Development and Performance Analysis of a Temporal Persistent Object Store POST/C++

Takayuki Suzuki
Doctoral Degree Program
in Engineering
University of Tsukuba
Tennohdai, Tsukuba, Ibaraki 305, Japan
suzu@dblab.is.tsukuba.ac.jp

Hiroyuki Kitagawa
Institute of Information Sciences
and Electronics
University of Tsukuba
Tennohdai, Tsukuba, Ibaraki 305, Japan
kitagawa@is.tsukuba.ac.jp

Abstract

ODBMSs are used in advanced database applications to manage complex objects and objects that have associated inherent procedures. Some of the applications also require temporal and historical information management in the context of such object management. In this paper, we first discuss a temporal persistent object model that treats past states of persistent objects as objects. This model supports uniform manipulation of current and past states of objects. Then, we describe design and implementation of a temporal persistent object management system, named POST/C++. POST/C++ persistently stores C++-based objects and manages their histories of updates by user transactions based on the temporal persistent object model. Implementation of the first version of POST/C++ has been completed on Sun OS 4.1.2. Finally, we show performance analysis of POST/C++ using the extended OO1 benchmark, and discuss basic performance issues on our design and implementation.

Keywords Temporal Database, Temporal Object Database, History Management, Database Benchmark

1 Introduction

Temporal data management is required in many advanced database applications. There have been many efforts to model temporal data, to design languages to manipulate such data, and to develop efficient storage structures [9,12,22,23]. Snodgrass and Ahn gave a useful taxonomy for researches on temporal data management [21]. According to the taxonomy, the time considered in this paper is the "transaction time," in other words, history of database states changed by update transactions. Management of database update history is tightly related to the transaction management of the DBMS. Therefore, support by the DBMS is indispensable to implement the history management. Various researches on the transaction time support by DBMSs have been done up to the present time [12,22,23]. However, most of them assume that files consist of traditional simple record structures as in the relational model [14, 15, 23]. It is well-known that DBMSs have to manage objects with complex structures and associated inherent procedures to meet requirements in advanced database applications. Therefore, it is often required for the DBMS to handle temporal information in the context of management of such objects.

In this paper, we present a temporal persistent object store named POST/C++. In common object database management systems (ODBMSs), updates to objects are done in place to overwrite the previous states. Thus, the previous object states are lost. To process temporal queries, however, the past object states must be managed by ODBMSs.

The temporal persistent object store POST/C++ is designed to implement a temporal persistent object model. The temporal persistent object model treats the past object states as objects, and supports uniform manipulation of both current and past object states. POST/C++ persistently stores C++-based objects and manages their histories of updates by user transactions. Past states of objects as well as their current states can be queried in application programs written in C++.

The first important contribution of this work is to demonstrate a concrete design and implementation scheme of a runnable temporal object database management system. Although a number of temporal persistent object models have been proposed in the research community [5, 7, 17, 18], little work has been reported on concrete design and implementation of underlying systems. Related works also include researches on version management in ODBMSs [1, 8, 10]. In most ODBMSs, however, versions are created by specific operators designated for version derivation rather than by ordinal updates to objects, and application depen-

Proceedings of the 7th Australasian Database Conference, Melbourne, Australia, January 29–January 30 1996.

dent semantics is associated with version derivation as exemplified by branches and merges of versions. Moreover, with the exception of some works [1, 11], most of the version object models stress management of version derivation relationships and are very weak in the temporal aspect of version management. Temporal persistent object management implemented in POST/C++ provides basic functionality for history management of objects, and various application oriented features can be developed on top of it.

The second important contribution of this work is to show performance study of the real running temporal object database management system. Ahn and Snodgrass studied performance issue of temporal databases [2, 3]. They are, however, based on the temporal relational model. We have devised an extended OO1 benchmark [4] and applied it to the first version of POST/C++. The benchmark results show some interesting features of POST/C++. One of them is that manipulation of current object states in POST/C++ is little affected by management of object histories.

The remaining part of this paper is organized as follows. In Section 2, we present a temporal persistent object model on which POST/C++ is based, and explain operations to handle temporal persistent objects. In Section 3, we explain architecture of POST/C++ and functions of key component modules. Furthermore, we discuss the major implementation issues. In Section 4, we give a detailed explanation of the extended OO1 benchmark to analyze basic performance of POST/C++, and discuss the experimentation results. Finally, in Section 5, we conclude this paper.

2 Temporal Persistent Object Model

This section describes the temporal persistent object model of POST/C++.

2.1 Modeling of Time

We assume that time is a linear and discrete collection of time units as in most temporal data models [15, 20, 23]. We term the smallest time unit a *chronon*, and the continual sequence of chronons an *interval*. As we can number the chronons, the time domain is isomorphic to the domain of natural numbers. An interval is represented by two chronons indicating the beginning and end of the interval. The special chronon "*now*" is used to designate the present time. When there is no possibility of confusion, we sometimes use the term "time" as a synonym of "chronon."

2.2 Persistent Objects

POST/C++ can manipulate two types of objects, namely *transient objects* and *persistent objects*. A transient object's lifetime is limited to the extent of the program in which it was created. Persistent objects, however, are written to the secondary storage so that those objects may be accessed during the subsequent program runs. A database managed by POST/C++ is a collection of persistent objects. Each object is identified by a unique identifier, called an *object identifier* (*oid*).

In POST/C++, an object's persistence is orthogonal to its type. A persistent object is simply one that is allocated in the persistent heap, as opposed to the conventional (transient) heap or in the activation stack.

The following gives the definitions of classes `Employee` and `Department` used in examples in the remaining part of this paper.

```
class Employee
{
  private:
    char*       name[SIZE]; // Name
    Department* dept;  // Department
        ...
  public:
    Employee(char*,Department*, ...);
    ~Employee();
    char* name();
    Department*  department();
        ...
};

class Department
{
  private
    char*   name[SIZE]; // Name
    Manager* manager; // Manager
        ...
  public:
    Department(char*,Manager*, ...);
    ~Department();
    char* name();
    Employee*  manager();
        ...
};
```

2.3 Temporal Persistent Object

In the life cycle of an object, the object is first created and then modified several times by update transactions. It may finally be deleted. A persistent object is created by the operation `pnew`, which takes as its arguments the class of the object and, possibly, values for initialization. The operation `pnew` returns an address in the persistent heap on which the object was allocated.

Figure 1 illustrates updates to objects. An update transaction changes object states at the commit time. Updates to conventional persistent objects overwrite the previous object states.

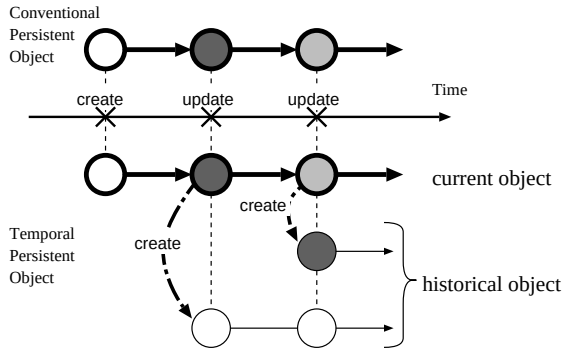

Figure 1: Temporal Persistent Object

Therefore, changes of object states are not maintained. In POST/C++, we introduce the concept of a *temporal persistent object* (shortly, *temporal object*). Update history of a temporal object is maintained with *current objects* and *historical objects*. A current object represents the current state of a temporal object. A historical object represents a past state of a temporal object. Current objects and historical objects are generically referred to as *version objects*. A current object and related historical objects form a set called a *version set*. Only current objects accept update requests. When a temporal object is to be updated, it is "logically" updated. The current object of the designated temporal object enters into the new state reflecting the update. Just before the update, a new historical object is created to preserve the previous object state. Each historical object is associated with the *current interval*, which begins at the commit time of the transaction that made the state represented by the historical object current and ends at the commit time of the transaction that made it historical. The current interval of the current object begins at the commit time of the last update transaction on the object, and the end time is a special value "now."

A temporal object is "logically" deleted by the operation `delete`. Deletion can be applied only to current objects. When deletion is applied to a current object, a historical object is created to preserve the last state just before the deletion. Then, the current object enters into the *tombstone* state. We refer to current objects with the tombstone state as *tombstone objects*. Values of all user-defined attributes of the tombstone object are NULLs. The current interval of the tombstone object begins at the deletion transaction commit time and continues forever. Figure 2 illustrates deletion of a temporal object.

A *lifespan* [6] is the interval which begins at the beginning time of the current interval of the first historical object and ends at the beginning time of the current interval of the tombstone object, if

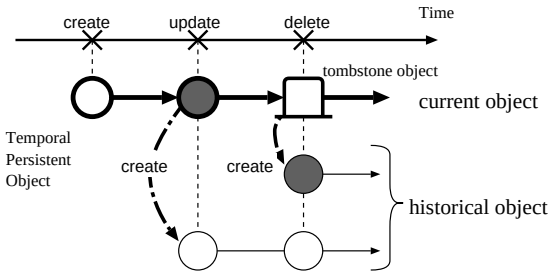

Figure 2: Deletion of a Temporal Object

any. If the tombstone object does not exist, the lifespan continues forever. All the version objects in a version set are associated with the same lifespan. Time contained in the lifespan designates a non-tombstone version object.

2.4 Primitive Operations to Temporal Objects

References to objects in application programs are done with pointers to the persistent heap, namely *persistent pointers*. In the disk format, the persistent pointers are 32-bit file offsets. When persistent objects are loaded into the virtual memory, those file offsets are automatically translated into virtual memory addresses referred to as *swizzled pointers*. We use the persistent pointers as oids of objects.

To facilitate accesses to version objects, the model provides a number of primitive functions. Function `current(`*OID*`)` returns the persistent pointer of the current object in the version set to which the object with OID belongs. Function `get_begin_time(`*OID*`)` and `get_end_time(`*OID*`)` return the beginning time and the end time of the current interval of the designated object, respectively. Function `snapshot(`*OID, time*`)` returns the persistent pointer to the version object which represents the current state at the *time*. It is usually used to access historical objects. History of a temporal object is represented by version objects in the version set. To chronologically navigate the version objects, we introduce the iterator class. An iterator gives a cursor to navigate the version objects in a version set in chronological sequence. Function `set_iterator(`*OID, time*`)` creates an iterator to navigate in the version set to which the object with OID belongs. The cursor position is initially set to the version object whose current interval contains the designated time. Member function `fetch()` returns the persistent pointer to the object at the cursor position. Member function `previous()` of the iterator moves the cursor in the reverse chronological direction and returns the persistent pointer to the object at the cursor position. Member function `next()` moves the cursor in the chronological direction.

The following example query retrieves the manager name of the "R&D" department as of "08/31/1990." Current objects can be tagged with *names* to make them persistent root objects. In this example, operator `lookup_object(`*name*`)` is used to get the persistent pointer associated with the *name*.

```
Department* dept;
Employee*   manager;
     ...
dept=MyDatabase->lookup_object("R&D");
manager=dept->snapshot("08/31/1990")
                              ->manager();
cout << manager->name();
```

The following program fragment shows the career of the employee since "08/31/1990" who is the present manager of the "R&D" department.

```
Department* dept;
Employee*   manager;
Iterator<Employee>* mit;
    ...
dept=MyDatabase->lookup_object("R&D");
manager=dept->manager();
mit=set_iterator(manager,"08/31/1990");
while (manager != NULL)
{
       cout << manager->department()
                      ->name() << '\n';
       manager=mit->next();
}
```

3 Design and Implementation of POST/C++

This section describes the design and implementation of POST/C++.

3.1 Architecture Overview

We give overview of the architecture of POST/C++ supporting the temporal persistent object model presented in Section 2. Key design decisions are as follows.

1. Objects in POST/C++ have associated procedures which are coded in C++. Functions coded in C++ cannot be applicable to objects unless they are loaded on the main memory from the secondary storage and translated into C++ language specific formats. Therefore, it is an important design issue how to efficiently attain the translation. For the following reasons, we have decided to follow the one-level storage scheme based the on page-wise memory-mapped architecture, which is one of well-known implementation schemes of ODBMSs [13, 19, 24].
   (a) Mapping between the on-memory page image and the disk page image is very simple and efficient in the scheme.
   (b) We can use usual C++ compilers rather than tailor-made ones. Therefore, we can save development costs.
2. Historical objects are read-only, and higher performance on their accesses is usually not required than current objects. Thus, historical objects are stored on pages in the *history file*, and the current objects are stored in the *current file*. The history file and the current file are implemented by separate physical files.
3. Each page in the current and history files is divided into smaller parts called *chunks*. The chunk size is fixed in a page, but the file may contain pages of several chunk sizes. The object and its header are stored on a page whose chunk has the minimum necessary size.
4. To maintain update histories of objects, we have to detect updates by the unit of object. In the page-wise memory-mapped architecture, it is easy to detect the updated pages with system calls. However, it is difficult to detect objects which are updated within the updated page. We have decided to compare the before image and the after image of the page by the unit of the chunk to identify updated objects.
5. To support some of the primitive operations presented in Subsection 2.4, we have introduced the *history index*. The index maintains the temporal relationship among objects in a version set. Function `snapshot()` and `set_iterator()` are implemented based on the history index.

Figure 3 shows architecture of POST/C++. The interface is a collection of class definitions and functions for object manipulation by C++ application programs and it provides the primitive operations as mentioned in Subsection 2.4. The persistent object manager is the main module of POST/C++ that manipulates objects and manages memory mapping of page images. The heap manager performs allocation of pages in the virtual memory space with system calls. The mapping and swizzling manager performs pointer swizzling and unswizzling [16, 25]. The history index manager maintains the history index to manage the relationship among version objects. Recovery facilities are provided by the recovery manager.

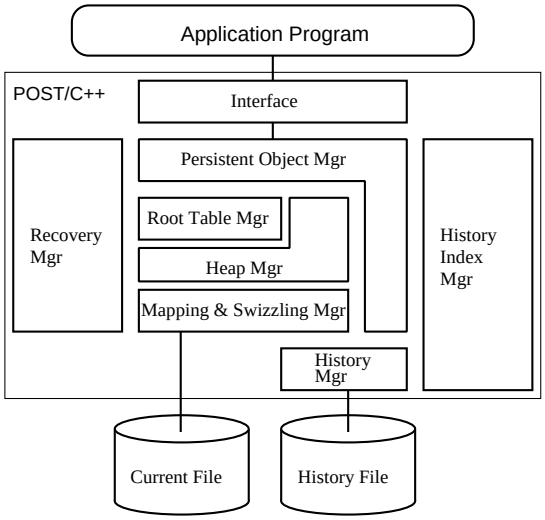

Figure 3: Architecture of POST/C++

3.2 Object Creation and Storage

When a new object is created with `pnew`, an area of the required chunk size is allocated to the object in the persistent heap. When the persistent heap is exhausted, a new page is allocated. As indicated in the preceding subsection, the object size is fixed when `pnew` is invoked and cannot change when the object is updated. The minimum chunk size is 32 bytes, and $2^n * 32$ bytes are allocated to an object. Each object is tagged with the object header, and it includes information required by functions `current()`, `get_begin_time()` and `get_end_time()`. The storage format of an object is shown in Figure 4.

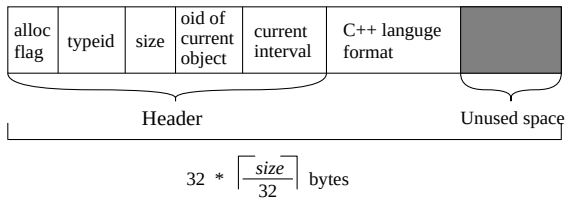

Figure 4: Storage Format of an Object

A current object is stored on a page in the current file which has the minimum required chunk size and one or more free chunks. A historical object is stored on a page in the history file in the same way as current objects. Therefore, historical objects of the same chunk size are stored in the chronological order within a page and chronologically clustered.

When an object is first activate in a transaction, the persistent object manager consults the VM(Virtual Memory) mapping table which manages the pages loaded into virtual memory. If the page containing the object does not reside on memory, it is loaded from the file. Then, it is mapped on the virtual memory by the persistent object manager. References to objects out-going from the page are swizzled by the mapping and swizzling manager.

3.3 History Index

As mentioned in Subsection 3.1, the history index manages the temporal relationship between current objects and historical objects. In the primitive operation `snapshot(`*OID, time*`)`, the index is used to retrieve the oid of the historical object which represents the current state at the designated *time*. To support functions `next()` and `previous()`, the index is used to create iterators to navigate in version sets.

The history index is based on the B+-tree structure. The history index is shown in Figure 5. At the leaf level, the history index stores a record of the format (OID, COID, beginning-time, end-time) for each history object, where OID is the oid of the historical object, COID is the oid of the current object associated with the historical object, and the beginning-time and end-time represent the current interval of the historical object. Note that oids are stored in the disk format in the history index. Those records are indexed by the composite key consisting of COID and beginning-time. In `snapshot`, the history index is searched with COID and time, to retrieve OID of the historical object whose current interval contains the designated time. Leaf nodes of the history index are linked with each other in both directions. This facilitates creation of iterators to navigate in version sets.

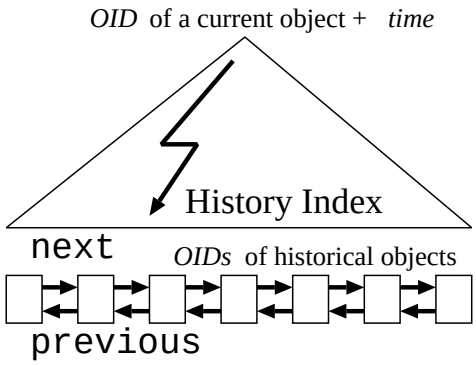

Figure 5: History Index

3.4 Commit Operation

Update to an object in a transaction makes the corresponding on-memory page image dirty. This action is detected by the page fault, if we make the page image write-protected by the system call `mprotect()` when it is first mapped on the virtual memory. Since pages in the history file images are not updatable, update to the memory area

which contains historical objects causes an error. The protection status of the memory is maintained in the VM mapping table. When the update transaction commits, on-memory pointers in all dirty pages are unswizzled, and the dirty page images are compared with the before page images in the current file to identify updated objects. Then, new historical objects are created. The history manager inserts the historical objects into the history file and the history index.

3.5 Implementation

We have decided to implement the first version of POST/C+ using Texas [19] as a base system to save development time and cost. Texas is a persistent object storage system developed by S. Kakkda and others at University of Texas. To implement the POST/C++ object manager, we have done the following several extensions and modifications to the original code of Texas:

- The object header is extended to manage historical objects as well as current objects.
- The history manager and the history index manager have been developed from scratch.
- Commit procedures are largely rewritten to manage histories as explained in Subsection 3.4. After the commit, pages containing current objects reside on the memory until the database is closed. They work as caches of current objects. This strategy is same as in the original design of Texas. However, as for historical objects, we have made a different design decision. Namely, pages containing historical objects are always purged at the commit time. The reason is that we assume repeated accesses to the same historical objects are not often and that caching historical objects may demand large memory space.
- The recovery procedure is modified to treat the history file and the history index.
- The interface to application programs is redesigned to include primitive operations to handle temporal objects. Modification has been also done to `pnew()` and `delete()`, too. The class Iterator<T> is introduced to navigate in version sets.
- Several internal tables are also modified and added.

Implementation of the first version of POST/C++ has been completed on SunOS 4.1.2 with GNU g++ compiler. The concurrency control facility is not included in the first version of POST/C++.

4 Basic Performance Analysis

This section explains the basic performance analysis of POST/C++, using a benchmark which is an extension of the OO1 benchmark [4].

4.1 Extension of OO1 Benchmark

The OO1 benchmark [4] is a standard benchmark to measure the performance of ODBMSs. The database used in the OO1 benchmark is independent of data models, and consists of a single object type, namely the Part type. The following is the definition of the Part type.

```
class Part {
    int id;
    char type[10];
    int x,y;
    int build;
    Part* to[3];
    Part* from[...];
}
```

In this experiment, the Part object size is 128 bytes including the object header. The OO1 benchmark database includes 20,000 Part objects, thus the storage space for the current objects is about 2.7 MB. Each Part object has 3 connections to other randomly selected Parts. The random connections between Parts are selected to produce some locality of reference. Precisely, the 90 percent of the connections are randomly selected among the 1 percent of Parts that are "closest," and the remaining connections are made to randomly selected Parts. Closeness is defined using the Part id associated with each Part object. In this experiment, Part objects are created in the sequential order of the Part id and the Part object size is fixed. Therefore, following the storage scheme as explained in Subsection 3.2, Part objects which have close Part ids are stored in physically close locations.

The original OO1 benchmark runs three types of operations: Lookup, Traversal and Insert, on the above database and measures the execution time. In this study, we are most interested in the effect of history management on the database update operation and performance of object accesses including historical objects. For this purpose, we have made the following four experiments.

Experiment 1: First, we create the initial database which includes 20,000 Part objects by one transaction. Let this creation time be t_0. At the creation of the database, there is one option to decide whether the database performs the history management or not. Using this option, two databases are actually created. One which

performs the history management is called *History*, and the other which does not perform the history management is called *No History*. Then, on both databases, the *Update* transaction is executed 20,000 times and their execution times are measured. One Update transaction does the following operations.

1. Open the database and report the execution time.
2. Select 100 current Part objects randomly, and update their x, y attributes. And report the execution time.
3. Commit and close the database. And report the execution time.

Let the commit time of the i-th transaction be t_i $(1 \leq i \leq 20000)$. In the History database, 2,000,000 historical Parts objects are created, and each Part object goes through 100 updates on average.

Experiment 2: This experiment is based on Lookup of OO1. We use the databases constructed in Experiment 1. The *Lookup* transaction is to generate 1,000 random Part id's and to fetch the corresponding Parts from the database. We consider the effect of caches as in OO1. The cold state represents the initial state and no data may be cached. In the warm state, the effect of caching gradually takes place. In the hot state, accesses to the same objects are repeated and all objects reside on the memory. The Lookup transaction is executed 21 times in series as follows:

1. Clear the operating system cache.
2. Open the database.
3. Perform the first Lookup transaction and report the cold execution time.
4. Repeat the Lookup transaction 10 times and report the warm execution time.
5. Repeat the Lookup transaction 10 times such that the same 1,000 part id's as the last warm Lookup are accessed, and report the hot execution time.
6. Close the database.

This measurement is done on the History database and the No History database. On the History database, two measurements are performed. One is to look up current objects, and the other is to lookup historical objects at t_0.

Experiment 3: This experiment is based on Traversal of OO1. The *Traversal* transaction is to find all Parts connected to a randomly selected Part, or to Parts connected to it, and so on, up to seven hops (total of 3,280 Parts). In addition to this *Forward* traversal, *Reverse* traversal is done in the reverse traversal direction. The Traversal transaction is executed 21 times in series as in Experiment 2. Measurements of accesses to historical objects as well as current objects are performed on the History database.

Experiment 4: OO1 does not have measurement corresponding to this experiment. The *History Traversal* transaction is to make an iterator for a randomly selected Part and to fetch every version object associated with it in the reverse chronological order from the current object to the historical object at t_0.

4.2 Experimentation Results

We used Sun workstation Sun 4/SLC with 8MB memory, SCSI disk controller (3.05 MB/sec transfer rate), and 1GB disk (14.5 ms avg. access time) for this experiment. The system ran on Sun OS 4.1.2. All measurements in the benchmark are shown by the second.

Table 1 summarizes results of Experiment 1. It gives execution times required for open, updates, and commit at t_{5000}, t_{10000}, t_{15000} and t_{20000}. It also gives the total execution times of Update and the overall averages. Note that the execution time of Update is almost constant, although the number of historical objects increases. Comparing measurements on History with the No History results, most of the penalty by the history management concentrates on the commit phase. The overhead is about 50 % at the commit phase, and 25 % for the total execution time of the Update transaction.

Table 1: Benchmark Results of Experiment 1

	open	updates	commit	Total
t_{5000}	0.28	8.28	11.19	19.75
t_{10000}	0.30	8.26	11.33	19.89
t_{15000}	0.29	8.22	11.28	19.79
t_{20000}	0.32	8.30	11.64	20.26
Average	0.30	8.22	11.86	20.38
No History	0.25	8.15	7.90	16.30

Table 2 summarizes results of Experiment 2. It shows the averages of 100 executions of Experiment 2. First, we discuss accesses to current objects. Comparing the History measurements with the No History results, the history management have no effect on accesses to current objects. When the cache state is cold, Lookup takes much time to read many pages from the disk. In the warm cache state, the fetched objects reside in the cache. Therefore,

the execution time decreases as less reads from the disk are required. In the hot cache state, all target objects exist in memory. Thus, the execution time is very short. In accesses to historical objects, accesses to the corresponding current objects have first to be done in POST/C++. The caching has a positive effect on current objects accesses. However, accesses to historical objects take much time because of a large amount of disk accesses to the history file. In the current implementation, pages of the history file are not cached after transaction commits as mentioned Subsection 3.5. Therefore, the history objects must be read from the disk at each iteration in Experiment 2. This is the reason why caching does not affect on the execution time of accesses to historical objects. So, it may be necessary to provide the option of caching historical objects to attain good performance for such object access requirements.

Table 2: Benchmark Results of Experiment 2

	No History (Current Object Access)	History (Current Object Access)	History (Historical Object Access: t_0)
cold	17.96	17.97	54.85
warm	0.62 – 0.01	0.65 – 0.01	36.10 – 35.62
hot	0.01	0.01	35.61

Table 3 summarized results of Experiment 3. It shows the averages of 100 executions of Experiment 3. First, we discuss accesses to current objects. As in Experiment 2, the history management has no effect on accesses to current objects. Comparing results of Experiment 3 with those of Experiment 2, although the number of accessed objects is three times more, the execution time of Traversal is less than Lookup. This is due to the "clustering" of objects in the disk. The locality of references between current objects corresponds to the locality of the storage positions in disk pages. Also, there is caching effect as in Lookup results. Next, we discuss accessing historical objects at t_0. The Traversal time is almost three times more than the Lookup time. The reason is that the locality of references does not coincide with the storage scheme of historical objects and the target historical objects are scattered in the history file.

In the Reverse Traversal, the number of reverse connections differs from object to object. For this reason, execution times show more distribution than in the Forward Traversal. However, the overall trend of the measurements is same as in the Forward Traversal.

Table 4 summarizes results of Experiment 4. It shows the averages of 100 executions of Experiment 4. For reference, Table 4 includes the execution time required to access to one historical object

Table 3: Benchmark Results of Experiment 3

Forward Traversal			
	No History (Current Object Access)	History (Current Object Access)	History (Historical Object Access: t_0)
cold	11.21	11.10	145.71
warm	3.21 – 0.17	3.39 – 0.06	136.12 – 130.73
hot	0.06	0.06	130.50

Reverse Traversal			
	No History (Current Object Access)	History (Current Object Access)	History (Historical Object Access: t_0)
cold	9.81	9.90	132.18
warm	4.00 – 0.22	2.79 – 0.31	122.93 – 139.20
hot	0.07	0.07	139.04

with the snapshot operator. Creation of an iterator takes little time because of the support by the history index. The time required for accessing a historical object by the iterator includes the time for swizzling the history page on which the historical object is stored. In contrast to this, the snapshot operator involves swizzling two pages: one for the page of the current file in which the current object is stored and the other is the page of the history file in which the designated historical object is stored. Therefore, as far as the average access time per a historical object is concerned, the iterator is more efficient than the snapshot operation.

Table 4: Benchmark Results of Experiment 4

Creation of an iterator	0.10
Total time of all history traversal	4.76
Average of access to one historical object	0.05
Average of access by snapshot	0.22

5 Conclusion

In this paper, we have presented design and implementation of a temporal persistent object store POST/C++. Then, we have analyzed basic performance of the first version of POST/C++ by the extended OO1 benchmark. POST/C++ implements the temporal persistent object model presented in Section 2. POST/C++ follows the page-wise memory-mapped architecture. The usual C++ compilers can be used for compiling application programs on POST/C++. By the benchmark, we have demonstrated that the history management has little effect on accesses to current objects, although it gives about the 30 % overhead on object updates.

There are a number of future research issues related to the topic. The cost of accessing historical objects is not so cheap in the first version of POST/C++. We are planning to analyze cost involved in accessing historical objects in detail to locate the bottleneck. Caching historical objects may be one option to get high performance for accessing historical objects. We must also consider the structure of the history file and the hindex file to cope with the monotonic growth of storage sizes to maintain histories. In the first version of POST/C++, the object size is fixed when it is first created. The next version of POST/C++ will be extended to handle variable length objects, and to implement the concurrency control facility.

Acknowledgements

The authors are grateful to many members of the Database Laboratory, University of Tsukuba, for fruitful discussion and their support. This work was partially supported by the Grant-in-Aid for Scientific Research from the Ministry of Education, Science, and Culture of Japan.

References

[1] R. Agrawal, S. Buroff, N. Gehani and D. Shasha. Object Versioning in Ode. In *Proceeding of 7th International Conference on Data Engineering*, pages 446–455, April 1991.

[2] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. *ACM SIGMOD RECORD*, Volume 49, Number 2, pages 96–107, 1986.

[3] I. Ahn and R. Snodgrass. Performance Analysis of Temporal Queries. *Information Sciences*, Volume 49, pages 103–146, 1989.

[4] R. G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, Volume 17, Number 1, pages 1–31, 1992.

[5] T. S. Cheng and S. K. Gadia. An Object-Oriented Model for Temporal Database. In R. T. Snodgrass (editor), *Proceedings of International Workshop on an Infrastructure for Temporal Databases*, June 1993.

[6] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespan. In *Proceedings of 3rd International Conference on Data Engineering*, pages 528–537, February 1987.

[7] U. Dayal and G. T. J. Wuu. A Uniform Approach to Processing Temporal Queries. In *Proceedings of 18th VLDB Conference*, pages 407–418, August 1992.

[8] K. R. Dittrich and R. A. Lorie. Version Support for Engineering Database Systems. *IEEE Transactions on Software Engineering*, Volume 14, Number 4, pages 429–437, 1988.

[9] C. S. Jensen (Editor). A Consensus Glossary of Temporal Database Concepts. *ACM SIGMOD RECORD*, Volume 23, Number 1, pages 52–64, 1994.

[10] R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Database. *ACM Computing Surveys*, Volume 22, Number 4, pages 375–408, 1990.

[11] P. Klahold, G. Schlageter and W. Wilkes. A General Model for Version Management in Databases. In *Proceedings of 12th VLDB Conference*, pages 319–327, August 1986.

[12] N. Kline. An Update of the Temporal Database Bibliography. *ACM SIGMOD RECORD*, Volume 22, Number 4, pages 66–80, 1993.

[13] C. Lamb, G. Landis, J. Orenstein and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, Volume 34, Number 10, pages 50–63, 1991.

[14] D. Lomet and B. Salzberg. Rollback Databases. Technical Report CRL 92/1, Cambridge Research Lab, Digital Equipment Corporation, 1992.

[15] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner and J. Woodfill. Designing DBMS Support for the Temporal Dimension. In *Proceedings of ACM SIGMOD Conference*, pages 115–130, June 1984.

[16] J. Eliot B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering*, Volume 18, Number 8, pages 657–673, 1992.

[17] N. Pissinou and K. Makki. The T-3DIS: An Approach to Temporal Object Database. In *Proceedings of International Conference of Information and Knowledge Management*, pages 185–192, November 1992.

[18] N. Pissinou, K. Makki and Y. Yesha. On Temporal Modeling in the Context of Object Database. *ACM SIGMOD RECORD*, Volume 22, Number 3, pages 8–15, 1993.

[19] V. Singhal, V. Kakkad and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceedings of 5th International Workshop on Persistent Object Systems*, pages 11–33, September 1992.

[20] R. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, Volume 12, Number 2, pages 247–298, 1987.

[21] R. Snodgrass and I. Ahn. A Taxonomy of Time in Database. In *Proceedings of ACM SIGMOD Conference*, pages 236–246, May 1985.

[22] M. D. Soo. Bibliography on Temporal Databases. *ACM SIGMOD RECORD*, Volume 20, Number 1, pages 14–23, 1991.

[23] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass. *Temporal Databases – Theory, Design, and Implementations* –. The Benjamin/Cummings Publishing, 1993.

[24] S. J. White and D. J. DeWitt. QuickStore: A High Performance Mapped Object Store. In *Proceedings of ACM SIGMOD Conference*, pages 395–406, May 1994.

[25] P. R. Wilson and S. V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proceedings of International Workshop on Object Orientation in Operation Systems*, pages 364–377, 1992.