World Wide Web (2019) 22:2633–2653
https://doi.org/10.1007/s11280-018-0552-7

Efficient time-interval data extraction in MVCC-based RDBMS

Haixiang Li[1] **· Zhanhao Zhao**[2,3] **· Yijian Cheng**[2,3] **·**
Wei Lu[2,3] **· Xiaoyong Du**[2,3] **· Anqun Pan**[1]

Received: 16 December 2017 / Revised: 4 March 2018 / Accepted: 21 March 2018 /
Published online: 11 April 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract Account reconciliation is the core business in banks and game companies. It regularly examines the account balance with the bank or expense statement for every user and reports the daily, weekly, or monthly balance. Once an account imbalance occurs, it is necessary to efficiently trace the transactions that possibly destroy the account balances. To help efficiently trace this kind of transactions, in this paper, we investigate the problem of doing efficient time-interval data extraction in MVCC-based RDBMS, i.e., extracting the incremental data that are valid between a given time interval in MVCC-based RDBMS. To this end, we propose a snapshot-based method to extract incremental data based on the fact that each record is inherently associated with lifetime, indicating whether the record can be

This article belongs to the Topical Collection: *Special Issue on Web and Big Data*
Guest Editors: Junjie Yao, Bin Cui, Christian S. Jensen, and Zhe Zhao

✉ Wei Lu
lu-wei@ruc.edu.cn

Haixiang Li
blueseali@tencent.com

Zhanhao Zhao
zhanhaozhao@ruc.edu.cn

Yijian Cheng
yijiancheng@ruc.edu.cn

Xiaoyong Du
duyong@ruc.edu.cn

Anqun Pan
aaronpan@tencent.com

[1] Tencent Inc., Shenzhen, China

[2] Key Laboratory of Data Engineering and Knowledge Engineering, Renmin University of China, Beijing, China

[3] School of Information, Renmin University of China, Beijing, China

accessed or not for a given time interval. We elaborate how to integrate our method into MySQL, an open-sourced RDBMS, and propose a declarative way to fetch the incremental data. Several optimization techniques are proposed to boost the extraction performance. Extensive experiments are conducted over the standardized Sysbench benchmark to show that our proposed method is robust and efficient.

Keywords RDBMS · MVCC · Incremental data extraction · Snapshot

1 Introduction

Account reconciliation is of great importance to corporate service charging and banking deposit systems. For example, in the Tencent's service charging system, a registered QQ [14] or WeChat [22] user can recharge his or her account so that he or she can utilize the paid business offerings that the system provides, like purchasing game equipments, watching digital fee TV, etc. From the perspective of the service charging systems, any access to the paid business offerings must guarantee that the account reconciliation is correct, i.e., any change of the account balances must be in consistency with the expense statements. To do this, the system is required to do regular account reconciliation and reports the daily, weekly, or monthly balance.

In some cases, however, due to the unpredictable system failures, results of doing the account reconciliation could be incorrect, i.e., changes of the account balances is not in consistency with the expense statements. Account imbalance can bring serious harms to financial systems, which could result in a loss of money or even a loss of customers. In this case, it is of great importance to dig out the harmful behaviors by tracing back to the transactions in the expense statements that possibly destroy the account balances. To this end, in this paper, we investigate the problem, namely *time-interval data extraction*, which is informally defined to extract the incremental data that are valid between a given time interval. By separating the time into a sequence of intervals and doing the account reconciliation with the help of the time-interval data extraction, once an account reconciliation in a time interval is checked to be incorrect, then transactions that occur in this time period are considered as suspicious transactions. For example, if we find the suspicious transactions occurring during 10:00 am and 11:00 am in Sep. 17, 2017, then we simply extract the transactions during 10:00 am and 11:00 am in Sep. 17, 2017 and then do account reconciliation to figure out the harmful transactions.

Execution of the time-interval data extraction tasks in RDBMS is not trivial. The reason is two-fold. First, implementation of time-interval data extraction should be transparent to users. One may think that by building an auxiliary relation recording all updates of account balance and the update time, it is able to do time-interval extraction by checking the update time based on the auxiliary relation. However, it not only incurs additional maintenance cost and modifies the database schema, but also degrades the system throughput since any update operation on the original relations will result in a cascading update of the auxiliary relation. Second, execution of the time-interval data extraction tasks should be easy to use. Consider that SQL has become a standard query language in existing commercial and open source RDBMS. Using SQL to execute the time-interval data extraction tasks is preferred.

We investigate the problem of doing interval data extraction in MVCC-based RDBMS. MVCC is widely adopted as a concurrency control mechanism in existing RDBMS, including Oracle, MySQL and PostgreSQL [19], etc. In MVCC mechanism, (1) each record is associated with multiple versions. More specifically, each insertion will generate a new version of the record. Each deletion will set a deletion flag in the latest version. Each update will set a deletion flag in the latest version and generate a new version of the record; (2) each version is associated with a transaction ID that operates (including insert, delete and update) the record. The transaction ID and the start time of the transaction are interchangeable; (3) a snapshot refers to the transaction states of the database system, and records all alive, completed transaction IDs [6]. To process a regular SELECT-FROM-TABLE query, the RDBMS engine will generate a snapshot for this query, and each record version, which is accessible by comparing the snapshot with the latest transaction ID that operates the record, is returned.

Consider the problem of interval data extraction in which we target to extract records that are accessible between time t_i and t_j. Direct applying snapshot to extract interval data is infeasible. The reason is two-fold. First, different from the regular SELECT-FROM-TABLE query, no snapshot is still preserved at time t_i or time t_j, and hence it is unable to verify whether a record is accessible. Second, even we can make effort to rebuild the snapshots, a judgment mechanism to verify the record is necessary. To address the above two issues, we first propose a snapshot reconstruction mechanism to preserve the transaction states during the runtime, and then present a carefully designed record visibility judgment mechanism, to extract the time-interval incremental data. To boost the query performance, we propose an index-based method to skip unnecessary pages to reduce the I/O cost. To sum up, our contributions are listed as follows:

– We propose a new type of query pattern, named as time-interval data extraction, which targets to extract incremental data for any given time interval in MVCC-based RDBMS.
– We propose a novel snapshot-based method, with a carefully designed record visibility judgment mechanism, to extract the time-interval incremental data. Various optimizations are proposed to reduce the I/O cost by skipping the pages without containing any records that are the results in the time-interval data extraction queries.
– We propose a declarative way, using SQL, to extract the time-interval data and provide a customizable parameter to fetch the inserted, updated and deleted data in the time-interval data extraction.
– We implement our proposed method in MySQL, a widely used open source RDBMS. Extensive experiments are conducted over the standardized Sysbench [20] benchmark, which demonstrates our proposed method is robust and efficient.

The rest of the paper is organized as follows. Section 2 defines our problem, discusses existing incremental data extraction methods, and reviews the MVCC mechanism. Section 3 gives an overview of our proposed method. Section 4 describes the implementation in details. Experimental results are discussed in Section 5. Section 6 concludes this paper.

2 Preliminaries

2.1 Problem definition

Let R be a relation with n records. Formally, R is represented as $\{r_1, r_2, ..., r_n\}$. Given two timestamps t_i, t_j ($t_i < t_j$), our objective is to efficiently identify the incremental data of relation R between time t_i and t_j, i.e., the data inserted, deleted or updated in this certain time-interval would be extracted. Let $S(R, t_i)$ represent the collection of records in R that are accessible at time t_i, and $I(R, t_i, t_j)$ represent the incremental records in R between time t_i and t_j. Apparently, records that are accessible at time t_i, but not accessible at time t_j, are deleted; on the contrary, records that are accessible at time t_j, but not accessible at time t_i, are inserted; Therefore, $I(R, t_i, t_j)$ can be formally defined in (1).

$$I(R, t_i, t_j) = (S(R, t_i) - S(R, t_j)) \cup (S(R, t_j) - S(R, t_i)), \quad t_i < t_j \tag{1}$$

Example 1 Figure 1a and b show a relation about accounts' balance at time t_1 and t_2. Suppose we want to figure out the incremental data between time t_i and t_j. We can see that the r_1 is deleted, r_2 keeps unchanged, r_3 is updated and r_4 is inserted during the given time-interval. Therefore, according to (1), records shown in Figure 2 can be reported as the results.

For reference, symbols listed in Table 1 will be used throughout the paper.

2.2 Related work

2.2.1 Incremental data extraction

Incremental data extraction is a key step of Extract-Transform-Load(ETL) process [8, 11, 12] in a wide spectrum of applications, including account reconciliation, data monitoring, archives management, entity retrieval [26], incremental recommendation [23] etc. Existing approaches target to do ETL more efficiently and effectively. In general, they can be divided into four categories, timestamp based extraction, data snapshot method, trigger based extraction and log based extraction.

Timestamp based extraction [15] needs to attach an additional timestamp attribute to the source table. The timestamp attribute will store the time when the record is inserted into the table. Therefore, given a relation named *mytable*, the incremental data between time t_i and t_j can be obtained by executing a query like *SELECT * FROM mytable WHERE timestamp BETWEEN* t_i *AND* t_j. As implemented as SQL statements, this method can easily be

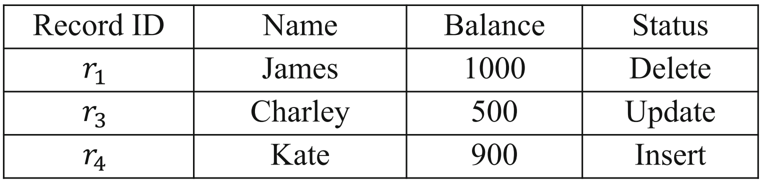
Record ID	Name	Balance
r_1	James	1000
r_2	Mark	2000
r_3	Charley	500

(a) Relation R at time $t1$

Record ID	Name	Balance
r_2	Mark	2000
r_3	Charley	1500
r_4	Kate	900

(b) Relation R at time $t2$

Figure 1 The relation of accounts' balance at time t_i and t_j



Record ID	Name	Balance	Status
r_1	James	1000	Delete
r_3	Charley	500	Update
r_4	Kate	900	Insert

Figure 2 The incremental data between time t_i and t_j

integrated into RDBMS. However, there are two main disadvantages of this method. First, records that have been deleted cannot be fetched because only the create time of records is stored but the deleted time is missing. Second, an expensive index should be applied on this attribute for accelerating the query efficiency, where maintaining this timestamp attribute needs additional costs.

Data snapshot method requires snapshots or backups of the physical data should be stored periodically. The changed data can be extracted by comparing snapshot of the current state with snapshot of a previously state. There are some methods like window algorithm [5] and sort-merge algorithm to compare the data snapshots and organizing data. It is an efficient way to find differences based on physical data but it cannot be combined with transactions, where the data may be incorrect since the consistency is not guaranteed. In other words, there may exist some dirty records that cannot be read at that time.

Trigger based extraction [15] means that there will be a trigger attached on the source relation to trace data insert, update and delete. Whenever the source relation changes, the corresponding trigger will be activated, and the changed data will be captured and stored in a temporary table. Therefore, the data is redundant, and a time-consuming trigger will lead to the slower response of the source database. Oracle synchronization CDC is implemented base on the trigger mechanism. Within this system, real-time synchronization can be achieved but business system performance will be degraded, and the efficiency of the source system is reduced by about 10%.

Log based extraction [4] is a method to obtain the incremental data by parsing the system log of RDBMS. This method is widely used due to its universality. Regarding this method, the log service would not affect the performance of the source system, and it is convenient to transfer the logs to other systems, like Kafka, which is now considered as a basic component for real-time analysis. There are some existing systems like Canal [3], Oracle asynchronized CDC, etc. However, there is a certain delay during the log generation. Besides, the development of log based extraction becomes quite difficult for the systems without providing log analysis interface.

Table 1 Symbols and their definitions

Symbol	Definition
R	a relation, $R = \{r_1, ..., r_n\}$
$I(R, t_i, t_j)$	the incremental records of relation R in time interval $[t_i, t_j]$
$S(R, t_i)$	snapshot at time t_i for relation R
A	attributes of R, $A = \{a_1, ..., a_m\}$
T_i	a transaction with ID i
opT	an insert, update or delete operation

2.2.2 *Multi-version concurrency control*

The MVCC mechanism was first proposed in 1978 by Reed and David Patrick [16]. It targets to solve the data inconsistency issue that occurs when the single record is accessed concurrently by multiple transactions. The MVCC is always combined with timestamp [1] or lock technology [24]. Snapshot Isolation is a detailed concurrently technique based on MVCC. In [2, 13, 17, 25], the serializable snapshot isolation has been conducted, which enable the transaction snapshot has been widely used in the RDBMS, like PostgreSQL [19] and MySQL(InnoDB as the storage engine).

In the MVCC-based database, a record has series of versions, which are generated when the update or insert operations are applied to this record. The snapshot can be defined as transaction snapshot, which is a view that shows the most recent version of a record can be seen by the current transaction. According to the multi-version record and the transaction snapshot, the concurrent transactions will not affect each other, for each transaction operates specific versions that definitely will not affect other transactions [6, 7, 10].

An example of MVCC-based transaction processing is shown in Figure 3. Suppose there is a record $A = \{10, 20\}$ and four transactions, denoted as T_0, T_1, T_2, T_3 respectively, are doing concurrent operations on record A. T_0 would like to update the first attribute of A to 30. Although the T_0 is still running when T_1 starts, T_1 is able to obtain the select result as $A = \{10, 20\}$ without blocking by T_0. As for T_2, a new version of record A is constructed after the commit operation of T_2. The latest version should be $A = \{40, 20\}$, the original version is $A = \{10, 20\}$. T_4 is able to get the result as $A = \{40, 20\}$. According to the MVCC mechanism, the consistency transactions can be processed efficiently and methodically.

3 System overview

In this section, we propose a novel snapshot-based method for incremental data extraction. Snapshot is a basic concept in RDBMS and used to record the execution status of all transactions. Typical status includes running, committed or aborted, indicating that a transaction

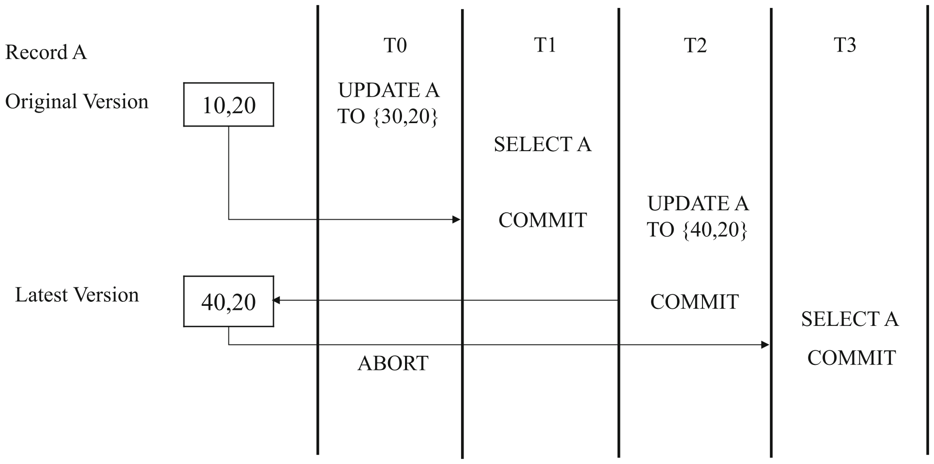


Figure 3 The concurrent transaction processing in MVCC

is running, has been committed, or aborted, respectively. When a transaction starts, its status will be set to running and added to the snapshot accordingly. When a transaction completes (committed or aborted), its status will be updated to committed or aborted in the snapshot accordingly. In RDBMS, snapshot helps determine which versions of records are visible when users issue queries, as well as concurrency control. With these knowledge, we aim to extract the incremental data based on the snapshot. In existing RDBMS, any historical versions of records are not maintained, i.e., we cannot trace the data that have been deleted or updated. Instead, with the help of the visibility judgment mechanism, any historical versions of records can be extracted and maintained from the system level. For this reason, the visibility judgment mechanism can help answer incremental data extraction queries automatically from the system level without any involvement from users, and hence reduce the extra maintenance overhead.

The overall system architecture is shown in Figure 4. Users can issue SQL statements using INCREDATA keyword to fetch incremental data. In general, usage of INCREDATA is similar to that of SELECT (details are shown in Section 3.2). By specifying the *startPoint* and *endPoint* in the time-interval data extraction query, it is able to generate two snapshots regarding two time points *startPoint* and *endPoint* , and we then extract the incremental data based on the two snapshots.

3.1 Snapshot-based incremental data extraction method

We can use the information stored in snapshot to fetch data versions generated between a certain time-interval [*startPoint*,*endPoint*], where the incremental data can be extracted according to these versions. Meanwhile, unless the transaction is committed, the data operated in this transaction would not influence other existing data. Therefore, combined snapshots with record versions, we can fetch the incremental data that is definitely correct.

For convenience, we present a typical snapshot schema for a regular MVCC-based RDBMS in Figure 5. Given two transaction IDs lowerBound and upperBound, there are

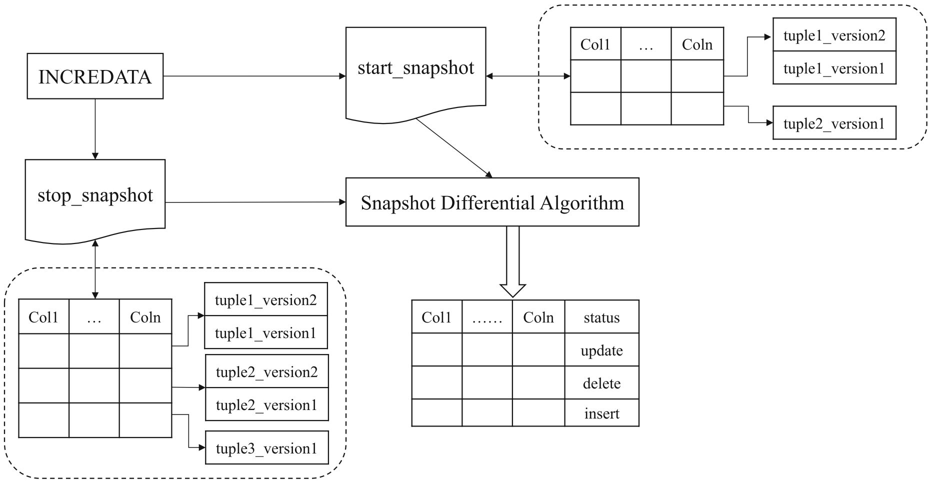


Figure 4 The overall system architecture

```
class Snapshot {
    ulint64      upperBound;
    ulint64      lowerBound;
    ulint64      trxCreator;
    ulint64      createTime
};
```

S1

upperBound:	50
lowerBound:	10
trxCreator:	25
createTime:	1511534986s

Figure 5 A typical snapshot schema and a nest snapshot example

several properties based on the snapshots. Transactions that finish before the lowerBound are definitely visible to the current snapshot. Transactions that start after the upperBound are definitely invisible to the current snapshot. The trxCreator represents the current transaction that creates this snapshot. CreateTime records a timestamp when this snapshot is created. According to these variables, the visibility judgment method for a certain time-interval is clearly conducted in Algorithm 2.

$S1$ in Figure 5 is a snapshot example, which contains the information that the transactions before lower-bound:10 are all finished and visible, the transactions after upperBound:50 are still running and invisible. This snapshot is created by transaction ID:25.

Given a relation R, the common record structure applied in MVCC-based RDBMS can be defined as Figure 6. It is composed with the system columns and user defined columns logically. The system columns are fixed columns generated by system including the unique row identifier, information about transaction effects on this record, etc. The commit_time represents when this record is inserted or updated, besides, the delete_flag will be marked when this record is deleted. The user defined columns, which are also represented as attributes $A = \{a_1, ..., a_m\}$, contain the structural data added by users. Therefore, we can define the record visibility as whether it can be seen at a specific time t, which can be judged by the transaction info and other system columns inherently existed in each record. The record that is non-visible at t_i but visible at t_j can be defined as incremental record. For each incremental record, we obtain an "expanded-record" which can be expressed as $r_i' = \{r_i, opT\}$, where opT represents the operation on this record and $opT = enum\{insert, update, delete\}$.

Now we consider about how to obtain the incremental data in MVCC-based RDBMS efficiently. Figure 7 illustrates some concurrency transactions circumstances based on a time-line. T_1 denotes a transaction starts at t_1 and commits at t_3. Besides, T_2 denotes a

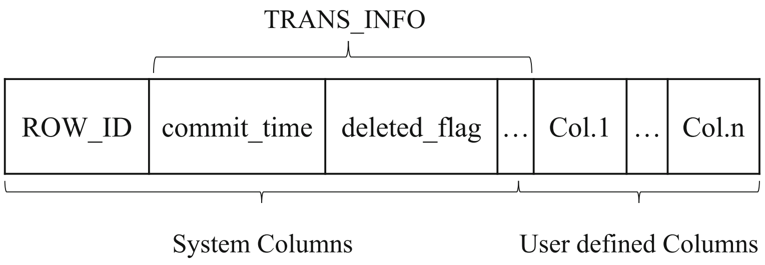


Figure 6 The common structure of a record in RDBMS

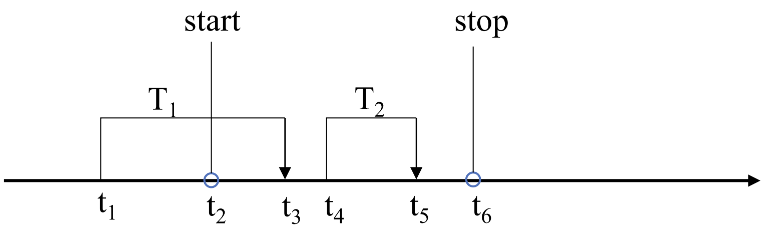


Figure 7 Concurrency transactions according to a time-line

transaction starts at t_4 and commits at t_5. Snapshots named as s_start and s_stop is token at t_4 and t_6 respectively.

When we do insert operations in T_1 and T_2, T_1 starts insert transaction before s_start and commits between s_start and s_stop, T_2 starts insert operation between s_start and s_stop and also commits between s_start and s_stop; In short, the insert data between s_start and s_stop can be generalized as (2).

$$\begin{array}{ll} s\_start.createTime < & r_i.commit\_time < s\_stop.createTime \\ \&\& & r_i.deleted\_flag == null \end{array} \tag{2}$$

When T_1 and T_2 are doing delete operations, the deleted data between s_start and s_stop can be generalized as (3).

$$\begin{array}{ll} s\_start.createTime < & r_i.commit\_time < s\_stop.createTime \\ \&\& & r_i.deleted\_flag \neq null \end{array} \tag{3}$$

When T_1 and T_2 are doing update operations, according to the MVCC, a newer version is generated and the older version is marked as deleted. Therefore, the update operations can be considered as a combination of delete and insert operations. The update records between s_start and s_stop should fit the condition in (4) and (5).

$$\begin{array}{ll} s\_start.createTime < & r_i.prev().commit\_time < s\_stop.createTime \\ \&\& & r_i.prev().deleted\_flag \neq null \end{array} \tag{4}$$

$$\begin{array}{ll} s\_start.createTime < & r_i.commit\_time < s\_stop.createTime \\ \&\& & r_i.deleted\_flag == null \end{array} \tag{5}$$

Algorithm 1 shows the framework that how the incremental data generated between given snapshots can be obtained. For each record, the correct version should be found out according to given snapshots. The version which passes the visible judgment is included in the incremental data set, where $I(R)$ can be generated.

Algorithm 1 Snapshot-based incremental data extraction algorithm

```
Input: $R$: Source relation; $s\_start$: Snapshot represents the start point; $s\_stop$: Snapshot
    represents the stop point;
Output: $I(R)$: Incremental data set
 1: initial $I(R) = \emptyset$;
 2: repeat
 3:     Get current positioned record $r_i$ in relation $R$;
 4:     opT = isVisible($r_i$, $s\_start$, $s\_stop$, 1)
 5:     if opT != 0 then                                      ▷ this version is visible
 6:         put $r_i$,opT into $I(R)$
 7:     end if
 8:     while $r_i.prev()! = NULL$ do
 9:         $r_i = r_i.prev()$;
10:         opT = isVisible($r_i$, $s\_start$, $s\_stop$, 0)
11:         if opT != 0 then                                  ▷ this version is visible
12:             put $r_i$,opT into $I(R)$
13:         end if
14:     end while
15: until Relation Scan Finished
```

Algorithm 2 indicates the detailed record visibility judgment algorithm for a certain time-interval. At first, the commit_time of the input record should meet the conditions indicated in (6). If so, this record is committed in the given time-interval and is an incremental record, otherwise it is not an incremental record in this time-interval. And then, the deleted tag of this record should be checked to identify if this is a deleted record. If the previous version is missing, this record can be defined as an inserted record, otherwise, this record is an updated record.

$$s\_start.createTime < r_i.commit\_time < s\_stop.createTime \tag{6}$$

Algorithm 2 Record visibility judgment algorithm

```
Input: $r_i$: A given version of a record; $s\_start$: Snapshot represents the start point; $s\_stop$:
    Snapshot represents the stop point; $is\_first$: 1-latest version, 0-previous version;
Output: opT: operation tag, 0-invisible, 1-insert, 2-update, 3-delete;
 1: initial $opT = 0$;
 2: if $s\_start.createTime < r_i.commit\_time < s\_stop.createTime$ then
 3:     if $r_i.isDelete == false$ then
 4:         return opT=1;
 5:     else
 6:         if $is\_first$ then
 7:             return opT=3;
 8:         else
 9:             return opT=1;
10:         end if
11:     end if
12: else
13:     return opT=0;
14: end if
```

3.2 INCREDATA Query

A specific SQL statement has been designed to answer time-interval data extraction query, which is shown as follows:

INCREDATA [col1,...]/[*] table [SNAPSHOT snapshot [TO snapshot2]] [WHERE ...]

- Users can use INCREDATA query to get incremental data in a given time-interval. The SNAPSHOT keyword attached to a certain relation defines the time-interval by two snapshots regarding two time points. Also, the usage of INCREDATA is similar to that of SELECT. The [col1,...]/[*] syntax gives the projection capacity to this command. The WHERE syntax can make this command handle filter operation and aggregation operations are also supported in INCREDATA query.
- We use snapshots to represent the logical time here, in other words, we could expand INCREDATA query to fetch data that changes between two transactions. The time-interval here is not only about clock-time, so that we can achieve much precise data extraction because the data is attached to correspond transactions.

4 System implementation

In this section, we would like to introduce the implementation of our proposed method in MySQL(InnoDB as the storage engine). With respect to InnoDB, it is a high-performance storage engine that can be plugged in MySQL, which is based on MVCC mechanism and uses snapshot isolation to ensure the read consistency.

4.1 Transaction status

The snapshot in InnoDB is defined as ReadView (shown in Figure 8), which is similar with the structure defined in Figure 5. With respect to ReadView, we can attach the definition of create_time with the m_creator_trx_id. The ReadView exists in the MySQL system inherently, which will generate when a transaction starts. The proposed incremental data extraction method is based on this structure.

However, when the transaction finishes, the ReadView correspond with this transaction will be released, so the transaction information will be missing for time-interval search. To solve this problem, we proposed a well-organized structure to store the transaction's information. The data structure has been shown in Figure 9. We use 16 bytes to store each transaction's start time, finish time and status, specified as follows:

- Start time (7 bytes). Represents the time when the transaction starts.
- Finish time (7 bytes). Represents the time when the transaction stops. Only finished transactions will have this feature.
- Status (2 bytes). Store transaction status includes not start, still running, committed and aborted.

One data page contains several transaction's information records. The transaction ID is an auto-increment number in MySQL, therefore, the transactions are sorted by transaction ID in this structure. The page structure is shown in Figure 9. Besides, the transaction ID is not stored in this structure, for the transaction ID can be clearly represented by the data

Figure 8 The ReadView in MySQL

```
class ReadView {
private:
        trx_id_t        m_low_limit_id;
        trx_id_t        m_up_limit_id;
        trx_id_t        m_creator_trx_id;
        ids_t           m_ids;
public:
        bool            changes_visible();
        bool            sees();
};
```

position. And the mapping function can be represented by (7). Each transaction can be targeted by the Page_No and the Offsets in that page. With an implementation of data buffer strategy, this structure gives a decent read-write performance.

$$\begin{aligned} &amount\_per\_page = \frac{page\_size}{info\_record\_size} \\ &Page\_No = \frac{transaction\_id}{amount\_per\_page} \\ &Offsets = (tansaction\_id\%amount\_per\_page) * info\_record\_size \end{aligned} \tag{7}$$

4.2 The implementation of visibility judgment algorithm

As the visibility judgment procedure shown in the Figure 10, a record is regarded as the input, where the id of transaction that creates or modifies this record is append on it. At first, the commit status should be checked according to the *transaction_information*. If this transaction has been committed, we should check if this transaction is committed between *s_start* and *s_stop* or not. According to Algorithm 1 and 2, the detailed transaction snapshot differential algorithm based on InnoDB is defined as Algorithm 3. We add this method in class ReadView, which could be called to judge the visibility of a certain record. Combined with the robust table-scan method in MySQL, the incremental data can be found out according to this method.

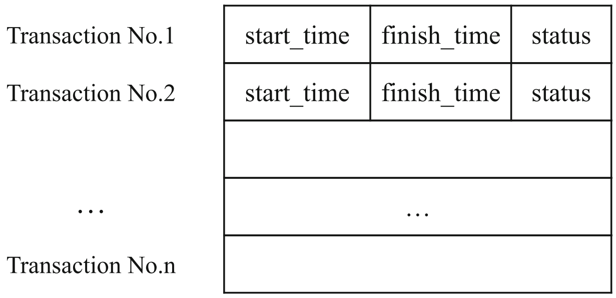


Figure 9 The page structure for transaction information

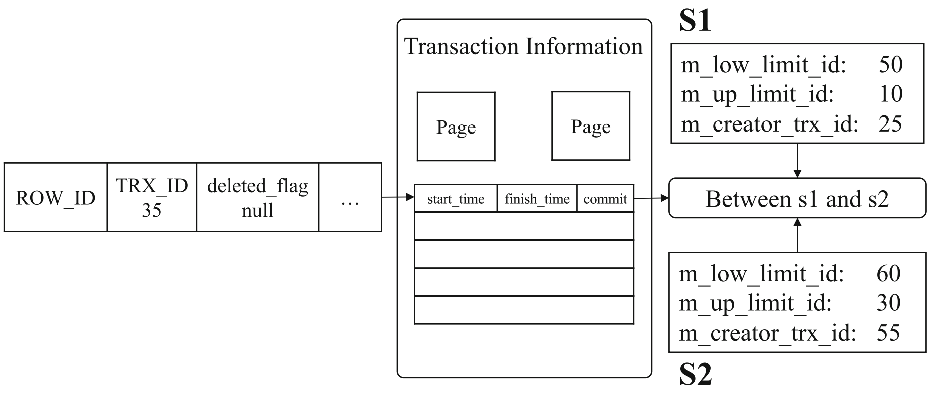


Figure 10 The visibility judgment procedure

Algorithm 3 ReadView. changes_visible_incre

Input: r_i: A given verison of a record; s_start: ReadView represents the start point; s_stop: ReadView represents the stop point; $transaction\_information$: stores the committed transactions and their commit time. is_first: 1-latest version, 0-previous version;
Output: opT: operation tag: 0-invisible,1-insert,2-update,3-delete;

```
1:  initial $opT = 0$;
2:  if $transaction\_information.search(r_i.trx\_id)$ then
3:      if $s\_start.m\_creator\_trx\_id < r_i.trx\_id < s\_stop.m\_creator\_trx\_id$ then
4:          if $r_i.isDelete == false$ then
5:              return opT=1;
6:          else
7:              if $is\_first$ then
8:                  return opT=3;
9:              else
10:                 return opT=1;
11:             end if
12:         end if
13:     else
14:         return opT=0;
15:     end if
16: else
17:     return opT=0;
18: end if
```

4.3 Storage strategy for historical versions

MySQL is built based on MVCC mechanism, therefore, physical records have been deleted or updated may still exist in the physical storage, which may become historical versions for concurrent transactions. However, the system cannot tolerate the immoderate expansion of these overdue physical records, which may cause the lack of storage space and effect the system efficiency.

To solve this problem, MySQL cleans up this kind of physical records periodically, which is called PURGE. Since PURGE removes batches of historical records periodically, we apply a method to obtain batches of records that have been removed, and restore them into another corresponding historical table. In consideration of a better query capacity, the whole record will be stored and an additional operation type attribute (define whether this version is generated by delete operation or update operation) is append to each record in the historical table. Also, based on the facility that tables in MySQL are index-organized tables, a cluster index on the transaction ID can be inherently applied. Consequently, the historical version that has been purged can be found in historical table and the time-interval query for incremental data has a decent efficiency provided by the original MySQL system structure.

5 Optimization

In this section, we designed an optimization theory to conduct an efficient index scan method to extract incremental data. As the method proposed in the above sections, almost each record in the relation should be checked whether it is included in the incremental data set for a certain time-interval. As we known, the minimum buffer unit is always corresponding to the page size, where the less page load from disk to buffer, the better incremental data query performed.

For the time-interval incremental data extraction, it is obvious that there may exist some data pages that all the records on them are not include in a specific time-interval. In this paper, we consider about the page in which all the records are created before a specific time-interval, we call this kind of page as "really old page". Exactly, some extremely old pages can be skipped for scanning and will not be loaded to the buffer for the records on them are definitely not included in the incremental data.

As is shown in Figure 11, we construct a B+tree index on transaction ID, where the pruning rule is according to the transaction ID attached on each record. According to this index, we can easily target to a place where transaction ID is equal with $s\_start.lowerBound$. Since transactions that finish before the lowerBound are definitely visible to s_start, the data operated by these transactions cannot be incremental data. Therefore, an efficient index scan can be conducted for "really old page" will be skipped.

For example, if the time-interval shows that incremental data is operated by transaction between 1000 and 1300, as to Figure 11, we can easily put the index scan pointer to the record whose transaction ID is 1000. Therefore, the pages pointed by key after 1000 need to be loaded to the buffer and do visibility judgment for each record, the pages that are pointed by key before 1000 do not need to do so.

Algorithm 4 indicates the index scan flow. We first get the first node in the index, and then keep scan the index. If the page should be scanned, we will call the record visibility judgment algorithm to check the visibility of each record. Suppose a relation is stored in N pages, so the total I/O for scan this relation should be $N * M$, where M represents the average time cost for loading one page. If there are P "really old page" in this relation, the total I/O for index scan should be $(N - P) * M$. Thus, the I/O cost would be reduced by cut down the page loading times, where the incremental data extraction performance would be boosted. Besides, suppose their are K records in each page, the visibility check operation times will reduce from $N * K$ to $(N - P) * K$.

Algorithm 4 Index Scan Algorithm

```
Input: $R$: source relation; $Index$ : index on $R$; $s\_start$: Snapshot represents the start point;
    $s\_stop$: Snapshot represents the stop point;
Output: Result set $R'$.
1: $iter \leftarrow Index.search(\text{s\_start}.lowerbound)$;
2: while $iter.hasNext()$ do
3:     $page \leftarrow iter.getPage()$;
4:     $R'.add(page\_scan())$;          ▷ according to record visibility judgment algorithm
5:     $iter \leftarrow Index.getNext()$;
6: end while
```

6 Experiment

In this section, extensive experiments are conducted over the standardized Sysbench [20] benchmark, which demonstrate the efficiency of our proposed method. Compared with other incremental data extraction methods, the proposed method in this paper gives a higher system performance and greater query efficiency.

All experiments are conducted on a Linux Server with Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 128G of memory and CentOS 7 operating system. Our comparison experiments are mainly based on MySQL 8.0.3, and approaches for incremental data extraction are generated into MySQL 8.0.3 respectively.

Comparative approaches We compare the following approaches in the experiments.

- Log based extraction (a.b.a. LBE). LBE is a method to obtain the incremental data by parsing the system log of RDBMS. Here we use the binary log in MySQL which stores the database changes to get the incremental data and evaluate the system throughput.
- Trigger based extraction (a.b.a. TBE). TBE means that there will be a trigger attached on the source relation to trace the insert, update and delete operation. Here we use the trigger provided by MySQL to conduct this method. Insert and update triggers are attached on the source table, and a corresponding table to store the incremental data is created.
- Timestamp based extraction (a.b.a. TSBE). TSBE appends an additional timestamp attribute to the source table, which represents the time when this data has been generated. Therefore, when a record is inserted or updated in the source table, the clock time will store in the additional timestamp attribute.
- INCREDATA represents our proposed method, which is a MVCC-based method to extract time-interval incremental data.

Datasets and experimental scenarios We conduct the experiments using the default datasets generated by Sysbench benchmark [20]. Dataset DS1 contains 2 millions records, which represents small amount of data, and DS2 has 5 millions records, which is a big data volume dataset. Besides, we construct two experimental scenarios to simulate the account balance changes in the Tencent's service charging system. TDSQL [21] is a distributed database system, which carries almost all the digital payment-related business in Tencent.

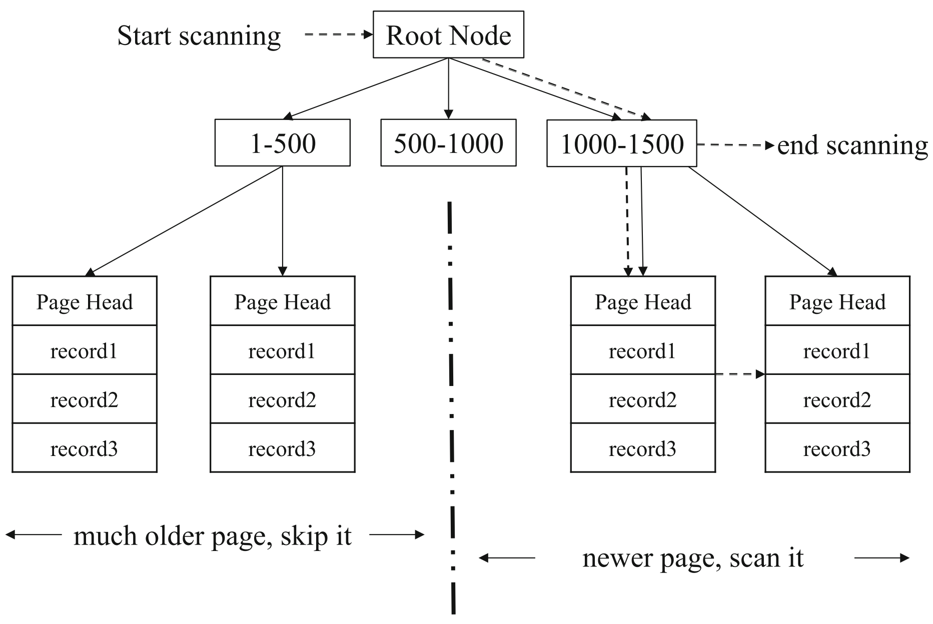


Figure 11 The index structure and index scan flow

Therefore, we collect and analyze the data in TDSQL and conclude the following experimental scenarios. We mainly consider about the alternative happened in the account table that contains information about user balance, user identification, etc.

- Scenario.A simulates the situation that Tencent are dealing with other companies. The number of companies in the charging system is quite stable, so we ignore the company registrations in a way. We perform the operations on the account table with proportion like $insert : delete : update = 0{:}1{:}9$ to illustrate this business-to-business situation.
- Scenario.B simulates the situation that QQ or WeChat [22] users are buying the service provided by Tencent. The users can register their accounts by themselves, so the insert operations on the account table are quite frequent. We perform the operations on the account table with proportion like $insert : delete : update = 2{:}1{:}7$ to show this customer-to-business situation.

6.1 Comparison among incremental data extraction methods

We can easily give a comparison among existing incremental data extraction methods with INCREDATA. Several aspects shown in Table 2 can be judged to show the advantages and disadvantages of each method. The *insert*, *delete* and *update* denote kinds of incremental data, each of them represents whether this kind of the incremental data can be distinguished. The *time-interval extraction* denotes whether the method can extract the incremental data in any time-interval. The *system delay* denotes whether this method will impact on the system efficiency. The *data invasion* denotes whether the method will affect the original data relation.

Table 2 Comparison among incremental data extraction methods

	Insert	Delete	Update	Time-interval extraction	System delay	Data invasion
LBE	√	√	√	√		
TBE	√	√	√	√	√	
TSBE				√		√
INCREDATA	√	√	√	√		

Table 2 shows that LBE and INCREDATA perform better in the given aspects. They not only can support to judge the incremental type, but also can get incremental data of any given time-interval. Otherwise, TBE will impact on system efficiency, which may cause system delay in a way.

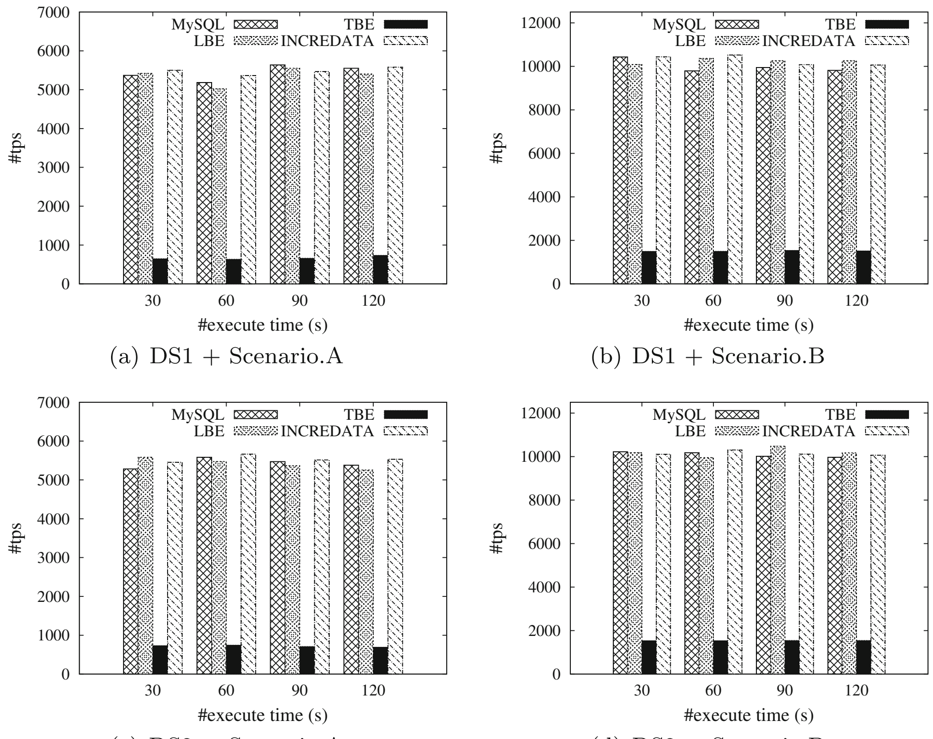
With respect to TSBE, it can neither avoid data invasion nor distinguish the incremental type. Therefore, TSBE method can hardly be used to extract incremental data in realistic production environments. Therefore, we will not evaluate the performance of TSBE method in the following experiments.

6.2 System throughput evaluation

We now apply several approaches into MySQL respectively, and evaluate the system performance. The system efficiency is reflected on the average number of transactions finished per second (a.b.a. tps), which can be collected by Sysbench benchmark [20]. By combining different datasets and experimental scenarios, we obtain 4 experiments, which are: (1) DS1 and Scenario.A; (2) DS1 and Scenario.B; (3) DS2 and Scenario.A; (4) DS2 and Scenario.B. The baseline system throughput is reflected by the original MySQL's tps. Meanwhile, the system influence can be indicated by comparing the tps of MySQL applied with a data extraction method with the original MySQL's tps. In this place, we mainly compare the throughput of LBE, TBE and INCREDATA, but TSBE is excluded for it cannot provide some key functions like judging the incremental type, etc.

We conduct the experiments with following steps. (1) Create a database and initialize the datasets. (2) Use the Sysbench script to execute insert, delete and update operations within the proportion defined in the specific scenario. We set the concurrent threads number to 500, and execute the script for 30s, 60s, 90s and 120s respectively. (3) Evaluate the tps within the certain time period.

Figure 12 shows the system performance based on different datasets and scenarios. The x axis *execute time* means how long the Sysbench script executes and the y axis represents *tps*. From the results, we observe that LBE and INCREDATA do not cause the system delay, but TBE affects the system efficiency. The *tps* of MySQL generated with LBE or INCREDATA is roughly same as *tps* of original MySQL, but *tps* of MySQL with TBE reduces to 15%. With respect to TBE, the time-consuming trigger operation will block the transaction execution to some extent. Every modification on the record will cause a correspond trigger operation, which will require a table-lock and hinder the transaction processing. Besides, the system performance of the LBE and INCREDATA is stable on different experimental datasets and scenarios. Therefore, LBE and INCREDATA perform better than TBE in system throughput evaluation, and which is better between LBE and INCREDATA cannot be distinguish by system throughput evaluation.



(a) DS1 + Scenario.A (b) DS1 + Scenario.B

(c) DS2 + Scenario.A (d) DS2 + Scenario.B

Figure 12 System throughput evaluation

6.3 Query efficiency evaluation

We now evaluate the query efficiency of extracting incremental data with different approaches. The experiments conduct as follows: (1) We use the datasets after executing Sysbench script for 120s, where the incremental data can be generated. (2) Extract the incremental data in a given time-interval. We set the start time of the time-interval as 5s, 35s, 65s, 95s after Sysbench script execution and assign the time-interval as 5s. (3) Evaluate the time cost.

As for LBE, the time cost is represented by the sequential scan of the binary log files provided by MySQL. As for TBE and INCREDATA, the time cost is calculated by the response time of a specific SQL command.

We collect the statistics of I/O cost of INCREDATA. Table 3 shows the total page size that contains all the pages that transfer between buffer and disk. From the results, we can see that the later the start time is, the smaller the total page size is. This conclusion matches the optimization theory that only "really old page" will be skipped during the index scan. Besides, compared with the full table scan, the index scan can reduce I/O cost to 25% approximately when start time is set as 95s.

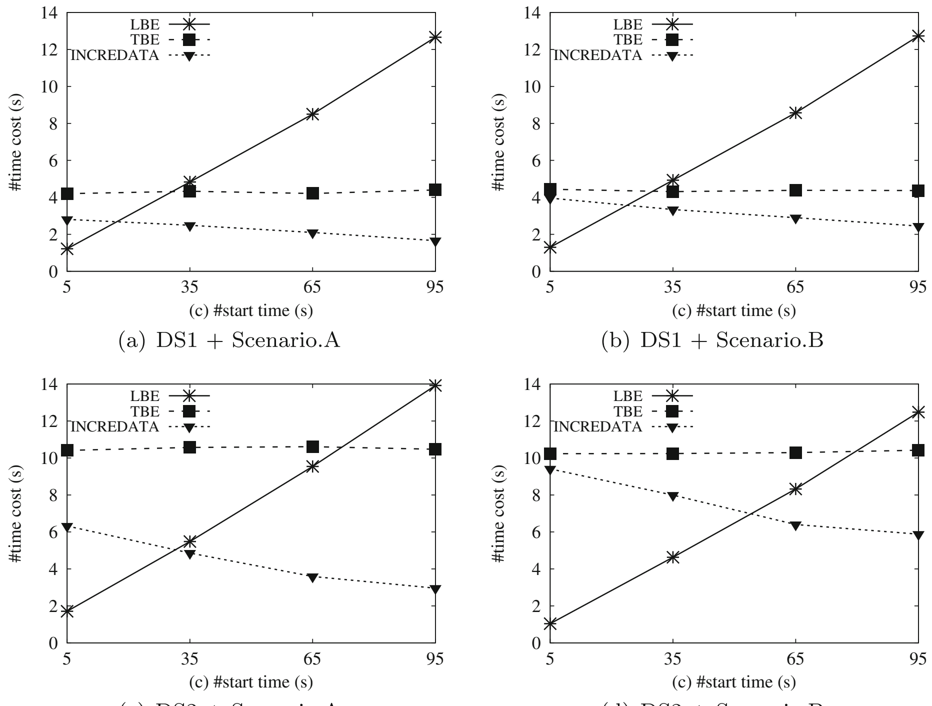
Figure 13 illustrates the time cost of several approaches when extracting a certain time-interval data, the x axis *start time* means how long after Sysbench script execution and the y axis $time\ cost$ means the time cost to get the time-interval incremental data.

Table 3 Statistics of I/O cost (KB)

	5s	35s	65s	95s	Table scan
DS1 + scenario.A	833312	657984	438784	219616	877664
DS1 + scenario.B	1055232	886064	666864	397696	1077696
DS2 + scenario.A	2187488	1753504	1210208	876736	2192416
DS2 + scenario.B	2554928	2150944	1712576	1274176	2589856

From Figure 13, we observe that the time cost of INCREDATA is less than the time cost of TBE. That is because TBE should conduct range search on the timestamp attribute, which needs a large amount of time because the physical records are not sort by the timestamp and the range search on an unsorted attribute is a time-consuming operation. However, INCREDATA applies an efficient index on transaction ID both in original and historical table, the physical records in historical table is sorted by transaction ID. When doing the query to extract time-interval incremental data, the index on transaction ID can quickly target a place where the records before it cannot be incremental data definitely.

Additionally, when the start time increases, the time cost of INCREDATA decreases, but the time cost of LBE rises rapidly. The reason can be explained as: (1) With the reduce of I/O cost, the query response time of INCREDATA can reduces correspondingly. (2) With



(a) DS1 + Scenario.A
(b) DS1 + Scenario.B
(c) DS2 + Scenario.A
(d) DS2 + Scenario.B

Figure 13 Query time cost

the continuous running of RDBMS, the log records will append to the log file and the log file will become larger and larger. Therefore, the time cost of LBE becomes longer with the increment of start time because much larger file should be scanned to extract the incremental data and the sequential scan will read log records from the beginning of the log file. According to the experimental results above, we can conclude that the performance of using INCREDATA to extract time-interval incremental data is much better than using other approaches.

7 Conclusion

In this paper, we investigate the problem of doing efficient time-interval data extraction in MVCC-based RDBMS. We propose a snapshot-based method to extract incremental data based on the fact that each record is inherently associated with some transaction information. We elaborate how to integrate our method into existing open-sourced RDBMS and propose a declarative way to fetch the incremental data. An index-based optimization techniques are proposed to boost the extraction performance. Extensive experiments are conducted over the standardized Sysbench benchmark to show that our proposed method is robust and efficient.

Acknowledgments We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (61502504, 61732014) and the Tencent Research Grant for Renmin University of China.

References

1. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. (CSUR) **13**(2), 185–221 (1981)
2. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Datab. Syst. (TODS) **34**(4), 20 (2009)
3. Canal. https://github.com/alibaba/canal
4. Doan, A., Naughton, J.F., Ramakrishnan, R., Baid, A., Chai, X., Chen, F., Chen, T., Chu, E., DeRose, P., Gao, B., et al.: Information extraction challenges in managing unstructured data. ACM SIGMOD Rec. **37**(4), 14–20 (2009)
5. Labio, W., Garcia-Molina, H.: Efficient Snapshot Differential Algorithms in Data Warehousing. Tech. rep., Stanford InfoLab (1996)
6. Li, H., Feng, Y., Fan, P.: The art of Database Transaction Processiong: Transaction Management and Concurrency Control. China Machine Press (2017)
7. Lu, W., Fung, G.P.C., Du, X., Zhou, X., Chen, L., Deng, K.: Approximate entity extraction in temporal databases. World Wide Web **14**(2), 157–186 (2011)
8. Lu, W., Hou, J., Yan, Y., Zhang, M., Du, X., Moscibroda, T.: MSQL: efficient similarity search in metric spaces using SQL. VLDB J. **26**(6), 829–854 (2017)
9. Ma, K., Yang, B.: Log-based change data capture from schema-free document stores using mapreduce. In: 2015 International Conference on Cloud Technologies and Applications (CloudTech), pp. 1–6 (2015).
10. McWherter, D.T., Schroeder, B., Ailamaki, A., Harchol-Balter, M.: Priority mechanisms for OLTP and transactional Web applications. In: ICDE. IEEE Computer Society, pp. 535–546 (2004)
11. Meehan, J., Tatbul, N., Zdonik, S., Aslantas, C., Cetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Pavlo, A., Stonebraker, M., Tufte, K., Wang, H.: S-store: Streaming meets transaction processing. Proc. VLDB Endow. **8**(13), 2134–2145 (2015)
12. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: Interactive analysis of Web-scale datasets. Proc. VLDB Endow. **3**(1-2), 330–339 (2010)

13. Ports, D.R.K., Grittner, K.: Serializable snapshot isolation in postgresql. Proc. VLDB Endow. **5**, 1850–1861 (2012)
14. QQ. https://im.qq.com
15. Ram, P., Do, L.: Extracting delta for incremental data warehouse maintenance. In: Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073), pp. 220–229 (2000).
16. Reed, D.P.: Naming and Synchronization in a Decentralized Computer System. Ph.D. thesis Massachusetts Institute of Technology (1978)
17. Revilak, S., O'Neil, P., O'Neil, E.: Precisely serializable snapshot isolation (pssi). In: 2011 IEEE 27th International Conference on Data Engineering, pp. 482–493 (2011)
18. Stonebraker, M.: The design of the postgres storage system. In: Proceedings of the 13th International Conference on Very Large Data Bases, VLDB '87, pp. 289–300. Morgan Kaufmann Publishers Inc., San Francisco (1987)
19. Stonebraker, M., Rowe, L.A., Hirohama, M.: The implementation of postgres. IEEE Trans. Knowl. Data Eng. **2**(1), 125–142 (1990)
20. Sysbench Benchmark. https://github.com/akopytov/sysbench
21. Tencent Distributed SQL System (TDSQL). http://tdsql.org
22. WeChat. https://weixin.qq.com
23. Wu, S., Ren, W., Yu, C., Chen, G., Zhang, D., Zhu, J.: Personal recommendation using deep recurrent neural networks in NetEase. In: 32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016. pp. 1218–1229 (2016)
24. Yabandeh, M., Gómez Ferro, D.: A critique of snapshot isolation. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 155–168. ACM (2012)
25. Zhang, C., Sterck, H.D.: Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. In: 2010 11th IEEE/ACM International Conference on Grid Computing, pp. 177–184 (2010)
26. Zhang, D., Li, Y., Cao, X., Shao, J., Shen, H.T.: Augmented keyword search on spatial entity databases. VLDB J. https://doi.org/10.1007/s00778-018-0497-6 (2018)