18th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems - KES2014

An implementation of concurrency control between batch update and online entries

Tsukasa Kudo*a, Yui Takeda, Masahiko Ishino, Kenji Saotome, Nobuhiro Kataoka

aShizuoka Institute of Science and Technology, 2200-2 Toyosawa, Fukuroi-shi, 437-8555, Japan
bMitsubishi Electric Information Systems Corp., 325 Kamimachiya, Kamakura-shi, 247-8520, Japan
cBunkyo University, 1100 Namegaya, Chigasaki-shi, 253-8850, Japan
dHosei University, 2-17-1 Fujimi, Chiyoda-ku, Tokyo 102-8160, Japan
eInterprise Laboratory, 1-6-2 Tateishi Fujisawa, Kanagawa 251-0872, Japan

Abstract

Databases of business systems are generally updated by two methods: the online entries reflect the input from a lot of terminals immediately; the batch update updates a great deal of data in a lump. Here, the batch update usually takes time. So, in the case where it is performed as a single transaction by using the lock, the conflicting online entries wait for a long while. For this problem, we had proposed the temporal update method, and showed the both can be executed concurrently by it. However, to process them by a serializable schedule, the execution timing of the online entries has to be divided by the batch update. So, it raises a problem that there are still latencies of some online entries by this. In this article, we propose four concurrency control methods to shorten this wait. And, we show their implementations and evaluation results. Moreover, we show the appropriate method should be chosen based on the operational requirements

Keywords: database update, batch processing, transaction, serializable schedule, concurrency control, nonstop service

1. Introduction

In the business systems, database updates are roughly classified into two methods. One method is used for the input from the online terminals (hereinafter, “online entry”), and they are reflected into the database immediately by short time transactions. For example, the input from many ATM (Automatic Teller Machine) in the banking systems, which are operated concurrently, is processed by the online entry. The other is used to update a great deal of data in a lump (hereinafter, “batch update”). And, since this process often needs a long while, it is processed by the batch processing to semi-automate. For example, a great deal of bank transfer in a banking system is processed by the batch update. In the case that the batch update is processed as a single transaction, it is necessary to lock a great deal data in a long

* Corresponding author. Tel.: +81-538-45-0201; fax: +81-538-45-0110.
E-mail address: kudo@cs.sist.ac.jp
while. Since this makes the online entries wait for a long while, two processing were executed in a different time zone in the past. However, such as ATM and internet shops, since the nonstop services became popular at present, the both have to be executed concurrently. So, the mini-batch is used widely, which performs a great deal of update like the continuous execution of the online transactions.

However, in this method, the ACID properties of the transaction can’t be maintained as the whole batch update. And, as a problem example caused by this, we showed the residence indication, which is the process to change the representation of the address of the designated area at the same time. Here, since the members of one family have the same address, this update is executed in each family. So, there is the case that a resident moves from one family to the other one: the former has been processed; the later hasn’t been processed yet. In this case, his resident card has an inconsistency: the current address is represented as before the residence indication, though the previous address is as after it.

To solve this problem, we proposed a novel batch update method, which is called the temporal update method. It extends the concept of data history into the future, and performs the batch update as future processing to avoid the conflict with the online entry that updates data at the current time. As a result, we showed that the batch update can be executed as a long time transaction concurrently with the online entry. Here, in order to perform the batch update as a single transaction, it needs to compose a serializable schedule with the online entry transactions. That is, as for the online entries that conflict with the batch update, their execution timings have to be divided by the batch update. Therefore, it makes a bottleneck of the execution of the online entries.

Our goal in this paper is to show that the batch update and online entries can be implemented as the efficient serializable schedule in the temporal update method. So, first, we show the wait time and its factor about the online entry. Second, we propose four concurrency control methods, and show its implementations, and evaluate the wait time in it. Finally, we show that the appropriate method should be chosen based on the requirements: tolerance of the wait time and retry as for the online entry; the operational requirement as for the batch update.

2. Related works

As for the concurrency control of the transactions, various kinds of method were proposed. First, as for the locking protocol, which is widely used, a transaction locks its access data beforehand to compose a serializable schedule. So, it makes the transaction, which conflicts with it, a wait state. As a result, in the case of applying to the long batch update, the latency of the online entry becomes a long while. On the contrary, as for the mini-batch, the transaction is divided into short time transactions, and executed sequentially to shorten the individual lock time. However, there is the problem that the ACID properties of the transactions can’t be maintained.

Next, as for the timestamp ordering, a unique timestamp is assigned to each transaction, and each data is also assigned this timestamp when the transaction accesses it. And, if the transaction accesses the data assigned a larger timestamp than this, it is aborted. By this way, serializable schedule is composed. Here, to prevent the cascading abort, the strict timestamp ordering method is necessary, in which the data is updated when the transaction commits. So, if the long time batch update is executed based on the timestamp ordering, some of corresponding data have to be already updated by the online entry in many cases. As a result, the whole batch update is aborted to maintain the atomicity.

Moreover, the multiversion concurrency control based on the snapshot isolation level is implemented in commercial databases. In this method, since the snapshot of the database at the transaction start is given to each transaction, it can execute its process by using the individual snapshots. Therefore, as for the read-only transactions, the long batch processing can be executed concurrently with the online entries. However, as for the update transaction, it needs to verify that the target data wasn’t be updated by the conflicting transactions before its commit. And, in the case of being updated, it has to be aborted. In other word, there is the same problem as the timestamp ordering. In this method, though an exclusive lock can be performed explicitly by such as SELECT FOR UPDATE statement in SQL, it makes the same problem as the locking protocol. Thus, we couldn’t find the method that executes the long batch update as a single transaction stably.

To solve this problem, we proposed the temporal update method, which uses the concept of the transaction time database that is one of the temporal databases. Here, the transaction time is the time that a fact existed in the database, and the data operations such as insertion, change and deletion are managed as the history of time. In this method, we
extend the concept of the transaction time into the future to avoid the conflict between the batch update and the online entries.

As shown in (1) in Fig. 1, in this method, even if both of the online entry and batch update are executed at time $t_0$, the batch update queries the past data at time $t_q$ and saves the update result as the future data at time $t_u$. In addition, the batch update is executed for the online entry data individually, and its result is also saved at time $t_u$. We call this process “online batch update” (hereinafter, “OB update”). As a result, the both processes don’t conflict, and all of their results are saved when time is $t_q$.

(2) in Fig. 1 shows the data history saved in the database table, and the queried data at the time of each $t_q$, $t_o$, $t_u$. Here, the notation $O$, $B$, $OB$ shows the online entry, the batch update, the OB update respectively. That is, by querying only the valid data, we can obtain the similar result to the case of executing them sequentially. The relation of the table is as follows. Incidentally, as for this case, we assume the balance of the account in the financial institute. So, $K$ and $A$ shows the account number and balance respectively.

$$R_e(K, T_o, T_d, P, D, A)$$ (1)

- $K$: Primary key attributes. It is the primary key of the projection ($K, A$), which is the attributes for business.
- $T_o$: Addition time. It shows the time when the data was added to the database.
- $T_d$: Deletion time. It shows the time when the data was deleted from the database. Here, since the deletion is performed logically by setting the deletion time to $T_d$, the data history remains. Incidentally, while data isn’t deleted, the time is expressed by now that is the current time and changing with the time passage.
- $P$: Process classification. This shows the process that updated the data: the OB update, the online entry, and 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 the query. So, it has the logical value: $\{true, false\}$. And, if $D$ is $false$ then the data doesn’t be queried. In the case of $D_3$ in Fig. 1, since $D$ of the OB update data is set to $false$, “Batch update 3” isn’t queried even after time $t_q$.

3. Concurrency control between batch update and online entries in temporal update method

3.1. Latency of online entry operations

In order to execute the batch update and the online entry as a serializable schedule in the temporal update, the concurrency control of both is required. Fig. 2 shows a schedule example of them. Here, $T_i$ shows the online entry transaction, in which $O_i$ is the online entry, $OB_i$ is the OB update. That is, they are processed in a single transaction, and the result of the OB update can be queried after $c$ executed as shown at time $t_u$ in Fig. 1.

Here, above-mentioned schedule is conflict equivalent to a serial schedule, which is composed of the online entries and batch update. Here, we express the batch update by $BU$; the online entry before it by $O_j$; one after it by $O_k$. Then,
that there are no conflicts among the online entries. Here, since the order of all the operations is same to equation (2), the schedule is conflict serializable. Incidentally, if the operation \( r_b \) and \( w_b \) don’t conflict with the operation of any \( O_j \), the batch update result about them is queried, instead of the OB update.

Next, to examine the latency of \( O_k \), we show the precedence graph of Fig. 2, in (1) in Fig. 3. This case assumes that there are no conflicts among the online entries. Here, since \( c \) waits for the completion of all the preceding \( O_j \), it becomes a bottleneck. However, it waits only the transactions that have already started when the batch update completed, like \( T_3 \) in Fig. 2. Moreover, \( O_k \) starts when \( c \) completes. Therefore, let \( \text{max\_time} \) is the maximum elapsed time of the operation, which is a function of the transaction. Then the maximum latency of \( O_k \) is \( \text{max\_time}(O_j + OB_j + c) \). That is the following total time: the longest elapsed time of the online entry transaction before \( c \); the commit of the batch update.

On the other hand, the online entry operations actually conflict with each other. We show this example in (2) in Fig. 3. Let \( T_3, T_x, T_y \) are the above-mentioned transactions. Then, in the case that they are executed in the above order, \( T_x \) waits for the completion of the \( T_x \) by the lock, and \( T_y \) waits for \( T_3 \). So, in this case, if these transactions started before the completion of \( BU \), \( c \) waits all the completion of them. Therefore, let \( N \) is the maximum number of conflicts among the online entry transactions at the same time, then the maximum latency of \( O_k \) is \( \text{max\_time}(O_j + OB_j) \times N + \text{max\_time}(c) \).

3.2. Proposal of concurrency control method for temporal update

The latency of the online entry in the temporal update can be considered from the viewpoint of the conflict with the batch update. As for the actual business systems, the method, which aborts a transaction as the victim to retry, is adopted for the conflict as shown in Section 2 or for the deadlock. Here, requirements for the actual online entry depend on the system operations. For example, it is not allowed to make the large latency of the ATM being used by many users, also to abort the operation dealing with money. On the other hand, it is allowed to make the latency and to retry, when user is looking for tickets and hotels in various conditions to reserve via the Internet.

From the viewpoint of the priority of the transaction, we show four concurrency control methods in Fig. 4. As for (a), it prefers the online entry: the batch update verifies before its commit that there isn’t the executing online entry conflict with it; if there is such a transaction, it is aborted and retried. On the contrary, as for (b), it prefers the batch update: the online entry verifies before its commit that the conflict batch update hasn’t been committed yet; if it has been committed, this online entry is aborted and retried. As for (c), it intends to execute all of them, and it corresponds to Fig. 2: no transaction is aborted, though there is the latency made by the serialization. As for (d), the restriction about (b) is relaxed: \( c \) is executed behind the designated time \( t_w \) from the completion of the batch update. Here, \( t_w = t_b - t_u \) in Fig. 4. So, the probability that the transaction is aborted becomes smaller than the case of (b). The maximum latency time of \( T_a \) of each case is as follows: as for (a) and (b), it is \( \text{max\_time}(c) \); as for (c), it is \( \text{max\_time}(O_j + OB_j) \times N + \text{max\_time}(c) \) as shown in Section 3.1; as for (d), it is \( \text{max\_time}(c + t_w) \).
4. Implementation of concurrency control between batch update and online entries

We build a prototype to examine the implementation of the concurrency control methods shown in Fig. 4, and evaluate its efficiency. As for the business of this prototype, we take up the case of the balance of the account shown in Fig. 1. In addition, we use Java, MySQL for the database, and InnoDB for the transaction processing, which is a database engine of MySQL.

4.1. Implementation of concurrency control methods

We show the overview of the composition of the prototype in Fig. 5. (1) shows the composition of the application software, and it is composed of application program, the table of the database, the Control class for the concurrent control, and view table for query. Here, we show the batch update program as the example of application program,
so the methods of Control class is also as for the batch update. We will show the program composition of the online entry in Fig. 9 of the latter part. (2) show the ER diagram of the tables: Bank Account is the business data table; Commit Time is the table to control the view table. Account Number and Balance of Bank Account table are the attributes of the business data corresponding to A and K in equation (1) respectively. The other items are the same in equation (1). Also, Commit Time table manages the time about the batch update for each business data table. Here, Update Time is the batch commit update time \( t_b \), and its value is null until the commit. (3) shows the class diagram, and attributes of Control class are as follows: cMode is the classification of the concurrency control shown in Fig. 4; status is the execution status of the batch update; cCount is the number of all the conflicts occurring between the online entry transactions and the batch update, privCount is the same name as for the each online entry transaction. additionTime is the Addition Time of the batch update results.

The execution state of the batch update is transferred as shown in Fig. 6 by the methods of Control class. Here, before shows the initial status: the batch update hasn’t been executed or has completed successfully. bPrepare method transfers the status to prepare. Here, to detect the conflict with the online entry, it initializes the instance of Control control, and set the key information about the batch update data. In this implementation, this information is composed like the key range lock: it has the minimum and maximum values of plural key range. Next, bBegin method transfers to batch updating status, and starts the batch update. When the batch update completed successfully, bCommit method is called. It sets the current time to Update Time \( T_b \) of Commit Time table, then the update results can be queried. And, it transfers the status to before. Incidentally, in this status, even if the online entry transaction completes, the value of cCount isn’t reduced to watch the existence or non-existence of the conflict with the online entry through the whole batch update.

In the case of abort, bRollback method is called. It transfers to the abort status, and deletes all the batch update and OB update results. bPrepare method transfers from abort to prepare status in the same way as before status. These transfers have to be executed exclusively from the transferences of the online entry: begin transaction, commit, and rollback. So, we implemented these transition functions including the online entry by getStatus method, using the synchronized method of Java. In addition, since the concurrency control method is specified in bPrepare method, we can select it for each batch update.

On the other hand, as for the online entry, it doesn’t use prepare method, so we implemented its transition by the following methods: begin, commit, rollback and commitOB. The three of the previous correspond to the following methods of transactions respectively: begin transaction, commit, and rollback. Here, begin method performs only the detection of the conflict with the batch update, and increases the value of cCount and privCount, of Control class. Also, commit verifies the presence or absence of conflict with the batch update. And, it starts the OB update in the case of conflict; it commits in the case of no-conflict. The OB update is committed by commitOB method to prevent the overlapping execution. In addition, in the case of (c) and (d), it waits for the commit of batch update in begin method, when it is executed first.

Here, the control of the transition, based on the concurrency control method shown in Fig. 4, is executed by the following methods: (a) is executed by bCommit; (b), (c) and (d) by commit (including commitOB). First, as above-mentioned, cCount doesn’t decrease after bBegin was executed. So, the case of cCount > 0 at bCommit means the online entry was executing at bBegin or was started after bBegin. Therefore, the operation of (a) can be realized by determining abort or commit, responding to the value of cCount in bCommit method. Incidentally, the online entry begins before bPrepare is aborted by commit method as shown in the following.

Next, the online entry transaction needs to determine its action corresponding to the presence or absence of a conflict with the batch update: commit; abort; start of its OB update. So, we composed the online entry transaction to control its action according to the state of the batch update status, in begin and commit method. We show this control in Fig. 7. In this figure, it is status at begin, which is described along with the number from (1) to (4) on top of each table. And, first row of each table shows status at commit. Here, “updating” shows batch updating. The column labeled “time” shows the version of additionTime \( T_a \) of Control class at commit, which is updated every batch update: \( T_a[n] \) shows the value at begin is the current version; \( T_a[b] \) before; \( T_a[b] \) prior to last. Each cell shows the action corresponding to status and this version. For example, as shown in (1), if status of the title (at begin) is before and status of the label (at commit) is prepare, a new batch update is preparing. So, commit method is performed in the case that “time” is before version, rollback method in the case of the others.
4.2. Data records and implementation of view table

We implemented the view table about Bank Account in SQL shown in (1) in Fig. 8. This view table returns only valid data at current time as a query result from the data history due to the condition shown in Fig. 1. (2) in Fig. 8 shows an example of data histories and their query results of this view table. In this case, the time is displayed as the date for the sake of simplicity.

As for the batch update, its completion time, that is, the addition time $T_a$ of the batch update and OB update is undecided at the start. So, as shown by Bank Account table in (2), $T_a$ is set to the start time which first digit is replaced by “@”. Here, since “@” is larger than the number as the dictionary code, it is treated as a future time in the case of time comparison. In order to obtain these data as the query result after the commit of the batch update, this table is joined with Commit Time table by $T_a$, as shown in (1). Thus, $T_a$ is replaced with the completion time of the batch update $T_b$.

Therefore, as for the batch update and OB update results of Bank Account table being set $T_a = @0140302$, the OB update result ($Balance = 200$), which is the valid data, is queried. On the other hand, as for the data being set $T_a = @0140310$, their commit doesn’t complete, and $T_b$ of Commit Time table is still set to null. Therefore, the batch update and OB update results doesn’t be queried, but the online entry result ($Balance = 1,100$) is queried.

5. Experiments and evaluations

5.1. Composition of experimental system

In the experiment, we prepared the bank account data having account number from 1 to 10,000, and performed the following process concurrently: the replacement of data between accounts as the batch update; the account transfer
between accounts as the online entries from five terminals. Here, the online entry transactions conflict with the batch update, but they don’t conflict with each other. And, they are composed to be executed in a constant time roughly. We show the online entry program in Fig. 9. A wait of 0.3 seconds is placed after each SELECT statement, and another wait, which length is calculated by multiplying 0.3 second by the uniform random numbers of 0 to 1, is placed after its commit and abort to change the timing of the execution. In addition, in the case of abort in (b) and (d) in Fig. 4, its retry is executed after the execution of the rollback method as shown in (2).

Concurrent executions of these programs were implemented by the thread class in Java. And, these experiments were performed on the stand-alone workstation with CPU Xeon E5-1620 (3.6GHz), 8 GB memory, Windows 7 (64 bit), MySQL 5.1.40 and Java 1.7.0_25.

5.2. Experimental results

First, in order to obtain the basic data to be compared with the experimental results, we experimented the case to lock all the target data by Begin method in the batch update. This is a form of the conventional batch update which prohibits the conflict online entry, to maintain the ACID properties. Fig. 10 shows the elapsed time of the online entries of each status of the batch update in this experiment with three kinds of value: maximum, median and minimum. Here, the median intended to obtain an average time of the normal transaction, excluding the extremely long or short ones: the transaction with a lock wait; the aborted transaction. Also, we repeat the same experiment three times and show each average of each status as the result. Here, “after” shows the status after the batch update completed successfully, and it corresponds to the status before in Fig. 6. As for before and after, the results are roughly equal to the elapsed time of the online entry (0.6 sec). However, as for batch update, even the minimum time is 5.6 sec, which is close to the batch update time (5.7 sec) shown at the top in Fig. 11.

Next, we show the experimental result corresponding to the each control method shown in Fig. 4: Fig. 11 shows the batch update average elapsed time including the above-mentioned result; Fig. 12 shows the elapsed time of the online entries like Fig. 10, with the number of their retries. Here, as for (d), we performed this experiment by setting the wait time of the batch update commit as follows: 0.6 sec (close to the elapsed time of the online entry), 0.3 sec and 0.1 sec.

As for the elapsed time of the batch update, the method to lock all the target data is most efficient, and the temporal update decreases up to about 35% in efficiency from this method. Incidentally, the efficiency in the case of “(commit)” of (a) is relatively high, since the conflict doesn’t occur. On the contrary, in the case of “(rollback)” of (a), all the data was deleted after the completion of the batch update, which had been added by the batch update and OB update. In the case of (2) in Fig. 8, this is performed by the simple process such as deleting all the data of \( T_a = @20140310 \) from Bank Account table. As a result, the difference in comparison with the case of “(commit)” is only 0.6 sec. The difference in the elapsed time in the other cases is within about 0.6 seconds. That is, the change of the elapsed time of the batch update depended on the maximum elapsed time of the online entry (including the OB update) as shown in Section 3.1.
As for the elapsed time of the online entry, the median is roughly equal to the elapsed time of independent execution, and the maximum is within twice of it. That is, it followed the equation \( \max_{time}(O_j + OB_j + c) \) shown in Section 3.1. Here, the maximum time is different by the control method. As for (a), since the conflicting batch update is aborted, there is no retry and wait time. So, it is close to the median time. As for (b), since the conflicting online entry is aborted, its retry occurred four times on average. However, since there is no wait time for the batch update completion, the time is shorter than the case of (c) and (d). On the other hand, as for (c), the retry didn’t occur but the maximum time is longest because of the wait time for the commit of the batch update. As for (d), the maximum time changed between (b) and (c), by setting the wait time to commit the batch update.

6. Discussion

In the experiments of this study, we evaluated four concurrent execution control methods for the temporal update, presupposing the online entry of the roughly constant time. As a result, we found that the latency of the online entries, which don’t conflict mutually, doesn’t depend on the elapsed time of the batch update; it was within the total elapsed time of the online entry (including the OB update) and the commit of the batch update. Therefore, we consider this method is effective in the following case: since the online entries conflict little mutually, the optimistic concurrency method is adopted. Furthermore, by using the method (c) in this case, even if it is performed concurrently with the batch update, it is possible to eliminate the risk of the retry of the online entry.

However, in the actual business systems, the elapsed time of the online entry transaction is often a long while because of the following reasons: the conflict among themselves as shown in (2) in Fig. 3, the latency of user input using a lock, and so on. For this problem, it is considered that the method (a), (b) and (d) is effective.

First, in the case that the online entry transactions have to be executed in a short time without the retry, the method (a) is effective, as shown in Fig. 12. However, since there is the risk of the retry of the batch update, the following conditions are required: the cost of its retry is small; its execution timing is not strictly specified; there is a time zone when the probability to conflict with the online entry is low. For example, the following batch update operation is considered: it is executed when few users use the system, such as the midnight; if it conflicts with the online entry, it is retried. Incidentally, as shown in (a) in Fig. 11, the rollback of the batch update is small in the temporal update.
On the contrary, in the case that the online entry transactions are allowed to execute its abort and retry, and to wait within a certain time, the method (d) can be adopted. For example, as shown in (d-1) in Fig. 12, by setting the normal maximum elapsed time to the wait time of the method (d), most of the online entry can be executed without retry. Moreover, by adding a margin to this time, it is possible to abort only the online entry executing for an exceptionally long time. In addition, in this case, the commit of the batch update should be executed upon the completion of the preceding online entries, to reduce the useless wait. Also, in the case that the online entry transactions are allowed to retry but not allowed its latency, the method (b) can be adopted.

That is, the appropriate method should be chosen based on the requirements: tolerance of the wait time and retry as for the online entry; the operational requirement as for the batch update. We show the summary of above discussion in Table 1.

7. Conclusions

We had proposed the temporal update method to update a great deal of data in a lump as the transaction during the online entry. In this paper, first, we showed the factor of the waiting time of the online entry conflicting with the temporal update: it doesn’t depend on the elapsed time of the temporal update, but depend on the elapsed time of the online entries and the conflict among them. Next, we showed four concurrency control methods between them, and implemented these methods to evaluate the latency of the online entries that don’t conflict mutually. As a result, we confirmed the latency of the online entry is as the above-mention hypothesis as for every method, though the conflicting online entries become the wait state during the conventional batch update. Moreover, we showed the appropriate method should be chosen based on the requirements about the system operations.

Future studies will focus on the evaluation of the effect on the actual business systems by the prototype that will be built assuming the actual business operations.

This work was supported by JSPS KAKENHI Grant Number 24500132. Also, the motivation of this study is the implementation of the patent\(^6\) for the purpose of the batch update in the non-stop service systems. We appreciate the members of Mitsubishi Electric Information Systems Corp. who supported to get this patent.

References