TKDE #106237

Hashing Methods for Temporal Data

George Kollios
Computer Science Department
Boston University
Boston, MA 02115
gkollios@cs.bu.edu

Vassilis J. Tsotras †
Department of Computer Science and Engineering
University of California
Riverside, CA 92521
tsotras@cs.ucr.edu

A B S T R A C T

External dynamic hashing has been used in traditional database systems as a fast method to answer *membership* queries. Given a dynamic set S of objects, a membership query asks whether an object with identity k is in (the most current state of) S. This paper addresses the more general problem of *Temporal Hashing*. In this setting changes to the dynamic set are timestamped and the membership query has a temporal predicate, as in: "find whether object with identity k was in set S at time t". We present an efficient solution for this problem that takes an ephemeral hashing scheme and makes it partially persistent. Our solution, also termed *partially persistent hashing*, uses linear space on the total number of changes in the evolution of set S and has a small $(O(\log_B(n/B)))$ query overhead. An experimental comparison of partially persistent hashing with various straightforward approaches (like external linear hashing, the Multiversion B-Tree and the R*-tree) shows that it provides the faster membership query response time. Partially persistent hashing should be seen as an extension of traditional external dynamic hashing in a temporal environment. It is independent from which ephemeral dynamic hashing scheme is used; while the paper concentrates on linear hashing, the methodology applies to other dynamic hashing schemes as well.

Index Terms: Hashing, Temporal Databases, Transaction-Time, Access Methods, Data Structures.

+ This research was partially supported by NSF grants IIS-9509527, IIS-9907477 and by the Department of Defense.

1. Introduction

There are many applications that require the maintenance of time-evolving[1] data [OS95, JS99]. Examples include accounting, marketing, billing etc. If past states of the data are also stored, the data volume increases considerably with time. Various temporal access methods (indices) have been proposed to query past data [ST99]. In particular, research has concentrated on supporting pure-snapshot or range-snapshot queries over past states of a time-evolving set *S*. Let *S(t)* denote the state (collection of objects) that *S* had at time *t*. Given time *t*, a pure-snapshot query asks for "*all* objects in *S(t)*", while a range-snapshot query retrieves "the objects in *S(t)* with identities (oids) in a specified range" [LS89, TK95, BGO+96, VV97]. Instead, this paper concentrates on the *temporal membership* query[2], i.e.: "given oid *k* and time *t* find whether *k* was in *S(t)*".

Temporal membership queries are important for many applications: (1) In admission/passport control and security applications we are interested whether a person was in some place at a given date. (2) Traditionally, efficient methods to address membership queries are used to expedite join queries [SD90, LR96]. Before joining two large sets, the membership property partitions them into smaller disjoint subsets and every subset from the first set is joined with only one subset from the second set. This idea can be extended to solve special cases of temporal joins [SSJ94, Z97, PJ99]. For example, given two time-evolving sets *X* and *Y*, join $X(t_1)$ and $Y(t_2)$. This temporal join is based on oids and on (possibly different) time instants (i.e., find the employees working in both departments *X* and *Y*, at times t_1 and t_2 respectively). (3) A by-product of partitioning (and thus of the membership property) is query parallelism. That is, the above temporal joins can be performed faster, if the corresponding set partitions are joined in parallel [LM92].

Conventional membership queries over a set of objects have been addressed through hashing. Hashing can be applied either as a main memory scheme (all data fits in main-memory [DKM+88, FNSS92]) or in database systems (where data is stored on disk [L80]). Its latter form is called *external hashing* [EN00, R97]. For every object in the set, a hashing function computes the *bucket* where the object is stored. Each bucket has initially the size of a page. For this discussion we assume that a page can hold *B* objects. Ideally, each distinct oid should be mapped to a separate bucket, however this is unrealistic as the universe of oids is usually much larger than the number of buckets allocated by the hashing scheme. If a bucket cannot accommodate the oids mapped to it in the space assigned to the bucket, an *overflow* occurs. Overflows are dealt in various ways,

1. In the rest of this paper, the terms time and temporal refer to *transaction-time* [J+94].
2. Pure-, range-snapshot and temporal membership queries are denoted as "**//-/S*","*R//-/S*" and "*S//-/S*" in [TJS98].

including rehashing (where another bucket is found using another hashing scheme) and/or chaining (create a chain of pages under the overflowed bucket).

If no overflows are present, finding whether a given oid is in the set is trivial: simply compute the hashing function for the oid and visit the appropriate bucket. With a *perfect* hashing scheme membership queries are answered in $O(1)$ steps (just one I/O to access the page of the bucket). If data is not known in advance, the worst case query performance of hashing is large (linear in the size of the set since a bad hashing function could map all oids to the same bucket). Nevertheless, practice has shown that in the absence of pathological data, good hashing schemes with few overflows and expected $O(1)$ query performance exist (assuming uniform data distribution). This is a major difference between hashing and indexing schemes. With a balanced index (like a B+ tree [C79]) answering a membership query takes logarithmic time on the set size. For many applications (for example in joins [SD90, LR96]) a hashing scheme with expected constant query performance (one or two I/O's) is preferable to the logarithmic worst case performance (four or more I/O's if the set is large) of a balanced index.

Static hashing refers to schemes that use a predefined collection of buckets. This is inefficient if the set of objects is allowed to change (by adding or deleting objects from it). Instead, a *dynamic* hashing scheme has the property of allocating space proportional to the set size. Various external dynamic hashing schemes [L78, FNPS79, ED88] have been proposed, among which *linear hashing* [L80] appears to be the most commonly used.

Even if the set evolves, traditional dynamic hashing is *ephemeral*, i.e., it answers membership queries on the most current state of set S. The problem addressed in this paper is more general as it involves membership queries for any state that set S exhibited.

Assume that for every time t when $S(t)$ changes (by adding/deleting objects) we could have a good ephemeral dynamic hashing scheme $h(t)$ that maps efficiently (with few overflows) the oids of $S(t)$ into a collection of buckets $b(t)$. One straightforward solution to the temporal hashing problem would be to separately store each collection of buckets $b(t)$ for each t. Answering a temporal membership query for oid k and time t requires: (i) identifying $h(t)$ (the hashing function used at t) and (ii) applying $h(t)$ on k and accessing the appropriate bucket of $b(t)$. This would provide an excellent query performance (as it uses a separate hashing scheme for each t), but the space requirements are prohibitively large. If n denotes the number of changes in S's evolution, storing each $b(t)$ on the disk could easily create $O((n/B)^2)$ space.

Instead we propose a solution that uses space linear in n but still has good query performance.

We call our solution *partially persistent hashing* as it reduces the original problem into a collection of partially persistent[1] sub-problems. We apply two approaches for solving these sub-problems. The first approach "sees" each sub-problem as an evolving subset of set *S* and is based on the Snapshot Index [TK95]. The second approach "sees" each sub-problem as an evolving sublist. In both cases, the partially persistent hashing scheme "observes" and stores the evolution of the ephemeral hashing *h(t)*. While the paper concentrates on linear hashing, the partially persistence methodology applies to other dynamic hashing schemes as well (for example, extendible hashing [FNPS79]).

We compare partially persistent hashing with three other approaches. The first approach uses a traditional dynamic hashing scheme to map all oids ever created during the evolution of *S*. However, this scheme does not distinguish among subsequent occurrences of the same oid *k* created by adding and deleting *k* many times. Because all such occurrences of oid *k* will be hashed on the same bucket, bucket sizes will increase and will eventually deteriorate performance (note that bucket reorganizations will not solve the problem; this was also observed in [AS86]). The second approach assumes that a B+ tree is used to index each *S(t)* and makes this B+ tree partially persistent [BGO+96, VV97, LS89]. The third approach sees each oid-interval combination as a multidimensional object and uses an R*-tree [G84, BKKS90] for storing it. Our experiments show that the partially persistent hashing outperforms the other three competitors in membership query performance while having a minimal space overhead.

The partially persistent B+ tree [BGO+96, VV97, LS89] is technically the more interesting among the competitor approaches. Conceptually, many versions of an ephemeral B+-tree are embedded into a graph-like structure. This graph has many root nodes, each root providing access to subsequent versions of the ephemeral B+-tree, organized by time. Searching for oid *k* at time *t* involves: (i) identifying the root valid at *t*, and, (ii) searching for *k* through the nodes of the B+-tree emanating from this root [BGO+96]. Part (i) is bounded by $O(\log_B(n/B))$, while part (ii) takes $O(\log_B(m/B))$ I/O's, where *m* corresponds to the number of objects in *S(t)*. In practice however, the I/O's for part (i) can be eliminated if large enough main memory is available. For example, [BGO+96] identifies the appropriate root using the root* structure which is kept in main-memory.

It was an open question, whether an efficient temporal extension existed for hashing. The work presented here answers this question positively. Searching with the partially persistent hashing

1. A structure is called persistent if it can store and access its past states [DSST89]. It is called partially persistent if the structure evolves by applying changes to its "most current" state.

scheme has also two parts. The first part is again bounded by $O(\log_B(n/B))$ and identifies the hashing function and appropriate bucket valid at time *t*. The second part takes expected $O(1)$ time (i.e., it is independent of *m*). In practice, with large enough main memory, the I/O's from the logarithmic overhead can be eliminated. This work supports our conjecture [KTF98] that transaction-time problems can be solved by taking an efficient solution for the non-temporal problem and making it partially persistent. Note that as with ephemeral hashing, the worst case query behavior of temporal hashing is linear (*O(n/B)*) but this is a rather pathological case.

The rest of the paper is organized as follows: section 2 presents background and previous work. The description of partially persistent linear hashing appears in section 3. Performance comparisons are presented in section 4, while conclusions and open problems for further research appear in section 5. Appendix A contains the main notations and symbols used throughout the paper while Appendix B discusses the update and space analysis of the evolving-list approach. Appendices C and D provide further performance comparisons (to be discussed later).

2. Background and Previous Work

A simple model of temporal evolution follows. Assume that time is discrete described by the succession of non-negative integers. Consider for simplicity an initially empty set *S*. When an object is added to *S* and until (if ever) it is deleted from *S*, it is called "alive"[1]. Each object is represented by a record that stores its oid and a semi-closed interval, or *lifespan*, of the form: [*start_time*, *end_time*). An object added at *t* has its interval initiated as [*t*, *now)*, where *now* is a variable representing the always increasing current time. If this object is later deleted, its end_time is updated to the object's deletion time. Since an object can be added and deleted many times, records with the same oid may exist but with non-intersecting lifespan intervals.

The performance of a temporal access method is characterized by three costs: space, query time and update time. Update time is the time needed to update the index for a change that happened on set *S*. Clearly, an index that solves the range-snapshot query can also solve the pure-snapshot query. However, as indicated in [TGH95], a method designed to address primarily the pure-snapshot query does not order incoming changes according to oid and can thus enjoy faster update time than a method designed for the range-snapshot query. Indeed, the Snapshot Index [TK95] solves the pure-snapshot query in $O(\log_B(n/B) + a/B)$ I/O's, using *O(n/B)* space and only $O(1)$ update time per change (in the expected amortized sense [CLR90] because a hashing scheme is

1. Clearly, *S(t)* contains all the alive objects at *t*.

employed). Here, *a* corresponds to the number of alive objects in *S(t)*. This is the I/O-optimal solution for the pure snapshot query [ST99].

For the range-snapshot query three efficient methods exist: the TSB tree [LS89], the MVBT [BGO+96] and the MVAS structure [VV97]. Both the MVBT and MVAS solve the range-snapshot query in $O(\log_B(n/B) + a/B)$ I/O's, using *O(n/B)* space and $O(\log_B(m/B))$ update per change (in the amortized sense [CLR90]). Here *m* denotes the number of "alive" objects when an update takes place and *a* denotes the number of objects from *S(t)* with oids in the query specified range. This is the I/O -optimal solution for the range-snapshot query. The MVAS structure improves the merge/split policies of the MVBT thus resulting in a smaller constant in the space bound. The TSB tree is another efficient solution to the range-snapshot query. In practice it is more space efficient than the MVBT (and MVAS), but it can guarantee worst case query performance only when the set evolution is described by additions of new objects or updates on existing objects [S99, ST99].

To the best of our knowledge, the temporal membership query is a novel problem [ST99]. One could of course use a temporal access method designed for range-snapshot queries [LS89, BGO+96, VV97] and limit the query range to a single oid. There are however two disadvantages with this approach. First, since the range-snapshot access method orders objects by oid, a logarithmic update $(O(\log_B(m/B)))$ is needed. Second, because the range-snapshot method answers the membership query by traversing a tree path, the same logarithmic overhead $(O(\log_B(m/B)))$ appears in the query time. Nevertheless, in our experiments we include a range-snapshot method for completeness. Since we assume that object deletions are frequent, we use the MVBT instead of the TSB. We use the MVBT instead of the MVAS since the MVBT disk-based code was readily available to the authors. As explained in the performance analysis section, the MVBT was also compared with a main-memory based simulation of the MVAS but there was no drastic performance difference among the two methods for the temporal membership query.

2.1 The Snapshot Index and Linear Hashing

The Snapshot Index [TK95] uses three basic structures: a balanced tree (*time-tree*) that indexes data pages by time, a pointer structure (*access-forest*) among the data pages and an ephemeral hashing scheme. The time-tree and the access-forest enable fast query response while the hashing scheme is used for fast update. When an object is added in set *S*, its record is stored sequentially in a data page. When an object is deleted from *S*, its record is located and its *end_time* is updated. The main advantage in the Snapshot Index, is that it clusters alive objects together. To achieve this, the method uses the notion of page *usefulness*; a page is called useful as long as it contains *uB* alive records $(0 < u \leq 1)$. When a page becomes non-useful (due to object deletions) its alive records are copied to another page. The usefulness parameter u is a constant that tunes the performance of the Snapshot Index. Larger u implies better clustering and thus less query time but more copying and hence more space. For more details we refer to [TK95].

Linear Hashing (LH) is a dynamic hashing scheme that adjusts gracefully to object insertions and deletions. The scheme uses a collection of buckets that grows or shrinks one bucket at a time. Each bucket is initially assigned one page. Overflows are handled by creating a chain of pages under the overflowed bucket. The hashing function changes dynamically and at any given instant at most two hashing functions are used. More specifically, let U be the universe of oids and h_0: U -> $\{0,...,M\text{-}1\}$ be the initial hashing function that is used to load set S into M buckets (for example: $h_0(oid) = oid \bmod M$). Insertions and deletions of oids are performed using h_0 until a bucket overflow happens. When the first overflow occurs (in *any* bucket), the *first* bucket in the LH file, bucket 0, is *split* (rehashed) into two buckets: the original bucket 0 and a new bucket M, which is attached at the end of the LH file. The oids originally mapped into bucket 0 (using function h_0) are now distributed between buckets 0 and M using a *new* hashing function $h_1(oid)$.

Further bucket overflows will cause additional bucket splits in a *linear* bucket-number order. A variable p indicates which is the bucket to be split next (i.e., initially $p = 0$). Conceptually the value of p denotes which of the two hashing functions that may be enabled at any given time, applies to what buckets. After enough overflows, all original M buckets will be split. This marks the end of a *splitting-round*. Variable p is reset to 0 and a *new* splitting round is started. In general, round i starts with $p = 0$, buckets $\{0,...,2^iM\text{-}1\}$ and hashing functions $h_i(oid)$ and $h_{i+1}(oid)$. The round ends when all 2^iM buckets are split. For our purposes we use $h_i(oid) = oid \bmod 2^iM$. Functions h_j, j=1,..., are called *split functions* of h_0. A split function h_j has the properties: (i) h_j: U -> $\{0,...,2^jM\text{-}1\}$ and (ii) for any oid, either $h_j(oid) = h_{j-1}(oid)$ or $h_j(oid) = h_{j-1}(oid) + 2^{j-1}M$. At any time, the hashing scheme is completely identified by the round number i and variable p. Given i and p, searching for oid k is performed using h_i if $h_i(k) \geq p$; otherwise h_{i+1} is used.

A split performed whenever a bucket overflow occurs is an *uncontrolled* split. Let l denote the LH file's *load factor*, i.e., $l = |S| / B|b|$ where $|S|$ is the current size (cardinality) of set S and $|b|$ the current number of buckets in the LH file. The load factor achieved by uncontrolled splits is usually between 50-70%, depending on the page size and the oid distribution [L80]. In practice, a higher storage utilization is achieved if a split is triggered not by an overflow, but when the load factor l becomes greater than some *upper threshold* g $(g \leq 1)$ [L80, EN00, FZR98]. This is called a *controlled split* and can typically achieve 95% utilization. (Note that the split is now independent of bucket overflows. Other controlled schemes exist where a split is delayed until both the threshold condition holds and an overflow occurs). Deletions in set S will cause the LH file to *shrink*. Buckets that have been split can be recombined (in reverse linear order) if the load factor falls below some *lower threshold* $f(0 < f \le g)$. Practical values for f and g are 0.7 and 0.9, respectively. In our experiments and analysis we use the *controlled* version of linear hashing.

3. Partially Persistent Linear Hashing

We will apply the partially persistent methodology on an ephemeral linear hashing scheme. Section 3.1 describes the evolving-set approach while the evolving-list approach is in section 3.2.

3.1 The Evolving-Set Approach

Using partial persistence, the temporal hashing problem is reduced into a number of sub-problems for which efficient solutions are known. Assume that an ephemeral linear hashing scheme (as the one described in section 2) is used to map the objects of $S(t)$. As $S(t)$ evolves the hashing scheme is a function of time, too. Let $LH(t)$ denote the linear hashing file as it is at time t. The two basic parameters that identify $LH(t)$ for each t are now time-dependent, namely $i(t)$ and $p(t)$.

An interesting property of linear hashing is that buckets are reused; when round i+1 starts it uses twice as many buckets as round i but the first half of the bucket sequence is the same since new buckets are appended at the end of the file. Let b_{total} denote the longest sequence of buckets ever used during the evolution of $S(t)$ and assume it reaches up to bucket $2^q M - 1$. Clearly $b(t)$, the collection of buckets used at time t, is a prefix of b_{total} and $i(t) \le q, \forall t$.

Consider bucket b_j from the sequence b_{total} $(0 \le j \le 2^q M - 1)$. State $b_j(t)$ is the set of oids stored in this bucket at t. If any state $b_j(t)$ can be reconstructed for each bucket b_j, answering a temporal membership query for oid k at time t is answered in two steps:

(1) find which bucket b_j, oid k would have been mapped by $LH(t)$, and,

(2) search through the contents of $b_j(t)$ until k is found.

The first step requires identifying $LH(t)$. This is easily maintained if a record of the form $< t, i(t), p(t) >$ is appended to an array H, for those instants t where $i(t)$ and/or $p(t)$ change. Given any t, $LH(t)$ is identified by simply locating t inside the time-ordered H in a *logarithmic* search (see discussion below).

The second step implies accessing $b_j(t)$. By observing the evolution of bucket b_j we note that its state changes as an *evolving set* by adding/deleting oids. Each such change can be timestamped with the time it occurred. At times the ephemeral linear hashing scheme may apply a rehashing procedure that remaps the current contents of bucket b_j to bucket b_j and some new bucket b_r. Assume that such rehashing occurred at time t' and its result is a move of v oids from b_j to b_r. For the evolution of b_j (b_r), this rehashing is viewed as a deletion (respectively addition) of the v oids at time t', i.e., all such deletions (additions) are timestamped with the same time t' for the corresponding object's evolution.

Figure 1 shows an example of the ephemeral hashing scheme at two different time instants. For simplicity $M = 5$ and $B = 2$. Figure 2 shows the corresponding evolution of set S and the evolutions of three buckets (a "+/-" denotes addition/deletion respectively). Controlled splitting with an upper threshold $g = 0.9$ is assumed. At time $t = 21$ the addition of oid 8 triggers a split since $|S| = 10$ oids and $M = 5$, $l > g$ (at the same time oid 8 causes an overflow on bucket 3). Thus the contents of bucket 0 are rehashed between bucket 0 and bucket 5. As a result oid 15 is moved to bucket 5. For bucket's 0 evolution this change is considered as a deletion at $t = 21$ but for bucket 5 it is an addition of oid 15 at the same $t = 21$. The records stored in each bucket's history are also shown. For example, at t=25, oid 10 is deleted from set S. This updates the lifespan of this oid's corresponding record in bucket 0's history from <10, [1, *now*)> to <10, [1, 25)>.

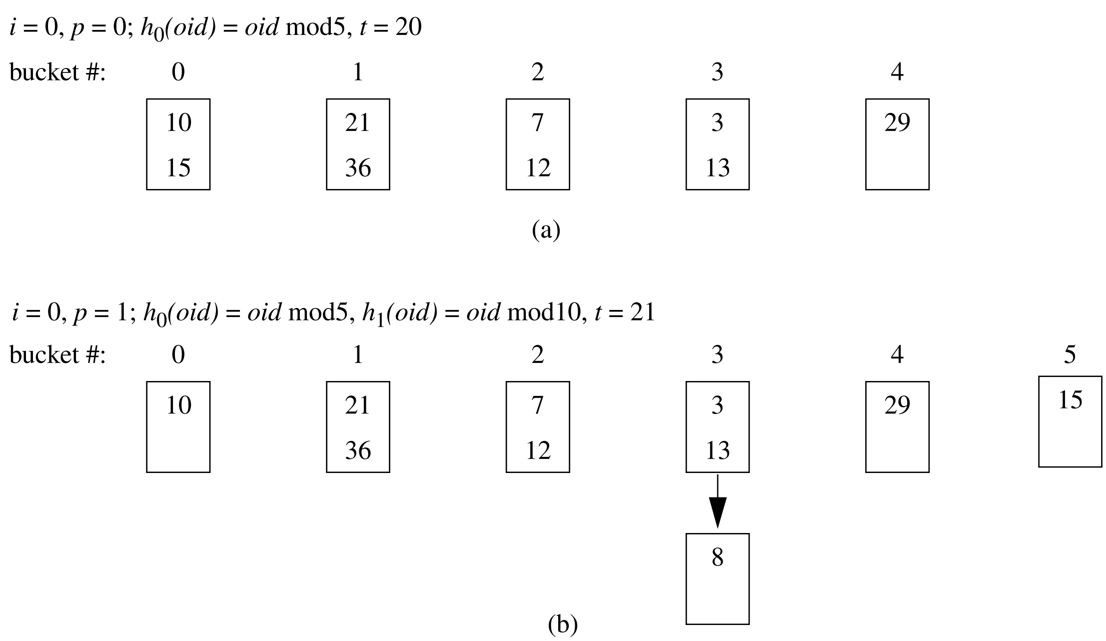


Figure 1: Two instants in the evolution of an ephemeral hashing scheme. (a) Until time $t = 20$, no split has occurred and $p = 0$. (b) At $t = 21$, oid 8 is mapped to bucket 3 and causes a controlled split. Bucket 0 is rehashed using h_1 and $p = 1$.

If $b_j(t)$ is available, searching through its contents for oid k is performed by a linear search. This process is lower bounded by $O(|b_j(t)|/B)$ I/O's since these many pages are at least needed for storing $b_j(t)$. (This is similar with traditional hashing where a query about some oid is translated into searching the pages of a bucket; this search is also linear and continues until the oid is found or all the bucket's pages are searched.) Since every bucket b_j behaves like a set evolving over time, the Snapshot Index [TK95] can be used to store the evolution of each b_j and reconstruct any $b_j(t)$ with effort proportional to $|b_j(t)|/B$ I/O's.

evolution of set *S* up to time *t* = 25:

(*t*	*oid*	*oper.*)
1,	10,	+
2,	7,	+
4,	3,	+
8,	21,	+
9,	15,	+
15,	36,	+
16,	29,	+
17,	13,	+
20,	12,	+
21,	8,	+
25,	10,	-

evolution of bucket 0:

(*t*	*oid*	*oper.*)
1,	10,	+
9,	15,	+
21,	15,	-
25,	10,	-

records in bucket 0's history

at *t* = 20:	at *t* = 21:	at *t* = 25:
<*oid*, lifespan>	<*oid*, lifespan>	<*oid*, lifespan>
<10, [1, *now*)>	<10, [1, *now*)>	<10, [1, 25)>
<15, [9, *now*)>	<15, [9, 21)>	<15, [9, 21)>

evolution of bucket 3:

(*t*	*oid*	*oper.*)
4,	3,	+
17,	13,	+
21,	8,	+

records in bucket 3's history

at *t* = 20:	at *t* = 21:	at *t* = 25:
<*oid*, lifespan>	<*oid*, lifespan>	<*oid*, lifespan>
<3, [4, *now*)>	<3, [4, *now*>	<3, [4, *now*)>
<13, [17, *now*)>	<13, [17, *now*)>	<13, [17, *now*)>
	<8, [21, *now*)>	<8, [21, *now*)>

evolution of bucket 5:

(*t*	*oid*	*oper.*)
21,	15,	+

records in bucket 5's history

at *t* = 20:	at *t* = 21:	at *t* = 25:
<*oid*, lifespan>	<*oid*, lifespan>	<*oid*, lifespan>
-	<15, [21, *now*)>	<15, [21, *now*)>

Figure 2: The detailed evolution for set *S* until time *t* = 25. Addition of oid 8 in *S* at *t* = 21, causes the first split. Moving oid 15 from bucket 0 to bucket 5 is seen as a deletion and an addition respectively.

The query performance of partially persistent linear hashing depends on the ephemeral hashing scheme used for $S(t)$. Since the worst case of ephemeral hashing is linear, the worst case of partially persistent hashing is $O(n/B)$. Nevertheless, a good ephemeral hashing scheme $LH(t)$ would use expected $O(1)$ I/O's for answering a membership query on $S(t)$. This means that on average each bucket $b_j(t)$ used for $S(t)$ would be of limited size, or equivalently, $|b_j(t)|/B$ corresponds to just a few (one or two) pages. In perspective, partially persistent linear hashing will reconstruct $b_j(t)$ in $O(|b_j(t)|/B)$ I/O's, which is expected $O(1)$.

The size of array *H* depends on *s*, the *maximum* number of alive objects at any time instant $(s = max_t(|S(t)|))$. An entry is added in *H* for each controlled split, i.e., when the number of alive objects is above the load factor threshold. If the scheme starts with a single bucket at worse a new split happens after adding *Bl* new oids. The number of splits is therefore bounded by *O(s/Bl)* and table *H* has $O(s/(B^2 l))$ pages. Even if all changes are new oid additions searching array *H* takes $O(\log_B(n/B))$ I/O's. Using *LH(t)* the appropriate bucket b_j is pinpointed and time *t* must be searched in the time-tree associated with this bucket. This search is bounded by $O(\log_B(n_j/B))$ where n_j corresponds to the number of changes recorded in bucket b_j's history (at worst n_j is $O(n)$).

In practice, we expect that: (i) *s* is small as compared to *n*, i.e., array *H* will use few pages and can thus be stored in main memory. (ii) most of *S*'s history will be recorded on the first $2^i M$ buckets (for some *i*); then n_j behaves as $O(n/(2^i M))$ and searching b_j's time-tree is rather fast.

3.1.1 Update and Space Analysis. It suffices to show that the partially persistent linear hashing scheme uses $O(n/B)$ space. An $O(1)$ amortized expected update processing per change can then be derived. Assume that the corresponding ephemeral linear hashing scheme *LH(t)* starts with *M* initial buckets and uses upper threshold *g*. Assume for simplicity that set *S* evolves by only adding oids (oid additions create new records and hence more copying; deletions do not create splits).

Clearly array *H* satisfies the space bound. Next we show that the space used by bucket histories is also bounded by *O(n/B)*. Since for partially persistent linear hashing, each split creates artificial changes (the oids moved to a new bucket are deleted from the previous bucket and subsequently added to a new bucket as new records), it must be shown that the number of oid moves (copies) due to splits is still bounded by the number of real changes *n*. We will first use two lemmas that upper and lower bound such oid moves during the first round of splits.

Lemma 1: For any *N* in $1 \le N \le M$, *N* splits occur after at least $(M+N-1)Bg$ real object additions happen.

<u>Proof:</u> Just before the *N*-th split we have exactly *M*+*N*-1 buckets and hence the load is $l = |S|/B(M+N-1)$. By definition, for the next controlled split to occur we must have $l > g$, which implies that $|S| > (M+N-1)Bg$ q

Lemma 2: For any *N* in $1 \le N \le M$, *N* splits can create at most $(M+N-1)B+1$ oid copies.

<u>Proof:</u> Assume that all buckets are initially empty. Observe that the number of oid copies made during a single round is at most equal to the number of real object additions. This is because during one round an oid is rehashed at most once. The number of real additions before a split increases with *g* (lower *g* triggers a split earlier, i.e., with less oids). Hence the most real additions occur when

$g = 1$. The N-th controlled split is then triggered when the load exceeds 1. Since there are already M+N-1 buckets, this happens at the $((M + N - 1)B + 1)$-th object addition. q

The basic theorem about space and updating follows.

Theorem 1: Partially Persistent Linear Hashing uses space proportional to the total number of real changes and updating that is amortized expected $O(1)$ per change.

Proof: As splits occur, linear hashing proceeds in rounds. In the first round variable p starts from bucket 0 and in the end of the round it reaches bucket M-1. At that point 2M buckets are used and all copies (remappings) from oids of the first round have been created. Since M splits have occurred, lemma 2 implies that there must have been at most (2M-1)B+1 oid moves (copies). By construction, these copies are placed in the last M buckets.

For the next round, variable p will again start from bucket 0 and will extend to bucket 2M-1. When p reaches bucket 2M-1, there have been 2M new splits. These new splits imply that there must have been at most (4M-1)B+1 copies created from these additions. The copied oids from the first round are seen in the second round as regular oid additions. At most each such oid can be copied once more during the second round (the original oids from which these copies were created in the first round, cannot be copied again in the second round as they represent deleted records in their corresponding buckets). Hence the maximum number of copies after the second round is $((2M - 1)B + 1 + (4M - 1)B + 1)$. The total number of copies C_{total} created in all rounds satisfies:

$$C_{total} \leq [\{(2M - 1)B + 1\} + \{(2M - 1)B + 1 + (4M - 1)B + 1\} + \ldots]$$

where each {} represents copies per round. Using that $M \geq 1$ and $B > 1$ we get:

$$C_{total} \leq [2MB + \{4MB + 2MB\} + \ldots] \leq MB\left[\sum_{k=0}^{i} \sum_{j=0}^{k} 2^j\right] = MB[2(2^{i+1} - 1) - (i + 1)] < 2^{i+2}MB \quad \text{(i)}$$

Lemma 1 implies that after the first round we have at least (2M-1)Bg real oid additions. At the end of the second round the total number of oid additions is at least $(4M - 1)Bg$. However $(2M - 1)Bg$ of these where inserted and counted for the first round. Thus the second round introduces at least 2MBg new additions. Similarly, the third round introduces at least 4MBg new oid additions etc. The number of real oid additions A_{total} after all rounds, is lower bounded by:

$A_{total} \geq [\{(2M - 1)Bg\} + \{2MBg\} + \{4MBg\} + \ldots] \geq [MBg + 2MBg + 4MBg + \ldots]$. Hence:

$$A_{total} \geq MBg \sum_{k=0}^{i} 2^k = (2^{i+1} - 1)MBg \geq 2^i MBg \text{ (ii).}$$

From (i), (ii) it can be derived that there exists a positive constant *const* such that $C_{total} / A_{total} < const$. Since A_{total} is bounded by the total number of changes *n*, we have that C_{total}= *O(n)*. For proving that partially persistent linear hashing has $O(1)$ expected amortized updating per change, we note that when a real change occurs it is directed to the appropriate bucket where the structures of the Snapshot Index are updated in $O(1)$ expected time. Rehashings must be carefully examined. This is because a rehashing of a bucket is caused by a single real oid addition (the one that created the split) but it results in a "bunch" of copies made to a new bucket (at worse the whole current contents of the rehashed bucket are sent to the new bucket). However, using the space bound any sequence of *n* real changes can at most create *O(n)* copies (extra work) or equivalently $O(1)$ amortized effort per real change. q

3.1.2 Optimization Issues. Optimizing the performance of partially persistent linear hashing involves the load factor *l* of the ephemeral Linear Hashing and the usefulness parameter *u* of the Snapshot Index. The load *l* lies between thresholds *f* and *g*. Note that *l* is an average over time of $l(t) = |(S(t))| / B|b(t)|$, where $|S(t)|$ and $|b(t)|$ denote the size of the evolving set *S* and the number of buckets used at *t* (clearly $b(t) \leq |b_{total}|$). A good ephemeral linear hashing scheme will try to equally distribute the oids among buckets for each *t*. Hence on average the size (in oids) of each bucket $b_j(t)$ will satisfy: $|b_j(t)| \approx |(S(t))| / |b(t)|$.

One of the advantages of the Snapshot Index is the ability to tune its performance through usefulness parameter *u*. The index will distribute the oids of each $b_j(t)$ among a number of *useful* pages. Since each useful page (except the acceptor page) contains at least *uB* alive oids, the oids in $b_j(t)$ will be occupying at most $|b_j(t)| / uB \approx |(S(t))| / uB|b(t)| = l(t) / u$ pages. Ideally, we would like the answer to a snapshot query to be contained in a single page. Then a good optimization choice is to maintain: $l / u < 1$. Conceptually, the load *l* gives a measure of the size of a bucket ("alive" oids) at each time. These alive oids are stored into the data pages of the Snapshot Index. Recall that an artificial copy happens if the number of alive oids in a data page falls below *uB*. At that point the remaining *uB*-1 alive oids of this page are copied to a new page. By keeping *l* below *u* we expect that the alive oids of the split page will be copied in a single page which minimizes the number of I/O's needed for finding them.

On the other hand, the usefulness parameter u affects the space used by the Snapshot Index and in return the overall space of the persistent hashing scheme. Higher values of u imply frequent time splits, i.e., more page copies and thus more space. Hence it would be advantageous to keep u low but this implies an even lower l. In return, lower l would mean that the buckets of the ephemeral hashing are not fully utilized. This is because low l causes set $S(t)$ to be distributed into more buckets not all of which may be fully occupied.

At first this requirement seems contradictory. However, for the purposes of partially persistent linear hashing, having low l is still acceptable. Recall that the low l applies to the ephemeral hashing scheme whose history the partially persistent hashing observes and accumulates. Even though at single time instants the $b_j(t)$'s may not be fully utilized, over the whole time evolution many object oids are mapped to the same bucket. What counts for the partially persistent scheme is the total number of changes accumulated per bucket. Because of bucket reuse, a bucket will gather many changes creating a large history for the bucket and thus justifying its use in the partially persistent scheme. Our findings regarding optimization will be verified through the experimentation results that appear in section 4.

3.2 The Evolving-List Approach

The elements of bucket $b_j(t)$ can also be viewed as an *evolving list* $lb_j(t)$ of alive oids. Such an observation is consistent with the way buckets are searched in ephemeral hashing, i.e., linearly, as if a bucket's contents belong to a list. Accessing the bucket state $b_j(t)$ is then reduced to reconstructing $lb_j(t)$. Equivalently, the evolving list of oids should be made partially persistent.

We note that [VV97] presents a notion of persistent lists called the C-lists. A C-list is a list structure made up of a collection of pages that contain temporal versions of data records clustered by oid. However, a C-list and a partially-persistent list solve different problems. (1) A C-list addresses the query: "given an oid and a time interval, find all versions of this oid during this interval". Instead, a partially persistent list finds all the oids that its ephemeral list had at a given time instant. (2) In a C-list the various temporal versions of an oid must be clustered together. This implies that oids are also ordered by key. In contrast, a persistent list does not require ordering its keys, simply because its ephemeral list is not ordered. (3) In a C-list updates and queries are performed by first using the MVAS structure. Updating/querying the partially persistent list always starts from the top of the list and proceeds through a linear search.

For the evolving-list approach, a list page has now two areas. The first area is used for storing oid records and its size is B_r (B_r is $O(B)$). The second area, of size B-B_r (B-B_r is also $O(B)$), accommodates an extra structure (array *NT*) which will be explained shortly. When the first oid k is added on bucket b_j at time t, a record <k, [t, *now*)> is appended in the first list page. Additional oid insertions will create record insertions in this page. When the B_r part of the first page gets full of records, a new page is appended in the list and so on. If oid k is deleted at t' from the bucket, its record in the list is found by a serial search among the list pages and its end_time is updated.

Good clustering is achieved if each page in $lb_j(t)$ is a *useful* page. A page is useful if: (i) it is full of records and contains at least uB_r alive records $(0 < u \leq 1)$, or, (ii) it is the last list page. A non-useful page is taken off list $lb_j(t)$ and its alive records are copied to the current last list page. If the last page in $lb_j(t)$ does not have enough free space, a new last page is appended.

Reconstructing $b_j(t)$ is equivalent to finding the pages in $lb_j(t)$. We will use two array structures. Array $FT_j(t)$ provides access to the first page in the list for any time t. Entries in FT_j have the form <*time, pid*> where pid is the address of the first page. When the first page of the list changes a new entry is appended in FT_j. This array can be implemented as a multilevel index since entries are added in increasing time order. Each remaining page of $lb_j(t)$ is found by the second structure, which is an array of size B-B_r, implemented inside every list page. Let $NT(A)$ be the array inside page A. This array is maintained for as long as the page is useful. Entries in $NT(A)$ are also of the form <*time, pid*>, where pid corresponds to the address of the next list page after page A.

To answer a temporal membership query for oid k at time t the appropriate bucket b_j, where oid k would have been mapped by the hashing scheme at t is found. This part is the same with the evolving-set approach. To reconstruct bucket $b_j(t)$, the first page in $lb_j(t)$ is retrieved by searching t in array FT_j. This search is $O(\log_B(n_j/B))$. The remaining pages of $lb_j(t)$ are found by locating t in the *NT* array of each subsequent list page. Since all pages in the list $lb_j(t)$ are useful, the oids in $b_j(t)$ are found in $O(|b_j(t)|/B)$ I/O's. Hence, the space used by the evolving-list approach is $O(n/B)$ while updating is $O(|b_j(t)|/B)$ per update (for details see Appendix B).

There are two differences between the evolving-list and the evolving-set approaches. First, updating using the Snapshot Index remains constant, while in the evolving list the whole current list may have to be searched for deleting an oid. Second, reconstruction in the evolving-list starts from the top of the list pages while in the evolving-set reconstruction starts from the last page of the bucket. This may affect the search for a given oid depending whether it has been placed near the top or near the end of the bucket.

4. Performance Analysis

For the Partially Persistent Linear Hashing we implemented the set-evolution (PPLH-*s*) and the list-evolution (PPLH-*l*) approaches, using controlled splits. Both are compared against Linear Hashing (in particular *Atemporal* linear hashing, which is discussed below), the MVBT and two R*-tree implementations (R_i which stores intervals in a 2-dimensional space, and R_p which stores points in a 3-dimensional space). The partial persistence methodology is applicable to other hashing schemes as well. We implemented Partially Persistent Extendible Hashing using the set-evolution (PPEH-*s*) and uncontrolled splits and compared it with PPLH-*s*. Implementation details and the experimental setup are described in 5.1, the data workloads in 5.2 and our findings in 5.3.

4.1 Method Implementation - Experimental Setup.

We set the size of a page to hold 25 object records (B=25). In addition to the object's oid and lifespan, the record contains a pointer to the actual object (which may have additional attributes).

We first discuss the *Atemporal* linear hashing (ALH). It should be clarified that ALH is <u>not</u> the ephemeral linear hashing whose evolution the partially persistent linear hashing observes and stores. Rather, it is a conventional linear hashing scheme that treats time as just another attribute. This scheme simply maps objects to buckets using the object oids. Consequently, it "sees" the different lifespans of the same oid as *copies* of the same oid. We implemented ALH using the scheme originally proposed by Litwin in [L80]. For split functions we used the hashing by division functions $h_i(oid) = oid \bmod 2^i M$ with $M = 10$. For good space utilization controlled splits were employed. The *lower* and *upper* thresholds (namely f and g) had values 0.7 and 0.9 respectively.

Another approach for Atemporal hashing would be a scheme which uses a combination of oid and the start_time or end_time attributes. However this approach would still have the same problems as ALH for temporal membership queries. For example, hashing on start_time does not help for queries about time instants other than the start_times.

The two Partially Persistent Linear Hashing approaches (PPLH-*s* and PPLH-*l*) observe an ephemeral linear hashing $LH(t)$ with controlled splits and load between f=0.1 and g=0.2. Array H is kept in main-memory (i.e., no I/O cost is counted for accessing H). In our experiments the size of H varied from 5 to 10 KB (which can easily fit in today's main memories). Unless otherwise noted, PPLH-*s* was implemented with u = 0.3 (other values for usefulness parameter u were also examined). Since the entries in the time-tree associated with a bucket have half the object record size, each time-tree page can hold up to 50 entries.

In the PPLH-l implementation B_r holds 20 object records. Here the usefulness parameter was set to $u = 0.25$. The remaining space in a list page (5 object records) is used for the page's NT array. Similarly with the time-arrays, NT arrays have entries of half size, i.e., each page can hold 10 NT entries. For the same reason, the pages of an FT_j array can hold up to 50 entries.

The Multiversion B-tree (MVBT) implementation uses buffering during updates; this buffer stores the pages in the path to the last update (LRU buffer replacement policy is used). Such buffering can be very advantageous since updates are directed towards the most current B-tree, which is a small part of the whole MVBT structure. In our experiments we set the buffer size to 10 pages. The original MVBT uses buffering for queries, too. For a fair comparison with the partially persistent methods, during queries we allow the MVBT to use a buffer that is as large in size as array H for that experiment. Recall that during querying the MVBT uses a *root** structure. Even though root* can increase with time, it is small enough to fit in the above main-memory buffer. Thus we do not count I/O accesses for searching root*.

As with the Snapshot Index, a page in the MVBT is "alive" as long as it has at least q alive records. If the number of alive records falls below q this page is merged with a sibling (this is called a *weak version underflow* in [BGO+96]). On the other extreme, if a page has already B records (alive or not) and a new record is added, the page splits (called a page *overflow* in [BGO+96]). Both conditions need special handling. First, a time-split happens (which is like the copying procedure of the Snapshot Index). All alive records in the split page are copied to a new page. Then the resulting new page should be incorporated in the structure. The MVBT requires that the number of alive records in the new page should be between $q+e$ and $B-e$ where e is a predetermined constant. Constant e works as a threshold that guarantees that the new page can be split or merged only after at least e new changes. Not all values for q, e and B are possible as they must satisfy some constraints; for details we refer to [BGO+96]. In our implementation we set $q = 5$ and $e = 4$.

Another choice for a range-snapshot method would be the MVAS [VV97] whose updating policies lead to smaller space requirements than the MVBT. In addition, the MVAS does not use the root* structure. With either method, a temporal membership query is answered by following a single path which is proportional to the height of the structure. We had access to a main-memory simulation of the MVAS and compared it with the disk-based MVBT implementation. We were able to perform up to $n = 120$K updates with the simulated MVAS code, using (a shorter version of) the uniform-10 workloads (see Table 1 in section 4.3). As expected, the simulated MVAS used less space (6209 pages against 6961 pages in the MVBT). However, the difference was not enough to affect the structures' heights. In particular, the simulated MVAS had height six, while the MVBT needed 2 levels in the root* and each tree underneath had height between three and four.

The R_i implementation assigns to each oid its lifespan interval. One dimension is used for the oid and one for the interval. When a new oid k is added in set S at time t, a record <k, [t, *now*), *ptr*> is added in an R*-tree data page. If oid k is deleted at t', the record is updated to <k, [t, t'), *ptr*>. Note that *now* is stored in the R*-tree as some large number (larger than the maximum evolution time). Directory pages in the R*-tree include one more attribute per record for representing an oid range. The R_p implementation has similar format for data pages, but it assigns separate dimensions for the start_time and the end_time of the object's lifespan interval. Hence a directory page record has seven attributes (two for each of the oid, start_time, end_time and one for the pointer). During updating, both R*-tree implementations used a buffer (10 pages) to keep the pages in the path leading to the last update. A buffer as large as the size of array H was again used during the query phase.To further optimize the R*-tree approaches towards temporal membership queries we forced the trees to cluster data first based on the oids and then on the time attributes.

Another popular external dynamic hashing scheme is Extendible Hashing [FNPS79], which is different from linear hashing in two ways. First, a directory structure is used to access the buckets. Second, an overflow is addressed by splitting the bucket that overflowed. Note that traditional extendible hashing uses uncontrolled splitting. To implement Partially Persistent Extendible Hashing one needs to keep the directory history as well as the history of each bucket. The history of each bucket was kept using the evolving-set approach (PPEH-*s*). In our experiments the directory history occupied between 7 and 18KB so we kept it in main memory. Since uncontrolled splitting is used, most buckets will be full of records before being split. This implies that the alive records in a bucket are close to one page. Consequently, to get better query performance (with the expense of higher space overhead), the history of such buckets was implemented using Snapshot indices with higher utilization parameter (u=0.5).

4.2 Workloads.

Various workloads were used for the comparisons. Each workload contains an evolution of a dataset S and temporal membership queries on this evolution. More specifically, a *workload* is defined by triplet $W=(U,E,Q)$, where U is the universe of the oids (the set of unique oids that appeared in the evolution of set S; clearly $s \le |U|$), E is the evolution of set S and $Q = \{Q_1, \ldots, Q_{|U|}\}$ is a collection of queries and Q_k is the set of queries that corresponds to oid k.

Each evolution starts at time 1 and finishes at time *MAXTIME*. Changes in a given evolution were first generated per object oid and then merged. First, for each object with oid k, the number n_k of the different lifespans for this object in this evolution was chosen. The choice of n_k was made using a specific random distribution function (namely *Uniform, Exponential, Step* or *Normal*) whose details are described in the next section. The start_times of the lifespans of oid k were generated by randomly picking n_k different starting points in the set {1,..., *MAXTIME*}. The end_time of each lifespan was chosen uniformly between the start_time of this lifespan and the start_time of the next lifespan of oid k (since the lifespans of each oid k are disjoint). Finally the whole evolution E for set S was created by merging the evolutions for every object.

For another "mix" of lifespans, we also created an evolution that picks the start_times and the length of the lifespans using Poisson distributions; we called it the *Poisson* evolution.

A temporal membership query in query set Q is specified by tuple *(oid,t)*. The number of queries Q_k for every object with oid k was chosen randomly between 10 and 20; thus on average, $\overline{Q}_k \sim 15$. To form the *(k,t)* query tuples the corresponding time instants t were selected using a uniform distribution from the set {1, ... , *MAXTIME*}. The *MAXTIME* is set to 50000 for all workloads. The value used in the R-trees for *now* was 100000.

Each workload is described by the distribution used to generate the object lifespans, the number of different oids, the total number of changes in the evolution n (object additions and deletions), the total number of object additions *NB*, and the total number of queries.

4.3 Experiments.

First, the behavior of all implementations was tested using a basic *Uniform* workload. The number of lifespans per object follows a uniform distribution between 20 and 40. The total number of distinct oids was $|U| = 8000$, the number of real changes $n = 466854$ and $NB = 237606$ object additions. Hence the average number of lifespans per oid was $\overline{NB} \sim 30$ (we refer to this workload as Uniform-30). The number of queries was 115878.

Figure 3.a presents the average number of pages accessed per query by all methods. The PPLH methods have the best performance, about two pages per query. The ALH approach uses more query I/O (about 1.5 times in this example) because of the larger buckets it creates. The MVBT also uses more I/O (about 1.75 times) than the PPLH approaches since a tree path is traversed per query. The R_i uses more I/O's per query than the MVBT (about 11.5 I/O's), mainly due to tree node overlapping and larger tree height (its height relates to the total number of oid lifespans while

MVBT's height corresponds to the alive oids at the time specified by the query). The R_p tree has the worse query performance (an average of 28.3 I/O's per query). The performance of the R-tree methods has been truncated in Figure 3.a to fit the graph. While using a separate dimension for the two endpoints of a lifespan interval allows for better clustering (see also the space usage in Figure 3.c) it makes it more difficult to check whether an interval contains a query time instant.

Figure 3.b shows the average number of I/O's per update. The best update performance was given by the PPLH-*s* method. In PPLH-*l* the NT array implementation inside each page limits the actual page area assigned for storing oids and thus increases the number of pages used per bucket. The MVBT update is longer than PPLH-*s* since the MVBT traverses a tree for each update (instead of quickly finding the location of the updated element through hashing). The update of R_i follows; it is larger that the MVBT since the size of the tree traversed is related to all oid lifespans (while the size of the MVBT structure traversed is related to the number of alive oids at the time of the update). The R_p tree uses larger update processing than the R_i because of the overhead to store an interval as two points. The ALH had the worse update processing since all lifespans with the same oid are thrown on the same bucket, creating large buckets that must be searched serially.

The space consumed by each method appears in figure 3.c. The ALH approach uses the smallest space since it stores a single record per oid lifespan and uses "controlled" splits with high utilization. The PPLH-*s* method has also very good space utilization, very close to ALH. The R-tree methods follow; R_p uses slightly less space than the R_i because paginating intervals (putting them into bounding rectangles) is more demanding than with points. Note that similarly to ALH, both R* methods use a single record per oid lifespan; the additional space is mainly because the average R-tree page utilization is about 65%. PPLH-*l* uses more space than PPLH-*s* because the *NT* array implementation reduces page utilization. The MVBT has the largest space requirements, about twice more space than the ALH and PPLH-*s* methods.

To consider the effect of lifespan distribution all approaches were compared using five additional workloads (called the exponential, step, normal, poisson and uniform-consecutive). These workloads had the same number of distinct oids ($|U| = 8000$), number of queries (115878) and similar n (~0.5M) and $\overline{NB}$ (~30) parameters. The *Exponential* workload generated the n_k lifespans per oid using an exponential distribution with probability density function $f(x) = \beta exp(-\beta x)$ and mean $1/\beta = 30$. The total number of changes was n = 487774, the total number of object additions was NB = 245562 and $\overline{NB}$ = 30.7. In the *Step* workload the number of lifespans per oid follows a step function. The first 500 oids have 4 lifespans, the next 500 have 8

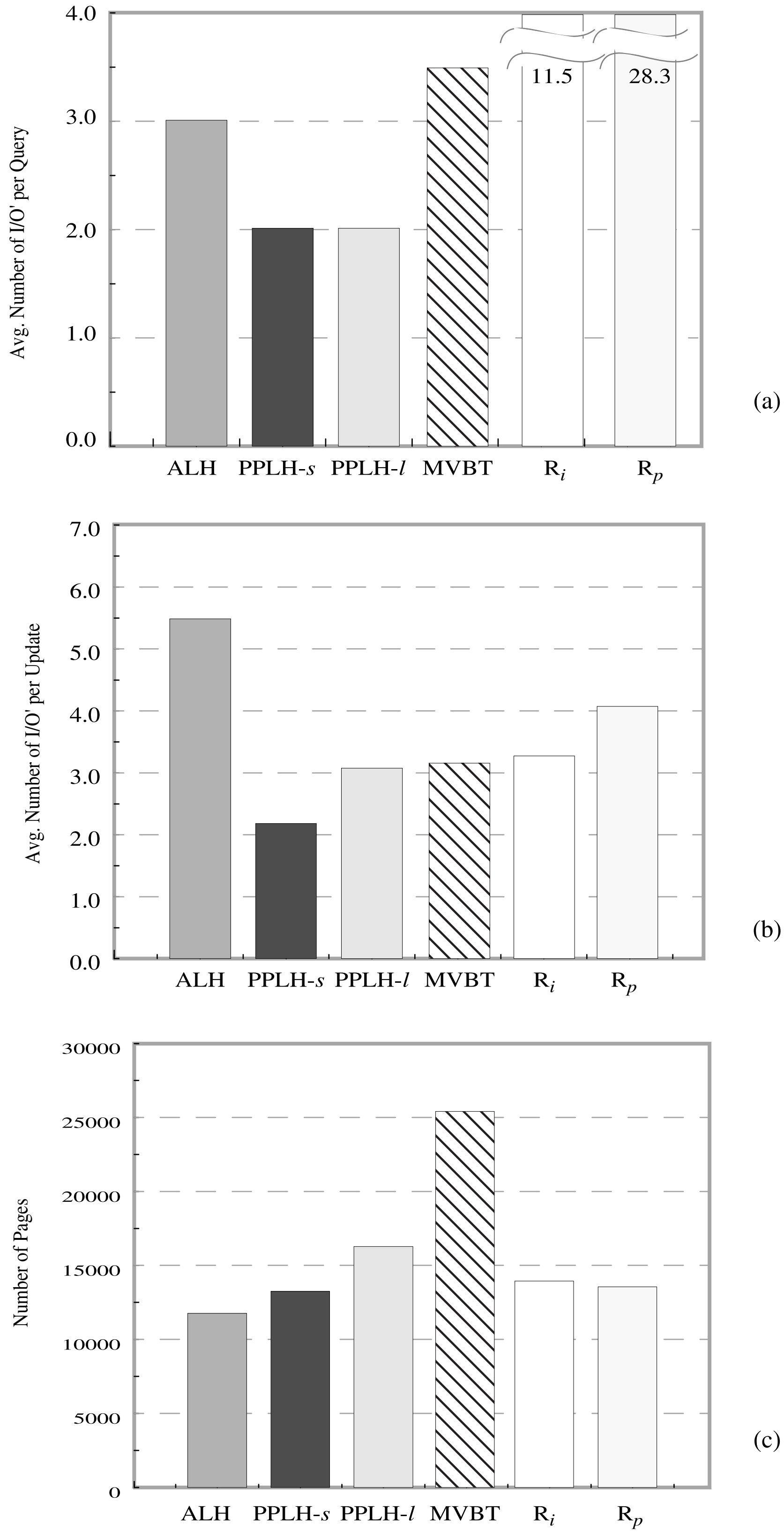


Figure 3: (a) Query, (b) Update, and, (c) Space performance for all implementations on a uniform workload with 8K oids, n ~ 0.5M and $\overline{NB}$ ~ 30.

lifespans and so on, i.e., for every 500 oids the number of lifespans advances by 4. In this workload we had n = 540425, NB = 272064 and $\overline{NB}$ = 34. The *Normal* workload used a normal distribution with $\mu = 30$ and $\sigma^2 = 25$. Here the parameters were: n = 470485, NB = 237043 and $\overline{NB}$ = 29.6.

For the *Poisson* workload the first lifespan for every oid was generated randomly between time instants 1 and 500. The length of a lifespan was generated using a Poisson distribution with mean 1100. Each next start time for a given oid was also generated by a Poisson distribution with mean value 500. For this workload we had n = 498914, NB = 251404 and $\overline{NB}$ = 31. The main characteristic of the Poisson workload is that the number of alive oids over time can vary from a very small number to a large proportion of $|U|$, i.e., there are time instants where the number of alive oids is some hundreds and other time instants where almost all distinct oids are alive.

The special characteristic of the *Uniform-consecutive* workload is that it contains objects with multiple but consecutive lifespans. This scenario occurs when objects are updated frequently during their lifetime. Each update is seen as the deletion of the object followed by the insertion of the updated object at the same time. Since the object retains its oid through updates, this process creates consecutive lifespans for the same object (the end of one lifespan is the start of the next lifespan). This workload was based on the Uniform-30 workload and had n = 468715, NB = 236155 and $\overline{NB}$ = 30. An object has a single lifetime which is cut into consecutive lifespans.The start_times of an object's lifespans are chosen uniformly.

Figure 4 presents the query, update and space performance under the new workloads. The results resemble the Uniform-30 workload. (For brevity we have excluded the R-tree based methods from the remaining discussion as they consistently had much worse query performance; the interested reader can find the detailed performance in [KT98]). As before, the PPLH-*s* approach has the best overall performance using slightly more space than the "minimal" space of ALH. PPLH-*l* has the same query performance with PPLH-*s* but uses more updating and space. Note that in Figure 4.c the space of the MVBT is truncated (MVBT used about 26K, 28K, 25K, 35K and 28K pages for the respective workloads). We have tried the consecutive workloads for the exponential, step, normal and poisson distributions, too. The comparative behavior of all methods in the consecutive workloads was similar and is thus not presented.

The effect of the number of lifespans per oid was tested using ten uniform workloads with varying average number of lifespans $\overline{NB}$ (from 10 to 1000). All used $|U|$ = 8000 different oids and the same number of queries (~115K). The other parameters are shown in Table 1 below.

Figure 5 shows the results for small to medium $\overline{NB}$ values. The query performance of atemporal

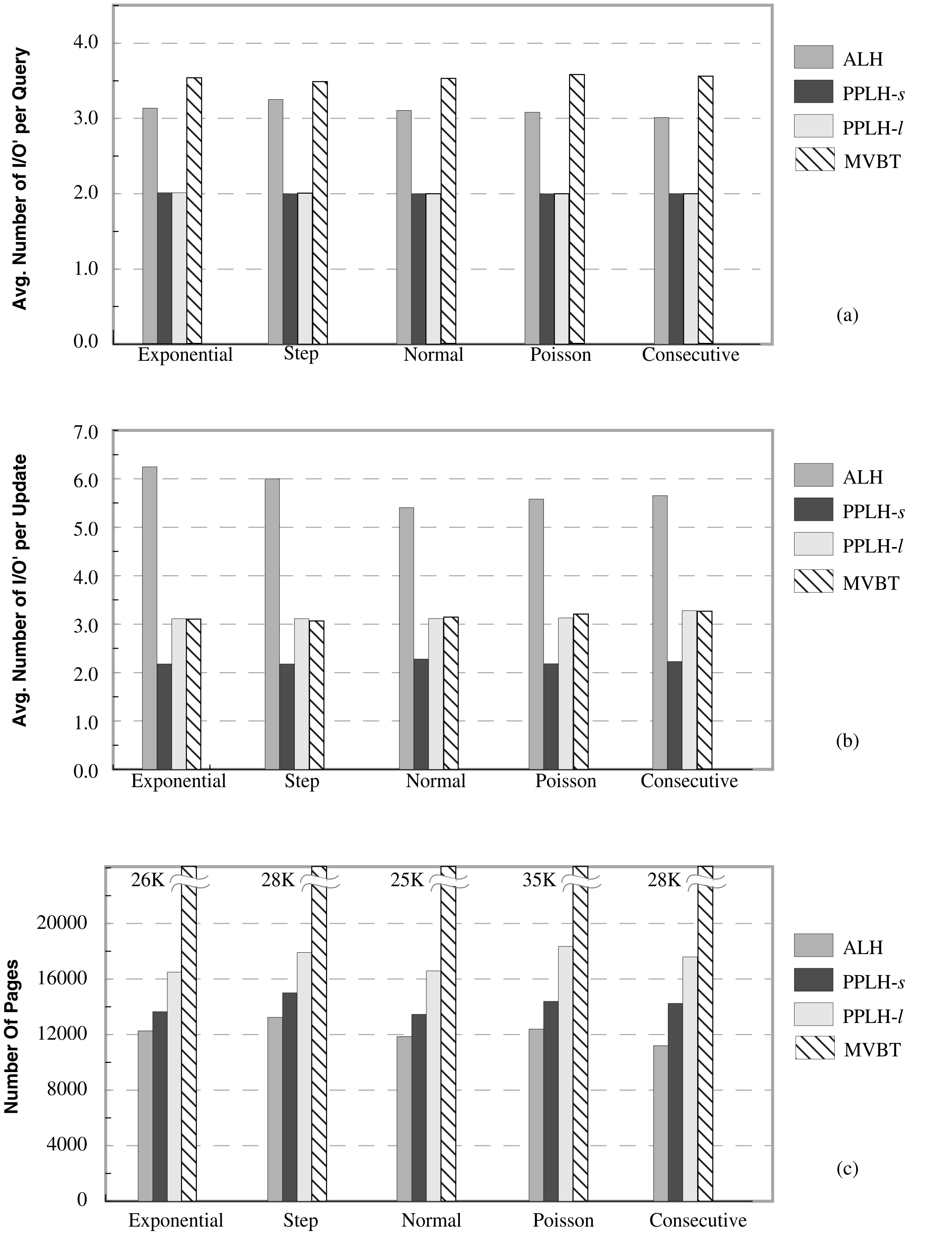


Figure 4: (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-*s*, PPLH-*l* and MVBT methods using the exponential, step, normal and poisson workloads with 8K oids, n ~ 0.5M and $\overline{NB}$ ~ 30.

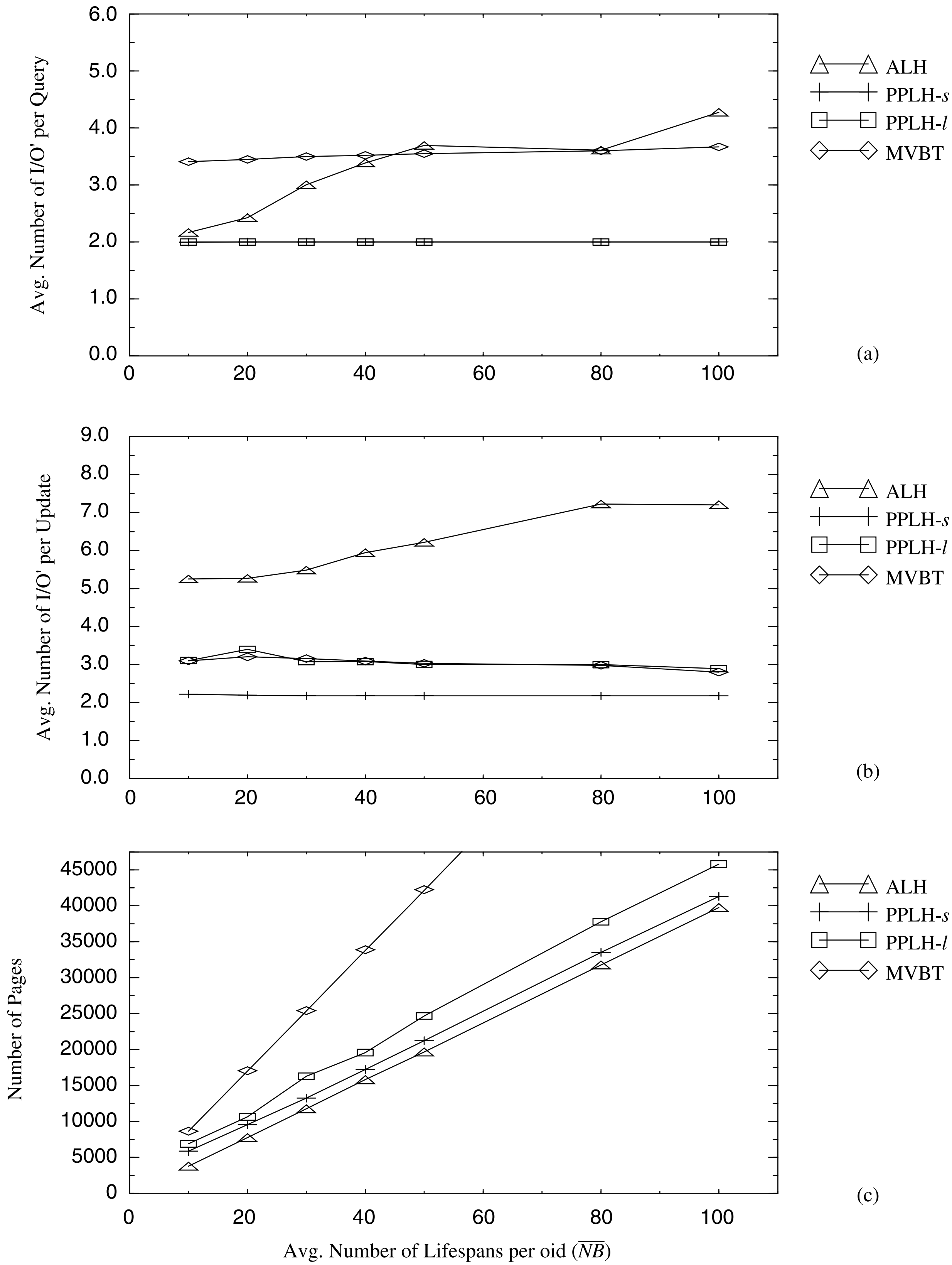


Figure 5: (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-*s*, PPLH-*l* and MVBT methods using various uniform workloads with varying $\overline{NB}$.

Table 1:

workload	*n*	*NB*	$\overline{NB}$
uniform-10	149801	75601	9.4
uniform-20	308091	155354	19.4
uniform-30	466854	237606	29.7
uniform-40	628173	316275	39.5
uniform-50	787461	396266	49.5
uniform-80	1264797	635604	79.5
uniform-100	1585949	796451	99.5
uniform-300	4787312	2393656	299.2
uniform-500	7996440	3998220	499.7
uniform-1000	15975955	7987977	998.4

hashing deteriorates as $\overline{NB}$ increases since buckets become larger (Figure 5.a). The PPLH-*s*, PPLH-*l* and MVBT methods have a query performance that is independent of $\overline{NB}$. PPLH-*s* outperforms all methods in update performance (Figure 5.b). The updating of PPLH-*s*, PPLH-*l* and MVBT is basically independent of $\overline{NB}$. Since increased $\overline{NB}$ implies larger bucket sizes, the updating of ALH increases. The space of all methods increases with $\overline{NB}$ as there are more changes per evolution (Table 1). The ALH has the least space, followed by the PPLH-*s*; the MVBT has the steeper space increase (for $\overline{NB}$ values 80 and 100, MVBT used 68K and 84.5K pages).

The methods performed similarly for high $\overline{NB}$ values namely 300, 500 and 1000 (see Appendix C). The query and update performance of PPLH-*s*, PPLH-*l* and MVBT remained basically unaffected by $\overline{NB}$. However, ATH was drastically affected as these workloads had many more occurrences per oid. The MVBT followed a steeper space increase than ATH, PPLH-*s* and PPLH-*l*.

The effect of the number of distinct oids per evolution was examined by considering four uniform workloads. The number of distinct oids $|U|$ was: 5000, 8000, 12000 and 16000, respectively. All workloads had similar average number of lifespans per distinct oid ($\overline{NB}$ ~ 30). The other parameters appear in Table 2. The results are depicted in Figure 6. The query performance of the hashing methods (PPLH-*s*, PPLH-*l* and ALH) is basically independent of $|U|$, with PPLH-*s* and PPLH-*l* having the lowest query I/O. In contrast, it increases for the MVBT. The increase is because more oids are stored in these tree structures, thus increasing the structure's height. Similar observations hold for the update performance (i.e., the hashing based methods have updating that is basically independent of $|U|$, while the updating of tree based methods tends to increase with $|U|$).

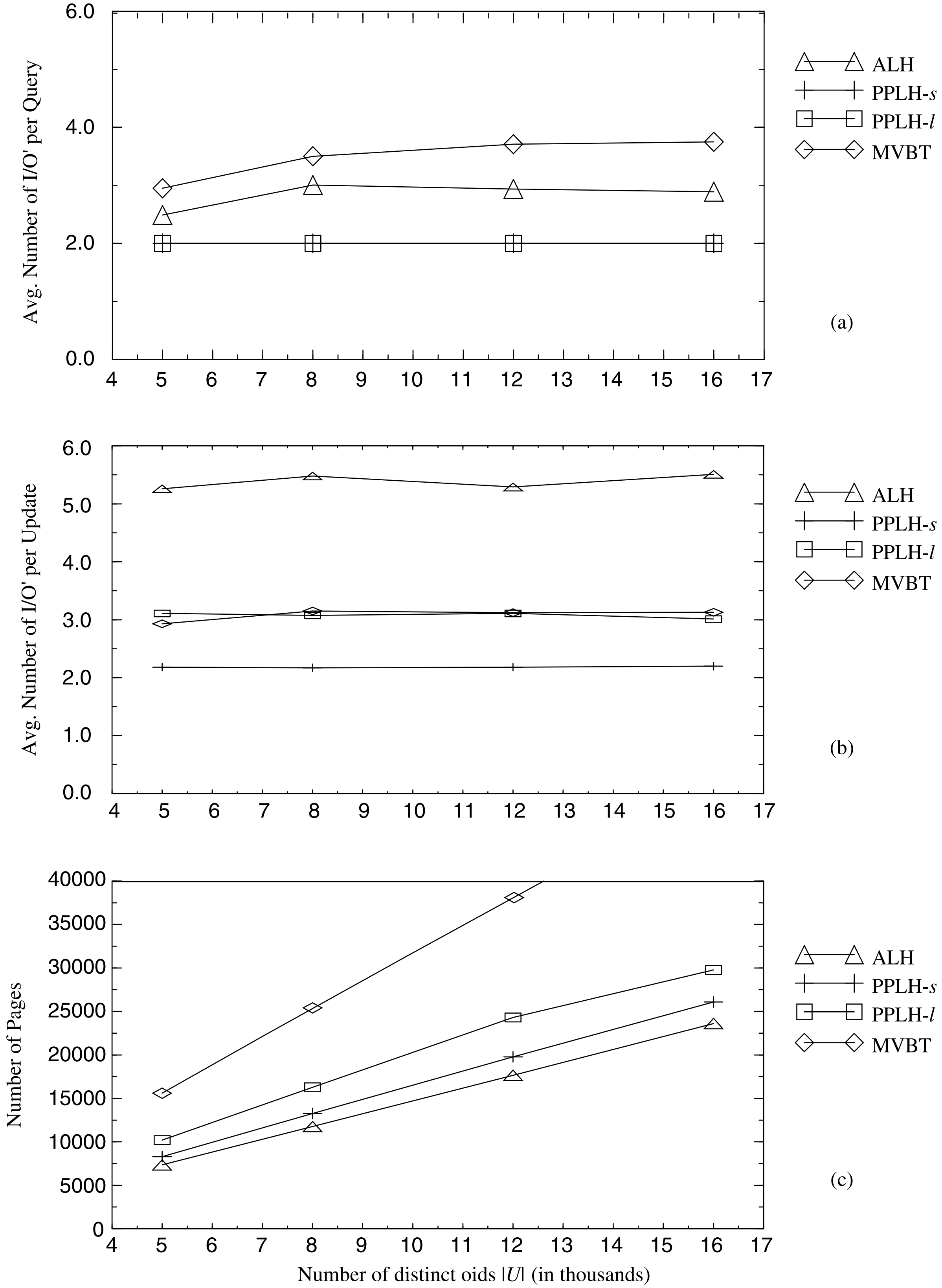


Figure 6: (a) Query, (b) Update, and, (c) Space performance for ALH, PPLH-*s*, PPLH-*l* and MVBT methods using various uniform workloads with varying |*U*|.

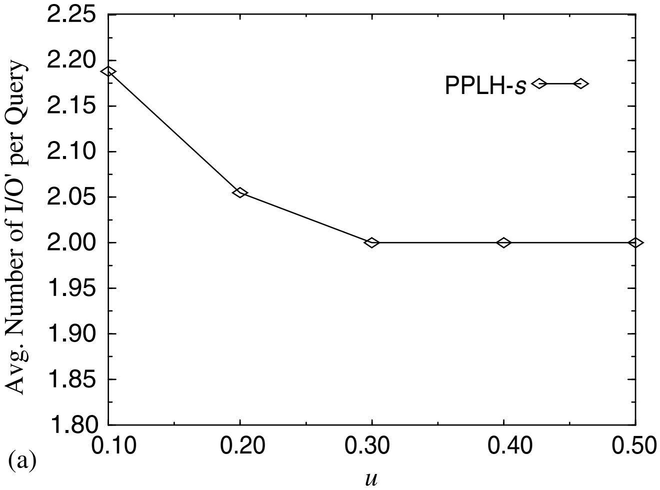
Finally, the space of all methods increases because n increases (Table 2).

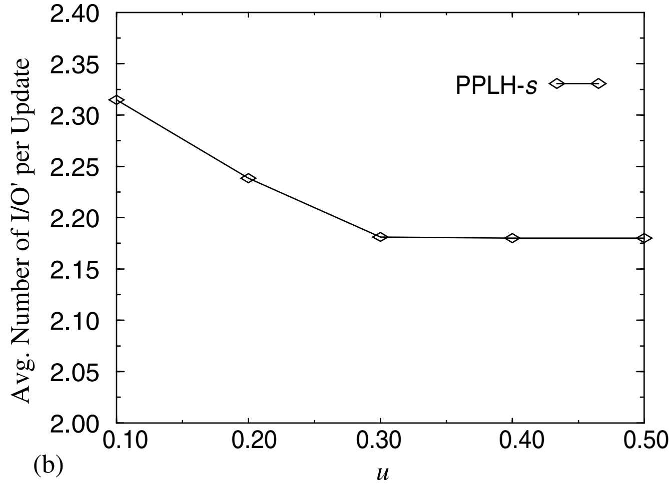
Table 2:

workload	n	NB	*#of queries*
uniform-5K	291404	146835	72417
uniform-8K	466854	237606	115878
uniform-12K	700766	353067	174167
uniform-16K	937443	472294	226456

From the above experiments, the PPLH-*s* method has the most competitive performance among all solutions. Its performance can be further optimized through usefulness parameter u. Figure 7 shows the results for the basic Uniform-30 workload ($|U| = 8000$, $n = 466854$, $NB = 237606$ and $\overline{NB} \sim 30$) but with different values of u. As expected, the best query performance occurs if u is greater than the maximum load of the observed ephemeral hashing. For these experiments the maximum load was 0.2. As asserted in Figure 7.a, the query time is minimized after $u = 0.3$. The update is similarly minimized (Figure 7.b) for u's above 0.2, since after that point, the alive oids are compactly kept into few pages that can be updated easier. Figure 7.c shows the space of PPLH-*s*. For u's below the maximum load the alive oids are distributed among more data pages, hence when such a page becomes non-useful it contains less alive oids and thus less copies are made, resulting in smaller space consumption. Using this optimization, the space of PPLH-*s* can be made similar to that of the ALH at the expense of some increase in query/update performance.

Finally, we compared the PPLH-*s* with partially persistent extendible hashing, a method that also uses the evolving-set approach but uncontrolled splits (PPEH-*s*). With uncontrolled splits a bucket b_j is split only when it overflows. Thus the "observed" ephemeral hashing tends to use less number of buckets but longer in size. As it turns out, this policy has serious consequences. First, for the Snapshot Index that keeps the history of bucket b_j the state $b_j(t)$ is a full page. This means that the Snapshot Index will distribute $b_j(t)$ into more physical pages in the bucket's history. Consequently, to reconstruct $b_j(t)$ more pages will be accessed. Second, the Snapshot Index will be copying larger alive states and thus occupy more space. We run the same set of experiments for the PPEH-*s*. In summary, PPEH-*s* used consistently larger query, update and space than PPLH-*s* (Appendix D presents some of our experimental results).





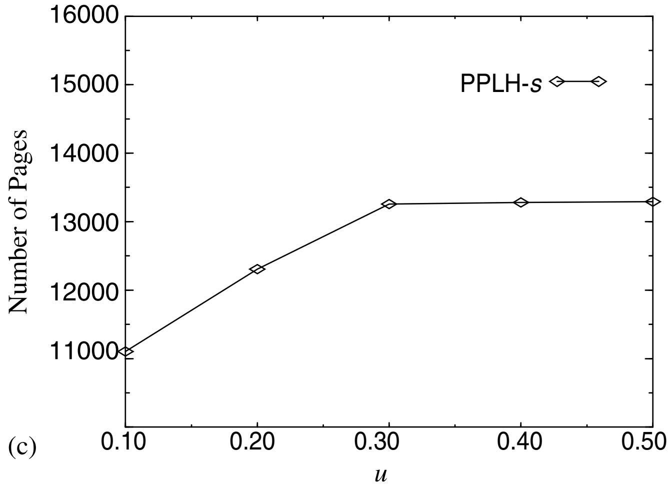


Figure 7: (a) Query, (b) Update, and, (c) Space performance for PPLH-*s* on a uniform workload with varying values of the usefulness parameter *u*.

5. Conclusions and Open Problems

We addressed the problem of *Temporal Hashing*, or equivalently, how to support temporal membership queries over a time-evolving set *S*. An efficient solution termed *partially persistent hashing* was presented. By hashing oids to various buckets over time, partially persistent hashing reduces the temporal hashing problem into reconstructing previous bucket states. The paper applies the methodology on linear hashing with controlled splits. Two flavors of partially persistent linear hashing were presented, one based on an evolving-set abstraction (PPLH-*s*) and one on an evolving-list (PPLH-*l*). They have similar query and comparable space performance but PPLH-*s* uses much less updating. Both methods were compared against straightforward approaches, namely: traditional (atemporal) linear hashing scheme, the Multiversion B-Tree and two R*-tree implementations. The experiments showed that PPLH-*s* has the most robust performance. Partially persistent hashing should be seen as an extension of traditional external dynamic hashing in a temporal environment. The methodology is general and can be applied to other ephemeral dynamic hashing schemes, like extendible hashing, etc.

An interesting extension is to find the history of a given oid that existed at time t. This query can be addressed if each version remembers the previous version of the same oid (through a pointer to that version). Note that this simple solution does not cluster together versions of the same oid, i.e., accessing a previous version may result to a distinct I/O. A better approach is to involve the C-lists of [VV97], however this is beyond the scope of this paper.

Traditionally hashing has been used to speed up join computations. We currently investigate the use of temporal hashing to speed up joins of the form: given two time-evolving sets X, Y, join $X(t_1)$ and $Y(t_2)$ (that is, find the objects common to both states). This temporal join is different than the valid-time natural join examined in [SSJ94, Z97]. There oids were matched as long as the lifespan intervals were intersecting (i.e., the join predicate did not include any specific temporal predicate). Also, our environment is based on transaction-time where changes are in time order. Note that one can easily manipulate the temporal hashing methods to answer interval membership queries ("find all objects that were in set S during interval I"). Thus, temporal hashing applies also in joins where the temporal predicate is generalized to an interval, i.e., find the objects that were both in set X during interval I_1 and in set Y during interval I_2. Of course one has to worry about handling artificial oid copies created by the Snapshot Index or the evolving-list approach. However, methodologies have been recently proposed to avoid such copies [BS96]. It would be an interesting problem to compare indexed temporal joins based on temporal hashing with temporal joins based on the MVBT, or MVAS indices. Finally, the discussion in this paper assumes temporal membership queries over a linear transaction-time evolution. It is interesting to investigate hashing in branched transaction environments [LST95].

Acknowledgments

We would like to thank the anonymous referees for their many careful comments. We also thank B. Seeger for providing the R* and MVBT code and R.M. Verma for the MVAS simulation code. Part of this work was performed while V.J. Tsotras was on a sabbatical visit to UCLA; we thus thank Carlo Zaniolo for his comments and hospitality.

References

[AS86] I. Ahn, R.T. Snodgrass, "Performance evaluation of a temporal database management system", *Proc. ACM SIGMOD Conf.,* pp.96-107, 1986.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, "An Asymptotically Optimal Multiversion B-tree", *Very Large Data Bases Journal*, Vol.5, No 4, pp 264-275, 1996.

[BKKS90] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, "The R*-tree: An efficient and Robust Access Method for Points and Rectangles", *Proc. ACM SIGMOD Conf.*, pp 322-331, 1990.

[BS96] J. Van den Bercken, B. Seeger, "Query Processing Techniques for Multiversion Access Methods", *Proc. of VLDB Conf.*, pp 168-179, 1996.

[C79] D. Comer, "The Ubiquitous B-Tree", *ACM Computing Surveys*, 11(2), pp121-137,1979.

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, The MIT Press and McGraw-Hill, 1990.

[DKM+88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnhert and R. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds", *Proc.* 29*th IEEE FOCS*, pp. 524-531, 1988.

[DSST89] J.R. Driscoll, N. Sarnak, D. Sleator, R.E. Tarjan, "Making Data Structures Persistent", *J. of Comp. and Syst. Sci.*, Vol 38, pp 86-124, 1989.

[ED88] R.J. Enbody, H.C. Du, "Dynamic Hashing Schemes", *ACM Computing Surveys*, Vol.20, No.2, pp. 85-113, 1988.

[EN00] R. Elmasri, S. Navathe, *Fundamentals of Database Systems*, Third Edition, Benjamin/ Cummings, 2000.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, H. Strong, "Extendible Hashing - a fast access method for dynamic files", *ACM Transactions on Database Systems*, Vol.4, No.3, pp. 315-344, 1979.

[FNSS92] A. Fiat, M. Naor, J.P. Schmidt, A. Siegel, "Nonoblivious Hashing", *JACM,* Vol 39, No 4, pp. 764-782, 1992.

[FZR98] M.J. Folk, B. Zoellick, G. Riccardi, *File Structures*, Addisson Wesley, Reading, MA, 1998.

[G84] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proc. ACM SIGMOD Conf.,* pp 47-57, 1984.

[J+94] C.S. Jensen, editor et. al., "A Consensus Glossary of Temporal Database Concepts", *ACM SIGMOD Record,* Vol. 23, No. 1, pp. 52-64, 1994.

[JS99] C.S. Jensen, R.T. Snodgrass, "Temporal Data Management", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 11, No. 1, pp 36-44, 1999.

[KT98] G. Kollios, V.J. Tsotras, "Hashing Methods for Temporal Data", Technical Report UCR-CS-98-01, Dept. of Computer Science, University of California-Riverside (http://www.cs.ucr.edu/publications/tech_reports/).

[KTF98] A. Kumar, V.J. Tsotras, C. Faloutsos, "Designing Access Methods for Bitemporal Databases", *IEEE Trans. on Knowledge and Data Engr.,* Vol. 10, No.1,pp 1-20,Jan/Feb1998.

[L78] P. Larson, "Dynamic Hashing", *BIT*, Vol.18, pp.184-201, 1978.

[L80] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", *Proc. of VLDB Conf.*, pp 212-223, 1980.

[L96] W. Litwin, M.A. Neimat, D. A. Schneider, "LH*-A Scalable, Distributed Data Structure", *ACM Trans. on Database Systems*, Vol. 21, No. 4, pp 480-525, 1996.

[LM92] T.Y.C Leung, R.R. Muntz, "Temporal Query Processing and Optimization in Multiprocessor Database Machines", *Proc. of VLDB Conf.*, pp. 383-394, 1992.

[LR96] M-L. Lo, C.V. Ravishankar, "Spatial Hash-Joins", *ACM SIGMOD Conf.*, pp 247-258, 1996.

[LS89] D. Lomet, B. Salzberg, "Access Methods for Multiversion Data", *Proc. ACM SIGMOD Conf.*, pp 315-324, 1989.

[LST95] G.M. Landau, J.P. Schmidt, V.J. Tsotras, "On Historical Queries Along Multiple Lines of Time Evolution", *Very Large Data Bases Journal*, Vol. 4, pp. 703-726, 1995.

[OS95] G. Ozsoyoglu, R.T. Snodgrass, "Temporal and Real-Time Databases: A Survey", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No. 4, pp 513-532, Aug. 1995.

[PJ99] D. Pfoser, C.S. Jensen, "Incremental Join of Time-Oriented Data", *Proc.* 11*th Intl. Conf. on Scientific and Statistical Database Management*, Cleveland, Ohio, July 28-30, 1999.

[R97] R. Ramakrishnan, *Database Management Systems*, 1st ed., McGraw-Hill, 1997.

[S88] B. Salzberg, *File structures*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[S99] B. Seeger, personal communication.

[SD90] D.A. Schneider, D.J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines", *Proc. of VLDB Conf.*, pp 469-480, 1990.

[SL95] B. Salzberg, D. Lomet, "Branched and Temporal Index Structures", College of Computer Science Technical Report, NU-CCS-95-17, Northeastern University.

[SSJ94] M.D. Soo, R.T. Snodgrass, C.S. Jensen, "Efficient Evaluation of the Valid-Time Natural Join", *Proc. IEEE Conf. on Data Engineering,* pp. 282-292, 1994.

[ST99] B. Salzberg, V.J. Tsotras, "A Comparison of Access Methods for Time-Evolving Data", *ACM Computing Surveys,* Vol.31, No.2, June 1999.

[TGH95] V. J. Tsotras, B. Gopinath, G.W. Hart, "Efficient Management of Time-Evolving Databases", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No. 4, pp 591-608, Aug.1995.

[TJS98] V.J. Tsotras, C.S. Jensen, R.T. Snodgrass, "An Extensible Notation for Spatiotemporal Index Queries", *ACM Sigmod Record*, pp 47-53, March 1998.

[TK95] V.J.Tsotras, N. Kangelaris, "The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries", *Information Systems, An International Journal*, Vol. 20, No.3, 1995.

[TW99] V.J.Tsotras, X. Sean Wang, "Temporal Databases", *Wiley Encyclopedia of Electrical and Electronics Engineering*, Vol. 21, pp 628-641, 1999.

[VV97] P.J. Varman, R.M. Verma, "An Efficient Multiversion Access Structure", *IEEE Trans. on Knowledge and Data Engineering*, Vol 9, No 3, pp 391-409, 1997.

[Z97] T. Zurek, "Optimisation of Partitioned Temporal Joins", Ph.D. Thesis, Dept. of Computer Science, University of Edinburgh, Nov. 1997.

Appendix A: Notation

Symbol	*Definition*	*Symbol*	*Definition*
g	upper threshold of linear hashing	B	page size in number of records (oids)
f	lower threshold of linear hashing	n	total number of changes in set S's evolution
l	load factor of linear hashing	s	maximum number of alive oids at any t
$LH(t)$	linear hashing scheme at time t	u	usefulness parameter
$i(t)$	round number of linear hashing at t	$lb_j(t)$	list of (useful) pages on bucket b_j at t
$b(t)$	sequence of buckets used at time t	FT_j	array with the first pages of bucket b_j
$\|b(t)\|$	number of buckets used at time t	$NT(X)$	next-page pointers array of page X
b_{total}	longest sequence of buckets used during the evolution of set S	B_r	page portion, available to store records (oids); $B_r < B$-1 (evolving list method)
n_j	# of changes recorded in bucket b_j	NB	total # of object additions in set S's evolution
$b_j(t)$	state of bucket b_j at time t	$\overline{NB}$	average # of lifespans per oid
$\|b_j(t)\|$	number of oids mapped to bucket b_j at t	$\|U\|$	# of distinct oids in set S's evolution

Appendix B: Update and Space Analysis of the Evolving-List approach

The *NT* arrays facilitate the efficient location of the next list page. However, if during the usefulness period of page A, its next page changes often, array *NT(A)* can become full. Assume this scenario happens at time t and let C be the page before page A. Page A is then *artificially* replaced in the list by page A'. The B_r part of A' is a copy of the B_r part of page A (A' is thus useful) but $NT(A')$ is empty. A new entry is added in *NT(C)* with the pid of A' and the first entry of $NT(A')$ stores the pid of the page (if any) that was after page A at t. If all list pages until page A had their *NT* arrays full, the above process of artificially turning useful pages to non-useful can propagate all the way to the beginning of the list. If the first list page is replaced, array FT_j is updated. However, this does not happen often. Figure 8 shows an example of how arrays *NT*() and FT_j are maintained. From each page only the *NT* array is shown. In this example B-B_r = 4 entries. At time t_6 array *NT(A)* fills up and an artificial copy of page A is created with array *NT(A')*. Array *NT(C)* is also updated about the artificially created new page.

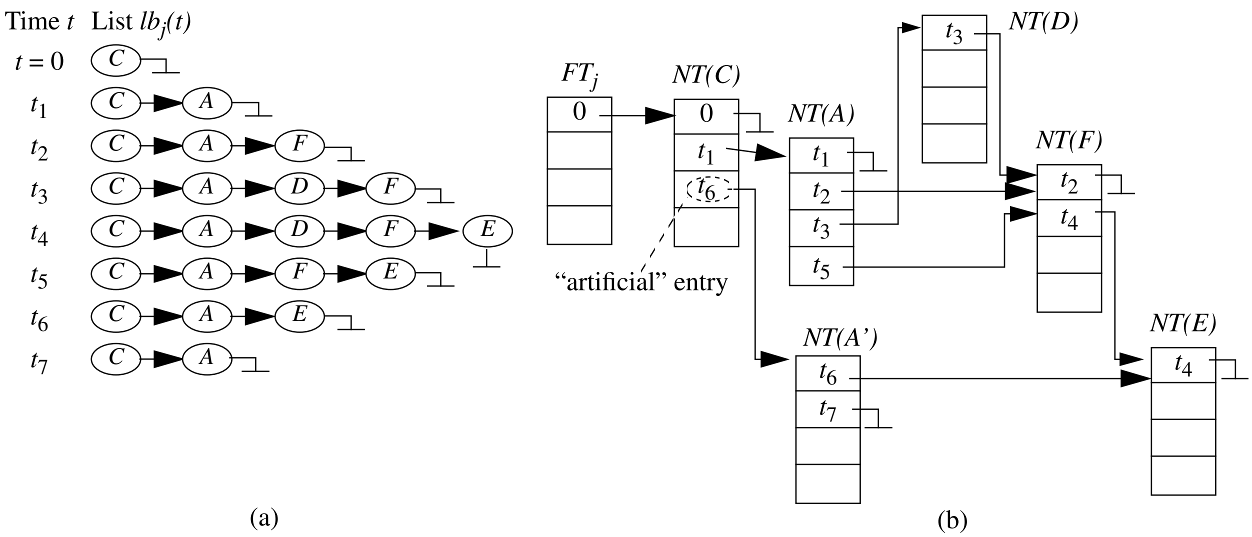


Figure 8: (a) An example evolution for the useful pages of list $lb_j(t)$. (b) The corresponding FT_j and *NT* arrays.

The artificial creation of page A' provides faster query processing. If instead of creating page A', *NT(A)* is allowed to grow over the B-B_r area of page A (using additional pages), the last entry on *NT(C)* still points to page A. Locating which is the next page after C at time t would lead to page A but then a serial search among the pages of array *NT(A)* is needed. Clearly this approach is inefficient if the page in front of page A changes many times. Using the artificial pages the next useful list page for any time of interest is found by one I/O! This technique is a generalization of the *backward updating* technique used in [TGH95].

In the evolving-list approach copying occurs when a page turns non-useful or when a page is artificially replaced. We first consider the evolving-list without the artificial page replacements and show that the copying due to page usefulness is linear to the total number of real oid changes. We use an amortization argument that counts the number of real changes that occur to a page before this page becomes non-useful.

A useful page can become non-useful only after it has acquired B_r records. A page that is not the last list page will turn non-useful if an oid deletion brought the number of alive oids in this page below uB_r. The last list page can turn non-useful only if it does not have uB_r alive records when it acquires B_r records. However a page never overflows; if it already has B_r records no new records can be added to this page.

In this environment a page is appended in the end of the list in two cases: either a real oid insertion occurs and its record cannot fit in the previous last page, or, a list page turned non-useful and some of its (copied) alive records cannot fit in the previous last page. At best, filling a page with real oid insertions accommodates at least B_r real changes for this page. However, at worst, a new page can be filled only with copies of alive records (from various other pages that turned non-useful). Copies do not correspond to real oid insertions but this page will need at least B_r-uB_r (which is $O(B_r)$) real oid deletions before it becomes non-useful. Hence the space used by this copying is $O(n_j/B)$ where n_j is the total number of real changes recorded in the evolution of bucket b_j.

Copying can also happen due to artificial page replacements. Note that a page is replaced only when its *NT* array becomes full. This implies that this page was not the last page in the list and hence its B_r part is already full. It can be easily verified that the extra space used by the *backward updating* technique [TGH95] is also $O(n_j/B)$. Note that the first replacement page is created after B-B_r (which is $O(B)$) pages in front of it became non-useful. Updating in the evolving-list approach is $O(|b_j(t)|/B)$ since the whole current list must be searched until a new oid is added or an existing oid is updated as deleted.

Appendix C: Performance comparison for high $\overline{NB}$ values.

	I/Os per Query				I/Os per Update				Space (thousands of pages)			
	PPLH-*s*	PPLH-*l*	MVBT	ATH	PPLH-*s*	PPLH-*l*	MVBT	ATH	PPLH-*s*	PPLH-*l*	MVBT	ATH
Uniform-300	2.19	2.21	3.69	6.5	2.20	2.84	2.79	12.2	118.5	118.9	345.4	118.2
Uniform-500	2.28	2.34	3.70	11.1	2.21	2.84	2.80	15.4	197.1	198.1	512.6	195.1
Uniform-1000	2.41	2.58	3.82	21.3	2.20	2.84	2.80	28.4	389.2	392.2	1145.5	387.2

Appendix D: PPLH-*s* (controlled splits) versus PPEH-*s* (uncontrolled splits)

	I/Os per Query		I/Os per Update		Space (pages)	
	PPLH-*s*	PPEH-*s*	PPLH-*s*	PPEH-*s*	PPLH-*s*	PPEH-*s*
Uniform-30	2	3.22	2.18	2.53	13256	17035
Exponential-30	2	2.8	2.18	2.43	13646	15985
Step-30	2	2.8	2.18	2.43	15003	16688
Normal-30	2	3.18	2.28	2.52	13462	16982
Poisson-30	2	2	2.18	2.23	14394	16760
Consecutive-30	2	2.41	2.23	2.60	14246	17636