A DIY Approach to Uni-Temporal Database Implementation *

Haitao Yang, Fei Xu, Lating Xia
GuangDong Construction Information Center (GDCIC)
Guangzhou, China

***Abstract*—when historical versions of data are concerned for a MIS (Management Information System) we naturally might resort to temporal database products. These bi-temporal products, however, are often extravagant and not easily mastered to most of MIS practitioners. Hence we present a plain DIY (do it yourself) solution, the Audit & Change Logs Mechanism-based approach--ACLM, to meet the uni-temporal requirement from restoring historical versions of data. With ACLM programmers can code SQL scripts on demand to trace and replay any snapshot of historical data version via RDBMS built-in functions, they need not to shift away from their usual way of coding stored procedures for data maintenance. Besides, the ACLM approach is compatible with meta-data change, and its additive overhead was instantiated imperceptible for throughputs of routine access with a typical scenario.**

Keywords—DIY solution; recurrence; historical snapshot; uni-temporal database; MIS

I. INTRODUCTION

Recent years' practices in developing web creditable MIS for numerous users made us all realized that maintaining enormous records of users-oriented data are an unbearable task to anybody (individuals or organs). Regarding that it is users or clients' own right and responsibility to keep their delivered data valid and complete, we were looking forward to an interactive and sharing pattern that all users involved should honestly maintain their information themselves, which is perhaps the only workable approach. Then, there comes the risk of abuse of self-maintenance right. To prevent such a risk we have no feasible solutions of instant response but can build a final line of defense by an ex post facto measure that is, logging all behaviors of maintaining data and offering a facility to restore or reveal any historical version of concerned data and the responsible manipulators. Herewith we get to the field of uni-temporal database application, and might assume products of temporal DBMS (in short, tDBMS) as a matter of course. tDBMS products, however, are bi-temporally-oriented, and not familiar to most of practitioners in ordinary MISs. Even worse, such products often offer extra functions well beyond need and bring with much greater complexity and higher cost than expected. Upon these considerations, we turned to explore an exercisable and methodological approach (that why we refer to it as DIY -- do it yourself).

II. RELATED WORK AND OUR DIRECTION

In realistic applications, data recorded in databases all have certain time properties either explicit or implicit, at least those indicate when the data are valid and when they are recorded [6] — the former is a time property with data semantics, classified as the valid-time property, and the latter is a time property with data operation, categorized into the transaction-time (time of manipulating data) property. Contemporary RDBMS products have granted us the ability to straightly store and manage all relational data including temporal data in the same database, notice that data's time property is also a datum. To certain extent, temporal data is an issue of data versions that concerns with data recordation at different times. It is not feasible for all time versions of data are treated in equivalence, since eventually the ever-increasing amount of historical versions will become overwhelming on all aspects of data storage and usage. Approaching such a problem and its related, a study of temporal DBMS has been developed for decades [1]. Despite lots of research on tDBMS, practical tDBMS products are rare, and even more, most of them are in fact an extension of traditional RDBMS [11], in general developing a tDBMS application is still a tough and often individualized task for a MIS (Management Information System) developer team.

In usual practices of developing MIS, we normally design and develop a database application around usage of the newest data version, because in default, people much concern with the current status rather than those historical. If no requirement of recalling a historical "snapshot" (we use this term to refer to *a picture of data at a historical time*), historical statuses of data will be updated or overlaid by the newest one, and everything is just simple as usual. But if the responsibilities of conducting data change is of the concern, e.g., they must be audited or traced afterwards (a lot of crucial MIS applications have this requirement), in which historical statuses of data need to be carefully and explicitly addressed, we actually step in the scope of tDBMS. Up to now, tDBMS approaches in mainstream, such as the famous ATSQL2 proposal are conducted in a direction of treating time properties of data as an abstract or super attribute with special disposal [10], and they are based on relational data processing, of which the most concerned are often associated with their special temporal data type, temporal manipulation on table or column-level, and dedicated temporal relation constraints etc. In these practices, accordingly, special time attribute-oriented extensions to SQL must be introduced to and well supported by tDBMS. Despite temporal SQL-compliant research has been very comprehensive these days, however, the related SQL-level support mechanism, temporal database model theories, etc., are sophisticated and too much for most of applications just involved with some plain temporal requirements as in ordinary MIS. Most of MIS practitioners would rather treat temporal parts of MIS applications in a similar way as in ordinary DBMS programming practice, e.g.,

* This work is sponsored by Guang-Dong Construction Information Center.

assigning each intended time property of data into a concrete data attribute, and so on. Thereby, we prefer a system mode of "traditional RDBMS" + "software". Here ,the "software" could be programs as a part of the hosting application itself, or in an embedded type as a third-product software product (often as a middleware), e.g., the well-known TimeDB [12] is a RDBMS-based embedded middleware for temporal data applications, TimeDB runs as a frontend to the hosting RDBMS (e.g., Oracle) and supports the temporal query language ATSQL2, where finally ATSQL2 statements are compiled by TimeDB into (sequences of) SQL-92 statements which are executed by the underlying RDBMS backend. The pure temporal disposal part (software) of TimeDB is to interface between the temporal usages (delivered in ATSQL2 statements by users) and SQL-92 executions.

As we know, tDBMS products such as TimeDB often store the transaction-time of data in the same data tuple. Such a device is inefficient for OLTP (On-Line Transaction Processing), considering that if data items are frequently updated, the data table where these items reside will soon be overwhelmed by historical versions of data, which we figured as a so-called 99 to 1 % phenomenon, i.e., 99% (symbolizing most) data records are for past statuses while 1% (symbolizing a little proportion) for the current or latest status, while most of accesses to the data table are just for the 1% records. On this aspect, we believe that many MIS practitioners like us would rather try an on-hand and less costly scheme than take an abstruse academic approach or purchase an often too costly or heavy tDBMS product.

For generality and practicability, basing on the above consideration and rules of engineering we believe that a good approach for the issue discussed should be a methodological one with ease of use or duplication, in the other word, a DIY (Do It Yourself) type, and which should adopt an outline pattern of separating historical versions of data from the current one, and provide guidelines for designing fundamental maintenance, management and utility services of data. Along this direction, we start our approach by introducing several key concepts in a simple but typical example about temporal data recordation:

P1 = ("John", 2000, *interval_1*) | a data record with its valid time indicating *John* has a salary of two thousands dollar.

P2 = (P1, 11/5/2012 4:44 PM) | the above data record P1 was created or updated at 11/5/2012 4:44 PM.

The statement from P1 is true only for its valid time period *interval_1*, but what from P2 that recorded a fact is always true. The valid period *interval_1* can be definite as [*time1*, *time2*], or indefinite as [*time1*, *unknown*] that spans from *time1* until something happens (e.g., John's salary is changed or he is dismissed, etc.) when the *unknown* becomes a certain value. Generally we are not likely to maintain valid time properties via an automatic mechanism since they are associated with concrete semantics of the data they modify, as in the above simple example, when and how to make the unknown time known is up to the intelligence of realizing the corresponding event and its relevancy; but it is different with transaction time properties since they simply denote a data manipulation event that can be monitored via certain DBMS built-in mechanisms. We doubt in nature there is any universal automatic scheme to cope well with storage, management and usage of valid time property of data, despite lots of techniques on related issues. In the other side, for a real relation object its valid time attribute and its other attributes are all in an equal position with respect to relational data theory and application semantics, thus they could be and should be treated equally if convenient.

Further, we clarify three key facts which are often ignored: 1) the **valid-time property** of data virtually can only be actively determined by who understanding the data meaning, perhaps an intelligent software can do this, but developing an intelligent software is far beyond the scope of applied tDBMS research; 2) for web data applications, the responsibility audit about data manipulation could not be done within DBMS since conventionally different web users share a common DBMS account; 3) data structure changes cannot be excluded in real applications, for instance, adding or retiring a field in a data table (in practices, a *relation* is often instantiated as a table of records, an *attribute* as a *field* of record in the table) for one or other reason is allowable.

Accordingly, we have three keynotes for a feasible tDBMS implementation: (1) the valid-time property of data should be considered in the context of data application; (2) the responsibility audit of data manipulation needs participation from higher layers of application outside DBMS; (3) a good implementation mechanism for tDBMS should be compatible with ordinary structure change of data tables. Following these guidelines, towards a generic design scheme for tDBMS-related MIS applications we focus on the **transaction-time** property of data (so we call this approach as of the *uni-temporal* database implementation) and treat it as a common attribute [9], this is contrasted with the nowadays so-called "bitemporal database" [11], i.e., a general temporal database.

III. V+A Frame For TDBMS Applications

Along the above decided direction of investigation, the underlying thing is to set up a software frame for tDBMS applications.

Firstly, concerning temporal evolvement of data content we noticed two main disposals of transaction-time of data: (1) using self-contained temporal recordation of data manipulation, often in a vitae form, i.e., each maintenance manipulation on a data item should append its execution time to all data records changed, there is no need for additional logging mechanism; (2) devising dedicated temporal logging mechanism for recording data manipulation activities, which notes down the transaction-time in a log separate from the data table.

Secondly, for the sake of practicability we shall take into account a common phenomenon of meta-data change — change to the composition of data table's PK (primary key), such as using different component fields or altering the data type of some component fields — which was often ignored by most of tDBMS approaches.

Thirdly, Application requirements on storing, managing, and hereby using time properties of data are versatile, but for usual cases of MIS they can be classified into two types: the time property that is used frequently or routinely should be accessed easily, whereas the others without routine usage can

do with less convenience of access for a much lower cost of implementation.

Accordingly, we proposed the Vita + Audit (in short, V+A) frame. The kernel of V+A frame consists of DRB (direct-retrieval base) + ACLM (audit & change logging mechanism). DRB is used to store usual data content, i.e., current-status data and temporal recordation in a vita form for direct accesses of routine transactions. Contrasting with DRB ordinary services for direct content access, ACLM is for dedicated audit accesses regarding data manipulations with transaction time — it keeps trace of each activity of data manipulation, memorizes into the change log the data snapshot of the data version just before the data manipulation exerts on DRB each time, and at the same time inserts into the audit log a record about what kind of data manipulation and who makes that manipulation. Under ACLM we should not miss any historical version of data being audited though we could not view directly its content in a single SQL manipulation.

In general, data query operations need not being logged in the audit log except for applications with extremely high safety demand since they do not create any new version of data, neither content of an *insert* operation needs recordation since it has no previous version. But the *insert* operation itself shall be recorded in the audit log in order to restore historical versions of data table before the operation timestamp. One main usage of the audit log is to record the responsible subjects of data maintenance – the actual operators from client end (terminal users) instead of those common DBMS accounts on the web data layer. In convention of software industries, user identity certification for web applications is fulfilled before calling functions of web data layer, and normal accesses to a web DBMS are requested via some shared DBMS account. To log the identity of a user (who instructed DBMS to execute a data manipulation) into a record of the audit log, the web user certification information should be passed into a corresponding inner procedure of the web DBMS. Manipulation of changing data content (**U**pdate, **D**elete, or i**N**sert) and its recordation in the audit and change logs should be treated within a single DBMS transaction as an atomic action (either both succeeded or anything they did will be withdrawn completely afterwards). Such transactions shall be coherently fulfilled through a stored procedure of DBMS script on the intermediate layer.

A. Audit Log

For each application the audit log is unitary, it is used in a way similar to keeping accounts of any change or comment on any data record of DRB: (1) recording any SQL-update, delete, and insert manipulation; and (2) logging any responsible comment on a data snapshot. The former is oriented to generic syntactic audit while the latter is about important semantic audit. The audit log in ACLM is application-oriented, i.e., all data tables from the same application share a unitary audit log. The structure of the audit log is defined as relation *Adt_log* described in Table I (data type in this paper are all given as in Oracle DBMS). Complementarily, a PK specification defined as relation *PK_spec* in Table II is introduced for all involved versions of PK structure of each data table from the same application, where one row of specification is for a member attribute of a PK.

TABLE I. AUDIT LOG (RELATION ADT_LOG)

Seq.	Attribute name	Data type	Remark
1	*Audit_ID*	Varchar(32)	The PK attribute
2	*D_table_name*	Varchar(32)	
3	*D_key_value*	Varchar2(128)	Convert to String
4	*timestamp*	date	Time and day
5	*comments*	Varchar2(512)	
6	*Operation_type*	Char(1)	C/D/U/N
7	*Operator_id*	Varchar(20)	
8	*signature*	Varchar(172)	Sha1RSA

TABLE II. PK SPECIFICATION (RELATION PK_SPEC)

Seq.	Attribute name	Description	Data type
1	*D_table_name*	Be referred in Table I	Varchar(30)
2	*PK_attribute_name*	PK member attribute	Varchar(512)
3	*PK_attribute_seq*	Sequence number of this member attribute	Integer>=0
4	*Struct_Valid_S_time*	When this PK structure became valid	Date

Further explanation of the audit log's definition and related usage are detailed as follows:

1) The basic design of the audit log is applied directly to data tables with a single attribute PK (unitary PK). In relation Adt_log, attributes D_table_name is used to store the name of the data table being audited, D_key_value is used to store the PK value of the data record being audited. As to data tables without a unitary PK, more additive disposal is needed, see later in section IV. In relation PK_spec, PK_attribute_seq=0 is corresponding for unitary PK cases, while PK_attribute_seq>0 for non-unitary PK cases.

2) In fact, all MISs should know the PK composition of their data tables via something like PK_spec in advance of executing data maintenance. With PK_spec we need not include PK_attribute_name in Adt_log, which can prevent a transition dependency <D_table_name, PK_attribute_name> occurs in Adt_log.

3) To cope with meta-data changes, Struct_Valid_S_time attribute of relation PK_spec is set to indicate the start time that a version of PK composition became valid. The expired time of a valid PK composition is given subsequently by a next value of Struct_Valid_S_time in sequences for the same data table.

4) Attribute Operation_type has a set of basic values {D (Delete), U (Update), N (iNsert)} and an extended value C (Comment). Attribute Comment is used to record any responsible literal comment (including endorsement) on the data record being audited, it is left empty (assigned a null value) when Operation_type<>C. Value usages of attribute Comment can be extended and further categorized if needed in applications, e.g., classified into Censor and Verification, etc.

5) Attribute Timestamp is used to note down the time when the current audit record was created. To avoid ambiguity it is stipulated that the time of a web server's clock be adopted, and relevantly-logged data changes take effect just after the instant of Timestamp.

6) Signature is for storing the result of RSA calculation of Hash value of objects being signed by terminal users with their private key [8, 3]. A signed object consists of all content attributes (except maintenance and auxiliary attributes) of the data record under audit, and attributes from seq. 1 to 7 in Adt_log. The Hash value of a signed object is computed on the concatenated contents (all converted into the string type) of each involved attribute.

7) Any Update manipulation to alter a PK's value shall be equivalently decomposed into a Delete manipulation on the data record with the present PK value, and a subsequent iNsert manipulation of the updated data record with a new PK value.

Locking a data table during submitting a comment on its data record could lower the risk of mismatching the comment with a newer data version that was being created in the same time. Of course, freezing the data table for the whole process of comment action can exclude such a risk completely, but which will bring along with a more serious problem that normal data maintenance might be blocked for an uncertain (at worse often rather long) time by some comment activity, and the situation probably become even worst if the comment right is abused. Thus we in practice shall set a threshold of time limit for locking (e.g., 10 minutes) to avoid involving sophisticated lock/unlock mechanism. Conclusively, we have several more principles of using audit log:

- Applicable to record verification results in a generalized form of literal comment.
- To log each behavior of deleting, inserting, updating or verifying a data record, and the manipulator's digital signature about the essential content of the audit record in the same audit record.
- Separating historical data's storage, i.e., they are kept elsewhere (in the change log).

B. Change Log

The direct usage of change log is to record any data version just before it become outdated, which enables the occurrence of any historical data snapshots later. We shall record in the change log the current value of each data attribute bound for a content change just before the change operation is carried out, and the change operation being taken shall be noted down in the audit log at the same time. The data structure of change log is as defined in Table III, where attribute *valbfchg* stores the value-before-change for attribute *chgfldname* that stores the name of an attribute undergoing a value change, while the expiration time of content of *valbfchg* is indicated by attribute *timestamp* from a correlated record (being correlated through the value of *Audit_ID*) in the audit log *Adt_log*. For example, *Adt_log.timestamp="time1"*, *Chg_log.chgfldname="name"* and *Chg_log.valbfchg="John smith"* specified that data attribute *name* had a value of *"John smith"* just before time *"time1"*.

The value-before-change of each data attribute (indicated with the content of *chgfldname*) that underwent a value change, except of lob type (Clob/Blob) shall be consistently converted into the string type and then put in attribute *valbfchg*. If a changed attribute is of lob type, its value-before-change shall be deposited in attribute *lob_value* while *valbfchg* is left empty. For data attributes of binary lob type, we shall use an additive attribute *ContentType* [4] to further specify their content type to facilitate web applications for presenting such content.

TABLE III. CHANGE LOG (RELATION CHG_LOG)

Attribute name	*Description*
Audit_ID	a PK attribute
Chgfldname	a PK attribute
valbfchg	Direct value before change
datatype	String/clob/blob
Lob_value	If dataype=C/Blob
ContentType	For lob type data
Hash	For lob type data
Chg_act	U/D

C. ACLM Operation

The procedure of ACLM operation is outlined as follows:

1) A web service of data maintenance calls a DBMS stored procedure [7] to execute a dedicated data manipulation (iNsert | Update | Delete | Comment), noticing that C type operation is writing to the comment attribute of the audit record;

2) The DBMS stored procedure fulfills the data change on each target data table and correspondingly inserts a new audit record into the audit log with the matched type assigned to the attribute Operation_type within a single transaction;

3) Triggers of each target data table are ignited [7] to insert corresponding log records into the change log.

This software mechanism is illustrated as Fig.1.

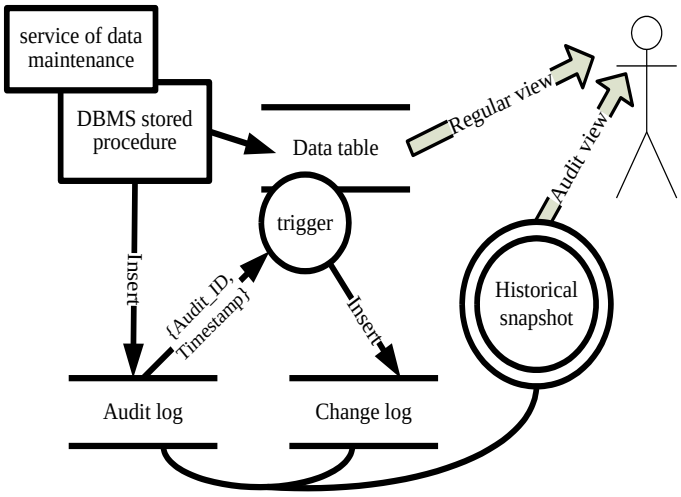

Fig. 1. Diagram of tDBMS software operation

Under ACLM, whenever calling a DBMS stored procedure to fulfill a process of data change (Update, Delete, iNsert) or data inspection (Comment) it is requested to specify whether to simultaneously write the audit and change logs, if yes (ACLM function enabled normally) then the calling program shall also ascertain the timestamp of logging a record into the audit log through an interface of the invoked DBMS stored procedure, regarding that a web server's clock is adopted, see paragraph 5) of subsection III.A. Since digital signature can only be made in

client ends where user private keys are available, and attribute *timestamp* in relation *Adt_log* is one of digital signature objects, thus the client end must get the time of web server just before user making digital signature. An individual ACLM operation involves two basic actions: A1 — exerting data changes, and A2 — logging such actions in both the audit and change logs. We propose not to execute A1 and A2 in separate web services, since combining two web services as a transaction (all done or nothing) will involve very sophisticated disposal, e.g., if A1 is successful but A2 is not, A1 has to be rolled back, then we have to pay off the cost of rolling back A1 due to A2; and what is more, sometimes (due to poor communication qualities) we cannot judge if A2 is successful or not (it might succeed but its reply was lost or simply delayed), unless we decide according to a time limit, but specifying a time limit is a trade-off issue, often very subtle.

Thus, we shall request a DBMS to conduct both A1 and A2 via a single web service call, which means whenever submitting a data change call we should have the companying *signature* value prepared for the audit record at the same time (must in advance obtain the values of *timestamp* and *Audit_ID* as parts of content to sign).

The above description implies a sequence of steps for carrying out a data change manipulation under ACLM: (1) pre-read content of a target data record to prepare the change action, (2) get the web server's time for digital signature, (3) submit the manipulation request to the hosting DBMS via a web service, (4) log the manipulation into the audit and change logs. Here, we have an order of timestamps: *timestamp* (pre-read) < *timestamp*(sign) < *timestamp*(submit) < *timestamp*(log). For better uniform simplicity, *timestamp* (pre-read) is adopted to substitute the rest of timestamps. This is because:

(1) it wouldn't influence consistency of retrospecting historical snapshots; (2) be competent for re-showing historical snapshots for cases without demand of extremely precise accuracy; (3) it is impossible within a one-off web calling to include into the digital signature a precise time of writing data table. Regarding that the web server functions as the centrum of ACLM, it is proper to grant attribute *timestamp* with the reading of the web server's clock.

Note:

1) Triggers of data table need to read contents of attribute Audit_ID and timestamp from the corresponding record of the audit log. Each record of the change log correlates to a unique record of the audit log, whereas each record of the audit log correlates to a group of records of the change log except those audit records of non-change type (no changing any existent data, e.g., insert or comment type), records from both logs are correlated via values of a common attribute Audit_ID.

2) If a round of data change process begins at a halfway phase (one or several rounds of data change were executed before, but none of them are regarded complete, i.e., all of their execution results are halfway, and saved into their DRB data table temporarily), then we strongly suggest that only enabling ACLM logging function for the first round of operation process since all midway versions of data change in the same process transaction need not logging.

3) We shall not enforce ACLM function indiscriminately for all data tables without considering the additive overhead. For example, when all historical data versions of a data table are in fact presented as direct content, there is no need to log data changes anyway. In developing ACLM for MIS, we suggest to set up a configuration table of data change audit individually per application to specify together all involved data tables and their involved fields whose value's change need to be logged.

D. Typical Applications of Audit & Change Logs

1) Restore a record's snapshot at an audit timestamp

Let's take a scenario of reverting to a historical snapshot of data record at an audit timestamp. For data table *Tx*, let *Ax* be the audit log record with the timestamp *ts*, *Kx* be the PK value of *Tx's* data record *Rx* that was audited by *Ax* at *ts*. If *Ax* is of C type, we have to track down along the time axis to the point (if any) when *Rx* underwent a change (update or deletion) after *ts*, see Fig. 2.

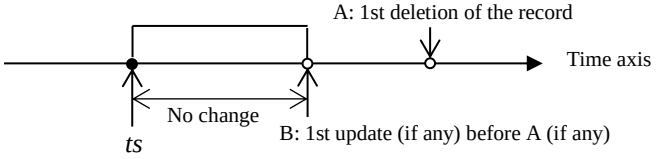

Fig. 2. Snapshot Recurrence Scene concerning a Specific Attribute

```
Create table snapshot(fieldname string not null, value string null, auditID
  string null, chgtime timestamp null);
snapshot_D obj=new snapshot_D(ts, Tx, Kx);
Class snapshot_D {
Var timestamp D_time=null;
Public snanpshot_D(timestamp tstamp, string tname, string keyval) {
Insert into snapshot (fieldname) select column_name from cols where
  table_name=tname;
Var string audiID=null;
Select min(a.timestamp) into D_time From adt_log a Where
  a.timestamp>=tstamp and a.D_table_name=tname and
  a.D_key_value=keyval and Operation_type="D";
If D_time=null then exit;  /* after tstamp Rx was not ever deleted */
Select a.Audit_ID into auditID From Adt_log a Where a.timestamp=D_time
  and a.D_table_name=tname and a.D_key_value=keyval and
  a.Operation_type="D";
Update snapshot s Set s.value = (Select b.valbfchg from chg_log b Where
  b.Audit_ID=auditID and b.Chgfldname=s.fieldname);}}
```

Fig. 3. Recurence of Rx snapshot just before Rx's first deletion after *ts*

Regarding that *Ax*'s existence implies *Rx* existed at *ts*, so we can restore the snapshot of *Rx* at *ts* through the following processes (be succinct, no datum of lob type is involved here):

Step 1: We shall check if *Rx* underwent a *D* type change or a PK value change after *ts* — the later is equal to a *D* type change being followed by a *N* type one, since in ACLM it is implemented by deleting the current record and subsequently inserting a record with the new PK value — if *yes*, then we shall firstly restore the snapshot of *Rx* just before *Rx*'s first deletion after *ts*, e.g., just before point A as in Fig.2.

```
snapshot_rollback roll_obj=new snapshot_rollback(ts, obj.D_time, Tx, Kx);
Class snapshot_rollback{
Public snanpshot_rollback(timestamp tstamp, timestamp Dtime, string
  tname, string keyval) {
SET ANSI_NULLS OFF;
Create view aud_chg_v AS  /* (var >= null) == false */
Select a.Audit_ID, a.D_table_name, a.D_key_value, a.timestamp,
   b.Chgfldname, b.valbfchg From adt_log a, chg_log b Where
   a.D_table_name=tname and a.D_key_value=keyval and
   a.timestamp>=tstamp and not(a.timestamp>=Dtime) and
   a.Operation_type="U" and a.audit_ID=b.audit_ID
Update snapshot s Set s.chgtime = (Select min(c.timestamp) From
   aud_chg_v c Where c.chgfldname=s.fieldname);
Update snapshot s Set s.auditID = (Select c.Audit_ID From aud_chg_v c
   Where c.timestamp=s.chgtime and c.chgfldname=s.fieldname);
Update snapshot s Set s.value = Coalesce((Select b.valbfchg From chg_log
   b Where b.Audit_ID=s.auditID and b.chgfldname=s.fieldname),
   s.value);}}
```

Fig. 4. Reverting to Rx's snapshot at time *ts*

The process of this step is illustrated by a procedure as coded in a pseudo-java+SQL language in Fig.3, where it is fulfilled by creating a class instance *obj=snapshot_D*(*ts,Tx,Kx*) regarding that *obj.D_time*!=*null* indicates *yes.*

Step 2: We shall search the earliest *U* type change during [*ts*, *obj.D_time*) for each attribute of *Rx*, and roll back together such a change (if any, e.g., at point B as showed in Fig.2) for any attribute whose value was changed during [*ts, obj.D_time*) to get the snapshot for *Rx* at time *ts*. The task of this step is illustrated by a procedure as coded in a pseudo-java+SQL language in Fig.4, where it is carried out by setting *roll_obj=snapshot_rollback*(*ts,obj.D_time,Tx,Kx*), regarding that *obj.D_time=null* means no change of D type was on *Rx* after *ts*, and then *not*(*a.timestamp>=Dtime*) becomes true accordingly due to *Dtime=null*.

Notice: different fields of a data record might undergo a content change at different times. The correctness of executing SQL scripts in Fig. 3 and 4 (where the procedure of opening database is omitted) rests with that all audits on the same object are sequential, i.e., no more than one audit action exerting on the same object is allowed at the same time.

2) Recuring to a table's snapshot at arbitrary time

To recur to the snapshot of table *Tx* at a given time *ts* is to restore exactly all data records that appeared at *ts*. Supposed ACLM has been functioning since *ts*, and then the recurring procedure can be outlined as follows:

a) To retrieve those Tx's records that have not undergone any change since ts, we have

S_1*={select * from Tx where (Tx's PK) not in (select D_key_value from adt_log where D_table_name="Tx" and operation_type<>"C" and timestamp>=ts)}.*

b) To restore S_2*={those Tx's records that existed at ts and underwent a change after ts}, we shall collect the set* S_c *of any audit record that logged the first change data operation on the same data record after ts:*

S_c*={select Audit_ID,operation_type from adt_log where timestamp in (select min(timestamp) from adt_log where D_table_name="Tx" and operation_type<>"C" and timestamp>=ts group by D_key_value)};*

and then we exclude those audit IDs whose audit records logged a *N* type operation (As to a data record, after *ts* the first change data operation is of *N* type implies that the data record didn't exist at *ts* otherwise it can not been inserted):

S_a*={select Audit_ID from* S_c *where operation_type<>"N"}.*

c) For any audit record Ax whose ID is in S_a *we restore the snapshot of the corresponding data record at Ax's timestamp through a process described in subsection III.D.1). It is easy to prove that* $\forall x \in S_2$ $(\exists y \in S_a)$ *that y logged the first U or D change of x after ts, whereas* $\forall y \in S_a$ $(\exists x \in S_2)$ *that the first U or D type change of x after ts was logged by the audit record with ID = y. As a result of the above process, we can regain each member of* S_2*. The* $S_1 \cup S_2$ *is the wanted.*

3) Practical simplicity for efficiency

The above applications are generally-oriented that each time a data record underwent a *U* type change the ACLM logged only those attributes that had undergone actual content change and their values. In practice, however, it was very clumsy to tell which field's value of data input interface has actually been made different from its existent value in a web submission of data maintenance input, and if such actual value change is judged within a trigger procedure then the execution efficiency of the trigger and thereby the hosted DML operation will be greatly abated. For the sake of simplicity and efficiency, the data input submitted (if accepted) is directly delivered to DBMS for an update operation to replace the existent values of the object attributes respectively without further distinguishing the existent and new values. In addition, to make easier the snapshot recurrence we should set ACLM to log all attributes and their existent values into the change log at the same time whenever a *U* type operation encountered though at more cost of storage space. Such a disposal can save a lot of computation, regarding that in this way each time an audit record closest behind to the given time is enough for snapshot recurrence without bothering to dig out all involved audit records for all attributes' snapshots that were logged at different timestamp.

IV. Testing ACLM's Influence On DBMS Operations

Applying ACLM means appending certain additional audit and log tasks to normal DML operations in exchange for the competence of tracing versions and their responsible persons. There comes an issue of evaluating additive overheads from ACLM. Intuitively, we have two plain measurements for this evaluation: (1) *the perceptible performance decline*, (2) *the increment in comparative execution time*. In fact, as for MIS applications featuring human-computer interaction it is the measurement (1) much more suitable than (2) though the latter is more precise than the former. The *perceptible performance decline* can be well evaluated in terms of success ratios of maintenance operation per unit time for a normal range of request throughput into the hosting DBMS. On this aspect we made a succinct test to checkout if the ACLM influence is acceptable or not: per one minute how many percentage of update or delete operations succeeded for a large scope of operations throughput. The data table for test is defined in Table IV, its content attributes (*Citizen_ID_num*, *Reg_name*, *Reg_text*) were under audit of ACLM. For comparison, during the test twin instances of relation *Cer_Reg* were created such that one instance is ACLM-enabled while the other is non-ACLM, and both were exerted hundreds of thousands *Update*

and *Delete* operations. The test results were depicted in Fig. 5 and 6.

TABLE IV. CER_REG

Seq.	Attribute name	Data type	ACLM
1	*Citizen_ID_num*	Number(18)	X
2	*Reg_name*	Varchar(32)	X
3	*Reg_text*	Varchar(1024)	X
4	*timestamp*	timestamp	
5	*Process_Status*	Char(1)	
6	*Lock_Person*	Varchar(32)	
7	*Operator_id*	Varchar(20)	
8	*signature*	Varchar(172)	

For a widely comparison of significance, we simulated a broad range of data maintenance frequencies covering and well beyond the statistic scope of **p**age **v**iew (PV) of our web site (a well known industry website in our province) whose home page was accessed about 350 times per minute (on average the frequency of home page access corresponds to that of the web underlying DBMS inner DML request in the case of data maintenance) at its all time peak -- in Fig.5 and 6 the peak PV value was indicated by the vertical green line. Both Fig.5 and Fig 6. actually recorded how many operations were successfully fulfilled within one minute.

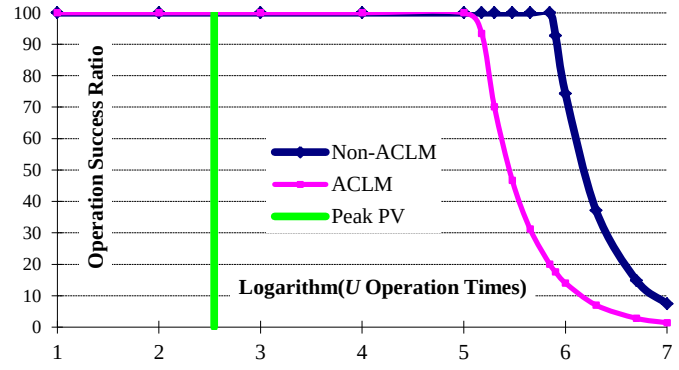

Fig. 5. Test results on Update (*U*) operations

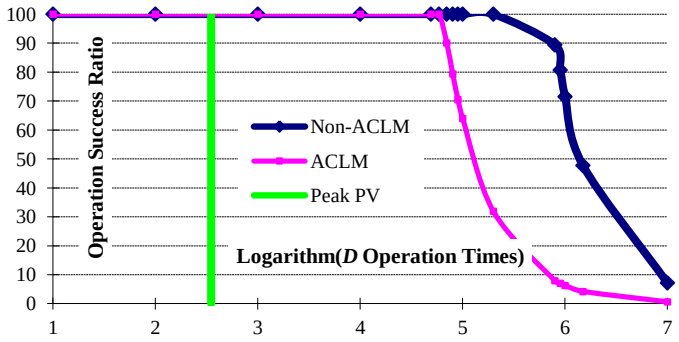

Fig. 6. Test results on Delete (*D*) operations

As in Fig. 5, both the ACLM-enabled (labeled ACLM) and ACLM-disabled (labeled non-ACLM) cases had similar performance lines (all in values of logarithm to base 10) for *Update* operations. Their success ratios hold 100% from the start at lower throughput of data operations up to some thresholds well above the site peak PV frequency, and then dropped rapidly as the impact of access throughput became very heavy. Although the success ratio of the ACLM-enabled system dropped relatively earlier, its abrupt decrease point only came about as the access frequency reached an extremely high level, say 150,000 times of inner update operations requested per minute in the test, which is pretty rare for a normal website and can be referred to as an ultra situation. Fig.6 told the same thing about *Delete* operations, where, the ACLM-enabled one encountered a threshold of success-ratio drop at 70,000 times of inner delete operations per minute, and the test case should be referred to as covering ultra situations too. Moreover, the higher the computing power of server and client ends became, the less the MIS performance loss from ACLM would be, regarding that the hardware configuration of the test is quite low (the hosting Oracle 9i DBMS was mounted on a Dell PowerEdge2950 PC server -- Xeon E5430 CPU with a 2.66 GHz frequency and 1G memory), all these showed that the performance decline of routine DBMS operation due to ACLM is normally acceptable or even neglectable.

V. FURTHER CONCERNS ABOUT ATTRIBUTES OF LOGS

A. *Log Attributes' Minimization*

First, relation *Adt_log* cannot be more simple, the reasons are: (1) attributes other than *Audit_ID* are semantic and all indispensable to specify a data manipulation; 2) the auxiliary attribute *Audit_ID* serves as a foreign key in *Chg_log* to correlate together all data attributes being logged there for the same data manipulation. Next, it is easy to verify that *Adt_log* is in the third normal form (3NF), while *Chg_log* is nearly of 3NF except a functional dependency <*Lob_value*→*Hash*>. Although <*Lob_value*→*Hash*> brings some redundancy (a list of *Hash* result), it in return offers a well-balanced performance or efficiency in the value comparison of judging if the content of lob type attribute underwent an actual change, regarding that a lob type attribute could have an unlimited variety of content data sequence length, but it can be stood for by its fixed length *Hash* value in comparison computing.

B. *Logs Construction's Completeness*

The proposed audit & change logs enable restoring any data record's value (if existed) at an arbitrary given time *ts* for data tables under ACLM governing: at first with them we can check if the data record underwent a change manipulation after *ts*; if no, the current status of the data record is the wanted, otherwise we can take steps as described in subsection III.D.1) to restore the data record's value at *ts*.

C. *Compatibility with data structure change*

First, adding or retiring a non-PK attribute would make no difference on restoring historical snapshots since the name and value of newly added or historical attributes had been recorded in attribute *chgfldname* of *chg_log* if they underwent a value change. Secondly, altering data type of a non-PK attribute would not mislead comprehension of the relevant record content (if any) logged by the change log, since the attribute's previous content was always recorded in a uniform type of character string (via *toString* translation) each time the attribute content underwent a value change. Besides, an alteration of data type without change content is of application-specific disposal which is none of the business of ACLM. Notice: normally retiring a data table field means that the field is no longer maintained but its previous values are still kept there.

Similarly we can elucidate that ACLM still performs its normal functions when data tables undergo a structure alteration in their PK attributes (attributes were added or retired, or their data type were changed) with the complementary PK structure specification from relation *PK_spec* (see Table II). So, ACLM is compatible with data structure or meta-data change.

VI. Handle Primary Key Of Multi-Attributes

The above ACLM solution has a precondition: all data tables governed must have a unitary-attribute PK. In practices, it is the 1NF [2] instead of a unitary-attribute PK that shall be the minimal requirement for relational schema designs, i.e., there is a PK (probably with multiple attributes) to exclude duplicate rows -- under 1NF at least all attributes together can uniquely fix on an instance of tuple. As to a data table without a unitary-attribute PK we shall introduce an artificial attribute to stand in as a unitary-attribute PK — named the Stand-In Unitary key, in short SIU key. Here, we propose two ways to set up an SIU key:

A. Map Multi-attributes into a unitary one

Let PK= $(F_1, F_2, \cdots, F_k)$, which is an ordered tuple of names of all attributes from the PK, where the subscript numbers are defined by the *PK_attribute_seq* in relation *PK_spec* regarding the *PK_attribute_name* (see Table II). If $\forall j\ (\mathrm{chr}(d) \notin \mathrm{str}(F_j))$ and $\forall j\ (F_j \leftrightarrow \mathrm{str}(F_j))$ we have

$$(F_1, F_2, \cdots, F_k) \leftrightarrow str(F_1) + \sum_{j=2}^{k} (\mathrm{chr}(d) + \mathrm{str}(F_j)) = \mathrm{SIU}\ .$$

Where, chr(*d*) denotes the character whose decimal ASCII code is *d* providing that chr(*d*) shall not appear in content of any involved relation attributes on high level MIS applications, "+" is the concatenating operator of character string, *str*(X) is the function that converts the value of variable X into a character string, " $\leftrightarrow$ " stands for a one-to-one mapping relation. Normally we choose *d*=24 for chr(24) is a non-printable character for a cancel signal in hardware control. Further, we recommend to turn SIU into a fixed length by *Hash* (SIU) for a better space efficiency and well balanced performance of comparative computation. Often a hash algorithm named *SHA1* is adopted to map different SIU values into distinct strings of 40 hexadecimal characters.

B. Create a DBMS self-maintained field as SIU key

Such built SIU key is normally self-incremental, it can label distinctly data records existed, and grant each newly inserted data record with a unique identity. In semantics, a SIU key is equal to its original PK for all *Update* manipulations except those altering any existent PK value. Whenever a data record underwent first a *Delete* and subsequently an *iNsert* operation (an equivalence of the *Update* operation altering an existent PK value), it will be assigned a new SIU key value different from its previous ones.

In this sense, any audit record of *N* type does not link to a historical snapshot of the data record it audited with respect to the PK value of that data record, because the PK value of the newly-inserted data record should not appeared before (the audit timestamp). As usual, the audit records of *N* type are used to exclude data records that are inserted after a given historical time in restoring the historical snapshot of a whole data table from its current status. But for restoring a specific data record with a given PK value (each member attribute value is given) at a given time it turns to be rather clumsy: we need to search the change log thoroughly for any *Audit_ID* value that is with each member attribute of the PK in change log records, and each member attribute from these records at least matched one time with their given value, this is because we don't know directly the SIU value at that given moment. But, if ACLM is merely oriented to the time property of data maintenance and historical snapshots of data table, this SIU key approach is to some extent simple and feasible.

VII. Discussion And Conclusion

The ACLM function is based on two basic conditions: 1) accurately logging all timestamps of maintenance operations that should be governed by ACLM and values of each involved attribute of the object data record just before each operation of change data; 2) being able to correlate together the values of all member attributes in the data tuple of snapshot. As illustrated in section III, the design and application of audit and change logs themselves have directly satisfy the second condition, and such a condition would be hold met ever since it was satisfied since both the audit and change logs permit only Insert-SQL maintenance manipulations; while the first condition is met by enabling DBMS trigger mechanism [5] that surely captures any event of maintenance operation.

Under ACLM we can flexibly program SQL scripts to recall any historical snapshot without difficulty. Compared with those powerful but extravagant tDBMS products, our ACLM-based solution is economical and exercisable (of DIY type) for MIS. This is because a) implementing ACLM is plain: three specific relational tables for logs and PK specification, a short script additive to usual RDBMS stored procedures of business logic, and a piece of SQL script (alone or embedded in existent SQL scripts) added in triggers of each data table audited, and b) the merits from ACLM: being compatible with changes to meta-data, and programmers can code in their usual way, e.g., implementing data maintenance via RDBMS stored procedures.

In fact, our ACLM approach does reflect a reality that actual tDBMS implementation is a workable evolution of RDBMS application rather than an innovation of nowadays RDBMS. The most valuable point thereby is, ACLM can be plainly deployed on current prevailing RDBMSs with less interference in routine MIS program practices, normal MIS users except who conduct audit would feel they are working with a familiar RDBMS for the current version data access as usual. We can conclude that the ALCM approach is a good option to replace current bitemporal database solutions for uni-temporal MIS applications in order to avoid unexpected or unnecessary cost and complexity.

Acknowledgment

The authors thank all members of their team and correlative colleagues for applying and validating ACLM in several e-

government MIS projects with requirements of uni-temporal database applications.

REFERENCES

[1] C.J. Date, H. Darwen, and N. Lorentzos, Temporal Data & the Relational Model, 1st ed. San Francisco, USA: Morgan Kaufmann, 2002.

[2] C. J. Date, "What first normal form really means," pp. 127–128. http://www.dbdebunk.com/page/page/629796.htm

[3] D. Eastlake 3rd and P. Jones, US Secure Hash Algorithm 1 (SHA1), RFC Editor, 2001. http://tools.ietf.org/html/rfc3174

[4] N. Freed and N. Borenstein, Multipurpose Internet Mail Extensions (MIME) - Part One: Format of Internet Message Bodies, RFC 2045, Nov. 1996.

[5] http://en.wikipedia.org/wiki/Database_trigger

[6] C. S. Jensen, "Introduction to temporal database research," Aalborg University, Denmark, Tech. Rep: No.1, 2000, pp. 1-27.

[7] J. Celko, Joe Celko's SQL for Smarties: Advanced SQL Programming, 4th ed. San Francisco, USA: Morgan Kaufmann, 2010. ISBN: 978-0-12-382022-8

[8] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," in Communications of the ACM, vol 21(2), 1978, pp: 120–126. doi:10.1145/359340.359342

[9] R.T. Snodgrass, Developing Time-Oriented Database Applications in SQL. San Francisco, USA: Morgan Kaufmann, 1999.

[10] H. Guo, Y. Tang, X. Yang, X. Ye, "Improvement and extension to ATSQL2," in Temporal Information Processing Technology and Its Application, Y. Tang et al, Eds. Tsinghua University Press, Beijing and Springer-Verlag Berlin Heidelberg, 2010, pp. 245-259.

[11] M. Kaufmann, P. M. Fischer, N. May, A. Tonder, and D. Kossmann, "TPC-BiH: a benchmark for bi-temporal databases," In TPCTC, 2013.

[12] TimeDB–A Temporal Relational DBMS. http://www.timeconsult.com/software/software.htm