# Recovery from “Bad” User Transactions

David Lomet
Microsoft Research
Redmond, WA 98052
lomet@microsoft.com

Zografoula Vagena[1]
Univ. of California Riverside
Riverside, CA
foula@cs.ucr.edu

Roger Barga
Microsoft Research
Redmond, WA 98052
barga@microsoft.com

## ABSTRACT

User written transaction code is responsible for the “C” in ACID transactions, i.e., taking the database from one consistent state to the next. However, user transactions can be flawed and lead to inconsistent (or invalid) states. Database systems usually correct invalid data using “point in time” recovery, a costly process that installs a backup and rolls it forward. The result is long outages and the “de-commit” of many valid transactions, which must then be re-submitted, frequently manually. We have implemented in our transaction-time database system a technique in which only data tainted by a flawed transaction and transactions dependent upon its updates are “removed”. This process identifies and quarantines tainted data despite the complication of determining transactions dependent on data written by the flawed transaction. A further property of our implementation is that no backup needs to be installed for this because the prior transaction-time states provide an online backup.

## 1. INTRODUCTION

### 1.1 The Problem

One strong reason for using database systems is their promise to guard the integrity of their data via transactions, whose properties have been described as “ACID”. Transactions provide atomicity (A) with its promise of “all or nothing” execution, hence preventing partial transaction executions in which, for example, money intended to be transferred between accounts is only withdrawn or only credited, but not both. They implement isolation (I) so that the effects of one transaction do not interfere with the effects of another, providing the illusion that transactions execute in some serial order. They implement redo recovery and forced logging so that once the database responds accepting responsibility for a transaction’s updates, those updates are durable (D), i.e. guaranteed to be included in the database state. However, these techniques, which are commonly known in the database technical community, do not entirely cope with the problem of consistency (C), which is primarily the responsibility of a user transaction to preserve.

1. Current Address: IBM Almaden Research Center

SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.

There is no system mechanism that prevents a user from entering a bad transaction, e.g. crediting the wrong amount to a bank account, or charging for an item that was not shipped. Existing systems do, however, have ways to “clean up” the database should this kind of problem be detected. We describe here a new recovery method for such “bad” transactions, where “recovery” refers to cleaning up after them. This is related but not identical to the traditional recovery task of restoring the data to consistency after aborts or system crashes.

### 1.2 Existing Solutions

Database systems already deal with bad data. A disk may fail, either catastrophically or by losing some bits. This is media failure, and dealing with it is called media recovery. Commonly, database systems take regular backups, a special form of replica optimized for high speed writing and reading. Media recovery involves backup load (restore) and redo recovery (roll forward) using a media recovery log that records transaction updates since the backup, hence restoring the database to its most recent state. This process can be arduous, and result in a long outage.

Media recovery does not deal with the problem of bad user transactions. The damage done by these erroneous transactions is particularly pernicious because not only is the data they write invalid, but the data written by all transactions that read this data may likewise be invalid. The usual database technique to deal with such invalid data is heroic, both in the cost to use it and the impact that it has on the database, its users, and those responsible for managing the database.

Removing inconsistency induced by bad user transactions is usually based on the media recovery technique described above, in which a backup is restored, and a media recovery log is used to roll the database state forward. But media recovery is halted before the current state of the database is recovered. Instead, recovery continues until just before the “bad” transaction executed. This is “point in time” recovery and it is effective.

However, “point in time” recovery is a heroic measure. It is costly to perform, introducing a long outage while the backup is used to restore the database, and the media recovery log is used to roll the state forward to the time desired. So this can seriously impair database availability.

Further, the impact of “point in time” recovery on the database state is extreme. All transactions that committed later than the erroneous transaction are “de-committed”, i.e. their effects are removed from the resulting database state. In a high performance transaction application, this can result in hundreds, even thousands, of transactions being de-committed. These transactions then need to be re-submitted in some way so as to try to reduce the negative impact. This can be a very labor intensive process since re-executing some of these transactions might result in different results than their original execution.

## 1.3 Our Approach

High human cost of existing practices suggests using machine resources to automate and speed this recovery. Our goal is to automatically identify and isolate the ill-effects of a flawed transaction, thus preserving much more of the current database state while also reducing the service outage time.

We want to attack both the problems mentioned above, long outages and the de-committing of large numbers of transactions. Long outages have previously been attacked by suggesting that a transaction time database can be used to provide an on-line backup that is always installed [4]. Reducing the number of transactions that need to be de-committed is the innovation that we introduce here to deal with "bad" transactions. Oracle Flashback [7] attacks long outages by keeping older versions available for point-in-time recovery. But they do not reduce the number of de-committed transactions in this process, as we do.

### *1.3.1Transaction Time Versions as Backup*

A transaction time database [5,8,9,10] stores and makes accessible prior states of a database. Each transaction execution produces a new database state. So there are a very large number of states. However, transactions typically change only a very small part of the database, and most data can be shared between many states. Each database update introduces a new version of the data it touches. The version is timestamped with the transaction time. One kind of query is an "as of" query that asks for the state of the database as of some selected time. This query is answered by finding the data that has the latest timestamp preceding the "as of" time. It is possible to construct a time based index for this data that will direct the query to exactly the data constituting the database state as of the time requested [3].

Because a transaction-time database contains prior states of the database, it can provide the function of a backup, i.e. a prior state of the database from which the media recovery log can be used to roll forward the state so as to reconstruct the current database state. This requires the prior state to be on a different medium than the current state, which we support. The prior state requires no restore process, since the past states are always on-line. Thus, the large availability outage is immediately avoided. An additional virtue of this approach is that the "backup" can be queried using various temporal queries, e.g., both "as of" querying and time travel querying where one asks for the history of some records in the database over some time period.

### *1.3.2 Identifying and "Removing" Invalid Data*

Like a conventional backup, a transaction time database backup can also be used to provide "point in time" recovery, greatly reducing any outage resulting from bad transactions. This can be accomplished by removing in some way all data versions that were introduced after the corruption occurred. However, this by itself does nothing in terms of reducing the number of de-committed transactions. The rest of the paper describes our innovations that

- reduce the number of de-committed transactions via the logging of transaction reads;

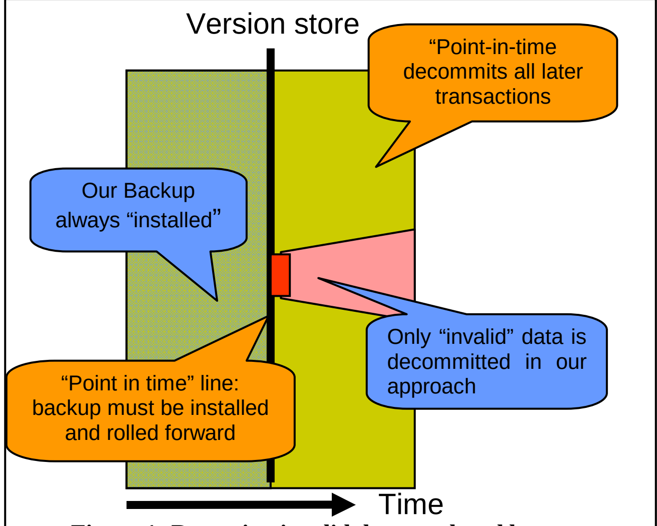
Version store
"Point-in-time decommits all later transactions
Our Backup always "installed"
Only "invalid" data is decommitted in our approach
"Point in time" line: backup must be installed and rolled forward
Time

**Figure 1: Removing invalid data produced by an erroneous user transaction.**

- quarantine data of de-committed transactions, making it invisible to normal querying so that earlier valid versions of the data are seen instead.

The result of these innovations, coupled with the use of a transaction time database as a backup, is that the outage period to provide recovery is almost eliminated, and the number of de-committed transactions is reduced to those reading tainted data, usually an enormous reduction. This is illustrated in Figure 1. This figure shows that the historical states of the transaction time database can be used as an online backup. Then, when bad user data is identified, because we log reads, we can identify exactly the data that is invalid and "de-commit" only this data. This is a very substantial improvement over "point in time" recovery.

## 1.4 Organization of Paper

We describe in section 2 how data is stored in Immortal DB [5], our transaction time database built into SQL Server. Section 3 details how we identify data made invalid by a bad user transaction (the potentially inconsistent data). In section 4, we discuss how we "invalidate" this data. We describe how our system works when invalidated data is encountered in section 5. Finally, section 6 summarizes what we have done and suggests some possible extensions.

## 2. MULTI-VERSION DATA

## 2.1 Record Versions

### *2.1.1 Transaction Time Data*

A transaction time database [10] stores multiple versions of data, each identified by a timestamp. When a transaction executes, it is assigned a timestamp for its execution. The timestamps for transactions are consistent with the serialization order of the transactions. A transaction's timestamp is stored in the versions of the data *d* that the transaction writes so as to identify exactly when its version became the current version. This transaction time *TT* denotes the start time for the data, i.e. $d.TT^{\vdash}$. *W(T)* denotes the exact versions of data that are written by *T*., i.e. those stamped with the timestamp of the transaction. For serializable transactions, that same timestamp identifies the versions of the data that are read by the transaction, i.e. the versions current ``as of" the time denoted by the transaction's timestamp.

A data item is identified by a key. A version of a data item d is valid during the time interval from its start time $d.TT^{\vdash}$ to its end time $d.TT^{\dashv}$, which is the start time for its succeeding version. Hence, it is valid during the semi-open interval $[d.TT^{\vdash}, d.TT^{\dashv})$.

We will need to identify exactly which version of a data item d is the version that is read. We use *R(T)* to denote the exact set of versions of data that are read by transaction *T*. We need to carefully make note of all times used in identifying versions of data that are read, so as to precisely define *R(T)*. For serializable transactions, these are the versions of the data items read that are current at the time *TT* for the transaction.

## 2.2 Intra-Page Organization

### *2.2.1 Version Chains*

Immortal DB record versions are accessed from a slot array, ordered by their identifying key, as is typical within a B-tree style primary index organization. Each version is timestamped with the time of the transaction that wrote it. Thus, the span of a version starts with this time, and ends with the time of the succeeding version that records an update for the record with the given key.

The versions of a record, which share a common key, are stored on linked version lists within a page, the most recent version for each data item at the head of the chain and each version linked to the immediately preceding version in time order. The head of the chain is stored in a slot array, a very common organization for database pages. This organization makes it easy to determine the version of interest within a page. It is simply the first version, starting at the head of the chain for the data item with a given key, and searching down the chain for the first version with a timestamp ≤ the time of interest. A page is illustrated in Figure 2, which shows two data items (records), one with key A, another with key B. These are chained together, with chain heads in the slot array at the end of the page.

### *2.2.2 Dealing with Deletes*

We need to provide an end time for versions immediately preceding deletes. To do this, we add a special new version to the head of the version chain, which is marked as a "delete stub. The delete stub includes a special delete stub flag, the identifying key for the record, and the timestamp for the delete. The stub serves to provide the end time for the immediately preceding version.

## 2.3 Inter-Page Organization

Versions of records can overflow the page on which the current version is stored. When this occurs, we can split the page on either time or identifying key. We then index these rectangular key-time regions using a TSB-tree [3], as shown in Figure 3.

Thus, chains of record versions can span pages, the earliest version on a page pointing to the slot in which to find the record latest version on the page containing immediately preceding versions. When a version's lifetime spans the time boundary used to in the time split, the version appears in both pages, and the version in the later page is linked to its clone in the earlier page. The back pointer that spans pages makes it simple to trace back the versions of a data item across pages. Figure 4 illustrates the back pointer between versions on different pages.

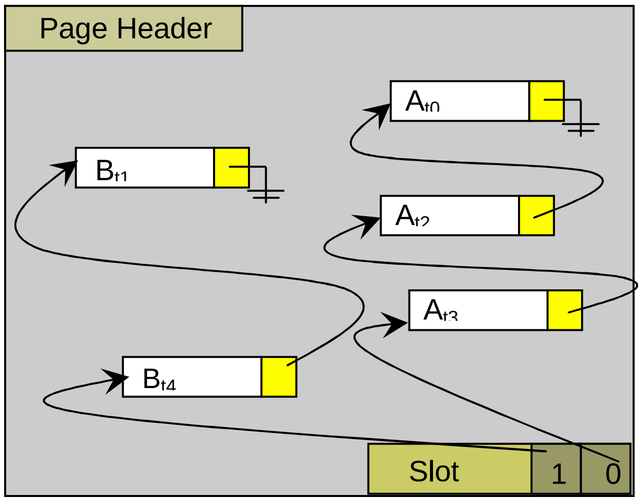

**Figure 2: A page containing versioned data for records with keys A and B.**

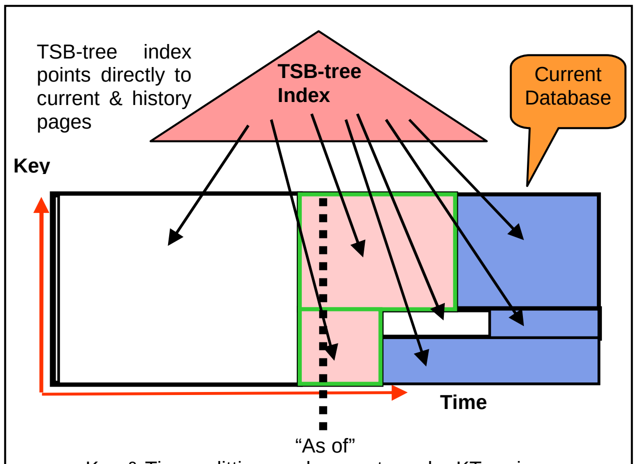
TSB-tree index points directly to current & history pages
TSB-tree Index
Current Database
Key
Time
"As of"
Key & Time splitting produce rectangular KT regions.

**Figure 3: A TSB-tree.**

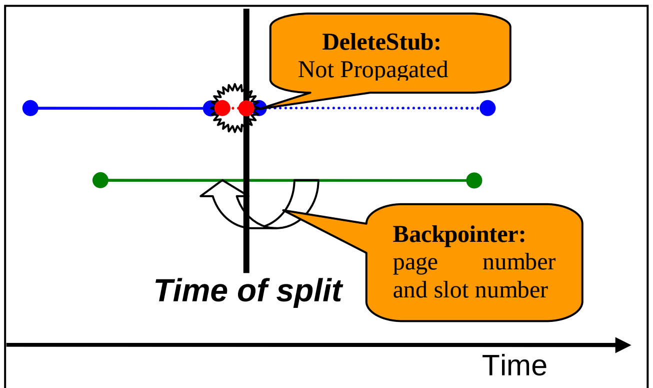
DeleteStub: Not Propagated
Backpointer: page number and slot number
Time of split
Time

**Figure 4: Handling versions between time-split pages.**

When a delete stub might cross the time split boundary, it is not cloned but only appears in the earlier page. The non-propagation of a delete stub prevents the database from filling up with deleted records. The delete stub's purpose is to provide an end time for the preceding deleted version, and it has already accomplished that. We will return to this in section 5, because not cloning delete stubs complicates dealing with invalidated data. Non-propagation of delete stubs is also illustrated in Figure 4, where the dotted line indicates that the delete stub is NOT present.

## 3. INVALID DATA

### 3.1 What Data is Invalid

We need to compute the closure of all data written by transactions impacted by invalid data. Essentially, this process involves computing *C(DB)*, which denotes the exact versions of data that are invalid (corrupt) in the database. This can be expressed in terms of the initially invalid versions of data and the versions of data written by transactions that have read invalid data.

1. $C(T_c)_0 = W(T_c)$, where $T_c$ is the transaction now known to be corrupt. Identifying $T_c$ is done outside of our technology by a process that will usually be a manual one.
2. $C(T_c)_{i+1} = C(T_c)_i \cup \{W(T_{i+1}) \mid \exists (T_{i+1})\ R(T_{i+1}) \cap C(T_c)_i \neq \varphi\}$ where $T_{i+1}$ has a timestamp $TS(T_{i+1})$ and there are no transactions $T_j$ such that $TS(T_i) < TS(T_j) < TS(T_{i+1})$.
3. $C(T_c) = C(T_c)_j$ such that there is no $T_{j+1}$, with $TS(T_j) < TS(T_{j+1})$ with $R(T_{j+1}) \cap C(T_c)_j \neq \varphi$.

We need to interpret item 2 of the preceding list somewhat carefully in order to deal correctly with phantoms [2], as we will describe later. In particular, the set $R(T_{i+1})$ can include ranges of records read, not simply single record reads.

#### 3.1.1 An Example of Invalidation

We illustrate the computation of invalid data using the example below in Figure 5. In Figure 5, transaction 1 at time T1 is determined to be an invalid transaction. So its write set $\{X_1\}$ is shown as invalid (dark box). Our analysis shows that transaction 3 at T3 reads $X_1$ and writes a new value $X_3$. This is also marked as invalid (striped lines). Further, transaction 5 at T5 reads $X_3$, so we show its write of $Z_5$ as invalid also (striped lines). Finally, transaction 6 at T6 reads $Z_5$ and writes $W_6$, which we also show as invalid (striped lines).

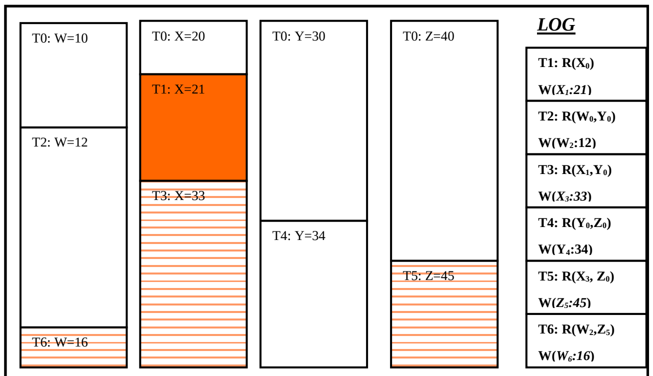

**Figure 5: Transaction time database with invalid data.**

#### 3.1.2 Conventional Database

Note that this same computation can be used to identify potentially invalid data in a conventional database by simply removing the version identification. This essentially collapses all invalid versions of a data item into the single (invalid) instance of the data that is present in the current database. So long as the only transactions being considered are those that execute after the $T_c$, this computation would be effective as well. It could be used to avoid de-committing transactions that are not impacted by the initial corruption.

### 3.2 Logging to Track Invalid Data

#### 3.2.1 Tracking Reads

The technology enabling us to avoid de-committing transactions is based on logging the identity of data that each transaction reads. This permits us to identify the transactions whose execution depended on the data now known to be invalid. Recovery logging in Immortal DB logs the identity of data items written by each transaction, i.e. *W(T)*, and includes the way in which they were changed. By logging the identity of the data items read by each transaction, i.e. *R(T)*, previously suggested in [1] to handle application corruption via wild stores of data in a main memory database system, we have enough information recorded on the log to perform the computation of $C(T_c)$ in section 3.1.

The "C" in "ACID" transactions stands for "consistency" and is largely the responsibility of the transaction code. When a transaction executes in its entirety, it promises to make a transition from one consistent database state to another. So, should a transaction be erroneous and make the database state invalid, then of necessity, the entire effects of the transaction are invalid. Hence, a transaction either invalidates the database or it does not. And if it does, all its updates to the data items it is writing are invalid.

Since a transaction in its entirety is either invalid or consistent, we can treat the activities of a transaction, both its reads and its writes as a whole. Where this is leading is that we do not need to remember when a read occurred within a transaction. A transaction that reads any invalid data will produce invalid updates, even when the invalid data is read after an update occurs. It is simply impossible to produce a consistent result were we to try and distinguish updates occurring before an invalid read from those occurring after such a read.

Thus, we do not need to remember when transaction reads occur relative to transaction updates. This permits us to consolidate all read information about a transaction into a single log record that we write just before we commit the transaction. Only if such a record is too long for our logging mechanism to deal with easily might it be useful to break the read log record into a number of records. Especially in transaction processing, this should always be unnecessary, and a single read log record should suffice. We thus store the information about what each transaction reads in a volatile data structure. Every such transaction is enhanced with a volatile structure which temporarily keeps track of the records that are read by the transaction under consideration. Access to this structure is synchronized with spinlocks to deal with multi-threading within the kernel.

Not all reads need to be recorded in the read log record. In an ARIES style recovery method [6], log operations document the data changes. As ARIES is used in SQL Server, update operations don't reveal whether the data is read prior to being written. To be conservative, we assume that all updates (including inserts) require reads of the updated data. Hence, only reads of versions not updated by the transaction need logging. Reads for updated data are already logged via update log records.

We only deal with user transactions that have performed at least one write operation, as read-only transactions by default do not make data invalid.

In the rest of this section, we discuss two aspects of how best to do this logging:

- how to identify the data items being read;
- the nature of the log records for reads.

### *3.2.2 Identifying Data Items Read*

Some database applications will do extensive amounts of reading prior to changing some much smaller part of the database state. This can make the size of the read log record very large. Further, the size of the data structure needed to remember this information during transaction execution until the moment of commit when the read log record is written can likewise be very large. It is always awkward to deal with large numbers of resources. Each resource must be identified unambiguously. Maintaining such a data structure is not difficult but it can put pressure on the database system's kernel which adds to the costs involved.

The above problem, i.e., of a very large number of resources, is not unique to our wanting to log transaction reads. It arises as well when transactions need to lock resources so as to provide transaction isolation. And we can deal with the issue in the same way, namely by introducing multi-granularity resources.

We capture the information about a transaction's activity at the interface between the query processor and the underlying database kernel. At that point, in our underlying system, we know about individual record reads and about reads of record ranges. We capture both forms of reads. Capturing key range accesses provides us with the multi-granularity resource information that permits us to avoid recording every record that is read in the range. Furthermore, knowing about ranges is critical if we are to deal correctly with phantoms (see the next subsection).

The solution we implemented in the current prototype is to preserve ranges of records that have been read. In order to identify the ranges that have been read, the system records the requests for range reads that come from the upper parts of the system (i.e. the query processor) and uses that in order to aggregate the information whenever possible. The system can currently record the reads at the granularity of single records, full table scans, and as (closed, right open, left open) ranges. While recording the reads we also make several checks, when those can be performed cheaply so that we avoid recording duplicate information.

### *3.2.3 Phantoms*

Some explanation is required to describe how phantoms arise when tracking data invalidation. Consider an invalid transaction that deletes a record. The deletion is invalid because the transaction has read invalid data. Hence the deletion should not have occurred. Were a transaction to read a range that includes the deleted record, that read should make the transaction invalid. However, if we were only recording record reads, we would not "see this". None of the records in the range that our transaction is reading are invalid. What is invalid is that the transaction should have read the deleted record but did not. This deleted record is a phantom. Fortunately, by recording the range that is read by this transaction, we can detect that the invalid deletion occurred in this range, and so make our "range reading" transaction invalid.

## 3.3 Writing Read Log Records

When a transaction that has performed user level updates is about to commit and before writing its commit record to the log, the system generates its read log record and writes it to the log. This log record contains the information that has been accumulated for the reads of this particular transaction. If the maximum size of the log record is reached during the writing process, multiple read log records may be created and written.

The format of the read log record is illustrated in Figure 6. The read log record header includes all the information of a basic log record structure (i.e. log record length, operation type, LSN of the previous log record, ID of the transaction that created this record etc) and in addition four more fields to denote the number of table scans, left open ranges, right open ranges, closed ranges and single record fields (in that order) that exist in the log record. This ordering permits us to decode the information that exists in the fields of the log record.

We chose to store the information in this particular order (i.e. first table scans, then ranges, the single records) so as to put large granularity ranges first. By checking the large granularity reads (i.e., the large ranges) first when reading this record, we increase the probability that we will discover that the transaction is an invalid one without looking at all the data that has been read. Of course, for valid transactions, which we expect to be the common case, there is no avoiding examining all the entries of the read log record.

The cost of logging reads is typically only a very small fraction of the overall logging costs for installations that need the ability to purge bad user transactions. These are typically enterprise systems such as OLTP. Even simple transactions will usually have several updates, e.g. debit/credit. And these systems do only a modest amount of additional reading of non-updated data. Adding one additional log record to transactions for these scenarios is a pretty most cost.

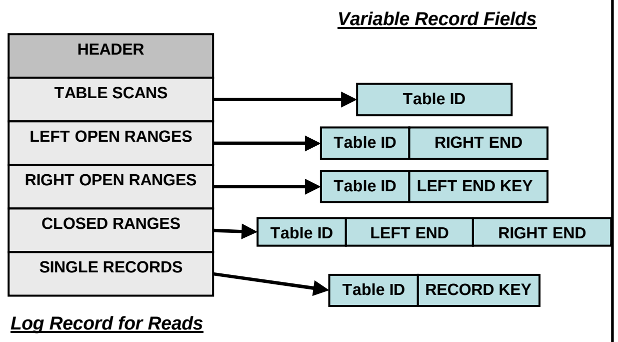

**Figure 6: Log Record for Reads**

## 4. THE INVALIDATION PROCESS

In section 3.1, we defined invalid data as data written by an invalid transaction ($W(T_c)$) or data written by transactions that have read invalid data ($C(T_c)$). This definition provides an abstract algorithm for doing the computation. When we log reads (see section 2.2) we have the information needed to compute the data corrupted by an initial invalid transaction $T_c$. In this section, we describe how to execute our abstract algorithm in the context of a database system.

### 4.1 Identifying Invalid Record Versions

The process of identifying all the invalid record versions consists of two steps.

1. identify the original invalid transaction and its versions;
2. recursively identify as invalid the versions that are written by transactions that have read invalid data.

We treat delete stubs as we treat regular record versions, i.e. we invalidate them exactly when the transaction that produced them either was the invalid transaction or has read invalid data.

We maintain a main memory data structure to record versions identified as invalid. Subsequently, we use the versions identified in this structure to mark the on disk versions in our transaction time database.

#### *4.1.1 Identification of Initial Invalid Transaction*

Step one is to identify the user transaction which produced the initial invalid data. This part of removing invalid data cannot be automated. A user who understands the applications being served by the database needs to identify in some way when an invalid transaction has executed. Recall that it is the user, not the database system, which provides the "C" in "ACID" transactions.

What we deal with here is the way in which the user identifies the invalid transaction to our system. We are aware that there might be several ways in which this might be done, and our invalidation framework can accommodate several methods. Some examples might be for the user to:

1. provide a request ID identifying the user level request that resulted in the invalid transaction being executed. The system might then map this request ID into a transaction timestamp.
2. provide the precise timestamp of the invalid transaction. Such a timestamp uniquely identifies the transaction within our transaction time database.
3. point to a record version that the user identifies as containing invalid data.

There are surely other methods that might also be employed.

It is the third alternative above that we pursue here. We assume in this that our knowledgeable user has executed an "as of" query and has seen what he considers to be invalid data. Our problem boils down to translating this identification of an invalid version into determination of the transaction on the log that wrote the invalid version and its timestamp. We can then identify its updates $W_0(T)$ and initiate the invalidation algorithm to produce $W_C(T)$.

The transaction we want to identify is the one which touched (with an insert, delete or update operation) the latest record whose timestamp is less than or equal to the "as of" time used in the query. Currently in our system, an "as of" query cannot read the timestamp of the data, so we need to discover it. Then we will use the log and our logged reads and writes to identify the rest of the invalid data.

We scan the log backward, starting at the current end of the recovery log for our transaction time database. During the scan the set of all the committed transactions (called the CS or commit set) is maintained in the form of a hash table, with transaction ID as the hashed key. The following actions are performed during the backward scan:

1. When a commit log record is encountered, the ID and commit timestamp of the transaction which created this log record is saved in the CS. The timestamp is given in the immediately earlier record, where it is recorded as an update to our transaction time table. Moreover, for reasons that become obvious later, we also maintain the LSN of the start record for this transaction in the CS. In Immortal DB, this information is provided in the commit log record.
2. When any update, insert or delete log records is encountered, the CS is probed and the transaction that performed this action is identified (we use the transaction ID which is part of the log record for any update log records for the probe value). This information will reveal the timestamp for this "modify" log record. If this timestamp is smaller or equal to the "as of" time for the record version identified as invalid, and the record has the same table and key values as the invalid record of the "as of" query, then we have identified the record version and its transaction that is the invalid one.

At the end of the first phase we have identified the invalid transaction $T_c$, the transaction that "corrupted" the database. The next step is to identify the set of all records that $T_c$ has updated (which are also invalid). This step is interleaved with the second pass of the invalid record identification task, and is described in the next section.

#### *4.1.2 Transitive closure of Invalid Record Versions*

In the second phase of the identification of all the record versions that are invalid, we perform a forward scan through the log, starting from the smallest LSN of the start record for any transaction that we have encountered with a timestamp that is later than or equal to the timestamp for $T_c$. All updates invalidated both directly and transitively by $T_c$ must have log records with LSNs later than that LSN, and hence occur later in the log. During this step we maintain a volatile structure which holds the set of all corrupted records that we identify during this forward scan. We call this set the IRS (Invalid Record Set). We also maintain the information about committed transactions that we have identified in the backward scan done in our first step, until we can decide whether those transactions are invalid.

We do not maintain in IRS an enumeration of record versions that we determine to be invalid. We assume that once a record version in our transaction time database is invalid, all subsequent versions of the record are invalid. With ARIES style recovery [6], with its physiological log records, any modification of an existing record is assumed to have also read the record. Hence, once a record version is invalid, all subsequent versions, which are produced as a result of having read the invalid version, will also be invalid. Thus, to identify corrupt versions of records, we need merely identify the earliest version of a record that is invalid. All subsequent versions will also be invalid. This observation permits us to keep the space required by the IRS much smaller than it would be otherwise.

As a result, we need only maintain the key that identifies a particular version, the table where this record resides and the timestamp of the earliest associated version that was identified as invalid. We proceed as follows:

1. We initially include in the IRS the records updated by $T_c$ (which we have identified in step one), adding them as we encounter them in our now forward log scan. When we encounter the commit log record of $T_c$, we know that we have found all the record versions modified by that transaction.
2. We now identify the other invalid record versions as follows:
   (a) if a "modify" log record (i.e. denoting insert, update, or delete) is encountered, we check whether there exists within the IRS another version with the same key that belongs to the same table. If so this version is also invalid. Hence the associated transaction and all its records are also invalid, as well as the remaining records that have been modified by that transaction.
   (b) when a read log record is encountered we check whether any of the reads that were performed by the associated transaction contain any of the records in the IRS. If so, then this transaction is invalid, and all its modifications to the database need to be identified as invalid. Because this check can only be done when we are at the end of a transaction, we need to remember until the commit record for each transaction is encountered, each update that a transaction has made.
   (c) at the end of each transaction encountered on the log, we know whether the transaction as a whole is either valid or invalid. If invalid, all the record versions modified by the transaction are entered into the IRS. Otherwise, we discard its temporarily remembered modified records.

Note that we know the validity or invalidity of all records read by any transaction before we encounter the transaction's commit record. Thus, when we do encounter a transaction's commit record, we can with certainty decide on the validity of the transaction.

## 4.2 Quarantining Invalid Record Versions

### *4.2.1 Handling Normal Record Versions*

The last step of the data corruption recovery process is to mark as invalid the data versions in the database itself, which are located on disk. This permits us to quarantine the records from normal data access. To denote a record as invalid, we use one bit in the record header. Setting the bit marks the record version as invalid. During this process, the database is off-line, as it would be when performing ordinary crash recovery.

Recall that the IRS contains the set of invalid records, each identified by its primary key, together with the timestamp of the earliest invalid version. All later versions of the record with the given primary key are likewise invalid.

We need to mark each version of every record in the IRS as invalid back to the timestamp of this earliest invalid version. Thus, we search for each record's current version, essentially performing a current time query for it. Then we traverse the associated record version chain backwards in time (using the previous version pointer of each on-page record in order to identify all its previous versions that reside in the same page and the previous page pointer of the associated page for record versions that reside in different pages). As we traverse this version chain, we mark the versions as invalid until we identify a record with timestamp equal to the earliest timestamp that we have saved in the IRS for this record.

### *4.2.2 Handling Delete Stubs*

In Immortal DB, a record version deletion is denoted with a delete stub. The delete stub records the fact that the record is deleted, and provides an end time d.TT, for the immediate earlier version. Recall that we need the delete stub only in the page containing this prior version. Any later pages will simply not have the record present. That is, we do not propagate delete stubs the way we propagate other versions across time split pages.

Not propagating the invalid delete stubs creates new problems.

1. When we look for a current version of a record in the IRS, and it has been deleted in an older page, the delete stub is not present in the current page. Hence there is no evidence of the record in the current page. How do we reach the versions of this IRS record so as to mark them as invalid?
2. When we do normal queries later, we want the prior, valid versions of the records requested to be part of the answer. But records with invalid delete stubs in earlier pages might not be present at all. So we would not even know that a record that would satisfy a range query that is an extension of the lifetime of an earlier valid record exists.

Here we address what we do during the invalidation process. This will directly solve problem (1) above. And it will set us up to solve problem (2), which is part of how we deal with the database during normal processing after records are marked as invalid (see section 5).

Recall that we do not want to propagate delete stubs for fear that they end up diluting later database states, where the delete stubs might be so large in number that the actual existing records become only a small fraction of the total utilization of later pages. However, we are prepared to mark the pages with a small and bounded amount of extra information.

When we execute our current time query for a record in the IRS that has a delete stub that has not been propagated to the current page and hence has an invisible invalid delete stub, we mark the page with an *IDS* flag (using one bit in the page header) as "containing" an invalid delete stub that was not propagated to it. We then follow the page pointer (our time split pages are chained from current time back, as we previously described) to the immediate preceding page. We continue marking pages in this way until we reach the page containing the invalid delete stub. At that point, we mark the delete stub as invalid, and commence traversing the record version chain as we did for ordinary record versions, marking each as invalid until we reach the version containing the timestamp of the earliest invalid version, as recorded in the IRS.

The result of the prior process is that we have marked each invalid version using one bit in the record header, including for versions that are delete stubs. In addition, each page that would have included delete stubs had they been propagated like normal record versions has been marked by IDS in the page header as "containing" unpropagated delete stubs.

## 4.3 Making the Quarantine Durable

The durability of the data invalidation process is achieved by marking all pages modified by invalidation as dirty pages in the cache. Then, when we are finished with the process, we flush all dirty pages to disk before making the database available for regular processing. This is the normal, non-failure case.

To guard against system crashes during invalidation requires that we make invalidation "recoverable". For this, we define two new log records which denote the beginning and completion of the data invalidation process.

1. The begin invalidation log record is designated with a log operation signifying this. It also contains the ID of the transaction which caused the invalidation. This log record is written to the log when we have established this transaction's ID, and before starting the identification of the invalid record versions. This permits us, upon system crash, to re-initiate the invalidation process as part of recovery, and before the system is made available for subsequent normal access.
2. The end invalidation log record is written to the log after all pages dirtied by the invalidation process have been flushed back to disk. At this point, we know that all invalidation has been done AND has been made stable. Hence, should the system subsequently crash, during recovery we can determine that we do not have to repeat the invalidation process, but rather can make the database available for normal processing.

## 5. DATABASE AFTER QUARANTINE

## 5.1 Normal Database Access

We want the effect of accessing our database after invalid data has been quarantined to be as if the bad user transactions that resulted in invalid database states had never been executed. Consider our example of Figure 5. In that example, transaction T1 was identified as invalid and its update of X produced an inconsistent database state. Subsequently, transactions T3, T5, and T6 read either X or other variables that had been made invalid by earlier transactions. This resulted in the versions of W written by T6, X written by T3 and Z written by T5 to also be invalid.

After our invalidation process, we want the effects of all the above invalid transactions to "disappear" from the database, as if these transactions had never executed. This quarantines them during normal access. Thus we want the database to be as shown in Figure 7. In that figure, the version preceding the invalid one has its lifetime extended up to the current time.

Our invalidation process has not, however, actually removed the invalid data. Nor has it propagated the prior valid versions of W, X, and Z as suggested in Figure 7. Rather, it has merely marked the versions shaded or striped in Figure 5 as invalid. It is how we interpret the data following invalidation that permits us to achieve the logical result suggested by Figure 7.

After data corruption recovery has taken place several data manipulation operations need to be modified in order to return, or operate on the database, as if the latter had no corrupted record versions. We describe how various database operations need to be changed in order to achieve the desired effect.

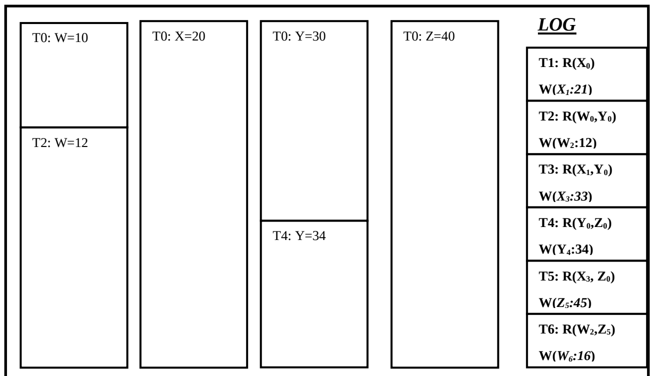
T0: W=10
T2: W=12
T0: X=20
T0: Y=30
T4: Y=34
T0: Z=40
LOG
T1: R($X_0$) W($X_1$:21)
T2: R($W_0$,$Y_0$) W($W_2$:12)
T3: R($X_1$,$Y_0$) W($X_3$:33)
T4: R($Y_0$,$Z_0$) W($Y_4$:34)
T5: R($X_3$, $Z_0$) W($Z_5$:45)
T6: R($W_2$,$Z_5$) W($W_6$:16)

**Figure 7: Versions from Figure 5 showing the logical view, as if invalid versions were not present.**

### *5.1.1 AS_OF Point Query*

This query requests record version in a transaction-time table identified by the record's primary key as of a particular time or the current time. The processing for that record proceeds as if data invalidation recovery has not taken place, until the page that contains the record version that would be returned if no data invalidation recovery has taken place (if one exists) is identified. At this point, what happens depends on what we find on the page.

**Page contains a version of the record:** If a version of the record is found, then we search, starting at the head of the chain for that record's version, and search the chain for the latest valid version earlier than the "as of" time as the answer. This is the empty set when that valid version found is a delete stub. If the later versions are marked as invalid, we need to search this version chain, potentially back through a number of pages until we encounter the first valid version. That then becomes the result that we return.

**Page does not contain a version of the record:** If we do not find a version of the record on the page there are two cases:

- If the page is not marked as *IDS* then we return the empty set as the answer. The requested version does not exist.
- For an *IDS* marked page, we follow our page chain backward to earlier pages that contain earlier versions for the key space of interest, repeating this process. That is, we look again for a version of the record on the older page, etc. Eventually we either find a version of the record (whether valid or not) and then switch to traversing the version chain instead of the page chain. Or we encounter a page not marked as *IDS* at which point we return the empty set as the answer.

### *5.1.2 AS_OF Range Query*

This query requests all records versions "as of" a given time in a specified table and that belong to a given range of keys. To efficiently handle this query when invalid data might be present in the range, we accumulate the valid versions for the range in a batch, rather than looking for each version individually. Thus, for each data pages accessed by normal processing via our temporal TSB-tree index, we identify the part of the key value range for which it provides the answer. We then have a number of cases:

**Page is NOT marked with *IDS*.** We collect valid normal record versions in the range and include them in *ANSWER* set. We follow intra-page versions chains looking for and including valid versions when the later version is marked as invalid. Versions remaining as invalid in the range as of the time requested are put in our *PENDING* set, including invalid delete stubs. The valid delete stubs we encounter are ignored. We follow the page chain back in time, and search each of these pages for members of *PENDING*. When we find the latest (first encountered in the backward chain) valid version for a member of *PENDING*, we remove that record from *PENDING* and include this valid version in *ANSWER*. When *PENDING* is empty, this process stops. Should the latest valid version be a delete stub, we remove the record from *PENDING* but do not include it in *ANSWER*.

**Page is MARKED with *IDS*.** We proceed as if the page is unmarked, placing record versions either in *ANSWER* or *PENDING*. However, it is possible that there are additional record versions "hiding behind" unpropagated invalid delete stubs. Thus when we traverse the page chain for the next older page, we not only search for records in *PENDING*. We also search for invalid delete stubs. When we find an invalid delete stub in the query range, we add its record to *PENDING*. We continue to traverse the page chain until *PENDING* is empty AND the page is not marked as *IDS*.

When collecting record versions in *ANSWER*, we make sure to (a) preserve only the latest valid version of a specific record and (b) preserve the record ordering, so that we do not disrupt operations that rely on that ordering. When processing is complete for the sub-range of the query range in a given page, we return the subrange in *ANSWER*. This permits the range query to be answered incrementally.

### *5.1.3 Update Record*

In Immortal DB, the update of a specific record is realized as follows: first the version of the record to be updated is identified by performing a retrieval operation. We then copy this version to a new version of the record and include it at the head of the version chain. This becomes the basis for the new version, which we then update in place. If the current version is invalid, instead using the current version as the basis for the new version, i.e., as the one that is updated in place, the system identifies the first valid version (if one exists) and copies the content of this valid version to the newly created record that is then updated.

For insertion of a new record, we need to make sure that an existing record with the same key does not already exist. This is answered by doing a retrieve for the record prior to the insert. An existing record may be "hiding" behind an invalid delete, but the retrieval will find it. When no prior record exists, the insert process is the usual one for multi-versioned databases.

A delete also requires a retrieve prior to deletion to make sure there is a record to delete. Such a record might also be "hiding" behind an invalid delete, but retrieval will again find it. When a prior record exists, delete involves creating a new delete stub in the otherwise usual way.

We also need to maintain the version chain. In all cases, if there is an immediate prior version in the current page, valid or not, the back pointer of the new version (or delete stub) points to it. If the immediate prior version is in the immediate prior page, we store its slot number in the back pointer. If there is no prior version (the insert case), the back pointer is set to null. Finally, if the prior version is further back in the page chain, due to unpropagated invalid delete stubs, we mark the back pointer with a special non-null value indicating that.

## 5.2 Database Healing after Invalidation

Pages that are marked as having unpropagated invalid delete stubs (*IDS* pages) are expensive to process. As our transaction time table grows via splitting, we would like to remove the *IDS* mark from new pages when the *IDS* mark is not needed. We call this "healing" the database. Once the *IDS* mark is removed, subsequent update performance and query performance for the current database will return to the level prior to the invalidation. Thus what we examine here is when the *IDS* mark needs to be propagated through page splits and, importantly, when it does not.

We need to understand when we need the *IDS* mark on a page. We need it exactly when there are records missing from a page that should contain them. We do not need the *IDS* mark if we are simply missing some versions on the version chain of a record. The version chain will tell us about that condition.

**Time splits:** In Immortal DB, data pages can be split both by key and by time. With time splits, both pages inherit the same record set. Hence both pages must inherit the *IDS* mark since if records are missing from the record set on the page prior to the split, they will be missing in both history and current page after the split.

**Key splits:** A key split divides a page's key range into two new ranges. While at least one of the resulting pages must have an *IDS* mark if the mark was present before the key split, one page might not. Hence we check whether the resulting pages (i.e. the original split page or the new page or both) still need the *IDS* mark. Thus, after we have performed a normal multi-version key split, for each resulting page we (a) identify its range and (b) we perform a backward search in its page version chain to determine if there is an invalid delete stub which belongs to this range. If we find one we mark the page, otherwise we remove the marking.

**Updates**: When we update or delete a record in an *IDS* marked page, we want to know if the new record version heals (removes) the *IDS* marking. We do a range based on the key range of the page. If there is **only one** invalid delete stub in this range and it has the same key as specified in the update or delete, we remove the *IDS* mark on the page, otherwise we leave it.

Healing operations are very important. Unpropagated invalid delete stubs negatively affect the performance of subsequent operations on an *IDS* marked page. Each time we may need to traverse several pages, to determine the valid contents of the page, without which we could not reliably execute the operations correctly. As a result, we have taken steps to only propagate this marking as necessary to avoid this search in later operations.

Last, in order to make the propagation of page marking recoverable, Immortal DB provides two new log records, one to set the *IDS* flag in a page, and the other to reset it. These permit us to make the setting of this flag durable, avoiding the messy and costly task of recomputing it after a system crash.

## 6. DISCUSSION

## 6.1 Controlling Costs

Not all database installations want or need the level of data integrity provided by our technique. For that reason, we make it optional both declaration of a table as immortal, and the logging of reads. However, it is clear that many enterprises need the kind of functionality that we provide. The existing technique, point-in-time recovery, documents this need, and is used even though it can result in very long availability outages.

For installations that need protection from bad user transactions, doing our invalidation promises to substantially shorten outages. The largest normal execution cost is in supporting the multi-versioning implied by transaction time databases. This cost, which is almost exclusively a storage cost, can be controlled by an administrator deciding how long database states should be maintained. Earlier states can be deleted at low cost, as Immortal DB does now for snapshot isolation versioning [5].

## 6.2 Querying Quarantined Data

We have quarantined the invalid record versions but have not shown how to query them. Querying is desirable as it permits system administrators and perhaps auditors to examine everything that has happened to the data in the database. To provide this functionality, we might provide a special SQL Select statement that indicates that INVALID data should be included. Perhaps one might ask for only INVALID data, or simply, for the purposes of this particular query, to treat invalid data as if it were valid.

Less clear is whether changing invalid data, perhaps making it valid again, is useful. For example, a bank account might have become invalid by having had an erroneous withdrawal posted. That may have resulted in the balance going negative. Were subsequent transactions re-executed against the corrected state, it would be possible to correct this erroneous state. However, posting compensating actions, which is the technique used currently may be a more effective may to deal with the problem. For example, a negative balance may have triggered a bank penalty, which now must be "undone" as well, presumably by posting an appropriate corresponding credit.

## 6.3 Other Invalidation Implementations

Dealing with unpropagated invalid delete stubs is the largest complication introduced by our invalidation implementation. We know of two different ways we might have proceeded, and discuss them here. Building a system is about making choices, frequently without complete knowledge. We made the choices described above, but would like to share some insights into alternatives that might have been used instead.

***IDS* Count Field:** We use a one bit flag to mark pages as having unpropagated invalid delete stubs. This restriction results from SQL Server having a small number of remaining flags in a page header. Without that limitation, we would have used a count of the number of unpropagated invalid delete stubs. Using a count would simplify the code for performing IDS propagation during record updates. Instead of a range search for all records in the key range on the page, we could have searched exactly for the key of the modified record, and reduced the IDS count as appropriate. A count of zero then indicates that the database page is "healed".

**Instant Current State Healing**: Without changing data on history pages, we cannot entirely avoid *IDS* pages. But there is a compromise position, i.e., instantly heal the current state. This avoids the complexity of dealing with invalid current states. This involves searching once for the valid record versions of a current page, and including them on the page during the invalidation process. Then only historical versions can be invalid. To do this, the valid version for each invalid record is copied to the head of its version chain in the current page. Where there is no record in the current page because of an unpropagated invalid delete stub, the missing valid version is inserted as a new record.

Including valid versions into a current page may cause a page to split, and that increases cost and complexity. Further, one still has to deal with the effects of unpropagated invalid delete stubs for historical queries. But added complexity for data modification operations, i.e. insert, delete and update, which only impact the current database state, are completely avoided. Similarly, range queries against the current database state never see invalid data.

## 6.4 Summary

We have dealt with the problem of bad user transactions that result in invalid data. Our method identifies the initial invalid data and all subsequent data that depends on it. Only transactions writing invalid data need to have their effects "de-committed". We identify this closure of invalid data, via logging data reads. Our method then removes only the effects of invalid transactions. Working with a transaction time database means that it is unnecessary to restore a backup as the historical state needed is already online. The bottom line is that our technique de-commits far fewer transactions in order to remove invalid data from the database, and the process for dealing with invalid data results in a much shorter outage than is currently the case for "point in time" recovery, the current method of choice.

## References

[1] Philip Bohannon, Rajeev Rastogi, S. Seshadri, Abraham Silberschatz, and S. Sudarshan: Using Codewords to Protect Database Data from a Class of Software Errors. ICDE 1999: 276-285

[2] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman: Concurrency Control and Recovery in Database Systems. Addison-Wesley 1987, ISBN 0-201-10715-5

[3] David Lomet and Betty Salzberg: Access methods for Multiversion Data. SIGMOD Conference, Portland, OR (May 1989) 315-324

[4] David Lomet and Betty Salzberg: Exploiting a History Database for Backup. VLDB 1993: 380-390

[5] David Lomet, Roger Barga, Mohamed Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu: Transaction Time Support Inside a Database Engine. ICDE 2006: (to appear).

[6] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst. 17(1): 94-162 (1992)

[7] Oracle: Oracle Database 10g Release 2 High Availability. http://www.oracle.com/technology/deploy/availability/pdf/TWP_HA_10gR2_HA_Overview.pdf, May 2005.

[8] Michael Stonebraker: The Design of the POSTGRES Storage System. *VLDB*, 289--300, 1987.

[9] Michael Stonebraker, Lawrence A. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE TKDE* 2(1):125--142, 1990.

[10] Abdullah Uz Tansel, James Clifford, Shashi K. Gadia, Sushil Jajodia, Arie Segev, Richard T. Snodgrass: Temporal Databases: Theory, Design, and Implementation Benjamin/Cummings 1993