Searching in Time

Christian Plattner
plattner@inf.ethz.ch
CS Department
ETH Zurich
Switzerland

Andreas Wapf
awapf@student.ethz.ch
CS Department
ETH Zurich
Switzerland

Gustavo Alonso
alonso@inf.ethz.ch
CS Department
ETH Zurich
Switzerland

ABSTRACT

This demonstration shows how to use external databases to provide an efficient implementation of a timetravel service. The timetravel semantics are defined using snapshot isolation. The system presented not only allows to retrieve older snapshots but also to identify snapshots of interest.

1. INTRODUCTION

Snapshot isolation based databases must keep different versions of each tuple, since every transaction has its own view (snapshot) of the database. By keeping the older versions even though they are not referenced by any running transactions, one effectively creates a history. There are already commercial implementations as well as research projects which allow clients to explore older snapshots of a database [1, 2]. Typically, these approaches allow the user to execute a given read-only transaction at a point in time in the past which is selectable by the user. This feature is commonly referred to as timetravel.

Timetravel creates a wealth of possibilities for new applications. Nevertheless the user is limited in his search as he can only 'peek' into the (possibly huge) history. The database offers no support for the user to find out which states of the database are the really interesting ones (and hence should be more closely investigated). For instance, if one wants to track the temperature of a sensor in the past, then it may be that only those snapshots are of interest where the temperature of the sensor actually changed. Alternatively one could think of a moving objects database used to track the position of a car for crime investigation. There, it is crucial that one extracts all the states where the car changed its position, one cannot rely on 'probing' at random the history of old snapshots. It is also not reasonable to rely only on a slider for browsing the entire history.

In addition to suffering from very limited query support, existing solutions often require special handling from the administrator (e.g., to be able to explore older states, tables have to be created with special attributes, rollback segments have to be carefully sized, etc.). This is also a severe limitation since it is not clear how to introduce timetravel in an already existing database.

In this demo we show our novel timetravel system. It allows clients to explore older snapshots through a powerful query interface. It also introduces a tool to search for interesting snapshots. Our system does not require changes in the database nor does it affect performance for non-timetravel transactions. All timetravel functionality is executed on a set of database replicas that are kept properly synchronized with a main database.

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

2. TIMETRAVEL ARCHITECTURE

Our architecture is based on the replication system used in the Ganymed project [3, 5]. The Ganymed system offers performance and functionality scale-out for snapshot isolation databases. A master database is used for consistency and is extended by a set of satellite database replicas. Updates are always executed on the master database, read-only transactions and extended query functionality not available on the master (e.g., skyline, text search) are offloaded to the satellite databases. Communication between the different parts is being performed through a thin adapter software which runs on each database machine. The system constantly keeps all replicas in-sync with the master. It extracts the *writeset* of each update transaction on the master, gathered writesets are then applied on the satellites in master-commit order. Clients access the system always through the adapter on the master machine and are guaranteed to see a consistent state. Independently of the underlying databases, the master adapter implements the standard PostgreSQL [4] interface on the client side. Clients therefore need no special drivers to access the system.

2.1 Adding Timetravel

As a general prerequisite for timetravel one needs to completely disable any form of garbage collection, otherwise the used databases will destroy older versions of tuples and cleanup index struc-

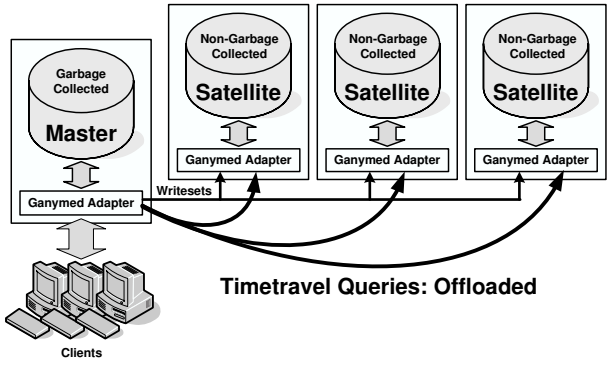

Figure 1: Timetravel Version of Ganymed.

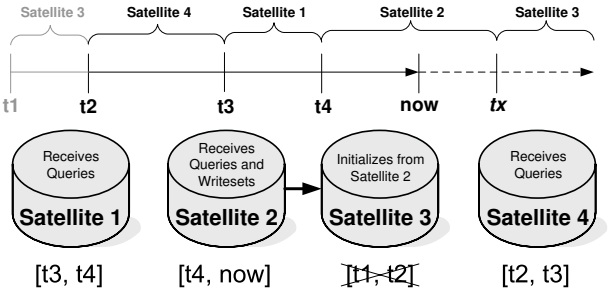

Figure 2: Time-Split Replication.

tures. Disabling garbage collection has also disadvantages: space requirements steadily increase and the performance decreases.

To build a timetravel system that offers unaffected performance for non-timetravel transactions we use a variant of the Ganymed system (see figure 1): the master database is a standard PostgreSQL 8.1 database which gets regularly garbage collected for optimal performance. All non-timetravel transactions are executed on the master. The satellites, used to handle timetravel transactions, consist of a set of non-garbage collecting PostgreSQL databases. The software of the PostgreSQL satellite databases had to be modified, since timetravel is not a supported feature.

Transactions that use timetravel are automatically offloaded to the satellite databases, since historical data is only located there. Also, if needed, any kind of special indexes can be applied to the satellites without having an impact on the master database.

2.2 Scaling to long Histories

To tackle the space problem for growing historical data, we use time-split replication (see figure 2). Instead of using full replication and sending every captured writeset to all satellites, we aggregate batches of historical data in a round-robin fashion, which leads to a time partition of historical data. Every satellite stores historical data for a certain time interval. At any given time, only one satellite receives writesets. Once a certain user specified limit (e.g., number of applied writesets or space limit) has been achieved, no more writesets are applied and further writesets will be applied on another satellite. To move on to another satellite, the underlying Ganymed system is used to consistently copy the currently receiving satellite to the next one. Once copying is done and the two satellites are in-sync, the stream of writesets can be redirected to the next satellite. When reusing satellites in a round-robin fashion, the oldest available historical data will be automatically removed.

The possibility of using varying numbers of satellite databases offers flexibility in terms of scalability and resources that one wants to assign to timetravel transactions.

3. TIMETRAVEL FUNCTIONALITY

PostgreSQL implements snapshot isolation and versioning by keeping different versions of tuples in the same tablespace. During the execution of a query (e.g., during a sequential scan or an index scan) the executor then needs to dynamically decide which version of a tuple is visible in the context of the current transaction. Each version of a tuple has several hidden attributes. The two most important ones are $xmin$ (the transaction id (xid) of the transaction that generated the tuple version) and $xmax$ (the xid of the transaction that either generated a new version of this tuple or deleted the tuple). Different versions of the same tuple use hidden pointers to form a linked-list (from the oldest to the newest version). If a transaction deletes a tuple, then the last version in the chain is marked as deleted.

Identification of the set of visible tuples for the current transaction, the so called $snapshot$, is performed as follows: at the beginning of a transaction a unique (monotonically increasing) xid is assigned to it. On the executing of the first statement in a serializable transaction (or on the start of each query in read-committed mode) the database takes a snapshot for the transaction. The snapshot represents all running transaction at the time it is taken and consists of three parts: xid_{min} is the the lowest xid amongst all transactions still in progress (transactions having $xid < xid_{min}$ are considered to be either committed or aborted), xid_{max} is the next xid that will be assigned (tuples versions generated by transactions having $xid \geq xid_{max}$ will never be visible to the current transaction), the third part is an array of xids in the range $xid_{min} \leq xid < xid_{max}$ (a list of still running transactions that has to be consulted when encountering tuple versions generated by transactions having $xid_{min} \leq xid < xid_{max}$).

In our distributed environment, the xids on the master database and the satellites are not in-sync. We therefore use a separate translation table in the Ganymed adapter on each satellite. Each translation table is automatically built when applying writesets from the master database. Incoming writesets are tagged with the xid of the corresponding update transaction on the master and the timestamp of the commit operation. Each writeset is then applied inside a fresh transaction on the receiving satellite. For each applied writeset the local adapter then adds an entry to the translation table as follows:

$$\{timestamp, xid_{master}, xid_{local}\}$$

On the master adapter, we keep a translation table which, instead of the xid_{local} entry, has an entry specifying the satellite that received the writeset.

3.1 Modifying PostgreSQL

To be able to add timetravel functionality to the PostgreSQL satellites we needed to make two modifications in the PostgreSQL source code: first, we inserted a special function so that a running transaction can modify its own snapshot data (and can therefore decide which versions of tuples it wants to see). Second, we modified the index scan and sequential scan code in the executor module. This change allows, together with a helper function, to identify the xids of transactions that generated (or deleted) tuple versions which match certain qualification criteria.

3.2 Exploring a historical Snapshot

On the satellites, every local xid corresponds to a committed update transaction on the master database. To perform timetravel and explore the database version generated by an update transaction, one has to override the current snapshot by setting xid_{min} and xid_{max} in the current snapshot data to xid_{local}. xid_{local} can be determined either by translating the timestamp or xid_{master} of the master transaction.

From the client's perspective, the determination of the responsible satellite, the involved translations and query-routing is invisible, it is all handled by the Ganymed adapters. Clients simply use a special query interface where they can specify a snapshot based on a master xid or timestamp:

```
START TRANSACTION TIMETRAVEL;
LOAD_SNAP_XID(114786);
SELECT * FROM sensor_data;
(...)
LOAD_SNAP_TIMESTAMP('2005-11-20 15:13:06');
SELECT * FROM sensor_data;
(...)
```

Upon the retrieval of such a transaction the master adapter intercepts all statements, determines continuously the responsible satellite and forwards the transaction's statements to the satellite's adapter. The target adapter in turn then uses the underlying, modified PostgreSQL database to answer queries.

3.3 Finding interesting Snapshots

To offer the possibility of identifying interesting snapshots, the system uses our second change to the PostgreSQL source code, the modified executor module. The task of the executor is to execute a plan tree prepared by the database's planner module. The plan tree consists of nodes which form a pipeline. Each time a node is called, it returns a tuple. Starting from the root node, complex upper nodes (e.g., join nodes) call the lower nodes. Nodes at the bottom level perform either sequential scans or index scans.

With our changes the executor can be switched into a special *find_snap* mode. When executing a query while being in *find_snap* mode, the database parses, rewrites and plans the query as normal. The difference lies in the behavior of the bottom nodes in the plan tree as well as the returned result from the root node. By default, a sequential scan node opens a relation, iterates over all tuples and returns those tuples that are visible with respect to the current snapshot data as well as with respect to all given field qualifications. In *find_snap* mode it logs the $xmin$ (and $xmax$ in case of deletions) attribute of all retrieved tuples, regardless of visibility, while all given field qualifications must match. A similar change has been introduced for B-tree index scan nodes.

At the end of a query that is executed in *find_snap* mode the result from the plan root node result will be discarded. Instead, a set containing all logged xids will be returned. These xids can then be used to perform further timetravel queries.

For database clients, the system offers the *FIND_SNAP* query functionality where clients can specify a query as well as a start and end time. The start and end times are used for two purposes: first, they serve as a filter for the results. Second, the Ganymed adapter on the master database uses them to locate the responsible satellite database where the query must be executed in *find_snap* mode. If the given interval spans several satellites, the query is forwarded to multiple satellites and results are collected and merged at the master adapter.

```
START TRANSACTION TIMETRAVEL;
FIND_SNAP 'SELECT * from sensor_data WHERE
    value > 35 and type=3'
  START '2005-11-20 15:13:06'
  END '2005-11-21 15:13:06';

master_xids
-----------
   113378
   113483
   114786
   114789
(4 rows)
```

The above shown FIND_SNAP query determines the xids of master update transactions which either created or deleted (in the given time interval) a tuple version in the sensor_data relation having the given properties.

4. THE DEMO APPLICATION

To demonstrate the features of our timetravel architecture we have implemented a demo program which simulates and visualizes a set of sensors (measuring temperature, presence of persons and status (open/closed) of doors and windows) in a small office. Reported sensor values are stored in a backend database.

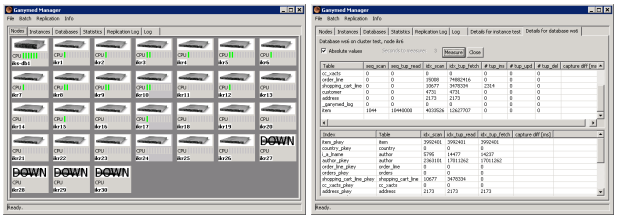

Figure 3: Impressions from the Ganymed Management GUI.

Investigation of the historical data is performed using a set of controls. There is a slider which can be used to scroll in the time-axis. By scrolling with the slider, one can observe the stored sensor data at a given time in the past (each move of the slider leads to a re-evaluation of a timetravel based query which retrieves the states of all sensors in the selected snapshot).

Also, there is a text field where one can enter a manual query. By evaluating this query with the underlying FIND_SNAP capability of the system, the possible positions where the slider can 'snap-in' can be massively reduced (these positions are marked with green lines below the slider). For instance, it is possible to show only those snapshots where a certain temperature sensor had a temperature higher than a given threshold and all doors were closed. Additionally, one can zoom by specifying start and end times for the range shown by the slider.

5. THE DEMONSTRATION

For the demo we will use a transportable setup consisting of four laptops. On the first laptop we will run the Ganymed management GUI (see figure 3) and the demo application. On the three other laptops we will run 1 PostgreSQL master database and 2 satellite databases, all machines also run the Ganymed adapter layer.

First, we will demonstrate how the Ganymed management GUI can be used to set up a timetravel environment by dynamically attaching the two satellite databases to the master database. Then, we will start the demo application and let it feed huge amounts of sensor data into the database system. During the whole time, one can observe the state (i.e., time-split replication) of the system through the Ganymed management console.

In the second part of the demonstration we will focus on the timetravel features. For this we will use a prepared environment which includes historical sensor data collected over a longer period. Inside the historical data, with the help of the demo application, we will identify interesting subsets of all possible snapshots by entering manual FIND_SNAP queries. We then will show how the identified snapshots can be explored with the slider interface.

6. REFERENCES

[1] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: Transaction Time Support for SQL Server. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 939–941, 2005.

[2] Oracle Corporation. Oracle Flashback Technology, http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm.

[3] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Middleware 2004, 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-22, Proceedings*, 2004.

[4] PostgreSQL Global Development Group. http://www.postgresql.org.

[5] The Ganymed Project. http://www.ganymed.ethz.ch.