Hashing Methods for Temporal Data

George Kollios and Vassilis J. Tsotras

February 12, 1998

TR-24

A TIMECENTER Technical Report

Title	Hashing Methods for Temporal Data
Author(s)	George Kollios and Vassilis J. Tsotras
Publication History	

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Dieter Pfoser
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta
Sudha Ram

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, University of California, Riverside, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract:

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 *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 the set *S* at time *t*". We present an efficient solution to the Temporal Hashing problem. Our solution, also termed *partially persistent hashing*, behaves as if a separate, ephemeral (i.e., non-temporal) dynamic hashing scheme is available on every state assumed by set *S* over time. However if the buckets of these hashing schemes were to be stored for each time of interest, the space would become prohibitively large (quadratic on the total number of changes in set *S*'s evolution); instead, our method uses linear space. We compare partially persistent hashing with various straightforward approaches (like the traditional linear hashing scheme, the R-tree and the Multiversion B-Tree) and 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 considers linear hashing, the methodology applies to other dynamic hashing schemes as well.

1. Introduction

Hashing has been used as a fast method to address membership queries. Given a set *S* of objects distinguished by some identity attribute (*oid*), a membership query asks whether object with oid *k* is in set *S*. 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* [EN94, R97] and a hashing function maps oids to *buckets*. For every object of *S*, the hashing function computes the bucket number 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. When more than *B* oids are mapped on the same bucket a (bucket) *overflow* occurs. Overflows are dealt in various ways, including rehashing (try to find another bucket using another hashing scheme) and/or chaining (create a chain of pages under the overflown bucket).

If no overflows are present, finding whether a given oid is in the hashed set is trivial: simply compute the hashing function for the queried oid and visit the appropriate bucket. If the object is in the set it should be in that bucket. Hence, if the hashing scheme is *perfect*, membership queries are answered in $O(1)$ steps (just one I/O to access the page of the bucket). Overflows however complicate the situation. If data is not known in advance, the worst case query performance of hashing is large. It is linear to the size of set *S* since all oids could be mapped to the same bucket

G. Kollios is with the Dept. of Computer & Information Science, Polytechnic University, Brooklyn, NY 11201; gkollios@paros.poly.edu. V. J. Tsotras is with the Dept. of Computer Science, University of California, Riverside, CA 92521; tsotras@cs.ucr.edu.

This research was partially supported by NSF grant IRI-9509527 and by the New York State Science and Technology Foundation as part of its Center for Advanced Technology program.

if a bad hashing scheme is used. Nevertheless, practice has shown that in the absence of pathological data, good hashing schemes with few overflows and constant average case query performance (usually each bucket has size of one or two pages) exist. This is one of the major differences between hashing and index schemes. If a balanced tree (B+ tree [C79]) is used instead, answering a membership query takes logarithmic (on the size of *S*) time in the worst case. For many applications (for example in join computations [SD90]), a hashing scheme that provides expected constant query performance (one or two I/O's) is preferable to the worst case but logarithmic query performance (four or more I/O's if *S* is large) of balanced search trees.

Static hashing refers to schemes that use a predefined set of buckets. This is inefficient if the set *S* is allowed to change (by adding or deleting objects from the set). If the set is too small and the number of pre-allocated buckets too large, the scheme is using more space than needed. If the set becomes too large but a small number of buckets is used then overflows will become more often, deteriorating the scheme's performance. What is needed is a *dynamic* hashing scheme which has the property of allocating space proportional to the size of the hashed set *S*. Various external dynamic hashing schemes have been proposed, among which *linear hashing* [L80] (or a variation) appears to be commonly used.

Note that even if the set *S* evolves, traditional dynamic hashing is *ephemeral*, i.e., it answers membership queries on the most current state of set *S*. In this paper we address a more general problem. We assume that changes to the set *S* are timestamped by the time instant when they occurred and we are interested in answering membership queries for any state that set *S* possessed. Let *S(t)* denote the state (collection of objects) set *S* had at time *t*. Then the membership query has a temporal predicate as in: "given oid *k* and time *t* find whether *k* was in *S(t)*". We term this problem as *Temporal Hashing* and the new query as *temporal membership* query.

Motivation for the temporal hashing problem stems from applications where current as well as past data is of interest. Examples include: accounting, billing, marketing, tax-related, social/ medical, and financial/stock-market applications. Such applications cannot be efficiently maintained by conventional databases which work in terms of a single (usually the most current) logical state. Instead, *temporal* databases were proposed [SA85] for time varying data. Two time dimensions have been used to model reality, namely *valid-time* and *transaction-time* [J+94]. Valid time denotes the time when a fact is valid in reality. Transaction time is the time when a fact is stored in the database. Transaction time is consistent with the serialization order of transactions (i.e., it is monotonically increasing) and can be implemented using the commit times of

transactions [S94]. In the rest, the terms time or temporal refer to transaction-time.

Assume that for every time *t* when *S(t)* changes (by adding/deleting objects) we could have a good ephemeral dynamic hashing scheme (say linear hashing) *h(t)* that maps efficiently (with few overflows) the oids in *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*. To answer a temporal membership query for oid *k* and time *t* we only need to apply *h(t)* on *k* and access the appropriate bucket of *b(t)*. This would provide an excellent query performance as it takes advantage of the good linear hashing scheme *h(t)* used for each *t*, but the space requirements are prohibitively large! If *n* denotes the number of changes in *S*'s evolution, flashing each *b(t)* on the disk could easily create $O(n^2)$ space.

Instead we propose a more efficient solution that has similar query performance as above but uses space linear to *n*. We term our solution *partially persistent hashing* as it reduces the original problem into a collection of partially persistent[1] sub-problems. We apply two approaches to solve 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 whose history is efficiently kept. In both cases, the partially persistent hashing scheme "observes" and stores the evolution of the ephemeral hashing in an efficient way that enables fast access to any *h(t)* and *b(t)*. (We note that partial persistence fits nicely with a transaction-time database environment because of the always increasing characteristic of transaction-time.)

We compare partially persistent hashing with three other approaches. The first one uses a traditional dynamic hashing function to map all oids ever created during the evolution of *S(t)*. This solution does not distinguish among the many copies of the same oid *k* that may have been created as time proceeds. A given oid *k* can be added and deleted from *S* many times, creating copies of *k* each associated with a different time interval. Because all such copies will be hashed on the same bucket, bucket reorganizations will not solve the problem (this was also observed in [AS86]). These overflows will eventually deteriorate performance especially as the number of copies increases. The second approach sees each oid-interval combination as a multidimensional object and uses an R-tree to store it. The third approach assumes that a B+ tree is used to index each *S(t)* and makes this B+ tree partially persistent [BGO+96, VV97, LS89]. Our experiments show that

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.

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. It corresponds to extending an ephemeral B+ tree in a temporal environment. Like the ephemeral B+ tree, it supports worst case logarithmic query time but for temporal queries. It was an open problem, whether such an efficient temporal extension existed for hashing schemes. The work presented here answers this question positively. As with a non-temporal environment, partially persistent hashing provides a faster than indexing, (expected) query performance for temporal membership queries. This result reasserts our conjecture [KTF98] that temporal problems that support transaction-time can be solved by taking an efficient solution for the corresponding non-temporal problem and making it partially persistent.

The rest of the paper is organized as follows: section 2 presents background and previous work as related to the temporal index methods that are of interest here; section 3 describes the basics of the Snapshot Index and Linear Hashing. The description of partially persistent hashing appears in section 4. Performance comparisons are presented in section 5, while conclusions and open problems for further research appear in section 6.

2. Background and Previous Work

Research in temporal databases has shown an immense growth in recent years [OS95]. Work on temporal access methods has concentrated on indexing. A worst case comparison of temporal indexes appears in [ST97]. To the best of our knowledge, no approach addresses the hashing problem in a temporal environment. Among existing temporal indexes, four are of special interest for this paper, namely: the Snapshot Index [TK95], the Time-Split B-tree (TSB) [LS89], the Multiversion B-Tree (MVBT) [BGO+96] and the Multiversion Access Structure (MVAS) [VV97].

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*. As time proceeds, objects can be added to or deleted from this set. When an object is added to *S* and until (if ever) is deleted from *S*, it is called “alive”. This is represented by associating with each object a semi-closed interval, or *lifespan*, of the form: [*start_time*, *end_time*). While an object is alive it cannot be re-added in *S*, i.e. *S* contains no duplicates. Deletions can be applied to alive objects. When an object is added at *t*, its start_time is *t* but its end_time is yet unknown. Thus its lifespan interval is initiated as [*t*, *now)*, where *now* is a variable representing the always increasing current

time. If this object is later deleted from *S*, its end_time is updated from *now* to the object's deletion time. Since an object can be added and deleted many times, objects with the same oid may exist but with non-intersecting lifespan intervals (i.e., such objects were alive at different times). The state of the set at a given time *t*, namely *S(t)*, is the collection of all alive objects at time *t*.

Assume that this evolution is stored in a transaction-time database, in a way that when a change happens at time *t*, a transaction with the same timestamp *t* updates the database. There are various queries we may ask on such a temporal database. A common query, is the *pure-snapshot* problem (also denoted as "*//-/*S*" in the proposed notation of [TJS98]): "given time *t* find *S(t)*". Another common query is the *range-snapshot* problem ("*R*//-/*S*"): "given time *t* and range of oids *r*, find all alive objects in *S(t)* with oids in range *r*".

[ST97] categorizes temporal indexes according to what queries they can answer efficiently and compares their performance using three costs: space, query time and update time (i.e., 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 (if no range is provided). However, as indicated in [TGH95], a method designed to address primarily the pure-snapshot query does not need to order incoming changes according to oid. Note that in our evolution model, changes arrive in increasing time order but are unordered on oid. Hence such method could enjoy faster update time than a method designed for the range-snapshot query. The latter orders incoming changes on oid so as to provide fast response to range-snapshot queries. Indeed, the Snapshot Index 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 employed). This is the I/O-optimal solution for the pure snapshot query. Here, *a* corresponds to the number of alive objects in the queried state *S(t)*.

For the range-snapshot query three efficient methods exist, namely, the TSB tree, the MVBT tree and the MVAS structure. They all assume that there exists a B+ tree indexing each *S(t)*; as time proceeds and set *S* evolves the corresponding B+ tree evolves, too. They differ on the algorithms provided to efficiently store and access the B+ tree evolution. Answering a range-snapshot query about time *t* implies accessing the B+ tree as it was at time *t* and search through its nodes to find the oids in the range of interest. Conceptually, these approaches take a B+ tree and make it partially persistent [DSST89]. The resulting structure has the form of a graph as it includes the whole history of the evolving B+ tree, but it is able to efficiently access any past state of this B+ tree.

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]). This is the I/O-optimal solution for the range-snapshot query. Here *m* denotes the number of "alive" objects when an update takes place and *a* denotes the answer size to the range-snapshot query, i.e., how many objects from the queried *S(t)* have oids in the query range *r*. The MVAS structure improves the merge/split policies of the MVBT thus resulting to 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. Since for the purposes of this paper we assume that object deletions are frequent we use the MVBT instead of a TSB.

3. Basics of the Snapshot Index and Linear Hashing

For the purposes of partially persistent hashing we need the fundamentals from the Snapshot Index and ephemeral Linear Hashing, which are described next. For detailed descriptions we refer to [TK95] and [L80, S88, EN94, R97], respectively.

3.1 The Snapshot Index

This method [TK95] solves the pure-snapshot problem using three basic structures: a balanced tree (*time-tree*) that indexes data pages by time, a pointer structure (*access-forest*) among the data pages and a hashing scheme. The time-tree and the access-forest enable fast query response while the hashing scheme is used for update purposes.

We first discuss updates. Objects are stored sequentially in data pages in the same order as they are added to the set *S*. In particular, when a new object with oid *k* is added to the set at time *t*, a new record of the form <*k*, [*t*, *now*)> is created and is appended in a data page. When this data page becomes full, a new data page is used and so on. At any given instant there is only one data page that stores (accepts) records, the *acceptor* (data) page. The time when an acceptor page was created (along with the page address) is stored in the *time* tree. As acceptor pages are created sequentially the time-tree is easily maintained (amortized *O*(1) I/O to index each new acceptor page). For object additions, the sequence of all data pages resembles a regular log but with two main differences: (1) on the way deletion updates are managed and (2) on the use of additional links (pointers) among the data pages that create the access-forest.

Object deletions are not added sequentially; rather they are in-place updates. When object *k* is deleted at some time t', its record is first located and then updated from <*k*, [*t, now)*> to <*k*, [*t, t′)*>. Object records are found using their oids through the hashing scheme. When an object is added in *S*, its oid and the address of the page that stores the object's record are inserted in the hashing scheme. If this object is deleted the hashing scheme is consulted, the object's record is located and its interval is updated. Then this object's oid is removed from the hashing function.

Storing only one record for each object suggests that for some time instant *t* the records of the objects in *S(t)* may be dispersed in various data pages. Accessing all pages with alive objects at *t*, would require too much I/O (if *S(t)* has *a* objects, we may access *O(a)* pages). Hence the records of alive objects must be clustered together (ideally in *a/B* pages). To achieve good clustering we introduce copying but in a "controlled" manner, i.e., in a way that the total space remains $O(n/B)$. To explain the copying procedure we need to introduce the concept of page *usefulness*.

Consider a page after it gets full of records (i.e., after it stops being the acceptor page) and the number of "alive" records it contains (records with intervals ending to *now*). For all time instants that this page contains *uB* alive records ($0 < u \leq 1$) is called *useful*. This is because for these times *t* the page contains a good part of the answer for *S(t)*. If for a pure-snapshot query about time *t* we are able to locate the useful pages at that time, each such page will contribute at least *uB* objects to the answer. The *usefulness* parameter *u* is a constant that tunes the behavior of the Snapshot Index.

Acceptor pages are special. While a page is the acceptor page it may contain fewer than *uB* alive records. By definition we also call a page useful for as long as it is the acceptor page. Such a page may not give enough answer to justify accessing it but we still have to access it! Nevertheless, for each time instant there exists exactly <u>one</u> acceptor page.

Let [*u.start_time, u.end_time)* denote a page's usefulness period; u.start_time is the time the page started being the acceptor page. When the page gets full it either continues to be useful (and for as long as the page has at least *uB* alive records) or it becomes non-useful (if at the time it became full the page had less than *uB* alive records). The next step is to cluster the alive records for each *t* among the useful pages at *t*. When a page becomes non-useful, an *artificial copy* occurs that copies the alive records of this page to the current acceptor page (as in a timesplit [E86, LS89]). The non-useful page behaves as if all its objects are marked as deleted but copies of its alive records can still be found from the acceptor page. Copies of the same record contain subsequent non-overlapping intervals of the object's lifespan. The copying procedure reduces the original problem

of finding the alive objects at *t* into finding the useful pages at *t*. The solution of the reduced problem is facilitated through the access-forest.

The access-forest is a pointer structure that creates a logical "forest of trees" among the data pages. Each new acceptor page is appended at the end of a doubly-linked list and remains in the list for as long as it is *useful*. When a data page *d* becomes non-useful: (a) it is removed from the list and (b) it becomes the next child page under the page *c* preceding it in the list (i.e., *c* was the left sibling of *d* in the list when *d* became non-useful). As time proceeds, this process will create trees of non-useful data pages rooted under the useful data pages of the list. The access-forest has a number of properties that enable fast searching for the useful pages at any time. [TK95] showed that starting from the acceptor page at *t* all useful pages at *t* can be found in at most twice as many I/O's (in practice much less I/O's are needed). To find the acceptor page at *t* the balanced time-tree is searched (which corresponds to the logarithmic part of the query time). In practice this search is very fast as the height of the balanced tree is small (it stores only one entry per acceptor page which is clearly *O(n/B)*). The main part of the query time is finding the useful pages. The performance of the Snapshot Index can be fine tuned by changing parameter *u*. Large *u* implies that acceptor pages become non-useful faster, thus more copies are created which increases the space but also clusters the answer into smaller number of pages, i.e., less query I/O.

3.2 Linear Hashing

Linear Hashing (LH) is a dynamic hashing scheme that adjusts gracefully to data inserts and deletes. The scheme uses a collection of buckets that grows or shrinks one bucket at a time. Overflows are handled by creating a chain of pages under the overflown bucket. The hashing function changes dynamically and at any given instant there can be at most two hashing functions used by the scheme.

More specifically, let *U* be the universe of oids and h_0: *U* -> {0,...,*M*-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 the first overflow happens. When this first overflow occurs (it can occur 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 $h_1(oid)$. The next overflow will attach a new bucket *M*+1 and the contents of bucket 1 will be distributed using h_1 between buckets 1 and *M*+1. A crucial

property of h_1 is that any oids that were originally mapped by h_0 to bucket j $(0 \le j \le M-1)$ should be remapped either to bucket j or to bucket $j+M$. This is a necessary property for linear hashing to work. An example of such hashing function is: $h_1(oid) = oid \bmod 2M$.

Further overflows will cause additional buckets to split in a *linear* bucket-number order. A variable p indicates which is the bucket to be split next. Conceptually the value of p denotes which of the two hashing functions that may be enabled at any given time, applies to what buckets. Initially p=0, which means that only one hashing function (h_0) is used and applies to all buckets in the LH file. After the first overflow in the above example, p=1 and h_1 is introduced. Suppose that an object with oid k is inserted after the second overflow (i.e., when p=2). First the older hashing function (h_0) is applied on k. If $h_0(k) \ge p$ then the bucket $h_0(k)$ has not been split yet and k is stored in that bucket. Otherwise $(h_0(k) < p)$ the bucket provided by h_0 has already been split and the newer hashing function (h_1) is used; oid k is stored in bucket $h_1(k)$. Searching for an oid is similar, that is, both hashing functions may be involved.

After enough overflows, all original M buckets will be split. This marks the end of *splitting-round* 0. During round 0, p went subsequently from bucket 0 to bucket M-1. At the end of round 0 the LH file has a total of $2M$ buckets. Hashing function h_0 is no longer needed as all $2M$ buckets can be addressed by hashing function h_1 (note: h_1: U -> {0,...,$2M$-1}). Variable p is reset to 0 and a *new* round, namely splitting-round 1, is started. The next overflow (in any of the $2M$ buckets) will introduce hashing function $h_2(oid) = oid \bmod 2^2M$. This round will last until bucket $2M$-1 is split. In general, round i starts with p = 0, buckets {0,...,2^iM-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-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 given time, the linear hashing scheme is completely identified by the round number and variable p. Given round i and variable p, searching for oid k is performed using h_i if $h_i(k) \ge p$; otherwise h_{i+1} is used. During round i the value of p is increased by one at each overflow; when p=$2^i M$ the next round i+1 starts and p is reset to 0.

A split performed whenever an overflow occurs is an *uncontrolled* split. Let l denote the LH file's *load factor*, i.e., $l = |S| / BR$ where $|S|$ is the current number of oids in the LH file (size of set S), B is the page size (in number of oids) and R the current number of buckets in the 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, to achieve a higher storage utilization a split is instead performed when an overflow occurs and the load factor is above some *upper threshold g*. This is a *controlled* split and can typically achieve 95% utilization. Deletions in set *S* will cause the LH file to *shrink*. Buckets that have been split can be recombined if the load factor falls below some *lower threshold f*($f \leq g$). Then two buckets are merged together; this operation is the reverse of splitting and occurs in reverse linear order. Practical values for *f* and *g* are 0.7 and 0.9, respectively.

4. Partially Persistent Hashing

We first describe the evolving-set approach which is based on the Snapshot Index; the evolving-list approach will follow.

4.1 The Evolving-Set Approach

Using partial persistence, the temporal hashing problem will be 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 3) is used to map the objects of *S(t)*. As *S(t)* evolves with time the hashing scheme is a function of time, too. Let *LH(t)* denote the linear hashing file as it is at time *t*. There are two basic time-dependent parameters that identify *LH(t)* for each *t*, namely *i(t)* and *p(t)*. Parameter *i(t)* is the round number at time *t*. The value of parameter *p(t)* identifies the next bucket to be split.

An interesting property of linear hashing is that buckets are reused; when round *i*+1 starts it has double the number of buckets of round *i* but the first half of the bucket sequence is the same since new buckets are appended in the end of the file. Let b_{total} denote the longest sequence of buckets ever used during the evolution of *S(t)* and assume that b_{total} consists of buckets: 0,1,2,..., $2^q M - 1$. Let *b(t)* be the sequence of buckets used at time *t*. The above observation implies that for all *t*, *b(t)* is a prefix of b_{total}. In addition $i(t) \leq q, \forall t$.

Consider bucket b_j from the sequence b_{total} $(0 \leq j \leq 2^q M - 1)$ and the observe the collection of objects that are stored in this bucket as time proceeds. The state of bucket b_j at time *t*, namely $b_j(t)$, is the set of oids stored to this bucket at *t*. Let $|b_j(t)|$ denote the number of oids in $b_j(t)$. If all states $b_j(t)$ can somehow be reconstructed for each bucket b_j, answering a temporal membership query for oid *k* at time *t* can be answered in two steps:

(1) find which bucket b_j, oid *k* would have been mapped by the hashing scheme at *t*, and,

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

The first step requires identifying which hashing scheme was used at time *t*. The evolution of the hashing scheme *LH(t)* 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 the values of *i(t)* and/or *p(t)* change. Given any *t,* the hashing function used at *t* is identified by simply locating *t* inside the time-ordered *H* in a logarithmic search.

The second step implies accessing $b_j(t)$. The obvious way would be to store each $b_j(t)$, for those times that $b_j(t)$ changed. As explained earlier this would easily create quadratic space requirements. The updating per change would also suffer since the I/O to store the current state of b_j would be proportional to the bucket's current size, namely $O(|b_j(t)|/B)$.

By observing the evolution of bucket b_j we note that its state changes as an *evolving set* by adding or deleting oids. Each such change can be timestamped with the time instant 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 a rehashing occurred at some 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 various buckets. At time $t = 21$ the addition of oid 8 on bucket 3 causes the first overflow which rehashes the contents of bucket 0 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 instant $t = 21$.

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 to store $b_j(t)$. (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.) What is therefore needed is a method which for any given *t* can reconstruct $b_j(t)$ with effort proportional to $|b_j(t)|/B$ I/O's. 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 the required efficiency.

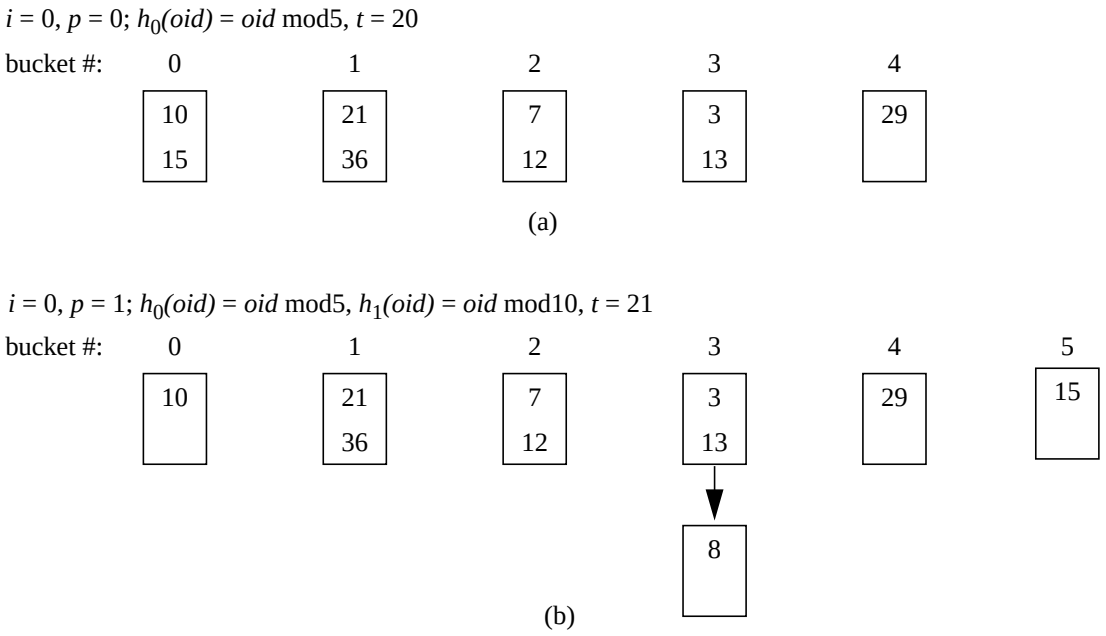

Figure 1: Two instants in the evolution of an ephemeral hashing scheme. (a) Until time t = 20, no split has occurred and p = 0. (b) At t = 21, oid 8 is mapped to bucket 3 and causes an overflow. Bucket 0 is rehashed using h_1 and p = 1.

We can thus conclude that given an evolving set *S*, partially persistent hashing answers a temporal membership query about oid *k* at time *t*, with almost the same query time efficiency (plus a small overhead) as if a separate ephemeral hashing scheme existed on each *S(t)*. A good ephemeral hashing scheme for *S(t)* would require an expected $O(1)$ I/O's to answer a membership query. 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 pages (in practice one or two pages). In perspective, partially persistent hashing will reconstruct $b_j(t)$ in $O(|b_j(t)|/B)$ I/O's, which from the above is expected $O(1)$.

The small overhead incurred by persistent hashing is due to the fact that it stores the whole history of *S*'s evolution and not just a single state *S(t)*. Array *H* stores an entry every time a page overflow occurs. Even if all changes are new oid additions, the number of overflows is upper bounded by *O(n/B)*. Hence array *H* indexes at most $O(n/B^2)$ pages and searching it takes $O(\log_B(n/B^2))$ I/O's.

Having identified the hashing function the appropriate bucket b_j is pinpointed. Then time *t* must be searched in the time-tree associated with this bucket. The overhead implied by this search is bounded by $O(\log_B(n_j/B))$ where n_j corresponds to the number of changes recorded in bucket

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 (a "+/-" denotes addition/deletion respectively). Changes assigned to the histories of three buckets are shown. The hashing scheme of Figure 1 is assumed. Addition of oid 8 in S at t = 21, causes the first overflow. Moving oid 15 from bucket 0 to bucket 5 is seen as a deletion and an addition respectively. 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)>.

b_j's history. In practice, we expect that the n changes in S's evolution will be concentrated on the first few in the b_{total} bucket sequence, simply because a prefix of this sequence is always used. If we assume that most of S's history is recorded in the first $2^i M$ buckets (for some i), n_j behaves as $O(n/(2^i M))$ and therefore searching b_j's time-tree is rather fast.

A logarithmic overhead that is proportional to the number of changes n, is a common characteristic in query time of all temporal indexes that use partial persistence. The MVBT (or MVAS) tree will answer a temporal membership query about oid k on time t, in $O(\log_B(n/B))$ I/O's. We note that MVBT's logarithmic bound contains two searches. First, the appropriate B-tree that indexes *S(t)* is found. This is a fast search and is similar to identifying the hashing function and the bucket to search in persistent hashing. The second logarithmic search in MVBT is for finding k in the tree that indexes *S(t)* and is logarithmic on the size of *S(t)*. Instead persistent hashing finds oid k in expected $O(1)$ I/O's.

4.1.1 Update and Space Analysis. We proceed with the analysis of the update and space characteristics of partially persistent hashing. It suffices to show that the scheme uses *O(n/B)* space. An *O*(1) amortized expected update processing per change can then be derived. Clearly array *H* satisfies the space bound. Next we show that the space used by bucket histories is also bounded by *O(n/B)*. Recall that *n* corresponds to the total number of real object additions/deletions in set *S*'s evolution. However, the rehashing process moves objects among buckets. For the bucket histories, each such move is seen as a new change (deletion of an oid from the previous bucket and subsequent addition of this oid to the new bucket). It must thus be shown that the number of moves due to rehashing is still bounded by the number of real changes *n*. For this purpose we will use two lemmas.

Lemma 1: For *N* overflows to occur at least *NB*+1 real object additions are needed.

Proof: The proof is based on induction on the number of overflows. (1) For the creation of the first (*N*=1) overflow at least *B*+1 oid additions are needed. This happens if all such oids are mapped to the same bucket that can hold only *B* oids (each bucket starts with one empty page). (2) Assume that for the *N* first overflows *NB*+1 real object additions are needed. (3) It must be proved that the *N*+1 first overflows need at least (*N*+1)*B*+1 oid additions. Assume that this is not true, i.e., that only (*N*+1)*B* oid additions are enough. We will show that a contradiction results from this assumption. According to (2) the first *N* of the *(N*+1) overflows needed *NB*+1 real object additions. Hence there are *B*-1 remaining oid additions to create an extra overflow. Consider the page where the last overflow occurred. This bucket has a page with exactly one record (if it had less there would be no overflow, if it had more, the *N*-th overflow could have been achieved with one less oid). For this page to overflow we need at least *B* more oid additions, i.e., the remaining *B*-1 are not enough for the (*N*+1)-th overflow. Which results in a contradiction and the Lemma is proved. (Note that only the page where the *N*-th overflow occurred needs to be considered. Any other page that has space for additional oids cannot have more than one oid already, since the overflow that occurred in that bucket could have been achieved with less oids). ❑

The previous Lemma lower bounds the number of real oid additions from *N* overflows. The next Lemma upper bounds the total number of copies (due to oid rehashings) that can happen from *N* overflows.

Lemma 2: *N* overflows can create at most *N*(*B*+1) oid copies.

Proof: We will use again induction on the number of overflows. (1) The first overflow can create at

most B+1 oid copies. This happens if when the first overflow occurs all the oids in that bucket are remapped to a new bucket. The deleted records of the remapped B+1 oids are still stored in the history of the original bucket. (2) Assume that for the N first overflows at most $N(B+1)$ oid copies. (3) It must be shown that the first (N+1) overflows can create at most (N+1)(B+1) oid copies. We will use contradiction. Hence let's assume that this is not true, i.e., the first (N+1) overflows can create more copies. Let (N+1)(B+1)+x be that number where $x \geq 1$. Consider the last N overflows in the sequence of overflows. From (2) it is implied that these overflows have already created at most $N(B+1)$ oid copies. Hence there are at least $B+1+x$ additional copies to be created by the first overflow. However this is a contradiction since from (1) the first overflow can only create at most B+1 oid copies. ❑

We are now ready to prove the basic theorem about space and updating.

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

Proof: Assume for simplicity that set S evolves by only adding oids (oid additions create new records, overflows and hence more copying; deletions do not create overflows). As overflows 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 overflows have occurred, lemmas 1 and 2 imply that there must have been at least MB+1 real oid additions and at most $M(B+1)$ 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 overflows. These new overflows imply that there must have been at least 2MB+1 new real oid additions and at most $2M(B+1)$ copies created from these additions. There are also the $M(B+1)$ copy oids from the first round, which for the purposes of the second round are "seen" as regular oids. At most each such copy 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: $2M(B+1) + M(B+1)$.

The total number of copies C_{total} created after the i-th round (i = 0,1,2,...) is upper bounded by:

$$C_{total} \leq [\{M(B+1)\} + \{2M(B+1) + M(B+1)\} + ... + \{2^i M(B+1) + ... + M(B+1)\}]$$

where each {} represents copies per round. Equivalently:

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

After the *i*-th round the total number of real oid additions A_{total} is lower bounded by:

$$A_{total} \geq [\{MB+1\} + \{2MB+1\} + ... + \{2^i M(B+1)\}] \quad .$$

Equivalently:

$$A_{total} \geq MB\sum_{k=0}^{i} 2^k + (i+1) = MB(2^{i+1}-1) + i + 1 > MB(2^{i+1}-1) \geq MB2^i \quad \text{(ii)}.$$

From (i), (ii) it can be derived that there exists a positive constant *const* such that $C_{total}/A_{total} < const$ and since A_{total} is bounded by the total number of changes *n*, we have that C_{total}= *O(n)*. To prove that partially persistent 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 have to be carefully examined. This is because a rehashing of a bucket is caused by a single real oid addition (the one that created the overflow) but it results into 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 we can prove that any sequence of *n* real changes can at most create *O(n)* copies (extra work) or equivalently *O*(1) amortized effort per real change. ❑

4.1.2 Optimization Issues. Optimizing the performance of partially persistent 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 $l(t) = |(S(t))|/BR(t)$, where $|S(t)|$ and *R(t)* denote the size of the evolving set *S* and the number of buckets used at *t* (clearly $R(t) \leq b_{total}$). 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: $|b_j(t)| \approx |(S(t))|/R(t)$.

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 $|b_j(t)|/uB$ pages, which is actually *l/u*. Ideally, we would like the answer to a snapshot query to be contained in a single page (plus probably one more for the acceptor page). Then a good optimization choice is to keep $l/u < 1$. 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 to find 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. As mentioned in section 3, 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 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. Due to 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 the next section.

4.2 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. This is because in practice each bucket is expected to be about one or two pages long. 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.

When bucket b_j is first created, an empty page is assigned to list lb_j. A list page has two areas. The first area is used to store oid records and its size is B_r where $B_r < B$. The second area (of size $B - B_r$) accommodates an extra structure (array *NT*) to 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 the list and more pages are appended as needed. 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 from *now* to t' (a logical deletion).

As with the Snapshot Index, we need a notion of page *usefulness*. A page is called useful as long as it contains at least *V* alive objects or while it is the last page in the list. Otherwise it is a non-useful page. For the following discussion we assume that $0 < V \leq B_r / 4$. Except for the last page in the list, a useful page can become non-useful because of an oid deletion (which will bring the number of alive oids in this page below the threshold). The last page can turn from useful to non-useful when it gets full of records (an event caused by an oid insertion). At that time if the page's total number of alive oids is less than *L* the page becomes non-useful. Otherwise it continues to be a regular useful page. When the last page gets full, a new last page is added in the list.

Finding the state $b_j(t)$ is again equivalent to finding the useful pages in $lb_j(t)$. We will use two extra structures. The first structure is an array $FT_j(t)$ which for any time *t* provides access to the first useful page in $lb_j(t)$. Entries in array FT_j have the form <*time, pid*> where pid is a page address. If the first useful page of the list changes at some *t*, a new entry with *time* = *t* and the pid of the new first useful page is appended in FT_j. This array can be implemented as a multilevel, paginated index since entries are added to it in increasing time order.

To find the remaining useful pages of $lb_j(t)$, every useful page must know which is the next useful page after it in the list. This is achieved by the second structure which is implemented inside every list page. In particular, this structure has the form of an array stored in the page area of size $B - B_r$. 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 useful page after useful page *A*.

If during the usefulness period of some page *A*, its next useful page changes many times, *NT(A)* can become full. Assume this scenario happens at time *t* and let *C* be the useful page before page *A*. Page *A* is then *artificially* turned to non-useful (even if it still has more than *V* alive records) and is replaced by a copy of it, page A'. We call this process artificial, since it was not caused by an

oid insertion/deletion to this page, rather it is due to a change in a page ahead. The new page A' has the same alive records as A but an empty $NT(A')$. A new entry is then added in $NT(C)$ with A''s pid. The first entry of $NT(A')$ has the pid of the useful page (if any) that was after page A at t.

If all useful list pages until page A had their NT arrays full just before time t, the above process of artificially turning useful pages to non-useful can propagate all the way to the top of the list. If it reaches the first useful page in the list, a copy of it is created and array FT_j is updated. However, this does not happen often. Figure 3 shows an example of how arrays $NT()$ and FT_j are maintained.

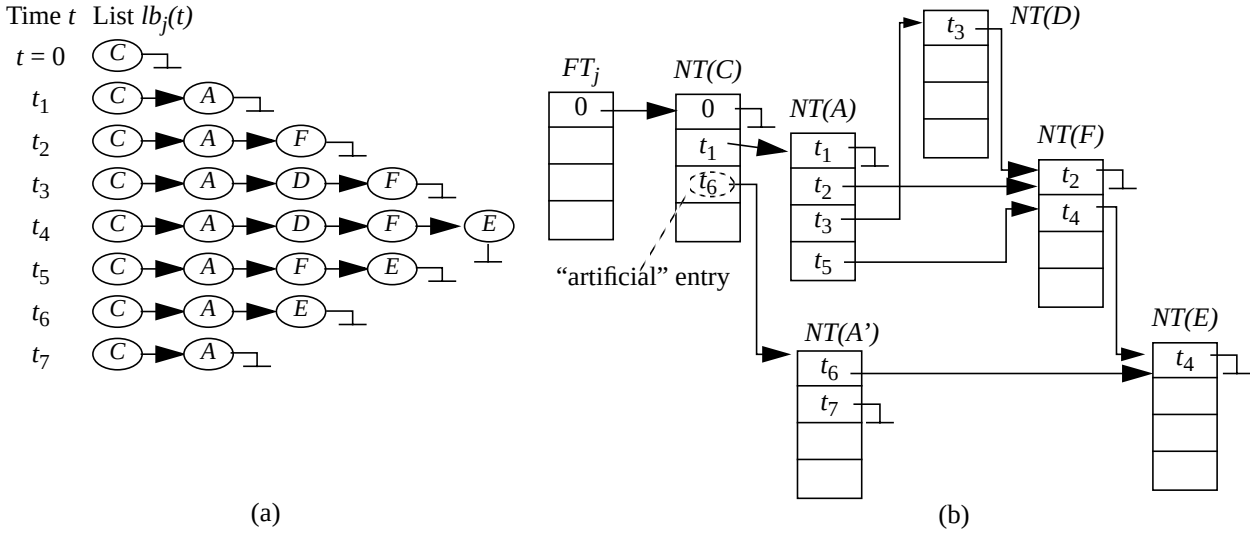

Figure 3: (a) An example evolution for the useful pages of list $lb_j(t)$. (b) The corresponding FT_j and NT arrays. From each page only the NT array is shown. In this example $B\text{-}B_r = 4$ entries. Since the page in front of page A changes often, its $NT(A)$ array fills up and at time t_6 an artificial copy of page A is created with array $NT(A')$. Array $NT(C)$ is also updated about the artificially created new page.

The need for artificial creation of a copy A' of page A is for faster query processing. The $NT(C)$ array enables finding which is the next useful page after C for various time instants. Assume for the moment that no new copy of page A is created, but instead $NT(A)$ is allowed to grow over the $B - B_r$ available area of page A, in additional pages. The last entry on $NT(C)$ would then still point 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 useful page in front of page A changes often. The use of artificial copies guards against similar situations as 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].

Special care is needed when a page turns from useful to non-useful due to an oid deletion/

insertion in this page. To achieve good answer clustering, the alive oids from such a page are merged with the alive oids of a sibling useful page (if such a sibling exists) to create one (or two, depending on the number of alive oids) new useful page(s). The new useful page(s) may not be full of record oids, i.e., future oid insertions can be accommodated there. As a result, when a new oid is inserted, the list of useful pages is serially searched and the new oid is added in the first useful page found that has space (in the B_r area) to accommodate it. Details are described in the Appendix.

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* must be found. This part is the same with the evolving-set approach. Reconstructing the state of bucket $b_j(t)$ is performed in two further steps. First, using *t* the first useful page in $lb_j(t)$ is found by searching array FT_j (which corresponds to searching the time-tree of each bucket in the evolving-set approach). This search is bounded by $O(\log_B(n_j/B))$. The remaining useful pages of $lb_j(t)$ (and thus the oids in $b_j(t)$) are found by locating *t* in the *NT* array of each subsequent useful page (instead, the evolving-set approach uses the access forest of the Snapshot Index). Since all useful pages (except the last in the list $lb_j(t)$) have at least *V* alive oids from the answer, the oids in $b_j(t)$ are found with an additional $O(|b_j(t)|/B)$ I/O's. The space used by all the evolving-list structures is $O(n_j/B)$.

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 adding or deleting an oid. Second, the nature of reconstructing $b_j(t)$ is different. In the evolving-list reconstruction 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.

5. Performance Analysis

We compared Partially Persistent Hashing (PPH) against Linear Hashing (in particular *Atemporal* linear hashing, to be discussed later), the MVBT and the R*-tree. The implementation and the experimental setup are described in 5.1, the data workloads in 5.2 and our findings in 5.3.

5.1 Method Implementation - Experimental Setup.

We set the size of a page to hold 25 oid records (*B*=25). An oid record has the following form, <*oid, start_time, end_time, ptr*>, where the first field is the oid, the second is the starting time and the third the ending time of this oid's lifespan. The last field is 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 not the ephemeral linear hashing whose evolution the partially persistent hashing observes and stores. Rather, it is a 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 [Lin80]. For split functions we used the hashing by division functions $h_i(oid) = oid \bmod 2^i M$ with $M = 10$. So as to get 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 Multiversion B-tree (MVBT) implementation is based on [BGO+96]. For fast updating the MVBT uses a buffer that stores the pages in the path to the last update (LRU buffer replacement policy is used). Buffering during updating can be very advantageous since updates are directed to 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 this buffer for queries, too. However, for a fair comparison with the other methods when measuring the query performance of the MVBT we invalidate the buffer content from previous queries. Thus the measured query performance is independent from the order in which queries are executed. Finally, in the original MVBT, the process of answering a query starts from a *root** array. For every time *t,* this array identifies the root of the B-tree at that time (i.e., where the search for the query should start from). Even though the root* can increase with time is small enough to fit in main memory. 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 has to be merged with a sibling (this is called a *weak version underflow*). On the other extreme, if a page has already *B* records (alive or not) and a new record has to be added, the page splits (a page *overflow*). 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 has to 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 buffer 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 $e = 4$. The directory pages of the MVBT have the same format as the data pages.

For the Partially Persistent Hashing we implemented both the set-evolution (PPH-*s*) and the list-evolution (PPH-*l*) approaches. Both approaches observe an ephemeral linear hashing *LH(t)* whose load *l(t)* lies between f=0.1 and g=0.2. Array *H* which identifies the hashing scheme used at each time is kept in main-memory, so no I/O access is counted for using this structure. This is similar to keeping the root* array of the MVBT in main memory. In all our experiments the size of array *H* is never greater than 15 KB. Unless otherwise noted, PPH-*s* was implemented with u = 0.3 (various other values for usefulness parameter u were also examined). Since the entries in the time-tree associated with a bucket have half the oid record size, each time-tree page can hold up to 50 entries.

In the PPH-*l* implementation, the space for the oid records B_r can hold 20 such records. The value of *V* is set equal to 5 since $0 < V \leq B_r / 4$ .This means that, a page in the list can be useful as long as the number of alive oids in the page is greater or equal to 5. The remaining space in a list page (of size 5 oid 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 each FT_j array can hold up to 50 entries.

For the R*-tree method we used two implementations, one with intervals (R*i*) in a two-dimensional space, and another with points in a three-dimensional space (R*p*). The R*i* implementation assigns to each oid its lifespan interval; one dimension is used for the oids and one for the lifespan intervals. 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*>. Directory pages include one more attribute per record so as to represent 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 use a buffer (10 pages) to keep the pages in the path leading to the last update. As with the MVBT, this buffer is not used for the query phase.

5.2 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*), *E* is the evolution of set *S* and $Q = \{Q_1, ... , Q_r\}$ is a collection of queries, where $r = |U|$ and Q_k is the set of queries 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* have to be 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.

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.

5.3 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 4.a presents the average number of pages accessed per query by all methods. The PPH 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 uses about twice as many I/O's than the PPH approaches since a tree has to be traversed per query. The R*i* uses more I/O's per query than the MVBT, mainly due to node overlapping and larger tree height (which in the R*i* structure relates to the total number of oid lifespans while in the MVBT corresponds to the number of alive oids at the time specified by the query). The problem of node overlapping is even greater with the query performance of the R*p* tree, which in Figure 4.a has been truncated to fit the graph (R*p* used an average of 44 I/O's per query in this experiment). In the R*p* all alive oids have the same end_time (*now*) that causes them to be clustered together even though they have different oids (that is, overlapping extends to the oid dimension as well). As observed elsewhere [KTF98], transaction-time lifespans are not maintained efficiently by plain R-trees.

Figure 4.b shows the average number of I/O's per update. The best update performance was given by the PPH-*s* method. The MVBT had the second best update performance. It is larger than PPH-*s* since MVBT is traversing 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 tree traversed is related to the number of alive oids at the time of the update). The ALH and PPH-*l* used even larger update processing. This is because in ALH all lifespans with the same oid are thrown on the same bucket thus creating large buckets that have to be searched serially during an update. In PPH-*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 R*p* tree uses even larger update processing which is due to the bad clustering on the common *now* end_time.

The space consumed by each method appears in figure 4.c. The ALH approach uses the smallest space since it stores a single record per oid lifespan and uses "controlled" splits with high utilization (*f* and *g* values). The PPH methods have also very good space utilization with the PPH-*s* being very close to ALH. PPH-*l* uses more space than PPH-*s* because the *NT* array implementation reduces page utilization. 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%. The MVBT has the largest space requirements, about twice more space than the ALH and

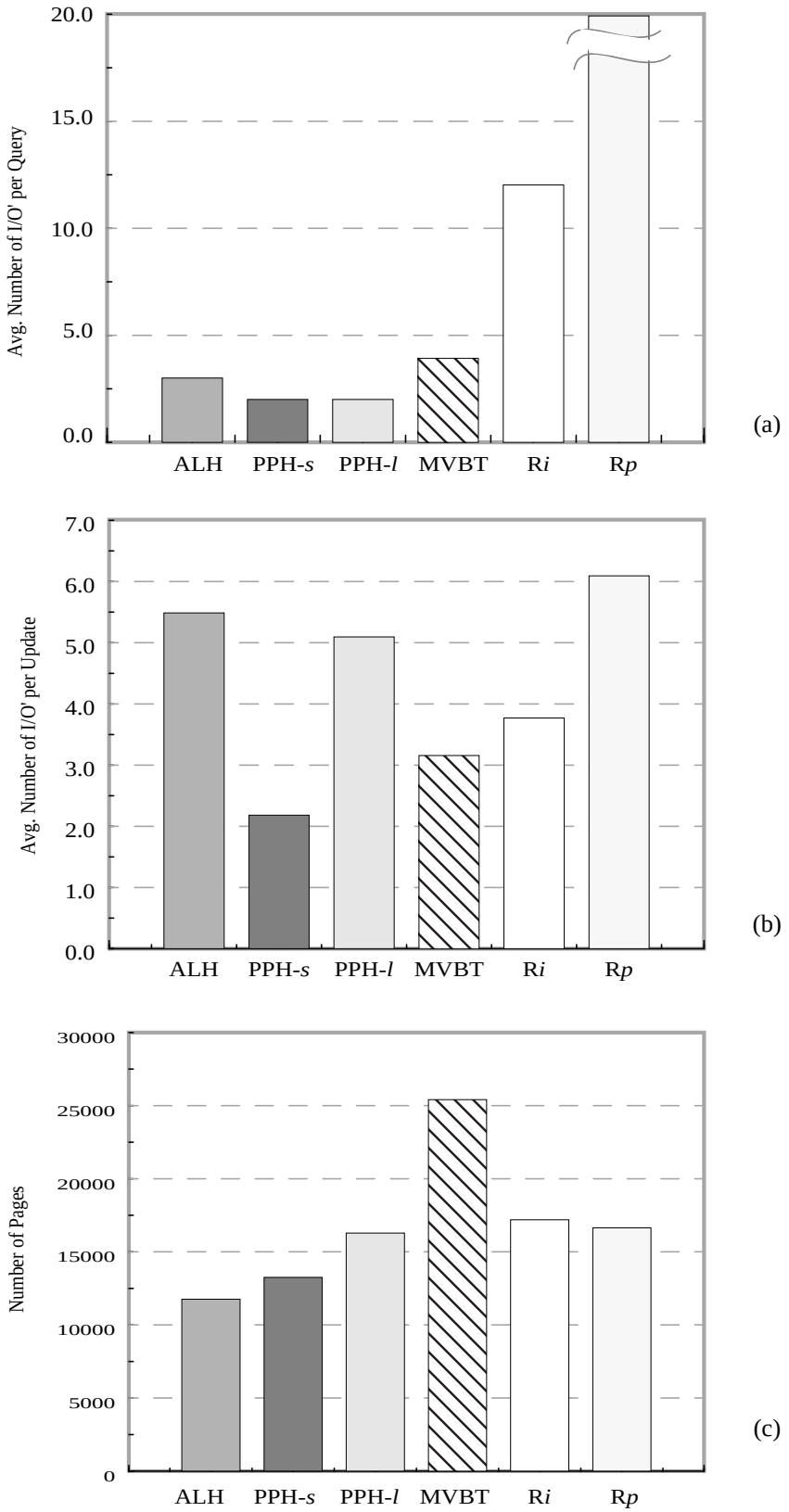

Figure 4: (a) Query, (b) Update, and, (c) Space performance for all implementations on a uniform workload with 8K oids, $n \sim 0.5$M and $\overline{NB} \sim 30$.

PPH-*s* methods.

In summary, the PPH-*s* has the best overall performance. Similarly with the comparison between ephemeral hashing and B-trees, the MVBT tree behaves worse than temporal hashing (PPH-*s*) for temporal membership queries. The ALH is slightly better than PPH-*s* only in space requirements, even though not significantly. The R-tree based methods are much worse than PPH-s in all three performance criteria.

To consider the effect of lifespan distribution all approaches were compared using four additional workloads (namely the exponential, step, normal and poisson). 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 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.

Figure 5 presents the query, update and space performance under the new workloads. For simplicity only the R*i* method is presented among the R-tree approaches (as with the uniform load, the R*p* used consistently more query and update than R*i* and similar space). The results resemble the previous uniform workload. As before, the PPH-*s* approach has the best overall performance using slightly more space than the "minimal" space of ALH. PPH-*l* has the same query performance and comparable space with PPH-*s* but uses much more updating. Note that in Figure 5.a, the query performance of R*i* has been truncated to fit the graph (on average, R*i* used about 10, 13, 11 and 10 I/O's per query in the exponential, step, normal and poisson workloads respectively).

Similarly, in Figure 5.c the space of the MVBT is truncated (MVBT used about 26K, 29K, 25K and 35.5K pages for the respective workloads).

The effect of the number of lifespans per oid was tested using eight uniform workloads with varying average number of lifespans. All used $|U| = 8000$ different oids and the same number of queries (~115K). The other parameters are shown in the following table:

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

The results appear in Figure 6. The query performance of atemporal hashing deteriorates as $\overline{NB}$ increases since buckets become larger (Figure 6.a). The PPH-*s*, PPH-*l* and MVBT methods have a query performance that is independent of $\overline{NB}$ (this is because in all three methods the $\overline{NB}$ lifespans of a given oid appear at different time instants and thus do not interfere with each other). The query performance of R*i* was much higher and it is truncated from Fig. 6.a. Interestingly, the R*i* query performance decreases gradually as $\overline{NB}$ increases (from 12.6 I/O's to 9.4 I/O's). This is because R*i* clustering improves as $\overline{NB}$ increases (there are more records with the same key).

PPH-*s* outperforms all methods in update performance (Figure 6.b). As with querying, the updating of PPH-*s*, PPH-*l* and MVBT is basically independent of $\overline{NB}$. Because of better clustering with increased $\overline{NB}$, the updating of R*i* gradually decreases. In contrast, because 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 *n* per evolution (Table 1). The ALH has the lower space, followed by the PPH-*s*; the MVBT has the steeper space increase (for $\overline{NB}$ values 80 and 100, MVBT used ~68K and 84.5K pages).

The effect of the number of distinct oids used in an evolution was examined by considering three variations of the uniform workload. The number of distinct oids $|U|$ was: 5000, 8000 and

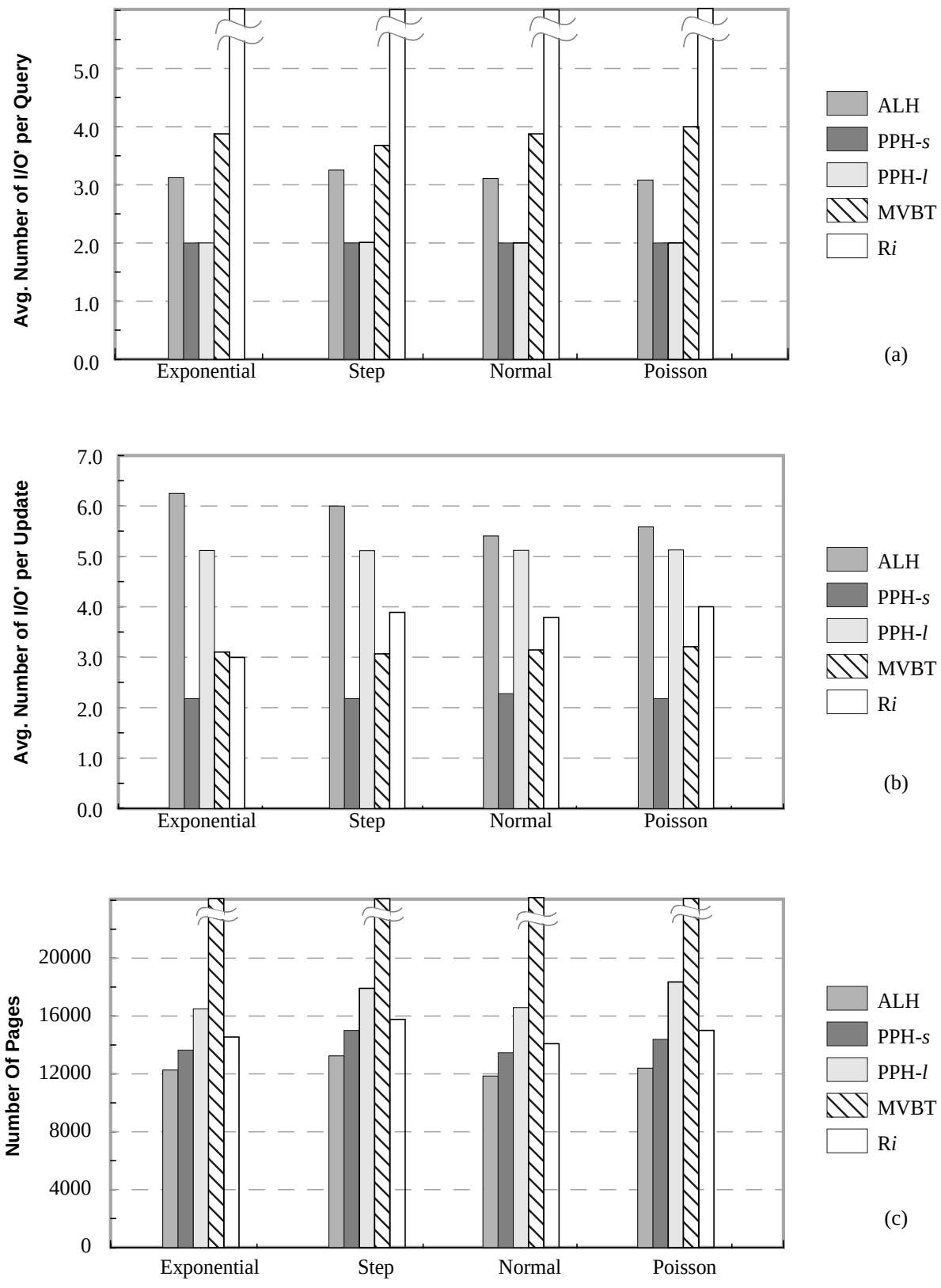

Figure 5: (a) Query, (b) Update, and, (c) Space performance for ALH, PPH-*s*, PPH-*l*, MVBT and R*i* methods using the exponential, step, normal and poisson workloads with 8K oids, $n \sim 0.5M$ and $\overline{NB} \sim 30$.

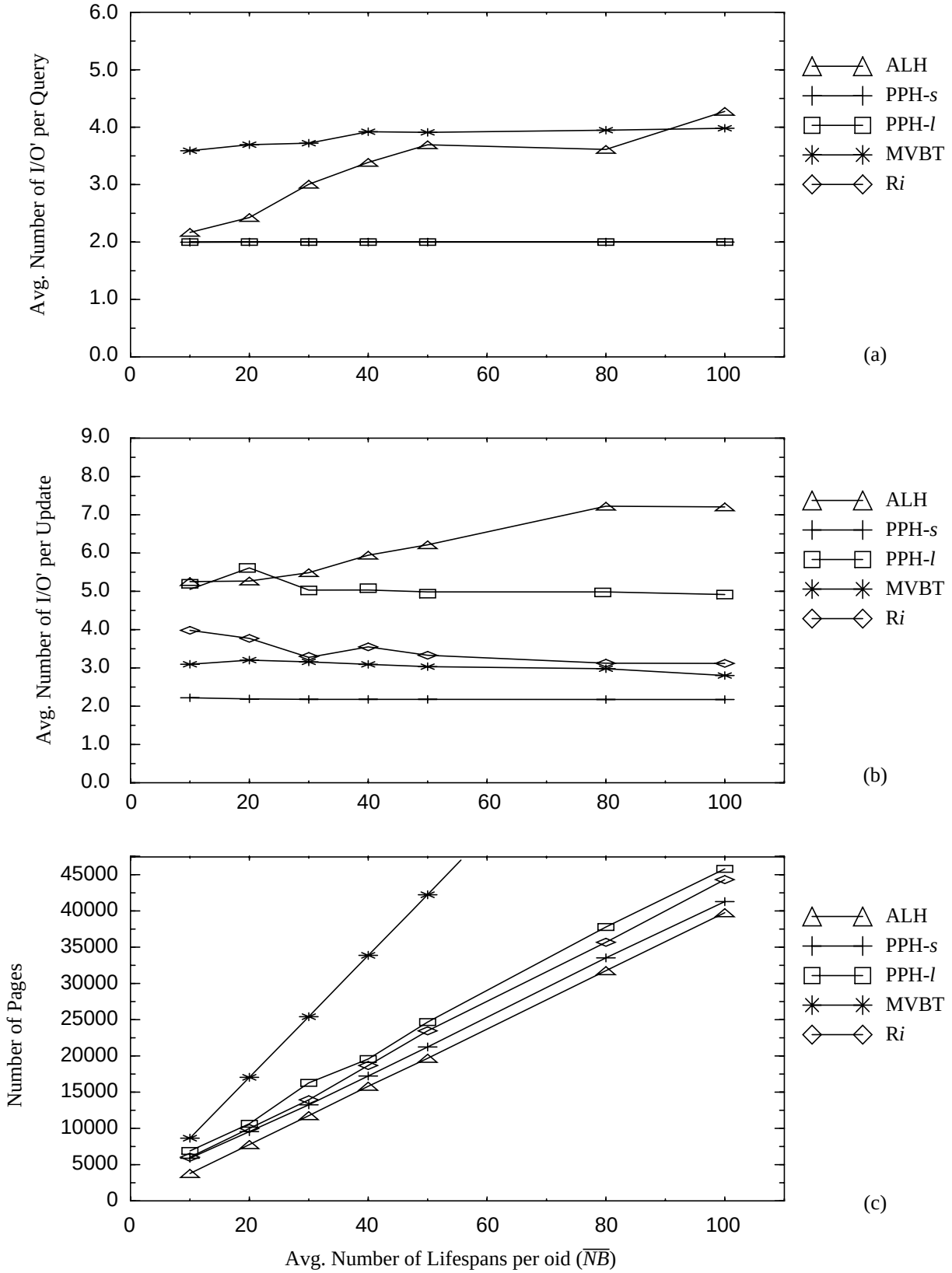

Figure 6: (a) Query, (b) Update, and, (c) Space performance for ALH, PPH-*s*, PPH-*l*, MVBT and Ri methods using various uniform workloads with varying $\overline{NB}$.

12000, respectively. All workloads had similar average number of lifespans per distinct oid ($\overline{NB}$ ~ 30). The other parameters appear in Table 2. The results appear in figure 7. The query performance of PPH-*s* and PPH-*l* is independent of |*U*|. In contrast, it increases for both MVBT and R*i* (the R*i* used about 10.4, 12 and 13 I/O's per query). The reason for this increase is that there are more oids stored in these tree structures thus increasing the structure's height (this is more evident in R*i* as all oids appear in the same tree). In theory, ALH should also be independent of the universe size |*U*|; the slight increase for ALH in Figure 7.a is due to the "controlled" splits policy that constrained ALH to a given space utilization. Similar observations hold for the update performance. Finally, the space of all methods increases because *n* increases (Table 2).

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

From the above experiments, the PPH-*s* method appears to have the most competitive performance among all solutions. As mentioned in section 4.1.2, the PPH-*s* performance can be further optimized through the setting of usefulness parameter *u*. Figure 8 shows the results for the basic Uniform-30 workload (|*U*| = 8000, *n* = 466854, *NB* = 237606 and $\overline{NB}$ ~ 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 8.a, the query time is minimized after *u* = 0.3. The update is similarly minimized (Figure 8.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 (for smaller *u*'s the alive oids can be distributed into more pages which increases the update process). Figure 8.c shows the space of PPH-*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 PPH-*s* can be made similar to that of the ALH at the expense of some increase in query/update performance.

6. Conclusions and Open Problems

This paper 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*

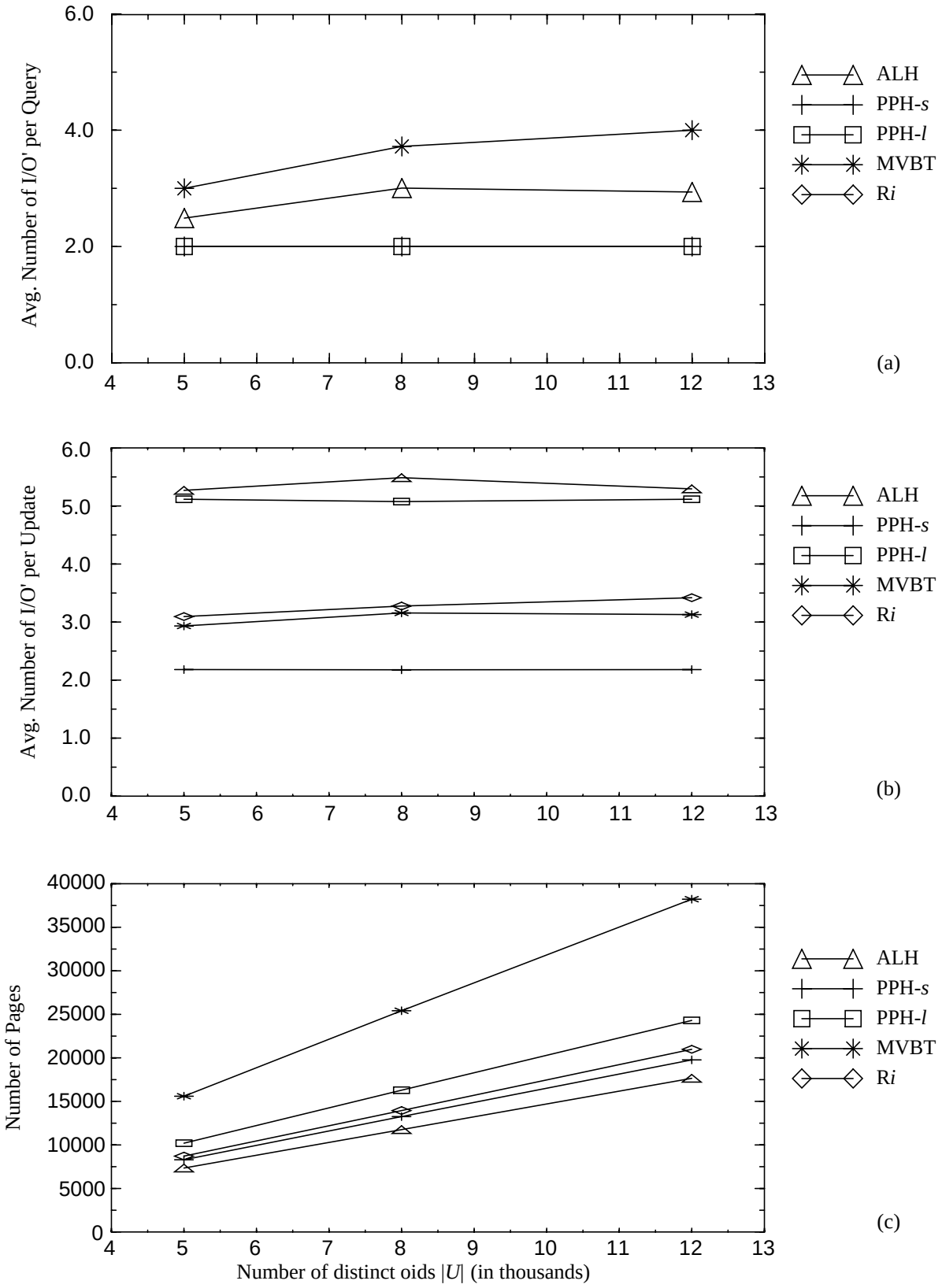

Figure 7: (a) Query, (b) Update, and, (c) Space performance for ALH, PPH-*s*, PPH-*l*, MVBT and R*i* methods using various uniform workloads with varying |*U*|.

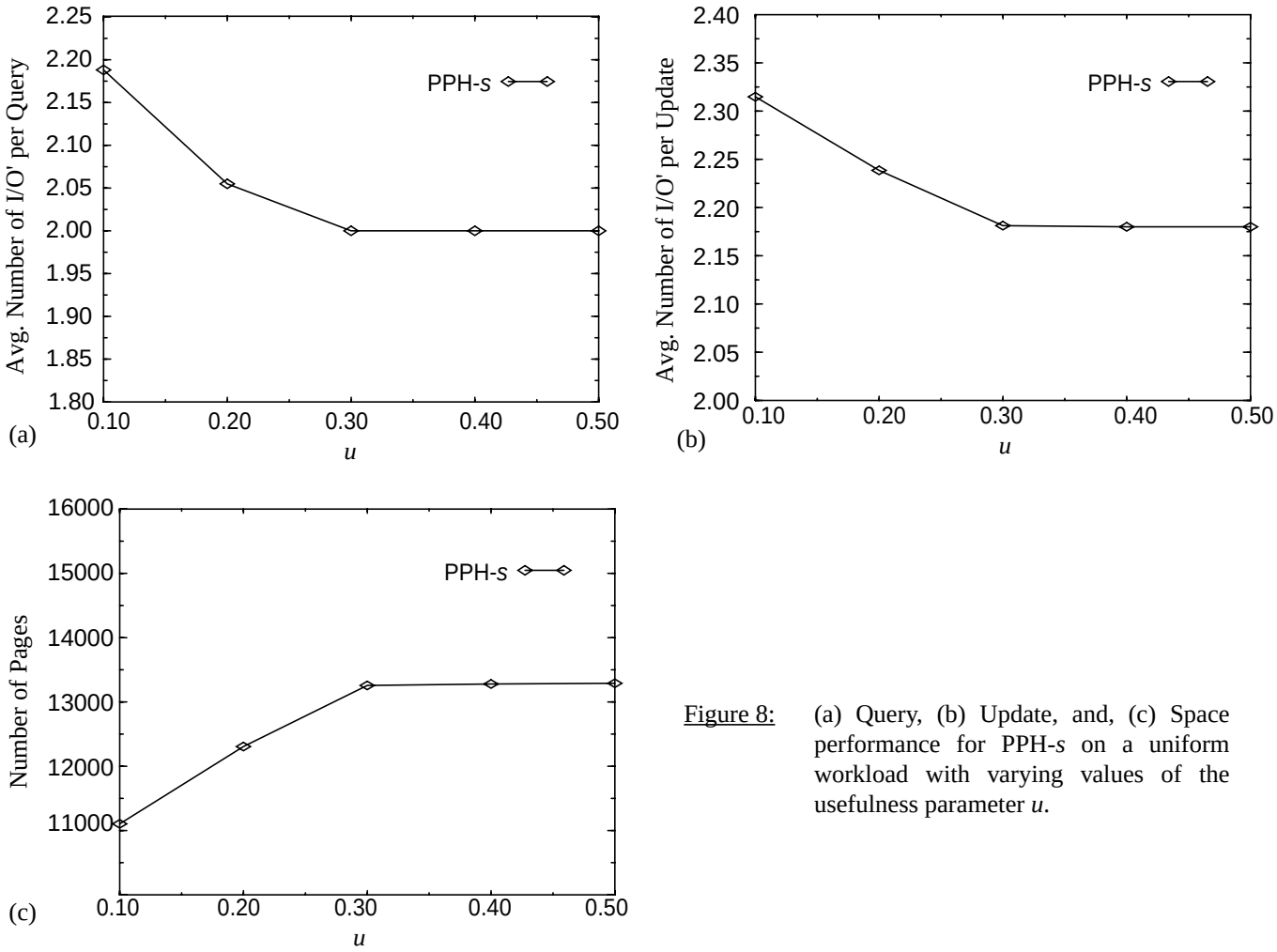

Figure 8: (a) Query, (b) Update, and, (c) Space performance for PPH-*s* on a uniform workload with varying values of the usefulness parameter *u*.

hashing (PPH) was presented. For queries and updates, this scheme behaves as if a separate, ephemeral dynamic hashing scheme is available on every state assumed by set *S* over time. However the method still uses linear space. By hashing oids to various buckets over time, PPH reduces the temporal hashing problem into reconstructing previous bucket states. Two flavors of partially persistent hashing were presented, one based on an evolving-set abstraction (PPH-*s*) and one on an evolving-list (PPH-*l*). They have similar query and comparable space performance but PPH-*s* uses much less updating. Both methods were compared against straightforward approaches namely, traditional (atemporal) linear hashing scheme, two R*-tree implementations and the Multiversion B-Tree. The experiments showed that PPH-*s* has the most robust performance among all approaches. Partially persistent hashing should be seen as an extension of traditional external dynamic hashing in a temporal environment. The methodology is independent from which ephemeral dynamic hashing scheme is used. While the paper considers linear hashing, it applies to other dynamic hashing schemes as well. There are various open and interesting problems.

Traditionally hashing has been used to speed up join computations. We currently investigate the use of temporal hashing to speed up temporal joins [SSJ94]. Another problem is to extend temporal membership queries to time intervals (find whether oid *k* was in any of the states set *S* had over an interval *T*). 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 B. Seeger for kindly providing us with the R* and MVB-tree code. Part of this work was performed while V.J. Tsotras was on a sabbatical visit to UCLA; we would thus like to 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.

[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. 29th 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.

[EN94] R. Elmasri, S. Navathe, *Fundamentals of Database Systems*, Second Edition, Benjamin/ Cummings Publishing Co, 1994.

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

[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.

[KTF98] A. Kumar, V.J. Tsotras, C. Faloutsos, "Designing Access Methods for Bitemporal Databases", *IEEE Trans. on Knowledge and Data Engineering*, Feb. 1998.

[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.

[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.

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

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

[S94] B. Salzberg, "Timestamping After Commit", *Proc. 3rd Intern. Conf. on Parallel and Distributed Information Systems*, pp 160-167, 1994.

[SA85] R.T. Snodgrass, I. Ahn, "A Taxonomy of Time in Databases", *Proc. ACM SIGMOD Conf.*, pp 236-246, 1985.

[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.

[ST97] B. Salzberg, V.J. Tsotras, "A Comparison of Access Methods for Time-Evolving Data", to appear at *ACM Computing Surveys*; appears as TimeCenter Tech. Report TR-18, Aalborg University, 1997 (http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/publications2.html).

[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.

[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.

Appendix: Description and Analysis of the Evolving-List approach

Method Description. Special care is needed when a list page becomes non-useful due to an oid deletion/ insertion. Assume that list page A moves from the useful state to non-useful at some time t; let V_A and R_A respectively be the number of alive oid records and the total number of records it contains (it is always true that:$V_A \leq R_A \leq B_r$). To achieve good answer clustering, the alive oids from such a page are merged with the alive oids of a sibling useful page (if such a sibling exists) to create one or two (depending on the number of alive oids) new useful page(s). We distinguish two cases:

(1) Page A is the first useful page in the list.Then we have:

(1.1) There is a useful page D that follows A in the list. Page A became non-useful because one of its alive oids was deleted; thus $V_A = V$-1. Let D contain V_D alive oids among a total of R_D oids.

(1.1.1) $R_D + V_A \leq B_r$ (i.e., the alive oids of *A* can fit in page *D*). Copy the V_A alive records of *A* into *D* and update the FT_j table to point to page *D* at time *t*. (Page *D* contains at least 2*V*-1 alive objects unless *D* is the last page in the list).

(1.1.2) $R_D + V_A > B_r$. Two subcases must be examined:

(1.1.2.a) $V_D \leq 3V$. Then $V_D + V_A < B_r$, i.e., the alive oids from both *A* and *D* can fit in a page). Create a new page *E* and copy all alive oids from *A* and *D* into *E*. Page *D* is artificially made non-useful. Add a new entry in array FT_j to point to page *E* at *t* and a new entry in array *NT(E)* to point to the next sibling (if any) of *D* at *t*. (New page *E* contains at least 2*V*-1 alive objects unless it is the new last page in the list).

(1.1.2.b) $V_D > 3V$. Create a new page *E* and copy the *V*-1 alive oids from page *A* and *V* alive oids from page *D* into page *E*. Since page *E* replaces page *A* on the top of the list, add a new entry in array FT_j to point to page *E* at time *t*. Add a new entry in *NT(E)* to point to page *D* at *t*. New page *E* starts with 2*V*-1 alive oids while page *D* continues with at least 2*V* alive oids.

(1.2) Page *A* is the last page in the list and $R_A = B_r$. Thus *A* became non-useful by an oid insertion that made it a full page but without enough alive oids (V_A<*V*). Because *A* became full, a new empty last page *D* is created after *A*. Copy the alive oids of *A* into *D* and update the previous sibling (if any) of *A* to point to *D*.

(2) Page *A* is not the first useful page in the list. Hence there exists a useful page *C* that is the previous sibling of *A* in the list.

(2.1) Page *A* became non-useful because one of its alive oids was deleted, i.e., $V_A = V$-1.

(2.1.1) If $R_C + V_A \leq B_r$, there is enough space in page *C* to hold the remaining alive oids of *A*. Hence copy the V_A alive oids into *C* and update *NT(C)* to point to the next useful list page after *A* (if any) at time *t*.

(2.1.2) If $R_C + V_A > B_r$, two subcases must be examined:

(2.1.2.a) If $V_C \leq 3V$, then as in case (1.1.2.a) create a new page *E* and copy all alive oids of *A* and *C* into *E*. Update *NT* arrays appropriately.

(2.1.2.b) If $V_C > 3V$ create a new page *E*. Similarly with case (1.1.2.b), copy the *V*-1 alive oids of page *A* and *V* alive oids from page *C* into new page *E*. Since page *E* replaces page *A*, array *NT(C)* is updated to point to page *E*. A new entry is added in array *NT(E)* to point to the useful page (if any) that was after *A*.

(2.2) Page *A* is the last page in the list and $R_A = B_r$. Thus *A* became non-useful by an oid insertion that made it a full page but without enough alive oids (V_A<*V*). As in case (1.2) above, create a new

empty last page *D* and copy all alive oids of *A* into *D*.

Update and Space Analysis. Updating in the evolving-list approach is $O(|b_j(t)|/B)$ since the whole current list has to be searched until a new oid is added or an existing oid is updated as deleted. Despite page copying, the space is $O(n_j/B)$, where n_j is the total number of changes recorded in the evolution of bucket b_j. Note that n_j contains both the real oid additions/deletions and the changes recorded due to bucket overflows. However, Lemmas 1 and 2 still apply, i.e., n_j is proportional to the number of real changes.

Copying occurs when a page turns from useful to non-useful. This can happen when a page's *NT* array becomes full. It can be easily verified that the extra space used by the *backward updating* technique [TGH95] is $O(n_j/B)$. Another way for a page to become non-useful is by an oid deletion/insertion into this page. Copying from such cases results to linear space, too. More specifically, using the "accounting" method of [CLR90] (also applied in [BGO+96, VV97]) we can show that for a sequence of n_j oid changes (insertions/deletions) the space consumed is $O(n_j/B)$. For each oid change that updates a page, a *credit* of $1/V$ is accumulated. The *charge* for each new page is 1. It is enough to show that adequate credit has been accumulated before a new page is created. The proof is based on the fact that each new useful page starts with at least $2V$-1 alive oids unless it is the last page in the list. In both cases such a useful page can become non-useful only after a substantial amount of new changes. For example, a new useful page with $2V$-1 alive oids can become non-useful only after at least V deletions. A new last page may start with less than V alive oids, but it can turn non-useful only after it becomes full of records, i.e., after at least $3V$ new additions.