THE PERFORMANCE OF A MULTIVERSION ACCESS METHOD

David Lomet
Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square, Bldg 700
Cambridge, Massachusetts 02139

Betty Salzberg*
College of Computer Science
Northeastern University
Boston, Massachusetts 02115

ABSTRACT

The *Time-Split B-tree* is an integrated index structure for a versioned timestamped database It gradually migrates data from a current database to an historical database, records migrating when nodes split Records valid at the split time are placed in both an historical node and a current node This implies some redundancy Using both analysis and simulation, we characterize the amount of redundancy, the space utilization, and the record addition (insert or update) performance for a spectrum of different rates of insertion versus update Three splitting policies are studied which alter the conditions under which either time splits or key space splits are performed

1. INTRODUCTION

A growing area of interest in the database community is in the support of multiversioned data [LoSa, AhSn, JeMR, Ston] Multiversioned data, when updated, results in a new version of the data being created Because the old version is retained, several versions of a record can exist, each appropriate to some particular time

There are many applications where multiversioned data is of interest[McKe, SnAh, SeSh] These include financial transactions, university transcripts, engineering design, legal and medical records, etc One usually wants faster access to the current records while tolerating slower access to the historical records It is thus useful to keep the *current* database small and keep it on a high performance medium The *historical* part can then be stored in a separate area, possibly on a slower medium In [LoSa], we developed the *Time-Split B-tree* for these applications

A Time-Split B-tree (TSB-tree) has a single unified index for retrieval from both the historical and the current database Data is written to the historical database by appending at its end Thus, while a non-volatile write many/read many (WMRM) medium is required for the current database, e g magnetic disk, it is possible to efficiently exploit a write-once/read-many (WORM) medium for the historical database, e g optical disk

There are two types of node splitting In both cases, an index term describing the split is posted to the parent index node

Key Splitting: As with B^+-trees, records with keys greater than or equal to the split key go to a new current node, records with keys less than the split key remain in the original node

Time Splitting: Records valid before the split time go to a new historical node, records valid at or after the split time remain in the original current node Records valid both before and after the split time have copies in both historical and current nodes

Different policies can be adopted for choosing split times and for choosing whether to split by time or by key or by key and time simultaneously These policy choices affect the performance characteristics of the structure This is explained in the next section

Time splitting introduces redundancy Records that exist across the split time need to appear in both resulting nodes The benefit of the redundancy is that a snapshot of the database as of some time has locality, i e , a node in the database contains all records in a given key range valid in a given time range Each node forms a rectangular partition of the key-time space as illustrated in Figure 1 The disadvantage is that long-lived records have many copies

In this paper, we both analyze and simulate the performance of the TSB-tree We provide asymptotic performance results under two assumptions

Uniform Growth Assumption: A new record is equally likely to be between any two existing records Hence, the probability that a record is inserted into a node is proportional to the number of records with unique keys in the node

Equal Probability Assumption: Each record with a unique key is equally likely to be updated

The two assumptions above do not imply that our results apply only to uniformly distributed keys, even

[0]This research was partially supported by NSF Research Grant IRI-88-15707

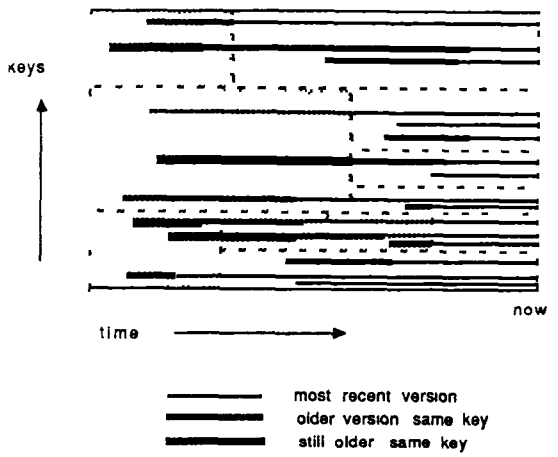

Figure 1 Historical and current nodes form a rectangular partition of the key-and-time space The shaded rectangles represent nodes in the historical database

though our simulation employed uniformly distributed keys When analyzing index-based access methods, the purpose of a uniform distribution is to realize the uniform growth assumption

We use a form of fringe analysis [EZGMW, BYLa] This computes a closure on node probabilities and produces asymptotic performance results directly The simulation entails multiple trials, each trial adding 50,000 records Node probabilities are determined by actual count of each type of node The simulation confirms the analysis and extends our results to nodes whose sizes are too large to analyze and to a split policy that did not succumb to analysis

Our base case split policy for multiversioned data is the write-once B-tree (WOB-tree) of Easton [East] The additional split policies of the TSB-tree exploit the fact that current data is stored on a WMRM medium, unlike the WOB-tree's WORM medium The impact of this difference, and of the additional split policies is shown to be substantial

Both the analysis and the simulation are parameterized in terms of the percentage of updates versus insertions The results that are presented graphically characterize the performance of the multiversioned index methods under a wide spectrum of insertion versus update rates Three split policies are studied and compared

The next section reviews the design of the TSB-tree In section 3, we introduce the fringe analysis model Section 4 describes our simulation model Performance results are presented in section 5 Finally, in section 6, we briefly discuss additional issues, e g splitting policy for index nodes, and draw some conclusions

2. THE TIME-SPLIT B-TREE

2.1 Description of TSB-tree Nodes

The leaves of the TSB-tree, like the B^+-tree, contain all the data records Each record contains a key, some data, and the commit time of the transaction that inserted it There may be many versions of the same record in the same leaf node That is, an update of a record is treated as an insertion of a new version with the same key but different timestamp Similarly, an index term of the TSB-tree contains a key, a timestamp and a pointer to a node on a lower level of the tree

2.2 Record Addition in the TSB-tree

To add a record (insert or update) to a TSB-tree, a search process is followed to find the correct leaf If there is room, the new record is placed in that leaf If there is no room, a "split" takes place, a new leaf node is allocated, and a new index term is posted to the parent Similarly, if index nodes are full, they too are split We discuss index node splitting briefly in section 6 What follows applies to data nodes.

There are key splits, time splits, and combinations of the two A key split is like a split in a B^+-tree The middle key of the node is used as the split key The considerations involved in performing a time split are more complicated

2.3 Time Splitting

Time splitting in a TSB-tree is derived from the time-splitting used in the WOB-tree [East] The WOB-tree does not have separate historical and current databases and has a more rigid splitting policy forced by non-erasability In particular, whenever a WOB-tree node needs splitting, a time split must be performed Sometimes a key split also occurs For the time split, the split time is always the current time A time split occurs whether or not the splitting reduces the number of records in the resulting current nodes

When we split by time in the TSB-tree, we may split by any convenient time AFTER the last time split In this case, the "older" versions of records are written into a node in the historical database while the newer versions are kept in the current database The versions of records that are valid across the split time must be present in both historical and current data bases

This redundancy makes it possible for records valid at a common time to be clustered in a small number of nodes Without such redundancy, regardless of what strategy is chosen for storing a long-lived record, some time based queries will be inefficient as the long-lived record cannot be stored near all records whose lifetimes overlap with it We give some examples of time splitting in Figure 2

2.4 Splitting Criteria

Current data being on a WMRM medium permits us to vary the split policy in the TSB-tree so as to improve its performance, i e reduce its cost A number of factors contribute to the cost of the method Two important ones are

Space Cost: the cost of space for current and historical databases

Expansion Cost: the cost per (version of) record added, in disk accesses, to expand the file

The kind of split chosen will depend on node contents If a node contains only current data, all of it must remain

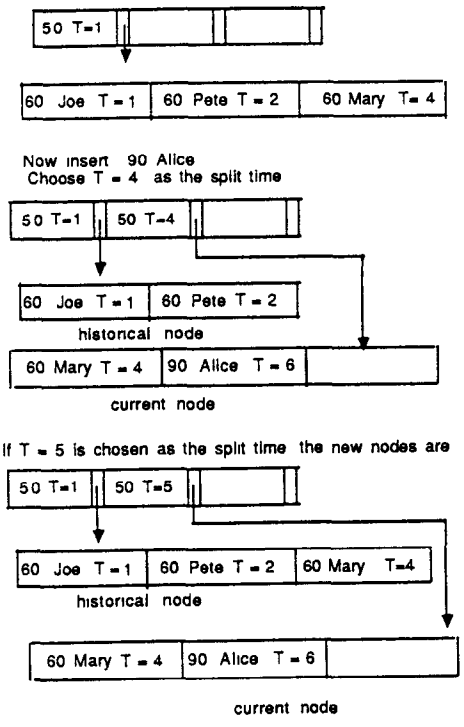

Figure 2 Time-Split B-tree time splits

in the current database Time splitting will be useless Key space splitting must be done (With the WOB-tree, time splitting always occurs, even when no current space is saved) If a node contains only repeated updates of a single record, all data is associated with the same key value and so cannot be key split Time splitting must be done

These boundary conditions suggest that the more historical data is in a node, the more likely it is that time splitting will be most effective The time splitting sweeps historical data out of the current database The more current data there is, the more likely key space splitting will be most effective The key splitting grows the index to the current database and does not introduce redundancy

2.5 Splitting Policies

This paper explores the consequences of the different forms of splitting, and the conditions under which they are employed We start with the splitting policy used for the WOB-tree, then introduce two additional policies with the intention of improving the performance or storage utilization

2.5.1 Write-Once B-tree Policy

The WOB-tree policy (WOB) is the policy used in the WOB-tree A TSB-tree using WOB policy when a data node overflows

1. always performs a time split, and uses the CURRENT time as the splitting time (The splitting node cannot be re-written in the WOB-tree, so this is done regardless of its effectiveness in reducing the current database size)

2. performs a key split whenever two thirds or more of the overflowing node consists of current data (This two thirds threshold is one of any number that might be employed We use it consistently with all our splitting policies See the discussion in section 6)

The WOB-tree is constrained to use this splitting policy because of its write-once medium The write-many medium of the current database in the TSB-tree permits a more flexible choice In particular, it permits us to reduce the amount of redundancy and hence the size of the historical database The WOB-tree uses the current splitting node as the historical node, in effect migrating it to the historical database Since it cannot be shrunk, there is no way for the WOB-tree exploit more flexible splitting policies

2.5.2 Time-of-Last-Update Policy

Suppose a number of insertions are done *after* the last update Choosing the split time to be the time of the last update (not the last insertion) avoids carrying these trailing insertions in the historical node because they do not live across the split time The contents of the resulting current node are not changed by this choice of split time, and remain at the minimum, i e it contains only current data, and no historical data

The TSB-tree using the time-of-last-update (TLU) policy

1. always performs a time split unless there is no historical data, and uses the time of last update as the splitting time

2. performs a key split whenever two thirds or more of the overflowing node consists of current data

Some of the current data may still persist across the split time But a wise choice of split time reduces the number of records that cross the split time boundary

If the split time is pushed back past updates as well as insertions, some historical data will end up in the current database This may still result in a smaller amount of redundant data overall as more data may be removed from the historical node than may be added to the current node But now, we are making a trade-off between amount of redundant data, and current database size We do not pursue this extension here

2.5.3 Isolated-Key-Split Policy

The preceding splitting policies always require that a time split be performed whenever a node overflows, unless there is NO historical data that can be removed from the node Sometimes, there is very little historical data that can be removed from an overflowing current node Expansion cost can be reduced if we do not force the creation of another historical node in these cases Further, redundancy is reduced as there are fewer split times for versions to live across Hence, fewer redundant copies of the data are generated

If we do only a key split under the same circumstances that we did a key split plus a time split previously, we do not increase the space for the current database initially, and only modestly eventually However, we dramatically reduce the number of time splits We call this the isolated-key-split policy (IKS) A TSB-tree following the IKS policy

1. performs a time split only when not doing a key split, and uses the time of last update as the splitting time

2. performs a key split whenever two thirds or more of the splitting node consists of current data

The number of key splits is slightly greater with IKS than with the other policies When current nodes also contain historical data, some key splits will occur at an earlier time than if that historical data had been swept into the historical database during prior splits Thus, while a node does not key split unless 2/3 of its records are current, it may be prevented from filling up with as much current data because of the presence of historical data

2.6 Notation

In the remainder of the paper, we make use of the following notation to denote the quantities of interest in our performance study

R	total non-redundant records
R_c	total current records
R_h	historical records (including redundant)
red	total redundant records
K	total distinct keys
N	total number of nodes
N_c	number of current nodes
N_h	number of historical nodes
k	number of distinct keys in a data node
r	number of records in a data node
l	insertions after last update in data node
b	current data node capacity (records)
p	probability of update
q	probability of insertion ($q = 1 - p$)

3. FRINGE ANALYSIS

We give an analysis of the TSB-tree for TLU and WOB policies like that made for B-trees in [BYLa, EZGMW] We characterize (or type) data nodes by their number of records, number of old versions (equivalently, the number of updates) and number of insertions since the last update Since there is a finite record capacity in each data node, the number of types of data nodes is finite

We model a TSB-tree with a vector indexed by the node types The sequence of random (vector) variables

$$X(R) = (x_1(R), x_2(R), \quad , x_m(R))$$

where $x_i(R)$ is the random variable representing the number of distinct keys in all current data nodes of type i after R records have been added to the database, is a Markov chain That is, $X(R)$ depends only on $X(R-1)$, not on the previous history of the process The study of trees by this Markov chain model of the (keys in the) leaves is called *fringe analysis*, where the leaves are the "fringe" of the tree

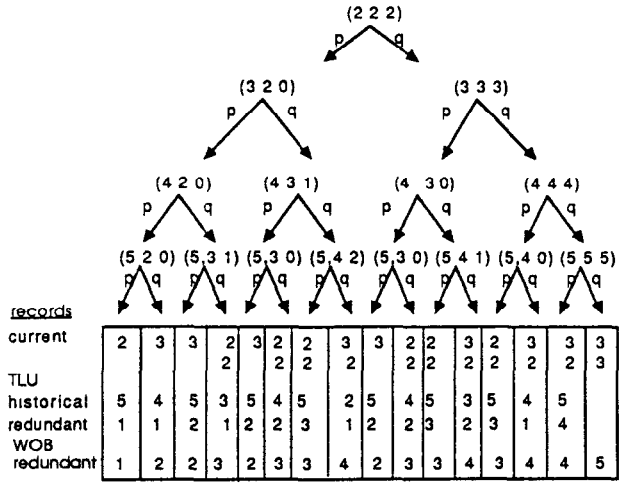

Figure 3 Node transition diagram

3.1 Transition Diagram

Figure 3 is a diagram of the transitions for the TLU and the WOB policies when $b = 5$ The current node types are indicated by three-tuples (r, k, l) The smallest number of records and keys at stable state in this case is $r = k = \lceil 5/3 \rceil = 2$

When a record is added to the database, it is an update with probability p and an insertion with probability q Figure 3 indicates this with arrows labeled by p for update and q for insertion For example, a node (3,2,0), with $r = 3$, $k = 2$ and $l = 0$ has 3 records, 2 distinct keys (that is, one of the records is an old version of another record in the same node), and the last record added was the update (the number of insertions since the last update is zero) If a record in a node of type (3,2,0) is updated, a node of type (4,2,0) results If a record is inserted (with a new key), a node of type (4,3,1) results

When a split occurs, because a record is added to a full node (of type (b, k, l)), the following occurs One new current node is created if k is 2 or 3 and the new record is an update, or if k is 2 and the new record is an insertion This is a "pure" time split Otherwise (when there are at least $4 = 2\lceil b/3 \rceil$ distinct keys including the new record), two new current nodes are created This involves a key split Except when there are no historical versions in a full node with the TLU policy, an historical node is always created in node splitting

For TLU and WOB, this characterizes the splitting process For IKS, the number of updates for each record is needed—this causes an explosion of node types Thus, IKS is studied only via simulation

3.2 Transition Equations

Transitions from one type of node to another occur whenever a record is added With uniform growth, it

is equally likely that keys of new records will fall in any interval between existing keys Each current data node *owns* a number of *intervals* equal to the number of keys in the node, k We let $f_R(r,k,l)$ be the probability that an interval lies in a node of type (r,k,l) when R new records have been added to the database This is also the probability that an *insertion* or an *update* (that is, any new record) will hit a node of this type

3.2.1 Non-Split Transitions

First, we treat transitions to node types that cannot be created by a split

Non-Split Transition Equation

$$(K+1)f_R(r,k,l) = p[(K+1)f_{R-1}(r,k,l)$$
$$-kf_{R-1}(r,k,l) + \delta_l' kf_{R-1}(r-1,k,*)]$$
$$+q[Kf_{R-1}(r,k,l) - kf_{R-1}(r,k,l)$$
$$+\delta_l kf_{R-1}(r-1,k-1,l-1)]$$

The left hand side of this transition equation is the number of intervals in nodes of type (r,k,l) after R records have been added to the database This is the total number of intervals, $(K+1)$, one more than the total number of distinct keys, multiplied by the probability that an interval is in nodes of this type

The right hand side of the equation has two major terms The first major term, with factor p, computes the change when the last record added is an update In this case, the total number of intervals before the last record is added does not change Thus the expected number of intervals in nodes of type (r,k,l) is $(K+1)f_{R-1}(r,k,l)$ before the transition

If the last record updated is in a node of type (r,k,l), then $kf_{R-1}(r,k,l)$ intervals of this type are lost because one node of that type is eliminated

If the last record updated is in a node with one less record and the same number of keys, then a gain in intervals of type (r,k,l) occurs The factor δ_l' indicates that this subterm is *present* only when l is zero (Note that l will be zero when the last record added is an update) The $*$ in this term indicates a wild card designating *any number of insertions since the last update*, since a new update changes l to zero Thus

$$f_{R-1}(r-1,k,*) = \sum_i f_{R-1}(r-1,k,i)$$

The second major term, with factor q, computes the change when the last record added is an insertion (a new key) In this case, there was one less interval in the database before the new record was added The loss if the insertion hits a node of type (r,k,l) is the same as in the case of an update

When the last record is an insertion into a node with one less record, one less key and one less insertion after the last update, a gain will occur The factor δ_l indicates that this subterm is *absent* when l is zero because l is zero only when the last addition is an update

Assume that a steady state probability vector is attained, i e , that

$$f_R(r,k,l) = f_{R-1}(r,k,l)$$

At steady state, therefore, if l is zero, we drop the subscript for f and obtain

$$f(r,k,l) = \frac{pkf(r-1,k,*)}{1-p+k}$$

At steady state for non-zero l, we have

$$f(r,k,l) = \frac{qkf(r-1,k-1,l-1)}{1-p+k}$$

3.2.2 Transitions Involving Splits

We now turn to the more complicated case, where the node type can also be created from a split of an overfull node This case only occurs for nodes where

$$\lceil b/3 \rceil \le r = k = l < 2\lceil b/3 \rceil$$

This is so because all splits sweep historical data from the current database We let $\delta_{\min}$ indicate that a term is zero when $r = \lceil b/3 \rceil$, the minimum number of records in a current node at steady state Then, a record addition will result in a transition as described below

Split Transition Equation

$$(K+1)f_R(r,r,r) = (\text{non\_split equation terms})$$
$$+p[rf_{R-1}(b,r,*) + \delta_{\min} rf_{R-1}(b,2r-1,*)$$
$$+rf_{R-1}(b,2r+1,*) + 2rf_{R-1}(b,2r,*)]$$
$$+q[rf_{R-1}(b,r-1,*) + \delta_{\min} rf_{R-1}(b,2r-2,*)$$
$$+rf_{R-1}(b,2r,*) + 2rf_{R-1}(b,2r-1,*)]$$

Nodes that can be generated by splitting can also be generated as a result of normal record addition where the node does not split Hence the non-splitting equation terms are also present here

A record update or insertion to a full node with less than $2\lceil b/3 \rceil - 1$ keys splits into one historical node and one new current node An update to a full node with *exactly* $2\lceil b/3 \rceil - 1$ keys has the same effect Otherwise two new current nodes and one historical node are created at split time

Briefly, let us consider splitting in the update case Updates occur with probability p, and hence the term prefixed with the factor p describes the update case

First, if the new record lands in a full node where the full node has the same number of keys as the resulting node, this represents a time split The expected gain in intervals of type (r,r,r), where $r = k$, is $rf_{R-1}(b,r,*)$

For a full node with $2r-1$ distinct keys, a key split results This creates two new current nodes for updates, as long as $2r - 1 \ge 2\lceil b/3 \rceil$ (That is, except when $r = \lceil b/3 \rceil$) One of the new nodes will have r keys and the other will have $r-1$ So the expected gain in the number

of intervals will be $rf_{R-1}(b, 2r-1, *)$ The reasoning for $2r+1$ keys is similar If there is a full node with $2r$ keys, an update will cause *two* new current nodes of this type to be created, with an expected gain of $2rf_{R-1}(b, 2r, *)$

We omit the similar arguments for the insert case

To obtain the steady state transition equations, we again eliminate the subscript and assume $f_R(r,k,l) = f_{R-1}(r,k,l)$ The result is quite complicated and we do not display it here We use these transition equations to create a matrix whose eigenvector for the largest eigenvalue is the vector of probabilities $\overline{f(r,k,l)}$ at steady state Multiple applications of the transition matrix to an arbitrary vector (non-zero) yields a sequence of vectors which converge (experimentally) to the eigenvector

3 3 Using the steady state solution to estimate space utilization

The steady state solution depends on the value of p, the percent of updates From the steady state solution, we can derive the number of current records, the number of historical records, the number of redundant records, and the number of nodes of each type, for a given number of records R added to the database

The number of total keys is $K = qR$ The number of keys in nodes of type (r,k,l) is $Kf(r,k,l)$ or the total number of keys times the probability that a key (or an interval) is in a node of this type To find the number of nodes of a type, we divide by k To find the total number of nodes in the current database, we sum these numbers over all current node types That is,

$$N_c = K \sum_{(r,k,l)} \frac{f(r,k,l)}{k}$$

Similarly, to find the number of current records, we use

$$R_c = K \sum_{(r,k,l)} \frac{rf(r,k,l)}{k}$$

These results apply for both TLU and WOB policies and are used to derive current space utilization

To derive the number of historical nodes for TLU, note that one historical node is created at each split of a full node except when it is an insertion into a node of type (b,b,b) That is, the number of historical nodes is

$$N_h[TLU] = R\Big(pf(b,b,b) + \sum_{(b,k\ l)\neq(b\ b\ b)} f(b,k,l)\Big)$$

The formula for the WOB policy is simpler

$$N_h[WOB] = R\Big(\sum_{(b,k\ l)} f(b,k,l)\Big)$$

This is the expectation of landing in a node which will cause a split when R records have been added to the database

When a record in a full node is updated, b records migrate to the historical node Only the new update is stored non-redundantly, only in the current database This is true for both TLU and WOB policies

For TLU, the number of records migrating to the historical node after an insertion to a full node is $b-l-1$ For WOB, there are still b records when there is an insertion Thus

$$R_h[TLU] = R\Big(pf(b,b,b)b + \sum_{(b,k,l)\neq(b,b,b)} f(b,k,l)(pb + q(b-l-1))\Big)$$

and

$$R_h[WOB] = Rb\Big(\sum_{(b,k,l)} f(b,k,l)\Big) = b \times N_h[WOB]$$

Last, we derive the number of redundant records If the new record added to a full node is an update, we expect $k-1$ redundant records in the new historical node If this record is an insertion, we expect $k-l-1$ redundant records in the historical node for TLU and k redundant records for WOB The total number of redundant records for the TLU policy is

$$red[TLU] = R\Big(pf(b,b,b)(b-1) + \sum_{(b,k,l)\neq(b,b,b)} f(b,k,l)(p(k-1) + q(k-l-1))\Big)$$

and the number of redundant records for the WOB policy is

$$red[WOB] = R\Big(\sum_{(b,k,l)} f(b,k,l)(p(k-1) + q(k))\Big)$$

Numbers of historical and redundant records in a split when $b = 5$ are given in the bottom of the diagram in Figure 3 We use ratios of the expressions in this section to obtain historical space utilization, percent of records which are redundant and so forth Note that the factor of R will cancel out in these ratios, that is, space utilization at steady state is independent of the total number of records

4. SIMULATION

4.1 Simulation Goals

Our intent in simulating the TSB-tree is to

1. confirm the steady state analytic results of the fringe analysis for the cases that were susceptible to analysis,

2. provide extrapolations of the analytic results to node sizes that could not be readily handled by the analysis simply because of a size explosion,

3 explore interesting splitting policies that proved difficult or impossible to analyze effectively In particular, fringe analysis was not employed with the IKS policy because of the explosion in node types and transitions

4.2 The Simulation Program

The detailed simulation effort resulted in a parameterized skeletal implementation of the TSB-tree Nodes are represented by vectors of records or index entries, timestamped appropriately All the logic to support the splitting policies described in section 2 5 is included Statistical information is accumulated during the execution trials of the TSB-tree model implementation

The skeletal implementation is driven by a program that generates random keys to be inserted or updated Insertions are drawn from a uniform distribution of keys The updates are, with equal probability to any one of the already existing keys The probability p that a record is being updated can be varied between zero and almost 100%, i e from all inserts to almost all updates Whether a particular record addition is an insert or an update is determined by whether another random number between zero and one is less than p or greater than p

A trial execution of the TSB-tree model implementation involves choosing the (i) split policy (one of WOB, TLU, or IKS), (ii) update probability p, (iii) total number of record additions R, (iv) key split threshold $T_k = k/b$ at which key splitting is to occur, and (v) node size b

The performance information reported for each run includes several forms of utilization, redundancy, record addition cost, and frequency of node types These are described in the next section

5. PERFORMANCE RESULTS

5 1 Introduction

We report asymptotic results Fringe analysis produces this directly The fringe analysis algorithm is executed until closure for each of its trial runs The detailed simulation confirms and extends the fringe analysis Each simulation trial made 50,000 record additions in an effort to provide a reasonable fit with the analysis

Both the analysis and simulations had difficulties reaching steady state behavior for very high update fractions, i e greater than 90% Except for these values, the results for simulation and analysis are the same to within 01 for all quantities

5.2 Single version current utilization

Single version utilization utilization tells us how effective our split policy is in minimizing the space for the current database The current database is stored on a write-many medium with cost per byte that is a factor of about ten times more expensive than space for the historical database on the write-once medium The single version current utilization U_{svc} is given by

$$U_{svc} = K/(N_c \times b)$$

The graph in Figure 4 plots U_{svc} for the full range of update probabilities, for nodes of size 11, and for the three split policies The results for multiple node sizes are tabulated in Table 1 in the appendix

What the figure and table both show for the three different split policies is that, as p increases, U_{svc} declines The end points for U_{svc} are explained as follows

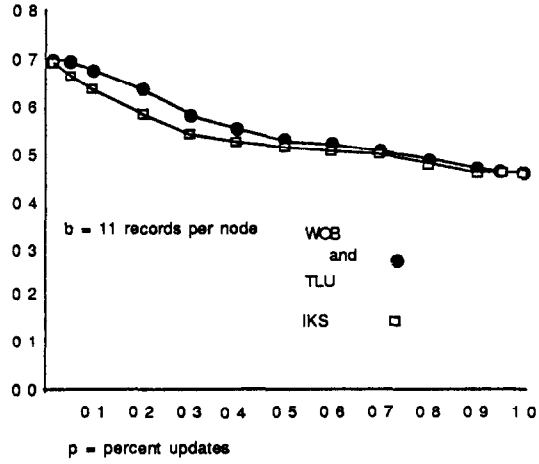

Figure 4 Single version current utilization

All Insertions [$p = 0$]: All policies behave as a regular B-tree behaves with respect to U_{svc} Generally, this utilization falls somewhat as node size b increases In the limit, as b increases, we expect $U_{svc} = \ln 2 = 0\ 693$

Almost All Updates [$p = 99$]: The maximum utilization, $U_{svc-max}$, becomes the key splitting threshold For most of our results, this threshold was set to 666 Hence, we expect U_{svc}, at high update rates, to be near $U_{svc-max} \ln 2 = 0\ 666 \ln 2 = 0\ 46$, and indeed that is what we see We experimented with thresholds of 0 5 and 0 833, and these cases yielded the anticipated results, i e 0 346 and 0 577

U_{svc} declines somewhat with increasing node size This is a well known phenomenon When a full node splits, its contents, PLUS the one additional record being added, are divided between the two resulting nodes The effect of this one extra record is significant at small node sizes but vanishes as nodes become large

WOB and TLU produce identical results for U_{svc} Both perform key splitting under identical situations The results for IKS are comparable at the extremes of p's range, but are up to about five percent less in the mid-range p This five percent utilization decline for IKS makes as much as a 10% difference in the space required for the current database This is the penalty for not sweeping historical data from the current database when a node is key split

5.3 Redundancy

The TSB-tree must copy versions of records that persist across the times used for time splitting We are interested in the fraction of records in the database that are duplicates This fraction redundant F_{red} is given by

$$F_{red} = red/R$$

The superiority of the IKS policy in reducing redundancy, and hence limiting the size of the historical database, is clearly demonstrated in Figure 5 Only at very high update fractions do the three policies converge to produce

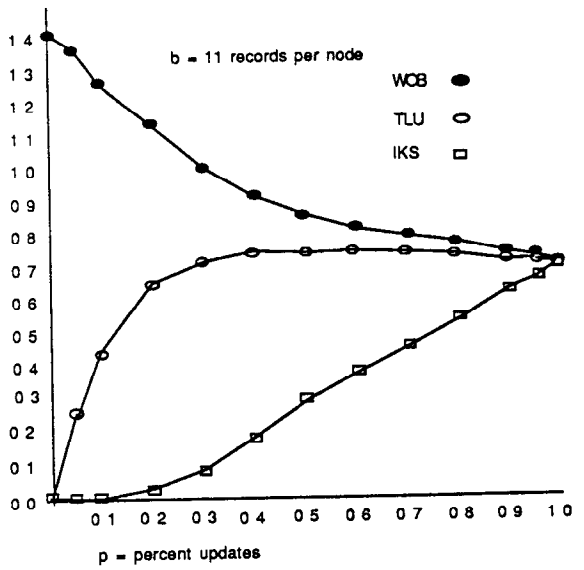

Figure 5 Fraction redundant

the same redundancy The redundancy for lower update fractions is dramatically lower with the IKS policy The WOB policy is by far the worst, demonstrating the limiting effect of the write-once medium At high update fractions, the time of last update optimization does not help much, since the last record addition to a node is almost surely an update

We can explain the results at the end points of the update fraction range, and hence understand the overall trends

All Insertions: F_{red} is zero for TLU and IKS as no time splits are done For WOB, F_{red} is about 1 4 In WOB, all current nodes were generated by a combined time and key-split Hence, each current node, on average, has left a half a node of historical data in its predecessor That predecessor has likewise left behind a half a node of data in its predecessor Hence, behind every current node there is $1/2 + 1/4 + \quad = 1$ node of historical data Each current node contains, on average, U_{svc} of current data Hence the fraction of redundant data is $1/U_{svc}$ This is independent of node size except as node size affects U_{svc} slightly (U_{svc} drops to 693 as node size increases)

Almost All Updates: All methods converge when p is large The kind of splitting performed by all the policies is largely pure time splits at these p values Further, the lack of insertions makes time-of-last-update the current time Hence, the policies produce approximately the same number of redundant records At each time split, no more than $U_{svc-max}$ of data is current and hence becomes redundant The records added between time splits is not less than $1 - U_{svc-max}$ Thus, F_{red} is bounded by $U_{svc-max}/(1 - U_{svc-max})$ In fact, F_{red} was close to $U_{svc}/(1 - U_{svc})$ except for very high key split thresholds Redundancy can increase without bound as the key split threshold approaches 100% Trials with higher thresholds

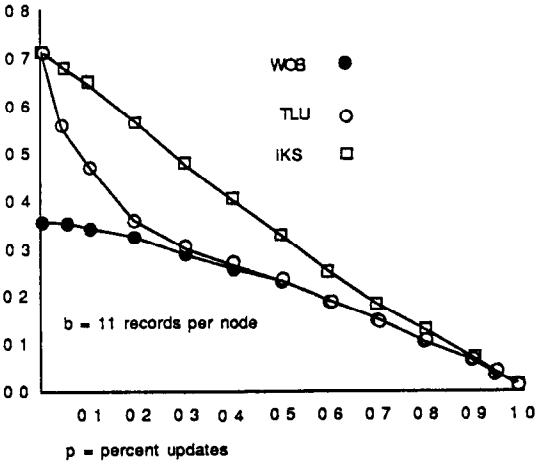

Figure 6 Single version total utilization

confirmed this

Table 2 provides data for a variety of node sizes and split policies The notable thing is the impact of increasing node size on redundancy We do not fully understand this However, the most important factor for redundancy in nodes with large numbers of records is the split threshold, not the node size

The TLU policy approaches the redundancy of the WOB policy as node size increases The number of records represented by trailing inserts remains approximately constant per node as b increases, and hence becomes an increasingly smaller fraction of the records of the node These are the records that do not need to be stored redundantly across a time split

5.4 Single version total utilization

Single version total utilization relates the cost of carrying multiple versions, in terms of total space consumed by these retained versions, to the storage needs for single version data This quantity is affected by the fraction of updates, well as how effectively the method utilizes storage and avoids redundancy The single version total utilization is

$$U_{svt} = K/(N \times b)$$

We graph our results for the three split policies for node size of 11 in Figure 6, with the results for other node sizes tabulated in Table 3 What we see is that U_{svt} tends to zero as the update fraction increases because the current data becomes an ever smaller part of the total database At low update factors (mostly inserts), the ability to perform isolated key splits clearly shows an advantage in holding down the size of the historical data base The dramatic difference between WOB and TLU policies results from time splitting frequently being ineffective TLU avoids the time split With WOB, the time split occurs regardless

Single version total utilization declines as node size increases due to the increase in redundancy as node size increases The change is dramatic only for small node sizes Larger nodes with the same split threshold have similar total utilizations

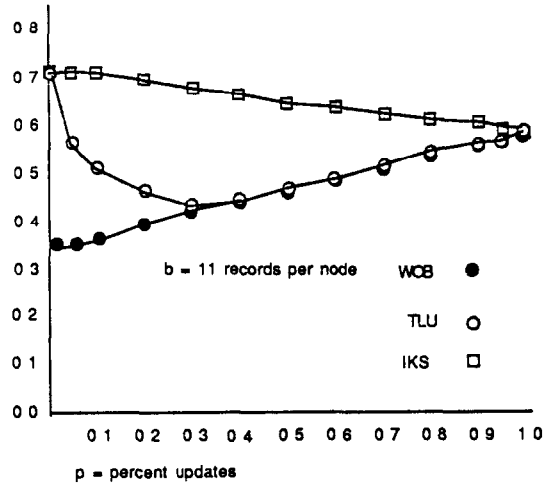

Figure 7 Multiple version utilization

5.5 Multiple Version Utilization

Multiple version utilization, U_{mv}, measures how effectively the TSB-tree, together with the particular split policy, support multiversion data This can be used to compare TSB-trees with other multiversion approaches It reflects the cost of maintaining the integrated index to the entire collection of versions, and the cost of storing redundant copies of the versions so as to support "as-of" queries Multiversion utilization is given by

$$U_{mv} = R/(N \times b)$$

Once again, we use node size of 11 as the case that we graph in Figure 7, with the remainder of the results tabulated in Table 4

The results here are consistent with our expectations in comparing the three split policies The WOB policy results in very substantial redundancy in the low update fraction cases, more than doubling the number of copies for these cases Note how well the IKS policy did At low update fractions, the utilization was comparable to having stored all the versions in a B+tree, without storing any versions redundantly That is, the redundant copies are being stored without compromising the storage utilization of the TSB-tree At higher update fractions, there is too much redundancy for this to happen Hence, the multiversion utilization trails off

This quantity is also sensitive to node size (especially for small nodes) because redundancy increases with node size Hence, U_{mv} decreases with increasing node size

5.6 File Expansion Cost

Storage is not the only cost for supporting access to data The performance of the record addition process is also important Here we treate the cost, in disk accesses per record addition, of the effect of the different policies on file expansion This expansion cost is derived from the frequency of the various kinds of node splitting that the three policies entail

We assume that an updated node needs to be written to disk as a result of record addition Hence, we exclude the cost of writing one node Further, we neglect the cost of non-leaf splits, which is very small What is included is the cost of writing the new nodes plus updating the index node that refers to these nodes

WOB: A split does not require re-writing the full node Rather, a new node (in the case of a pure time split) or two new nodes (in the case of a key and time split) are written, along with the parent index node Thus

$$\text{expand[WOB]} = (\text{timesplits} + 2 \times \text{time\&keysplits})/R$$

TLU: This policy requires a separate medium for historical data, and hence the current node cannot become the historical node The original full node must be re-written This policy can have three kinds of node splits Thus

$$\text{expand[TLU]} = (2 \times \text{timesplits} + 2 \times \text{keysplits} + 3 \times \text{time\&keysplits})/R$$

IKS: Like TLU, this policy requires two media and hence pays the larger node splitting cost Here, however, combined time and keysplits do not occur Hence

$$\text{expand[IKS]} = (2 \times \text{timesplits} + 2 \times \text{keysplits})/R$$

The above expansion costs all decline with increasing node size since the frequency of node splitting varies inversely with node size

All policies have equal expansion costs when the update factor is zero, i e only insertions are performed This is so because all splits are pure key splits (with WOB, the splits are ineffective time and key splits), the number of splits is the same and the cost of the splits are equal

As the update factor increases, the cost for the WOB policy declines with its decline in redundancy For TLU and IKS, expansion costs increase modestly with increasing update factor This is related to increased redundancy as update factor increases, resulting in a larger number of splits being required The TLU and IKS policies have higher expansion costs than WOB because the historical node never needs writing with WOB Expansion costs for nodes of size 11 are plotted in Figure 8, with selected values for a number of node sizes tabulated in Table 5

6. DISCUSSION

There are a number of points worth discussing that have not fit conveniently into the results that we have reported above

6.1 Index Node Time-Splitting

Our faithful simulation of the TSB-tree access method revealed an unexpected attribute of the method We did not believe that performing time splitting of index nodes would be necessary We knew it would be difficult for the TSB-tree as index terms in an historical node cannot

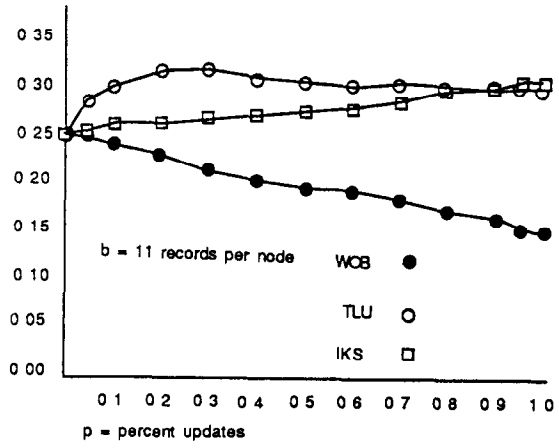

Figure 8 Expansion cost in disk access

refer to nodes in the current database since those nodes can move

We discovered the need for index node time splitting during trials with a node size of five The problem is that small nodes soon degenerate to having only a single current entry and its updates, all other entries having been removed by key splitting This makes further key splits impossible

There are basic two approaches to index node splitting

1 Find a split time at which historical index terms can migrate to an historical node without any current index terms ending up there as well This involves finding the oldest current index term and using its time as the split time This approach was adopted for the simulation, though not without having to deal with several subtle bugs in the splitting process

2. Force the descendent nodes of the index terms to split as of some time so that that time can be used as a split time This introduces additional redundancy, but might be useful for migrating/archiving the TSB-tree data as of the time chosen We did not pursue this approach

6.2 Other Key Split Thresholds

Only the 2/3 threshold value was used for purposes of deciding whether to time split or key split An interesting question is what happens when the threshold is varied The obvious happens, in terms of direction of effect Increasing the threshold makes key splitting less likely, hence increasing U_{svc} Because time splitting frequency is increased, redundancy is increased, and hence U_{mv} and U_{svt} are decreased File expansion cost also increases modestly Decreasing the threshold has the opposite effect

What threshold to use depends on what cost function one is attempting to minimize We discuss this below in the context of choosing a splitting policy Similar considerations apply to the threshold value The 2/3 threshold appears to be a decent compromise with tolerable performance over the spectrum of update factors

6.3 Some Conclusions

The purpose of a performance study is to assist in choosing design parameters so that users can optimize their application, with its particular characteristics One would like for this task to be simple, by finding techniques that are universally good regardless of application Unfortunately, the splitting policies studied here do not lend themselves to universal interpretations

Obviously, if the entire TSB-tree must reside on a WORM medium, then one must use the WOB policy The other policies are impossible in their vanilla form since they require that the full splitting node be reused for current data The beauty of the WOB policy is that it exploits the full node by transforming it into the historical node This makes a virtue of the necessity of the WORM medium However, on a WORM medium, it is clumsy to manage the current data and its representation tends to be very wasteful of space

If WMRM storage is available and should per byte WORM cost be more than a factor of ten less than WMRM storage cost, then using TLU may be interesting U_{svc} is always equal to the WOB policy's result, and always better than IKS by a modest, but potentially significant amount

If WORM storage is less than a factor of ten cheaper than WMRM storage cost, then the IKS policy is a good choice It gives up a modest amount of current utilization to gain a substantial reduction in redundancy, and hence in the size of the historical database Its record addition cost is modestly higher than WOB, but as good or better than TLU (or WOB when data must be migrated to the WORM medium) When the cost differential is less than ten, there is no update factor at which the TLU (WOB) storage cost is less than the IKS storage cost

Currently, it would appear that WORM devices are about a factor of ten less costly than magnetic disk Thus, the choice of policy is not clear

References

[AhSn] Ahn, I and Snodgrass, R , "Partitioned Storage for Temporal Databases," *Information Systems*, **13**, 4, 1988 pp 369-391

[BYLa] Baeza-Yates, R and Larson, P A , "Performance of B^+-trees with Partial Expansions," *IEEE Trans on Knowledge and Data Engineering*, **1:2**, June 1989, pp 258-257

[East] Easton, M , "Key-Sequence Data Sets on Indelible Storage," *IBM J Res Develop* , **30:3**, May 1986, pp 230-241

[EZGMW] Eisenbarth, B , Ziviani, N , Gonnet, G , Mehlhorn, K and Wood, D , "The Theory of Fringe Analysis and Its Application to 2-3 Trees and B-Trees," *Inform Contr* , **55**, 1982, pp 125-174

[JeMR] Jensen, C S , Mark, L , and Roussopoulos, N , "Incremental Implementation Model for Relational Databases with Transaction Time," University of Maryland UMIACS-TR-89-63 CS-TR-2275 July, 1989

[LoSa] Lomet, D and Salzberg, B , "Access Methods for Multiversion Data," *Proc ACM SIGMOD*, Portland, 1989, pp 315-324

[McKe] McKenzie, E , "Bibliography Temporal Databases," *SIGMOD Record*, **15 2**, Dec 1986, pp 40-52

[SeSh] Segev, A and Shoshani, A , "Logical Modeling of Temporal Data," *Proc ACM SIGMOD*, May 1987, pp 454-466

[SnAh] Snodgrass, R , and Ahn, I , "A Taxonomy of Time in Databases," *Proc ACM SIGMOD*, March 1985, pp 236-246

[Ston] Stonebraker, M , "The Design of the POSTGRES Storage System," *Proc 13th VLDB Conference*, Brighton, 1987, pp 289-300

Appendix: Tables of Results

Table 1 Single Version Current Utilization

Policy	Node Size	Update Probability 01	10	30	50	70	90	99
WOB	5	74	70	61	55	51	48	47
	11	71	67	59	53	50	48	46
	17	70	67	58	52	49	48	47
	35	69	66	57	52	51	48	46
TLU	5	74	69	61	55	50	48	46
	11	71	67	59	53	50	48	47
	17	70	66	59	52	49	47	46
	35	69	66	57	52	49	46	50
IKS	5	73	65	57	53	50	48	47
	11	70	64	55	52	50	47	46
	17	69	63	54	52	49	47	45
	35	69	63	53	52	50	47	45

Table 2 Fraction of Records Redundant

Policy	Node Size	Update Probability 01	10	30	50	70	90	99
WOB	5	1 37	1 22	96	76	61	49	43
	11	1 39	1 27	1 00	85	79	73	71
	17	1 41	1 26	1 00	87	86	81	79
	35	1 42	1 28	1 00	90	93	90	88
TLU	5	02	20	44	51	50	46	42
	11	06	43	71	73	73	71	67
	17	08	61	82	80	82	81	80
	35	17	89	92	87	92	90	92
IKS	5	00	00	06	15	26	37	43
	11	00	00	09	27	44	61	68
	17	00	00	10	32	51	70	77
	35	00	00	10	39	60	79	85

Table 5 Expansion Cost per Record

Policy	Node Size	Update Probability 01	10	30	50	70	90	99
WOB	5	54	52	47	41	36	31	29
	11	25	24	22	20	18	16	15
	17	17	16	14	13	12	11	11
	35	08	08	07	06	06	06	05
TLU	5	55	59	63	52	60	58	57
	11	26	30	32	31	31	31	30
	17	17	21	21	21	21	21	21
	35	09	11	10	10	10	11	11
IKS	5	54	55	56	57	58	57	58
	11	26	26	27	28	30	31	31
	17	17	17	17	19	20	21	21
	35	08	08	09	10	10	11	11

Table 3 Single Version Total Utilization

Policy	Node Size	Update Probability 01	10	30	50	70	90	99
WOB	5	37	35	30	24	17	06	01
	11	35	34	29	23	15	06	01
	17	35	34	29	23	14	05	01
	35	35	33	29	22	14	05	01
TLU	5	72	55	36	26	17	06	01
	11	67	46	30	23	15	06	01
	17	64	41	29	23	15	05	01
	35	60	36	29	22	14	05	01
IKS	5	73	65	50	35	21	07	01
	11	70	64	48	32	19	06	01
	17	69	63	47	31	18	06	01
	35	69	63	47	30	17	05	01

Table 4 Multiple Version Utilization

Policy	Node Size	Update Probability 01	10	30	50	70	90	99
WOB	5	37	39	43	48	56	64	70
	11	35	37	42	46	50	56	59
	17	35	37	41	46	48	53	56
	35	35	37	41	44	47	51	53
TLU	5	72	61	51	51	56	65	70
	11	67	51	43	46	51	56	60
	17	65	45	42	45	49	53	55
	35	60	40	41	44	47	51	52
IKS	5	74	72	71	70	69	70	69
	11	71	71	68	64	61	60	59
	17	70	70	68	62	59	57	56
	35	70	70	67	60	57	54	54