0130601	null

View table

K	Ta	Td	Process
1	20130501	20130529	O
1	20130531	now	B
1	20130529	now	O
1	20130531	now	OB
1	@0130601	now	B

Figure 3: Change of addition time of business data.

of Fig. 3, the addition time T_a, which is equal to the completion time, is set to “@0130530” and “@0130601”. Also, its “Process” column shows the classification of the update process of the data: O (the online entry), B (the batch update) and OB (the OB update). In this case, the data entered by the online entry on 5/1 of 2013 was updated by the batch update on 5/29, which above-mentioned completion time (T_a) is “@0130530”. And, the OB update was performed along with this. In addition, the batch update with the completion time “@0130601” was performed after this.

When the batch update completed, its completion time is set (“Time” of “Correspondence table”), which corresponds to the temporary time T_a. Incidentally, it is set to $null$ until this completion. Only if the the completion time is set to “Time”, T_a of data of “View table” is replaced with it. In the case of Fig. 3, “@0130530” of this is replaced with “20130531”; “@0130601” isn't replaced.That is, even if there are many target data, all T_a of them can be replaced by updating only one data. Therefore, this update process can be executed by a short time transaction. Incidentally, since the value of “@” is larger than any numerical value, the data having the temporary time (with “@”) isn't queried by Equation (2) with designating time.

3.2 Serialization Method without Lock Feature

To solve the problem shown in Fig. 2, we propose the serialization method between the online entry and batch update. In (A) and (B) of the Fig. 4, we show the case that the online entry is executed across the batch update completion time t_u. Incidentally, (C) shows the case that there is no online entry, as the reference. Here, the black hatching shows a transaction of the online entry; the broken line shows the data that should not be queried though it is exists.

First, (A) of Fig. 4 shows the success case of the online entry. Transaction (A) continues across the batch update completion time t_u. If the batch update result $(a-1)$ is queried between time t_u and the completion time of this transaction, it becomes a phantom read. Therefore, the mechanism, which makes this data not to be queried, is necessary to maintain the consistency. Incidentally, after the transaction completed, data (a) is deleted logically by setting the deletion time; the

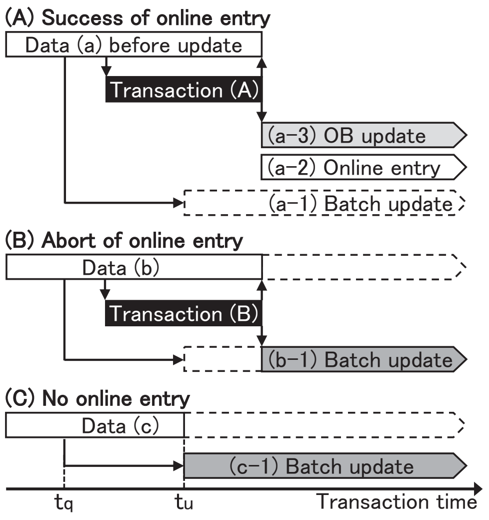


Figure 4: Query result as for online entry spanning t_u.

online enrty data $(a-2)$ is inserted; the OB update data $(a-3)$ is inserted if data (a) was the target of the batch update. So, $(a-1)$ is not queried, though either $(a-2)$ or $(a-3)$ is queried.

Next, (B) of Fig. 4 shows the abort case of the online entry. Transaction (B) continues across the batch update completion time t_u similar to (A), and its result is undecided at t_u: success or abort. Therefore, as for data $(b-1)$, the mechanism similar to (A) is necessary. In the case of the abort, data (b) remains without change, since the rollback of the transaction is performed. And, since the batch update was executed, its result $(b-1)$ has to be queried after the completion of the transaction. That is, for the serialization, the temporal update has to be composed to obtain the following query result.

- **During the online entry:** the batch update result is not queried, but the data just before the start of the online entry transaction is queried.
- **After the online entry completion (success):** either the online entry result or the OB update result is queried. Incidentally, in only the case of this data being target of the batch update, the latter occurs.
- **After the online entry completion (abort):** the batch update result is queried.

Therefore, in the case that the online entry transaction continues across the batch update completion time t_u, the consistency of the query result is maintained by the mechanism: the batch update result is not queried until the completion of the transaction. Therefore, we propose a method using a table that manages the key of the data being updated by online entry transactions. And the data, which key is registered in this table, is excluded from the query result of the batch update. This table has the following relation.

$$R_e(t\_name, t\_key) \tag{3}$$

Here, t_name shows the name of the target table; t_key shows the value of the primary key of the online entry data.

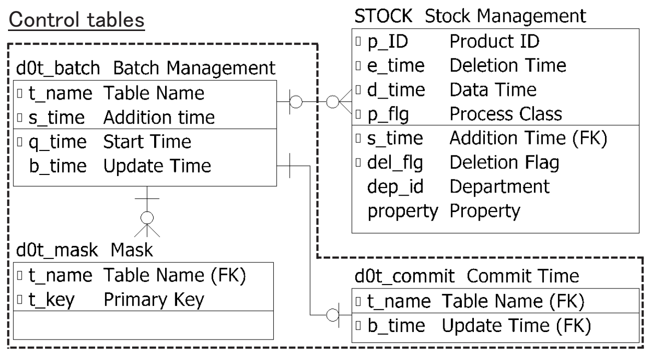


Figure 5: ER diagram of control tables.

And, (t_name, t_key) composes the primary key of this table. In this way, it is possible to perform the serialization by the table R_e without the lock feature between the onlien entry and batch update.

4 IMPLEMENTATION

First, Fig. 5 shows the ER diagram of the Control tables, which are used for the implementation of the proposal method. These tables save the data to control the temporal update and are placed in each target database of the temporal update. Batch management table $d0t\_batch$ manages the time of the temporal update, and it saves the following data: Table name t_name; Addition time s_time, Start time q_time; Update time b_time. Here, b_time is used to set the completion time of the batch update at the timing of its completion dynamically as shown in Fig. 3. That is, if the estimation of the completion time of the batch update is difficult beforehand, a temporary time and null are set to each s_time and b_time at the timing of its beginning. Then, the completion time t_u is set to b_time at the timing of its completion to query the updated data having the addition time t_u. Also, Mask table $d0t\_mask$ is an implementation of the relation of Equation (3). Commit time table $d0t\_commit$ stores the latest completion time t_u in b_time for each target table which name is set to t_name. And the result of the batch update and OB update which s_time is before b_time is queried.

Stock Management table $stock$ is an example of the business data table, and we use it for the experiment in Section 5. Here, as for the stock management, various kinds of distributed databases are used. For example, in the retail companies, each branch has its database system and manages its stock. On the other hand, the database serves as a part of the distributed database in the case of the stock movement among branches or the delivery from the distribution sector. Similarly, in the manufacturing industry, the supply chains of the parts are built among the related companies.

As for the attributes of $stock$, the following are correspond to the attributes (K, T_a, T_d, A) of the relation R_t shown in Equation (1): Product ID p_ID; Addition Time s_time; Deletion Time e_time; Department dep_id and Property $property$. In addition, we add the following properties for the temporal update. Data Time d_time shows the update order of the data as shown in Section 2.2; Process Class p_flg show the classification of the update process of the data shown as "Process" column in Fig. 3; Deletion Flag del_flg shows the data was deleted if it is set. Here, del_flg is used to exclude all data having the same primary key K from the query result. For example, in the case that the OB update (3) is deletion in Fig. 1, the other data (1) and (2) also not have to be queried. For this purpose, we exclude the unnecessary queried data (3) after the query, and as a result, the other data is also not queried.

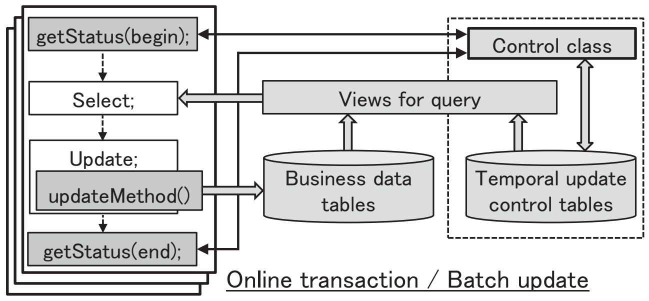


Figure 6: Composition of temporal update.

Second, the concurrency control between the online entries and batch update is necessary as shown in Fig. 4. For this purpose, we implemented the Control class by the Java to perform this control as shown in Fig. 6. It exposes a method $getStatus$ that is the interface of business programs, and the programs call this method at the timing of the start and completion each of the online entry transaction and batch update. Then, the updating of the control tables and the control about the serialization are performed by this method.

In the case of the batch update, $getStatus(begin)$ is called at the timing of start to set the control data of the batch update to $d0t\_batch$ of all related databases. $getStatus(end)$ is called at the timing of its completion similarly, and it sets the completion time to b_time of $d0t\_batch$ and updates $d0t\_commit$. Thus, the batch update result can be queried. In the case of the online entry, $getStatus(begin)$ is called at the timing of its transaction start similarly, and confirms whether or not the batch update updates the target table. If it is not being updated, only the usual online entry transaction is performed. On the contrary, if it is being updated, the transaction's data is set to $d0t\_mask$ by this method. And, the transaction performs the OB update after the online entry, and calls $getStatus(end)$ to delete the data of $d0t\_mask$ at the timing of its completion.

Third, The business programs query the table through its view table, if it is the target of the temporal update. The view table is created by the DDL (Data Definition Language) shown in Fig. 7 and has the following feature.

(a) It changes the addition time T_a of the view table as shown in Fig. 3.

(b) It controls query results of the batch update and OB update by using the time changed in (a): only the data, which Addition Time s_time is older than Update Time b_time of $d0t\_commit$ (including b_time), is queried. Incidentally, every online entry result is queried.

(c) It excludes the data from the query result of (b) by using

```
CREATE VIEW stock_v AS
SELECT p_id, COALESCE(b_time, a.s_time, b.s_time),
   e_time, d_time, p_flg, del_flg, dep_id, property FROM stock a
LEFT OUTER JOIN d0t_batch b USING (s_time)
WHERE p_flg= '1'
   OR (p_flg = '0' AND p_id NOT IN
      (SELECT t_key FROM d0t_mask
         WHERE t_name = 'stock' AND q_time = a.d_time)
      OR p_flg = '2')
   AND COALESCE (b_time, a.s_time, b.s_time) <=
      (SELECT b_time FROM d0t_commit
         WHERE t_name = 'stock');
```

Figure 7: Example of DDL to create view table.

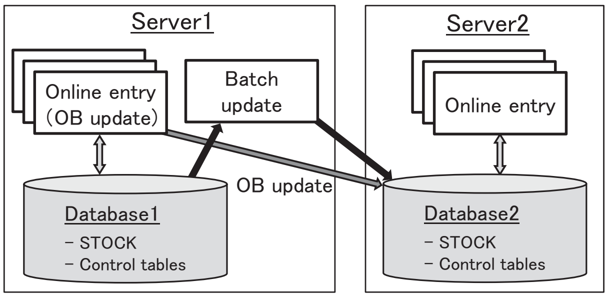


Figure 8: Experimental system composition.

$d0t\_mask$, which is being updated by the online entry transaction.

Using these view table, there are the following effects: since it doesn't allow to query the intermediate state of the temporal update shown in Fig. 4, the ACID properties of the database can be maintained; the developer can build the program efficiently by it, since it conceals the above-mentioned procedure.

On the other hand, these view tables aren't always updatable. So, though it is necessary that the table is updated directly by the business programs, transaction time attributes of the table are not exposed to users: e_time, d_time and s_time of $STOCK$. Therefore the method $updateMethod$ is provided, and the business programs update tables using this method. In addition, the above-mentioned features about databases were implemented by MySQL: InnoDB for the transaction feature; XA transaction for the distributed transaction [5].

5 EXPERIMENTS AND EVALUATIONS

5.1 Overview of Experiments

To evaluate the proposal method, we built the prototype of a stock management system, and performed experiments by it. We show the composition of the experimental system in Fig. 8. This system is a distributed system consisting of two servers, and each server has its own database. Its business table is the stock management table shown in Fig. 5. And, we assume this system is a non-stop service system. That is, its data can be deleted by the online entry transactions at any time when the corresponding goods are sold.

Data	Time (Sec)								Remarks
	2	4	6	8	10	12	14	16	
Server1 (Local)									
R010	●	●	●						Sold
R011	●	●	●	●	●				Sale canceled, Move out
R020	●	●	●	●	●	●			Sold
R021	●	●	●	●	●	●			Sale canceled, Move out
R100	●	●	●	●	●				Move out
Server2 (Remote)									
R010									
R011						●	●	●	Move in
R020									
R021							●	●	Move in
R100						●	●	●	Move in

Batch update

Figure 9: Query result of proposed temporal update.

The stock movements from $Database1$ to $Database2$ are performed in a lump-sum when necessary. This is built by using the temporal update method, and its batch update results, which is shown by (1) in Fig. 1, are inserted to $stock$ of each database. Among them, as for the data of $Database1$, Deletion Flag del_flg is set not to be queried after the completion of the temporal update. On the one hand, the data is inserted into the table of $Database2$, and they can be queried after the completion time. In addition, in the case that the data of $Database1$ is deleted by the online entry, its movement to $Database2$ has to be also canceled by the OB update. Concretely, as for $Database2$, the data with setting del_flg is inserted by the OB update, so the batch update results are also excluded from the query result.

In the experiment, we performed the online entry in both of the server, and performed the above-mentioned temporal update on its way. In Fig. 9, we show the query result of the typical data along the passage of time. Among the data, $R010$ was sold during the data movement by the temporal update; the sale of $R011$ was canceled, that is, the transaction was canceled by the rollback. As for $R020$ and $R021$, the sale of them continued across the completion time of the data movement, and $R020$ was sold; the sale of $R021$ was canceled. $R100$ was only moved. Also, the black circle shows that the data was queried at the time through the view table in Fig. 7; the blank shows not to be queried;

5.2 Evaluations of Validity of Proposal Method

First, in order to complete the temporal update immediately after the batch update, we set the temporary time "@0" (we show only its second, the same in the following) at the timing of the batch update start. Then, we set its completion time "11" to Update Time b_time of both of $d0t\_batch$ and $d0t\_commit$ at the timing of the batch update completion. As shown by $R011$ and $R100$, we could complete the temporal update immediately after the batch update and query their update result.

Next, for the serialization between the online entry and batch update without the lock feature, we inserted $d0t\_mask$ the corresponding data to the online entry at the timing of its start; we deleted the data at its completion. As a result, during online entry, the batch update results are not queried, but the data before the online entry is queried as shown by $R020$ and $R021$. That is, as for the sold data $R020$, the data before its selling is queried until the completion of the online entry; it can't be queried after the completion. Also, it didn't be moved to $Database2$. As for $R021$ in the case of sale being canceled, since it was moved at the timing of the completion of the online entry transaction, it can't be queried in $Database1$ after the completion; it can be queried in $Database2$. As described above, the temporal update results are queried after the online entry transaction, that is, the serialization between them could be controlled.

5.3 Evaluations of Implementation in Distributed Environment

As for the data movement by the temporal update, its business programs in the destination server could be composed by only local transactions. That is, as shown in Fig. 8, as for Stock Management Table $stock$ in $Database2$, since the existing data in another database is only inserted by both of the batch update and OB update, there is no competition with the online entry. Also, since Control tables are used only inside of the view table and hidden from the above-mentioned programs, these tables could be implemented without influences on the existing programs of $Server2$ except the implementation about the view tables.

On the other hand, as for the business program in the server where the temporal update is performed, the implementations about it were necessary. As for the updating of Control tables, by developing the control class for the temporal update, the business programs could be configured to call its methods at the timing of start and end of the online entry transactions as shown in Fig. 6. So, these tables are hidden from the business programs. However, since the OB update has to be executed as a part of the online entry transaction, it had to be integrated into the corresponding online entry program. By the way, as for the implementation of this method, since the data of $d0t\_mask$ has to be inserted before the online entry transaction start, the transaction of this method had to be executed separating from it. Similarly, in the case of abort, since the rollback of the online entry transaction is executed, the data of $d0t\_mask$ had to be deleted by another transaction. In addition, in the case of success, the deletion could be executed in the same transaction.

As for the batch update, the serialization control between the temporal update and the online entry transactions had to be executed at the start and end of the transaction. On the other hand, since this control wasn't necessary except these timing, the business tables of each database could be updated by the local transactions one after another. In particular, in the case of the number of the updating data was small, its commit was not necessary in the middle of the updating. So, the updating features could be implemented by only using SQL statement without the cursor operations, and we could implement it more efficiently than the mini-batch.

6 CONSIDERATIONS

We found that the temporal update can be completed immediately after the completion of the batch update by this method. As for the temporal update, since the OB update has to be executed as a part of the online entry transaction, its execution time is considered to become long particularly in the distributed environment. Therefore, from the viewpoint of the efficiency, to reduce the execution time of the temporal update is effective.

Incidentally, we consider that the method to reserve the completion time of the temporal update is also useful in both of the centralized and distributed systems. It can be used in the case to change great deal of data in a lump-sum at the designated time and so on. For example, we discuss the case that the share of product of each branch is changed at the prearranged date in Fig. 8. In this case, we execute the temporal update with reserving its completion time at 0 a.m. of the designated date and the stock of each branch office can be updated at once with reflecting the sale earlier.

Also, we found that the serialization between the temporal update and online entry can be composed without the lock feature. It has been pointed out that the long time transactions and the lock feature cause the fault in the distributed systems. So, we consider that the lump-sum update can be composed more secure by this method.

Moreover, in the case of the data movement from one server to another server, we found that we need to implement only the control tables and the view tables as for the latter; the existing business programs are not affected except the implementation about the view table. In particular, even the lock feature for update isn't necessary, because the lump-sum update can be executed by only the insertion of data. That is, this method is valid in the case of data transfer from the specified administration server to the other servers widely.

7 CONCLUSIONS

In IWIN2012, We showed the temporal update method in a centralized database is effective in the viewpoint of maintaining the consistency and updating data efficiently. However, to apply this method to the distributed database, there are problems: it is difficult to estimate its completion time; an online entry wait is spread to the other servers at the completion timing of this method, because the serialization between the batch update and the online entry transactions has to be performed. For these problems, we propose the method in this paper for the following purpose: the beforehand estimation of its completion time becomes unnecessary; the serialization can be executed without the online entry waits. And, we confirmed by experiments that these feature was valid and could be implemented in the distributed database. Moreover, we find through these experiments this method is also valid in the case of data transfer from the specified administration server to the other servers in a wide area.

The future challenge is the evaluation of the operational efficiency and performance in the viewpoint of the business system in order to adopt it to the actual distributed systems.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 24500132.

REFERENCES

[1] J. Gray, and A. Reuter, "Transaction Processing: Concept and Techniques," San Francisco: Morgan Kaufmann (1992).

[2] T. Kudo,, et al., "A batch Update Method of Database for Mass Data during Online Entry," Proc. of 16th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES2012), pp. 1807–1816 (2012).

[3] T. Kudo, et al., "Evaluation of Lump-sum Update Methods for Nonstop Service System," Proc. of Intenational Workshop on Informatics (IWIN2012), pp. 3–10 (2012).

[4] P.M. Lewis, A. Bernstein, and M. Kifer, "Databases and Transaction Processing: An Application-Oriented Approach," Addison-Wesley (2001).

[5] ORACLE, XA Transactions, http://dev.mysql.com/doc/refman/5.1/en/xa.html.

[6] M.T. Özsu, and P. Valduriez, "Principles of Distributed Database Systems," Springer (2011).

[7] U. Shanker, ,M. Misra, A.K. Sarje, "Distributed real time database systems: background and literature review," Distributed and Parallel Databases, Vol. 23, Issue 2, pp. 127–149 (2008).

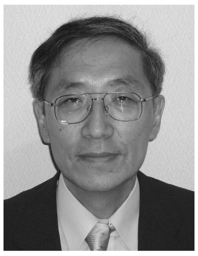
[8] R. Snodgrass, and I. Ahn, "Temporal Databases," IEEE Computer, Vol. 19, No. 9, pp. 35–42 (1986).

[9] B. Stantic, J. Thornton, and A. Sattar, "A Novel Approach to Model NOW in Temporal Databases," Proc. of 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic, pp. 174–180 (2003).

[10] 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).

[11] J. Yang, I. Lee, O. Jeong, S. Song, C. Lee, and S. Lee, "An architecture for supporting batch query and online service in Very Large Database systems," Proc. of IEEE International Conference on e-Business Engineering (ICEBE '06), pp. 549 – 553 (2006).

(Received December 2, 2013)
(Revised March 4, 2014)



Tsukasa Kudo received the M. Eng. 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.

Yui Takeda received the B.E. from Keio University, Japan in 1987. In 1987, she joined Mitsubishi Electric Corp. She was an engineer of artificial intelligence and application software. Since 2001, she joined Mitsubishi Electric Information Systems Corp. Now, she manages intellectual property rights.

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. Since 2009, he is a professor of Fukui University of Technology. Now, His research interests include Management Information Systems, Ubiquitous Systems, Application Systems of Datamining, 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.