Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 22 (2013) 502 – 511

Procedia
Computer Science

17th International Conference in Knowledge Based and Intelligent Information and Engineering Systems - KES2013

A Mass Data Update Method in Distributed Systems

Tsukasa Kudo[a,*], Yui Takeda[b], Masahiko Ishino[c], Kenji Saotome[d], Nobuhiro Kataoka[e]

[a]*Shizuoka Institute of Science and Technology, 2200-2, Toyosawa, Fukuroi-shi 437-8555, Japa*
[b]*Mitsubishi Electric Information Systems Corp., 325, Kamimachiya, Kamakura-shi 247-8520, Japan*
[c]*Fukui University of Technology, 3-6-1, Gakuen, Fukui City 910-8505, Japan*
[d]*Hosei University, 2-17-1 Fujimi, Chiyoda-ku, Tokyo 102-8160, Japan*
[e]*Interprise Laboratory, 1-6-2 Tateishi Fujisawa, Kanagawa 251-0872, Japan*

Abstract

Today, distributed business systems are used widely, and the consistency of their databases are maintained by the distributed transactions. On the other hand, a great deal of data is often updated in a lump-sum in the business systems. And, since the non-stop service has become general, it is necessary to perform this lump-sum update concurrently with the online transactions that service to users. So, some methods are utilized to avoid the influences on the online transactions, like the mini-batch that splits a lump-sum update into small transactions and executes them one after another. However, in the distributed systems, since it has to be executed by the distributed transactions, there is a problem on the efficiency to update a great deal of data by this method. For this problem, we propose to apply an update method to the distributed systems, which utilizes the records of data about the time to avoid conflicts between the lump-sum update and the online transactions. Moreover, through experiments using a prototype, we confirmed that it can update data more efficiently than the chain of small transactions even in the distributed systems.

© 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
Selection and peer-review under responsibility of KES International

Keywords: database; distributed system; batch processing; mini-batch; distributed transaction

1. Introduction

With the spread of the Internet and distributed processing technologies, the decentralization of the business systems in corporations is developing [7, 15, 13]. At present, it has also spread to the various Web services, such as net shops [1, 11]. For example, a corporation having branch offices in each region introduces a server to each office, and the system operations adapted to the business of the individual office is performed. In this case, each server has the database for each office business to reduce its communication cost, and the databases of other offices are accessed in order to perform comprehensive operations only when it is needed. In such a decentralized system environment (hereinafter, "distributed system"), some distributed processing technologies are introduced: the distributed transaction to control the updates across multiple databases and maintain their consistency; the

*Corresponding author. Tel.: +81-538-45-0201 ; fax: +81-538-45-0110.
E-mail address: kudo@cs.sist.ac.jp.

1877-0509 © 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
Selection and peer-review under responsibility of KES International
doi:10.1016/j.procs.2013.09.129

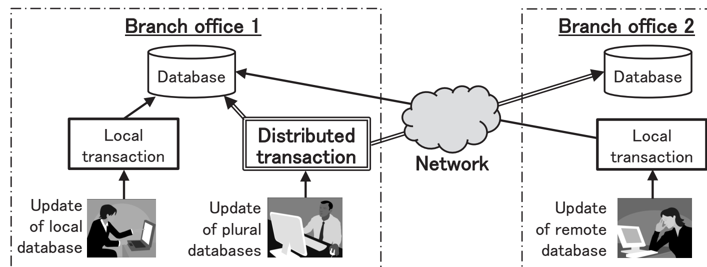


Fig. 1. An example of distributed system.

replication to improve the efficiency of the query at each office by placing replicated databases of the master database in each server [10].

On the other hand, often a great deal of data is updated by the lump-sum update. However, today, nonstop services have become to be used widely because of the spread of the internet business, the improvement of the convenience for users and so on. For example, in banking systems, ATM (Automatic Teller Machine) is provided for users as a nonstop service, and the entry data is reflected in the database by the transaction. That is, it is necessary to execute the lump-sum update processing without affecting the transactions of nonstop services [14]. So, for example, it is executed as the mini-batch that performs a great deal of update as a chain of small update transactions [2]. However, the distributed transaction is necessary for the distributed system. So, there is a problem that it is inefficient, because it is accompanied by access control over the network [13]. Similarly, the replication has a problem that it takes a long while to reflect the data, when a great deal of data is updated [5, 6].

Here, we had proposed a method to update a great deal of data in a lump-sum without stopping online services for centralized systems (hereinafter, "temporal update") [3]. This method uses an extended transaction time database [8] to avoid the conflict between the lump-sum update processing and the transactions that process the entries of users from online terminals (hereinafter, "online entry"). Concretely, in this method, the former updates the data at the future time, though the later updates the data at the present time. Therefore, since it can commit plural updated data collectively in the lump-sum update processing, we expect that this method can be executed efficiently even in the distributed systems.

Our goal in this paper is to examine the efficiency of this method to update a great deal of data in a lump-sum in the distributed systems. In this study we composed a prototype to evaluate it. And, our empirical results show that our method can execute a lump-sum update much more efficiently than the mini-batch even in the distributed systems. The rest of this paper is organized as follows. In section 2, we show the problem of the lump-sum update in the distributed systems, and the related study. In section 3, we show an implementation of the temporal update in a distributed system. In section 4, we show the evaluations of its efficiency based on experiments, and we show our considerations in section 5.

2. Conventional Update Methods and Related Study

2.1. Problem of conventional update method in distributed systems

In Fig. 1, we show an example of the business system that is constructed as a distributed system. The database of each branch office stores the data of the office, and users update the database of their office or another office depending on their business. Such an update is processed by the update transaction for the single database (hereinafter, "local transaction"). On the other hand, the processing to update the plural databases of the branch offices simultaneously has to be processed by the distributed transaction, which has the function such as two-phase commit, to maintain the integrity of these databases. For example, in the inventory management system constructed as the distributed system as shown in Fig. 1, there is the processing to move the stock from some office to the other one. In this case, a single distributed transaction executes both of the processing for this stock

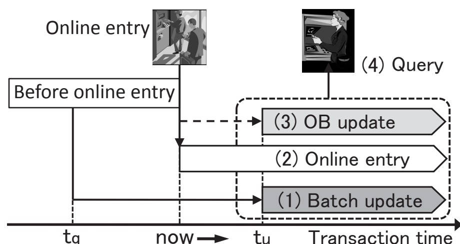


Fig. 2. Lamp-sum update method using transaction time.

movement: reduction from the database of some office; addition to the database of the other one. So, the integrity of the stock amount as a whole system is maintained.

Here, the database of the business system is often updated as a lump-sum update of a great deal of data. For example, as for the above-mentioned stock movement, when a great deal of stock is moved among the offices in a lump-sum, a great deals of lump-sum update must be executed to the plural databases at the same time. In the past, a great deal of lump-sum update was executed efficiently as the night batch to occupy the target tables, except the time zone of online services. However, at present, since the nonstop service is provided widely, it is necessary to execute a great deal of lump-sum update concurrently with the online entries.

For this process, some methods are put to practical use. For example, there is a mini-batch to execute the lump-sum update as the chain of the divided small transactions. However, in the distributed systems, it is executed as a chain of a great deal of distributed transactions, so there is a problem on the efficiency. Moreover, as for the mini-batch, since the update is committed one after another, there are the following problems: in the case of failure, it isn't possible to be restored at the start point of the update [12, 13]; it cannot maintain the integrity in the case of updating the data related mutually [4].

2.2. Related study

We have proposed the temporal update, which is the update method for a great deal of lump-sum update for the single database update. And, its table has a relation that is extended from the relation of a transaction time database table. The relation R_t of the transaction time database table is expressed by the following.

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

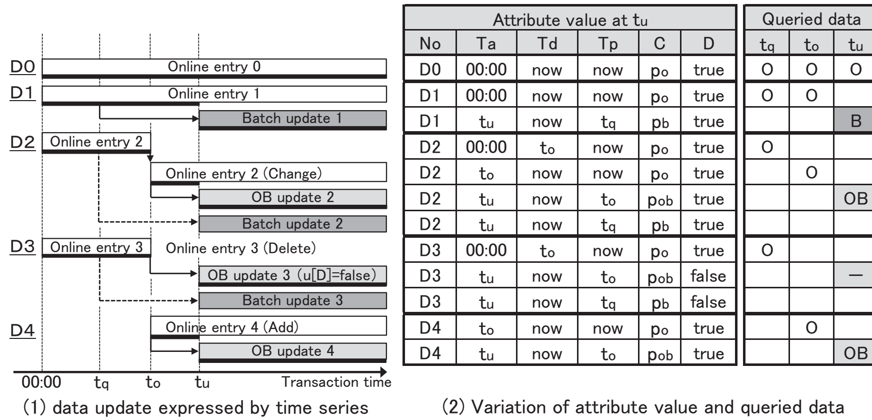
Here, T_a shows Addition Time that the data was added to the database, T_d shows Deletion Time that the data was deleted from the database. Even in the case of data deletion, since the data is deleted logically by setting the deletion time, the record of this data is left. Incidentally, while data isn't deleted, the value of attribute $q[T_a]$ is expressed by *now*, the current time of update or query. Here, $q[T_a]$ shows the value of attribute T_a of data q. And, it is changing with the passage of time [9]. Therefore, the data set of the snapshot at the time t is expressed by the following $Q(t)$. And, above-mentioned K shows its primary key attributes; A shows the other attributes.

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

Fig. 2 shows the change of data by the time series in the case of database update by this method. First, the batch update (1) queries the data at the past time t_q for its update, and stores its update result at the future time t_u. Second, the online entry (2), which is executed concurrently with the batch update, updates the data at the current time *now*. So, these times are different, there is no conflict between the batch update and the online entry. On the other hand, it is necessary to execute the batch update individually on the result of the online entry. This process is executed as the OB update (online-batch update) (3). Thus, in this method, the lump-sum update is performed by both of the batch update (1) and OB update (3), though it is performed by only the batch update (1) in the conventional method. Here, the relation of the target table is expressed by the following R_e extended from R_t.

$$R_e(K, T_a, T_d, T_p, P, D, A) \tag{3}$$

The attributes that were added for R_e are as follows



Attribute value at t_u						Queried data		
No	Ta	Td	Tp	C	D	t_q	t_o	t_u
D0	00:00	now	now	p_o	true	O	O	O
D1	00:00	now	now	p_o	true	O	O	
D1	t_u	now	t_q	p_b	true			B
D2	00:00	t_o	now	p_o	true	O		
D2	t_o	now	now	p_o	true		O	
D2	t_u	now	t_o	p_{ob}	true			OB
D2	t_u	now	t_q	p_b	true			
D3	00:00	t_o	now	p_o	true	O		
D3	t_u	now	t_o	p_{ob}	false			—
D3	t_u	now	t_q	p_b	false			
D4	t_o	now	now	p_o	true		O	
D4	t_u	now	t_o	p_{ob}	true			OB

Fig. 3. Queried data about each update case.

- T_p**:** Data Time. It is the time to show the processed order of update data: as for the batch update, its query time t_q is set; as for the online entry and OB update, the addition time of the online entry data $q[T_a]$ is set.
- P**:** Process Class. This shows the process that updated the data: the OB update and the online entry, the batch update. The corresponding value set is expressed by $\{P_{ob}, P_o, P_b\}$. Here, we make $P_{ob} > P_o > P_b$.
- D**:** Deletion Flag. This shows whether the queried data is the target of query. So, it is the logical value set $\{true, false\}$. In the case that the online entry (2) delete the data, the OB update data q is set to $q[D] = false$. In this way, the batch update (1) of Fig. 2 is made not to be queried even after time t_u. Otherwise, it is set to $q[D] = true$ and queried.

After the time t_u of Fig. 2, since there are plural data q having same above-mentioned primary key value $q[K]$, we extract valid data by the following algorithm.

(a) We extract the data with the latest value of Data Time T_p.
(b) If there is plural data, we extract the target data by the value of above-mentioned Process Class P.
(c) If the value of the Deletion Flag D of this extracted data is $false$, we exclude it from the query result.

Fig. 3 shows the query results by this way. (1) of this figure shows the update situation of the data by the time series: as for $D0$, there is no update after the data entered; as for $D1$, a batch update was executed between time t_q and t_u; As for $D2$, $D3$, $D4$, the change, deletion and addition of data was executed respectively by the online entry, and with this, the OB update is executed. (2) of this figure shows the attribute values at the time t_u, and the queried data at the time t_q, t_o and t_u. According to the above-mentioned algorithm, following data is queried: as for $D0$, the online entry data; as for $D1$, the batch update data; as for $D2$, $D3$ and $D4$, the OB update data because of the execution of the online entry during the batch update. However, as for $D3$, since the data was deleted by the online entry, Deletion Flag of it was set to $false$ and no data is queried. In this way, batch update is executed without confliction with the online entry, and the result data can be queried.

In the single database environment, we had showed that the lump-sum update by this method could reduce its elapsed time to about 1/8 comparing to the mini-batch, which commit was executed by one update [4]. However, in the distributed environment, there is the update in plural database or the processing by the distributed transaction. And, it isn't confirmed that the application of this method to this environment makes these processing efficient. Therefore, in this paper, we implement this method to the distributed system and evaluate the update efficiency compared to the mini-batch.

3. Implementation in Distributed System

In this section, we show the implementation of this method to the inventory management system that is constructed as a distributed system as shown in Fig. 1, and our target processing is the movement of a great deal of

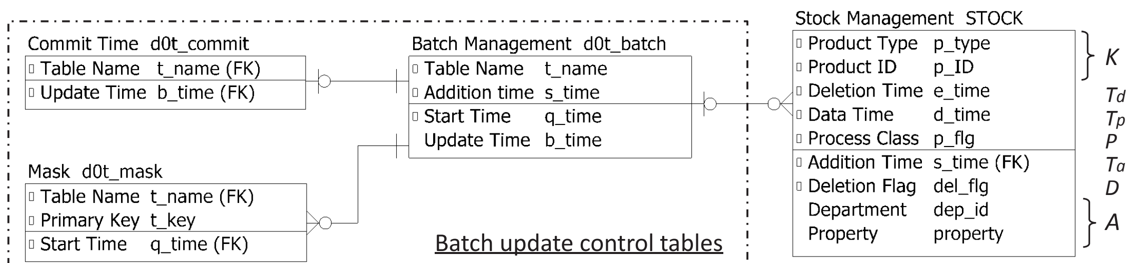


Fig. 4. ER diagram of database tables

stock in a lump-sum during the online entry. In this figure, the distributed transaction corresponds to the lump-sum update; the local transaction corresponds to the online entry. While a great deal of stock is being moved in a lump-sum, the following conditions must be maintained for the online entry transactions: the integrity of the stock data among branch office 1 and 2 is maintained; all products are present in only one of the database in this system. That is, the lump-sum update must be processed as the transaction viewed from the online entry transactions.

3.1. Implementation of tables

In the temporal update, the OB update is executed for the online entry that is executed during the batch update, and the results of the batch update and OB update is made valid when the batch update completes. In other words, these updates are processed as the transaction, and rollback can be executed in the case of failure. For this control, we add the following batch update management tables that are shown in Fig. 4. These exist in every database of the offices in Fig. 1. Incidentally, Stock Management table $STOCK$ is a table for the business and has the relation R_e. We show the corresponding attribute of R_e at the right side of the table in the figure.

- **Commit Time table (*d0t_commit*):** It stores Update Time *b_time* when the results of the batch update and OB update become valid for each target table that name is Table Name *t_name*. That is, it stores the time t_u of Fig. 2. The time is set when the batch update completes. And, since it isn't set in the case of failure, these update results aren't queried even at the estimated completion time of the batch update.
- **Batch Management table (*d0t_batch*):** It stores three kind of time to control the batch update for each target table: Start Time *q_time* of the batch processing, which is same as the query time t_q; Update Time *b_time*, which is same as Commit Table. And, Addition Time *s_time* shows the temporary time that was set to the corresponding time of Stock Management table in the case that Update Time is uncertain at the time of its beginning. In this case, Update Time of this table is updated simultaneously with Commit Time table. On the other hand, in the case that Update Time is established when the temporal update begins, it is set before the processing.
- **Mask table (*d0t_mask*):** It stores Primary Key *t_key* of the online entry to the batch update target table during the batch update, which corresponds to K of R_t. As shown in section 3.2, since the temporal update needs the serialization with the online entry, this table manages the status of the online entry.

3.2. Implementation of temporal update processing

Fig. 5 shows the process flow about the temporal update: (1) shows the processing composition of the batch update; (2) shows the processing composition of the online entry and a OB update; (3) shows the order of the processing by the time series.

In (1), the box expressed by the double line shows the distributed transaction processing, and the other shows the local transaction processing. Also, the hatching boxes show the processing that updates the batch update management tables of Fig. 4. First, before the batch update, the pre-process set the time for the batch update to Batch Management table of all target databases. Since these processing is executed beforehand, they can be executed one after another by the local transactions. Next, the batch updates are executed. Since its results cannot

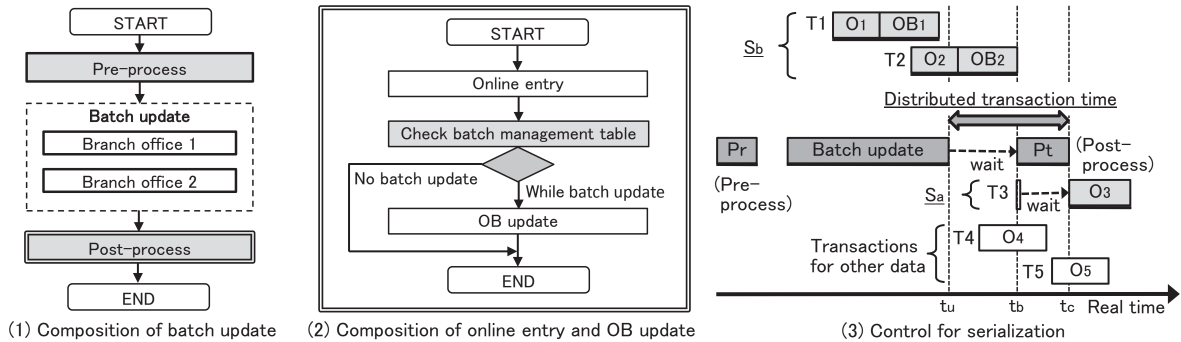


Fig. 5. Process flow of transactions.

```
CREATE VIEW STOCK_v
AS SELECT a.* FROM STOCK AS a, d0t_commit AS b
WHERE TRIM(t_name) = 'STOCK' AND (p_flg = '1' OR p_flg IN ('0', '2') AND s_time <= b_time);
```

Fig. 6. View table of Stock Management table to query the valid data.

be queried by the online entry transactions until the update of Commit Time table, they also can be executed by the local transactions for each database. Lastly, Commit Time table and Batch Management table are updated by the post-process, and the batch update results can be queried from the online entry transactions after this.

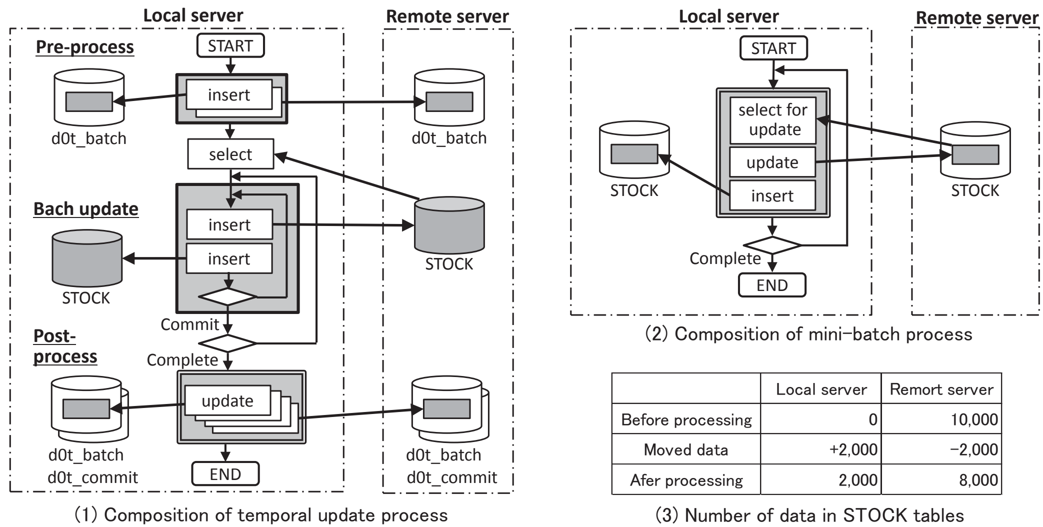
The online entry in (2) is executed by both the kind of transaction: as the local transaction in the case of updating tables of the single database; as the distributed transaction in the case of plural databases. It queries Batch Management table at the beginning. And, in the case during the batch update, the OB update is executed. Incidentally, since the consistency between the both processing results has to be maintained, they are executed as the single transaction.

To execute the whole batch update as a transaction, the serializable schedule must be composed among it and the online entry, which is executed at the same time. In this method, since its update results become able to be queried from the online entry transactions after the post-process *Pt* of the batch update process as shown in Fig. 5, the serializable schedule at this point becomes important. (3) of this figure shows the control flow for serialization by this method. Here, S_b shows the online entry transactions started before the batch update completion time t_u; S_a shows the transactions started after this time and its update data conflicts with the batch update. Here, since the target tables of the batch update are contained in the databases of the plural offices, the serializable schedule is controlled by the following process by the distributed transaction: *Pt* waits until all the transactions of S_b complete; the transactions of S_a wait until the completion of *Pt*.

Here, if it isn't executed by the distributed transaction, each database is committed one after another. So, for example, since the OB update OB_2 complete after *Pt*, the corresponding batch update result is queried as a phantom until the completion of OB_2. Incidentally, the transactions $T4$, $T5$ that update the data other than those that is the target of the batch update can be executed without waiting the completion of *Pt*. In this way, in the batch update processing of the temporal update, only the post-process has to be executed as the distributed transaction; the others can be executed as the local transactions.

3.3. Implementing of query function

To execute the temporal update as a transaction, the results of the batch update and OB update have to be queried after the completion of the post-process, that is, after the update time t_u was set to Commit Time table. Here, since the online entry is executed as another transaction, its result has to be queried even before the time t_u as shown in Fig. 3. We implemented these query function by the following conditions using Process Class *P*.



	Local server	Remort server
Before processing	0	10,000
Moved data	+2,000	−2,000
Afer processing	2,000	8,000

Fig. 7. Composition of experimental environment.

Table 1. Spec of servers.

server type	CPU	Memory	OS
Local serve	Xeon E5-1620 3.6GHz	8GB	Windows7 (64bit)
Remote server	Core i5 3.2GHz	4GB	Windows7 (32bit)

- **Online entry data** ($P = P_o$)**:** it is queried unconditionally.
- **Batch update data** ($P = P_b$) **and OB update data** ($P = P_{ob}$)**:** only the data, which Addition Time *s_time* is newer than Update Time *b_time* of Commit Time table, is queried.

We implemented this feature as the view table. Fig. 6 shows the DDL(Data Definition Language) to make the view table for Stock Management table. By this view table, in the case of failure, rollback can be executed without affecting the online entry. That is, we conceal the results of the batch update and OB update by this view table, and delete them by their compensating transaction. Incidentally, in the actual business, the valid data is extracted from this view table by the query algorithm shown in section 2.2.

4. Experiments and Evaluations

To evaluate the efficiency of the temporal update in the distributed system, we experimented by the prototype as for the case of a great deal of stock movement among the branch offices as shown in Fig. 1. Fig. 7 shows the experimental environment. We built the local and remote servers in the distributed environment by two PCs shown in Table 1, and connected them by a 1Gbit HUB. Stock Management table and the batch update control tables are stored in every server, and their compositions are shown in Fig. 4 respectively. We use MySQL for the database; InnoDB for the storage engine; XA transaction of MySQL for the distributed transaction [16]. To use the distributed transaction, we set the isolation level of the transaction to *serializable*. And, we used Java, and accessed MySQL by using JDBC.

In this experiment, we moved 2,000 stocks from Stock Management table of the remote server, which stores 10,000 stocks at the beginning, to the local server as shown in Fig. 7. We did this process by both the temporal update and the mini-batch, and evaluated the efficiency of the temporal update processing in the comparison with the mini-batch.

4.1. Composition of temporal update processing

(1) of Fig. 7 shows the program composition of temporal update processing, which is executed by 3 steps of transactions. First, the pre-process set the start time of the temporal update to Batch Management table *d0t_batch* before the batch update begins. This processing is executed by the local transactions as shown in section 3.2. Here, it inserts 1 record to each above-mentioned table.

Second, the batch update moves the stocks. It queries the target data from the table of the remote server (select). Next, it inserts the corresponding OB update data into this table, which Deletion Flag is set to $false$ as shown by $D3$ in Fig. 3. In this way, these queried data become to be deleted logically at the completion time. On the one hand, it inserts the target data into the table of local server as shown by $D1$. Since it also doesn't need the synchronization among both of the database, it is executed by the local transactions and committed for each specified number of updates. Incidentally, since the transaction time to query the data is the past time, the data isn't updated by the online entry as shown in Fig. 2. Therefore, the data can be queried in discrete. In this experiment, 100 data is queried at a time.

Last, the post-process updates Batch Management table *d0t_batch* and Commit Time table *d0t_commit*. In this processing, since the serialization between the online entry and the batch update is necessary as shown in Fig. 5, it is executed by the distributed transaction.

4.2. Composition of mini-batch processing

(2) of Fig. 7 shows the program composition of the mini-batch processing. The mini-batch competes with the online entry. So, it is not only executed by the distributed transactions, but also its data has to be processed one after another: each data is queried with lock mode (select for update), updated and committed. Incidentally, in this experiment, since we used Stock Management table that has the same composition with the temporal update, the data is deleted logically in the same way as the transaction time database. That is, the data queried from the Stock Management table of the remote server is deleted logically by setting Deletion Time, and the data is added to the Stock Management table of the local server.

In further details, in the mini-batch processing, the following distributed transaction is repeated: above-mentioned processing is executed on one data, which is query, update and insertion; next, the prepare and commit of the two-phase commit are executed.

4.3. Evaluations

(1) of Fig. 8 shows the elapsed time of the temporal update processing in the case that the commit is executed for each 200 updates; (2) shows the elapsed time of the mini-batch processing. This elapsed time is the average time of the result of measuring each processing five times. The elapsed time to move 2,000 data by the mini-batch is about 120 second. On the other hand, the elapsed time by the temporal update is less than 2 second. Therefore, it is reduced to about 1/60.

Here, in the temporal update, the elapsed time of the pre-process shown between "START" and "Pre" was about 0.2 second as shown in (1); the post-process shown between "2000" and "Post" is about 0.1 second. Incidentally, the pre-process contains the connection processing to the database. So, the elapsed time for the database update is shown between "Pre" and "2000". On the other hand, in the mini-batch, since there isn't the pre-process nor post-process, the whole elapsed time is spent for the data update. In both the method, the elapsed time nearly increases in proportion to the number of update data as shown in Fig. 8.

As for the temporal update, the number of the update data for one commit can be specified. Fig. 9 shows the change of the elapsed time with this number. The elapsed time is the same average as Fig. 8. Also, the temporal update, which number of data for the commit is 200, and the mini-batch correspond to Fig. 8. In Fig. 9, the temporal update, which number for the commit is 1, executes the commit for each update similar to the mini-batch. However, the batch update processing is executed by the local transaction and query 100 data at a time. And, this elapsed time was about 58.3 seconds, that is, about 1/2 of the mini-batch processing.

As for the temporal update, the whole elapsed time became shorter with increasing the numbers of update for one commit. In the case of 200, the elapsed time is about 1.9 second, and was reduced to about 1/30 compared to the case of 1; in the case of 1,000, about 1.4 second, and to 1/40. That is, in the case to commit for each of 1000

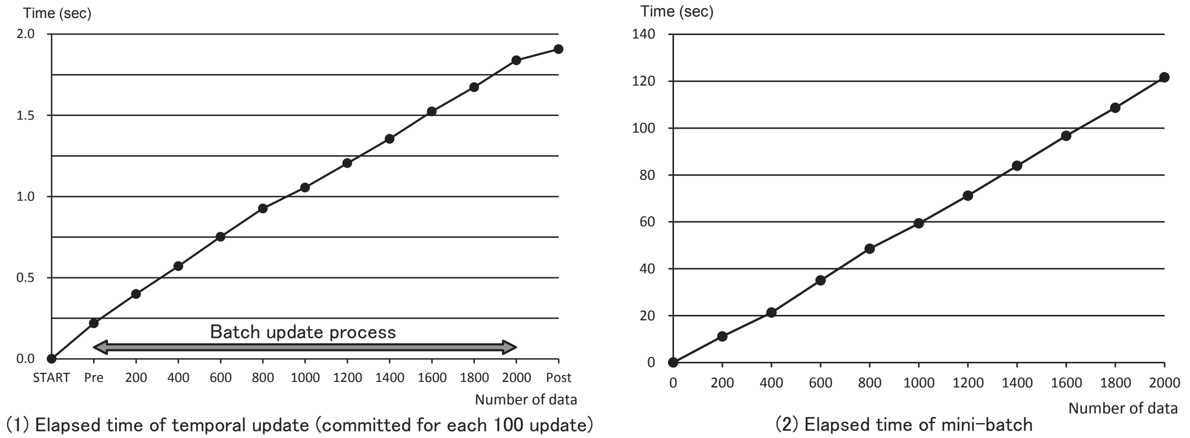


Fig. 8. Evaluation of elapsed time.

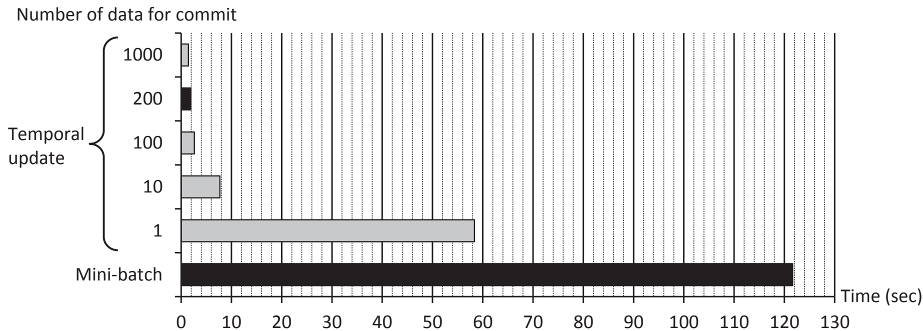


Fig. 9. Comparison of elapsed times.

update in the temporal update, its efficiency is more than 80 times compared to the mini-batch. Incidentally, as for the pre-process and post-process processing, since they don't depend on the number of update data, their elapsed time was almost constant even by changing this number for the commit.

5. Considerations

The mini-batch has to be executed as the chain of the small distributed transactions as shown in (2) of 7. On the other hand, as for the temporal update, it needs to synchronize only in the post-process by the distributed transaction as shown in (1) of Fig. 7. That is, the processing of the batch update can be flexibly composed. It is considered that the temporal update is effective in the aspect of the update efficiency, the development efficiency and the safety of system operations.

First, as for the update efficiency, its update processing can be executed by the local transaction. For example, as shown in Fig. 9, even in the case that the commit is executed for each of update, the elapsed time is reduced to about 1/2 comparing to the mini batch. And, in the case that the commit is executed for each of 1,000 update, the elapsed time is reduced to about 1/80. Here, the number of the data and the elapsed time are proportional. Therefore, we consider that the temporal update is effective to update a great deal of data in a lump-sum.

Second, as for the development efficiency, the flow of the update processing of the temporal update can be flexibly rearranged. For example, we updated each database individually by the local transaction in this experiment. In the actual business processing, since there are the updates with the complicated procedures, they can be

composed more simply by the temporal update: by dividing of the lump-sum update into simple updates or by rearranging of the process flow.

Third, as for the safety of system operations, the data being updated by the temporal update cannot be queried by the online transactions until the completion of the post-process. In the actual system operations with the lump-sum update, the failures often occur. As for the mini-batch processing, since its update is immediately committed, the update cannot be cancelled. So, the user has to continue the process until the end with removing the cause of the failure. Here, in the distributed environment, since the processing is executed through the network, the probability of the failure generally increases more than processing in the single server. On the other hand, in the temporal update processing, its update can be cancelled in the case of failure. So, the user can rerun the processing at any time after the disposal of the failure cause.

6. Conclusions

With the spread of the Internet and distributed processing technologies, the decentralization and nonstop services of the business systems are developing. On the other hand, often a great deal of data is updated in a lump-sum in the business systems. To update plural databases with maintaining the consistency among them, the mini-batch is used. However, since it executes a great deal of update as a chain of small update transactions, there is problem that its efficiency is reduced in the distributed environment. In this paper, we applied the temporal update method to a distributed system, which utilized the records by the time series to avoid the conflicts between the lump-sum update and the online entry. And, we evaluated its efficiency compared to the mini-batch. As a result, we found that its elapsed time can be substantially reduced than the mini-batch, since it is executed by the local transaction and commits the plural updates at a time.

Future study will focus on the evaluation in the actual business system environment: updating concurrently with the online entry in the distributed environment; the efficiency of querying the update results.

This work was supported by JSPS KAKENHI Grant Number 24500132.

References

[1] DeCandia, G., et al., Dynamo: amazon's highly available key-value store, Procs. twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07); 2007, p. 205–220.
[2] Gray, J., Reuter, A., Transaction Processing: Concept and Techniques, San Francisco: Morgan Kaufmann; 1992.
[3] Kudo, T., et al., 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; 2012, p. 1807–1816.
[4] Kudo, T., et al., Evaluation of Lump-sum Update Methods for Nonstop Service System, Procs. Int. Workshop on Informatics (IWIN2012); 2012, p. 3–10.
[5] Maalouf, H.W., Gurcan, M.K., Minimisation of the update response time in a distributed database system, Performance Evaluation, Vol. 50, Issue 4; 2002, p. 245–266.
[6] Schwartz, B., et al., High Performance MySQL, Oreilly & Associates Inc.; 2008.
[7] Shanker, U.,Misra, M., Sarje, A.K., Distributed real time database systems: background and literature review, Distributed and Parallel Databases, Vol. 23, Issue 2; 2008, p. 127–149.
[8] Snodgrass, R., Ahn, I., Temporal Databases, IEEE COMPUTER, Vol. 19, No. 9; 1986, p. 35–42.
[9] Stantic, B., Thornton, J., Sattar, A., 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; 2003, p. 174–180.
[10] Tanenbaum, A. S., Van Steen, M., Distributed Systems: Principles and Paradigms. Prentice Hall; 2006.
[11] Thomson, A., Diamond, T., Weng, S., Ren, K., Shao, P., Abadi, D.J., Calvin: fast distributed transactions for partitioned database systems, Procs. the 2012 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '12); 2012, p. 1–12.
[12] Vonk, J., Grefen, P., Cross-Organizational Transaction Support for E-Services in Virtual Enterprises, Distributed and Parallel Databases, Vol. 14, Issue 2; 2003, p. 137–172.
[13] Wang, T., Vonk, J., Kratz, B., Grefen, P., A survey on the history of transaction management: from flat to grid transactions, Distributed and Parallel Databases, Vol. 23, Issue 3; 2008, p. 235–270.
[14] Yadav, D.S., Agrawal, R., Chauhan, D.S., Saraswat, R.C., Majumdar, A.K., Modelling long duration transactions with time constraints in active database, Procs. the Int. Conf. on Information Technology: Coding and Computing (ITOC' 04), Vol. 1; 2004, p. 497 – 501.
[15] Yang, J., Lee, I., Jeong, O., Song, S., Lee, C., Lee, S., An architecture for supporting batch query and online service in Very Large Database systems, IEEE Int. Conf. on e-Business Engineering (ICEBE '06); 2006, p. 549 – 553.
[16] ORACLE, XA Transactions, http://dev.mysql.com/doc/refman/5.1/en/xa.html.