ELSEVIER

Data & Knowledge Engineering 29 (1999) 1–15

DATA & KNOWLEDGE ENGINEERING

An adaptive split policy for the Time Split B-Tree

Laura Amadesi[b], Fabio Grandi[a,*]

[a]Dipartimento di Elettronica Informatica e Sistemistica, Università di Bologna, Viale Risorgimento 2, I-40136 Bologna, Italy

[b]Bain & Company, Piazza Ungheria 6, I-00198 Rome, Italy

Received 19 August 1997; revised 20 August 1997; accepted 11 June 1998

Abstract

The Time Split-B Tree is an efficient storage and access structure proposed by Lomet and Salzberg for transaction-time temporal data. This structure forces strong data duplication, due to storage space splits along the time dimension, to improve selectivity for temporal queries. However, redundancy increases storage space requirements and, as a consequence, can be considered as a weakness for a large class of transaction-time database applications. From this viewpoint, almost all the merit figures of the Time Split B-Tree suffer from the high redundancy degree usually reached by the structure during its lifetime. An improvement of the Time Split B-Tree is presented in this work. It is based on a new adaptive split policy which aims to dynamically limit data duplication within given bounds. An evaluation of the Time Split B-Tree with the adaptive split policy has been done via extensive simulations in comparison with the previous Time Split B-Tree performance. The Time Split B-Tree with the new split policy proved to have a better and more uniform behaviour. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Temporal database; Transaction-time; Storage and access structures; Time Split B-Tree; Data redundancy; Split policies

1. Introduction

Many application areas (e.g. CAD/CAM/CIM, financial, legal and medical records) require non-deletion policies and support of data versioning along time. One time notion of interest in this context is *transaction time* [2], which is system-managed and represents the period in which data are considered *current* in the database. A transaction-time database [2,8] maintains not only current records but also past record versions generated by updates.

The time and version management in a transaction-time database is system-managed and completely transparent to the user. Almost standard (i.e. non-temporal) applications using standard query languages, like SQL-92, run in the database accessing current data only. Temporal applications,

* Corresponding author. E-mail: fgrandi@deis.unibo.it

0169-023X/99/$ – see front matter © 1999 Elsevier Science B.V. All rights reserved.
PII: S0169-023X(98)00030-5

exploiting the transaction-time database *rollback* facilities, can also be executed, for auditing purposes, by database administrators or special users from time to time. Therefore, the past version maintenance and access support should not dramatically affect the system response to standard applications. Users may ignore the temporal nature of the database as long as its performance is not heavily influenced.

Notice that, due to the non-deletion policy, the number of versions of each data object stored in the database is continuously increasing in time. For instance, if ten or one hundred different versions are present for each data item, on average, the secondary storage space required for the complete database is consequently, in the absence of *redundancy*, ten or one hundred times larger than in the non-temporal case. This is the main implementation problem for a temporal database, as the explosion of the amount of data to be managed harms performance and forces heavy storage space costs. In this framework, temporal indexing is required in order to make the temporal applications *feasible*, by avoiding the exhaustive searching of a huge and fast growing database.

Several index and storage structures for temporal data have been proposed in the literature. Reference [7] is an excellent survey on temporal access structures, also including the Time Split B-Tree proposed by Lomet and Salzberg [3,4,5] for transaction-time databases. The Time Split B-Tree is an integrated index structure for current and historical data. The historical database consists of out-of-date versions of current records, which migrate when index nodes split. When a node split involves the time dimension, copies of all the current records are both maintained in the current node and also copied in a historical node, giving rise to *redundancy*. As shown in [4], redundancy, which may be higher than 100%, severely affects the Time Split B-Tree performance, in terms of secondary storage utilization (and, consequently, of access costs) and maintenance costs.

It should be clear that, if the high-rate data growth is the main problem in a temporal database, the introduction of data duplication results in further harm. If in a spatial database, for example, quite a high degree (e.g. about 70%) of duplication can be well-tolerated as a trade-off to improve index selectivity [6], the bad impact of redundancy on performance is increasingly *amplified* in a temporal database as time elapses, due to the steady data growth rate.

In this work we present an improvement of the Time Split B-Tree, consisting of an adaptive split policy that enables maintaining the data duplication degree within desired bounds, fixed by the user on his/her necessities. In the second part of the work we will show that the new policy is really effective in controlling redundancy, and how it improves the overall performance of a database adopting the Time Split B-Tree organization.

The paper is organized as follows. The Time Split B-Tree organization is briefly introduced in Section 2. In Section 3 the new adaptive split policy is presented. Section 4 is devoted to performance evaluation of the Time Split B-Tree with the new policy: experimental performance figures obtained via simulations are matched against the Time Split B-Tree behaviour with the old policies reported in [4]. Finally, our conclusions are drawn in Section 5.

2. The Time Split B-Tree

Time Split B-Tree (TSB-Tree) is an access method for multiversion records that clusters data by key value and transaction time. Each record version is time-stamped with the commit time of the transaction that inserted it. The TSB-Tree gradually makes data migrate from a current database

(maintained on a rewritable and fast medium like a magnetic disk) to a historical database (possibly on a WORM and slower medium like an optical disk) when nodes split, while it provides a single unified index to all multiversion data. Data are stored in the leaves of the TSB-Tree, whereas index nodes contain triples consisting of a time value, a key value and a pointer to a lower-level tree node. The time and key values represent, respectively, the lowest transaction time and the lowest key of all the data stored in the data space portion associated with the pointed sub-tree. There are three kinds of data space split allowed when a data node overflows:

Key Split (K-Split). It is a split on the key, like the splits in a standard B^+-tree. It is the only one applicable in a node which only contains current records and which overflows after the insertion of a new record. It implies the creation of one new node and does not produce archival nodes. It does not introduce redundancy at all as it does not require data duplication. It does not improve index selectivity on transaction time.

Time Split (T-split). It is a split on transaction time. It is the only one applicable in a node which only contains versions of a single record and the overflow is due to an update of the same record. It implies the creation of one new node and the production of one archival node. It requires the duplication of all the record versions which intersect the time point chosen for the splitting. It does not improve the index selectivity on the key.

Key and Time Split (KT-split). It is a combination of a K- and T-split. It is applicable in an overflowing node containing more than one version of more than one record. It implies the creation of two new nodes and the production of one archival node. It requires the duplication of all the record versions which intersect the time point chosen for the splitting. It improves index selectivity on the key and on the time.

In each case, an index term describing the split is posted to the parent index node. Index nodes can also be split, resulting in a modification of the tree structure (see [3] for details). Different policies can be adopted for the choice of the kind of split, and different choices affect the performance of the index. In particular, three policies have been considered in [4]:

Write-Once B-tree (WOB) Policy. When a data node overflows:
- It *always* performs a time split, and uses the current time as the splitting time;
- It performs a key split whenever two thirds or more of the overflowing node consist of current records.

Time-of-Last-Update (TLU) Policy. When a data node overflows:
- It *always* performs a time split unless there is no historical data and uses the time of the last update as the splitting time;
- It performs a key split whenever two thirds or more of the overflowing node consist of current records.

Isolated-Key-Split (IKS) Policy. When a data node overflows:

- It performs a time split *only* when not doing a key split and uses the time of the last update as the splitting time;
- It performs a key split whenever two thirds or more of the overflowing node consist of current data.

When a key split is done, the splitting key is chosen as the *middle* key of the node, as in a standard B^+-tree.

The TSB-tree behaviour with a given split policy has been evaluated under the following assumptions:

Uniform Growth. A new record has the same probability of being between any two records already present in the key space, so its probability of being inserted into a node is proportional to the number of different records in that node.

Equal Probability. Each different record has the same probability of being updated, so each of them have almost the same number of versions.

All the simulation results presented in [4] for each of the split policies considered show good performance figures. The only problem evidenced for the TSB-tree is that, since a high number of time splits must usually be effected, the redundancy may reach very high levels.

3. A new adaptive split policy

An adaptive split policy should take advantage of the freedom of choice of the split parameters—type of split, splitting key and/or time—in order to provide the structure with a great ability to adapt itself to the distribution and the dynamics of data, allowing punctual optimization of data storage and index efficiency. In this section we present our new **Adaptive Split Threshold (AST) Policy** which can be used for the TSB-tree. The purpose of the policy is to control data duplication within a fixed range, which is usually chosen to keep redundancy low in order to improve the transaction-time database overall performance. This is effected by trying to avoid or delay time splits, which create data redundant copies.

It should be noticed that, in general, among all the current records in an overflowing node, the ones updated by the present transaction only contribute to improving index selectivity, as their old version is put in the historical node while the new version is put in the new current node. All the other current records (which are very likely all but one) contribute to redundancy, as they are both stored in the historical and current nodes, and do not improve index selectivity in any way. Index selectivity for time queries is indeed improved by the presence of past record versions, which are stored in the archival node only. Therefore, delaying T-splits as long as possible has also a *positive* influence on index selectivity on time, as it tries to maximize the ratio between past and current records in a current node. However, the same adaptive policy can also be used, for special purposes (e.g. to privilege some kind of queries requiring high time selectivity), to obtain a high degree of duplication.

In any case, the policy is based on the management of two thresholds for the choice of the kind of

split, which are changed in an adaptive way, according to the data and update characteristics. The policy can be formalized as follows:

- Two thresholds (Θ_T, Θ_K) are used for the choice of the kind of split in case of overflow owing to an update or insertion. Each threshold is given in terms of percentage of different keys with respect to the total number of records stored in the node subject to overflow.
- If, at the time of the split, the percentage of different records (i.e. records with different key values) in the destination leaf is under the lower threshold Θ_T, it means that there is a large number of versions of the same records. In this case we operate a T-split, in order to relegate in an archival node past versions, which are never updated again.
- On the other hand, if there are so many different records that their percentage with respect to the capacity of the leaf exceeds the upper threshold Θ_K, the choice is to make a K-split. In this case, there are so many different records that it is better to separate them into two new current nodes, in order to delay the next overflow as long as possible.
- Otherwise, if the same percentage is in the range between the two thresholds, it should be advantageous to operate a KT-split, because there are enough past record versions to make the creation of an archival node profitable, and there are also enough different records to make their separation into two new current nodes profitable too.

We can simply summarize the new policy with the following algorithm:

```
Using the TSB-tree,
  find the data node in which the new record (version)
  should be inserted;
r := total number of record versions in the node;
If (r < b)
Then Insert the record in the node
Else Begin
  k := number of different records in the node;
  If (k/b ≤ Θ_T) Then Perform a T-split;
  Else If (k/b ≥ Θ_K) Then Perform a K-split;
    Else Perform a KT-split;
  End;
```

Moreover, in the case of a split involving the key (K- or KT-split), the splitting key is chosen as the *median* between the values contained in *all the record versions* to be stored in the new current nodes. In this way we try to optimize the overall node occupation, maintaining the occupancy around 50% in the new resulting leaves even in the presence of skewed key distributions and uneven relative update frequencies. In the case of a split involving time (T- or KT-split), the splitting time is chosen as the *current* transaction time, in order to leave a 100% occupancy in the archival node, which will never be updated again.

By changing the values of the split thresholds, it is possible to tune the organization to fit the application requirements on redundancy. In fact, time splits necessarily introduce redundancy as all records which persist through the splitting time must have copies in two nodes: the new current node and the archival one. Nevertheless, time splits cannot be avoided unless we abandon the requirement

of data clustering by transaction time, which only enables the TSB-tree to provide access selectivity on multiversion data. In general, an optimal choice of the threshold values represents a trade-off between redundancy and desired index selectivity. As a heuristic approach to this problem, an adaptive threshold adjustment mechanism has been embedded in the TSB-tree in order to control data duplication within a user-defined band. Every time there is an insertion of a new record or an update of an already present record, the current duplication F_{red} is evaluated as the ratio between the total number of (physical) record versions in the nodes (including duplicates) and the number of different (logical) record versions inserted. If this figure is not included in the desired range, the thresholds are automatically changed (they are increased when the redundancy is too low, decreased otherwise). The exact mechanism is given by the following algorithm:

```
/* [F_min .. F_max] = user-defined redundancy range */
If (F_red < F_min)
Then Begin
  If (Θ_T < Θ_max) And (Θ_K ≥ Θ_min + Δ)
  Then Θ_T := Θ_T + θ;
  If (Θ_K < Θ_max)
  Then Θ_K := Θ_K + θ;
  End
Else If (F_red > F_max)
Then Begin;
  If (Θ_K > Θ_min) And (Θ_T ≤ Θ_max − Δ)
  Then Θ_K := Θ_K − θ;
  If (Θ_T > Θ_min)
  Then Θ_T := Θ_T − θ;
  End;
```

In this algorithm, θ is a percentage (of different records in a bucket with respect to the total bucket capacity) that is added (or subtracted) to the thresholds. In our implementation we used a value of 5% for this parameter. While Θ_{min} and Θ_{max} represent the minimum and maximum values that the two thresholds can reach (set to 5% and 95%, respectively, in our implementation), Δ is a maximal distance that must be maintained between the two thresholds (the value we used is 30%), in order to enable KT-splits. The algorithm is designed in order to make the gap between the thresholds shrink to under Δ only when one of them reaches an extreme value (Θ_{min} or Θ_{max}) and to leave it unchanged (i.e. equal to Δ) otherwise. Thresholds are jointly moved in this case. The values of all the constants involved have been chosen to obtain an almost 'optimal' behaviour and their values have been tuned through extensive simulation. However, their values could also be changed in order to fit different specific users' aims.

4. Performance

The performance of the TSB-tree with the AST policy has been evaluated in comparison with the 'traditional' TSB-tree by means of simulation experiments. Hence, the same merit figures and

Table 1
The symbols used

R	Total non redundant records
R_c	Total current records
red	Total redundant records
F_{red}	Fraction of redundant records
K	Total distinct keys
N	Total number of nodes
N_c	Number of current nodes
k	Number of distinct keys in a data node
r	Number of records in a data node
b	Current data node capacity
p	Probability of update

notations introduced in [4] have been used here (see Table 1). Most of our experiments were based on the same **uniform growth** and **equal probability** assumptions introduced for the TSB-tree (see Section 2) but other experiments were explicitly designed to test the structure behaviour when such assumptions are not justified. In particular, we effected experiments where the key values of the records to be inserted or updated were drawn from a very skewed distribution (Zipf's distribution [9]). In this case, the arrival of new records and of new record versions due to update activity is very unevenly distributed over the key space: most of the records have one or two versions only, while a very few records have the rest (i.e. hundreds or thousands of versions each). Therefore, partitioning and clustering techniques based on uniform data space splits usually fail in introducing key selectivity. In our experiments, we did not find substantial differences in the performances with respect to the uniformity case. This is mostly due to the *median* strategy for the partitioning key choice, which really cancels the effects of skewness in data distributions. Although this canceling behaviour is, in principle, local to the split occurrences in the data space, the simulation trials evidenced how a global benefit is reached. Since the performance of the AST policy is almost the same with uniform or skewed data, we only consider uniformity in the following, for direct comparison with original TSB-tree results. It should be kept in mind that the 'traditional' TSB-tree performance may degrade in the presence of skewed data as node under-utilization may occur.

The comparison between the adaptive and the traditional TSB-tree was made taking into account all the split policies introduced for the TSB-tree in [4] and sketched in Section 2, even if the most significant comparison concerns the TSB-tree with the IKS policy, which is the most similar to the one introduced here and provides on average the best 'traditional' TSB-tree performance.

The choice between insertions of new records or updates of already present records is determined in simulation experiments according to an update probability p, which was varied in the range 0–100%, in order to analyze the TSB-tree behaviour under different input characteristics. All the effected simulation trials:

- insert 50,000 record versions;
- use a current node capacity $b = 35$;
- try to maintain the redundancy between 10% and 15% (by means of the AST policy).

The values chosen for the number of inserted records per experiment and for the node capacity are the same used in [4] for the original TSB-tree evaluation, in order to obtain immediately comparable results.

All the simulation results briefly displayed and commented below have been obtained by repeating the experiments three times, and computing average values. In the figures which follow, 'AST' stands for the TSB-tree with the new split policy, whereas 'IKS', 'TLU' and 'WOB' represent the TSB-tree with the corresponding split policy. Performances of the 'traditional' TSB-tree are directly derived from experimental data reported in [4].

Table 1 summarizes the symbols used for TSB-tree characterization.

We did not consider deletions, since most 'problems' in a temporal database are only connected with a fast data growth, due to very frequent insertions and updates. A faster data growth more heavily emphasizes redundancy shortcomings. However, if the deletion rate is low with respect to the sum of the insertion and update rates, the behaviour of the TSB-tree is not particularly affected. On the other hand, in a transaction-time database with a very high deletion rate, the duplication problem is less dramatic: the deletions improve the ratio between past and current records in a current node and, thus, enable T-splits to cause lower redundancy and higher index selectivity. In this work we chose the same evaluation framework adopted in [4] for the TSB-tree, which does not consider deletions; a more complete analysis is, thus, beyond the scope of the present paper.

4.1. Split kind distribution

Several simulation trials were effected to test the adaptive split policy and to find the optimal configuration of split kinds reached under different work conditions.

It should be kept in mind that a high proportion of K-splits maintains redundancy at a very low level but requires the use of a large number of current nodes, since it does not move past versions in historical nodes. On the other hand, a high percentage of T-splits enables an efficient indexing of versions along time and makes it possible to organize data in a modular way, keeping only the most recent information on the fastest media, but it promotes record duplication. The KT-splits, as a combination of T- and K-splits, combine the pros and the cons of the two.

The effects of the adaptive split policy can be seen in Fig. 1, that shows the experimental distribution of the kind of splits versus varying update probabilities in a steady state (i.e. after the transient in which thresholds are moved). The number of T- and KT-splits is very low for update percentages under 70% due to the strict constraint (<15%) imposed on duplication. Almost only T-splits are indeed effected for extremely high (>90%) update percentages. In this case, current buckets only contain a few different records (one or two) in a steady state along with their numerous non-redundant versions. In this case, the duplication introduced by T-splits is 'minimal' and non-dangerous for redundancy constraint violation, as can be seen in the next chapter.

4.2. Redundancy

Any temporal structure creates copies of the same records when a temporal split occurs. The fraction of duplicated records is an important factor to evaluate the performance of an access structure for temporal data and is defined as:

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

where *red* is the total number of redundant records. As we can see in Fig. 2, in all cases but WOB policy, F_{red} is almost null for small values of p, because there are no temporal splits. However, when

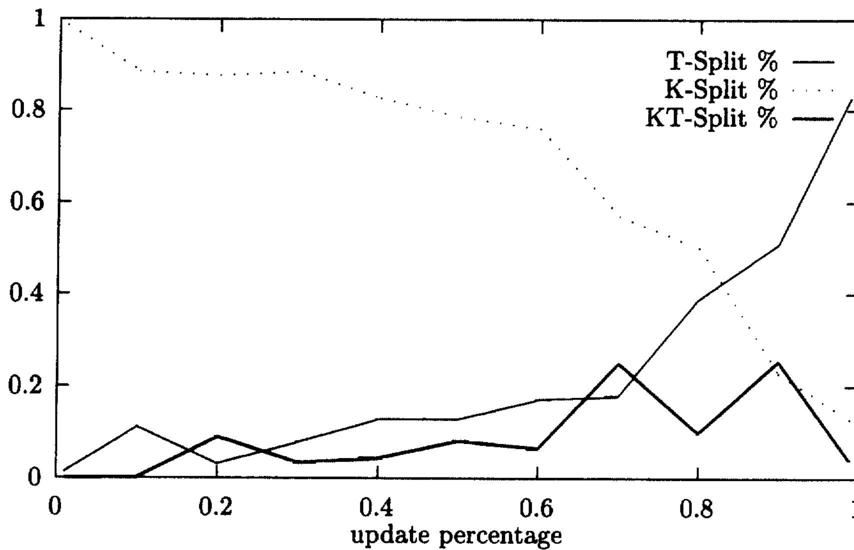


Fig. 1. Split kind resulting configuration against update probability.

p is increased also F_{red} increases with IKS and TLU policies, because we have a key split only when two thirds or more of the split node consists of current records (of course, this situation becomes more unlikely as p increases). On the contrary, it can be verified from the figure that redundancy F_{red} is always maintained between 10% and 15% with the adaptive split policy.

Other experiments showed how the AST policy is always effective in (asymptotically) approximating any user-defined redundancy range, since the adaptive mechanism acts as a negative-feedback on redundancy. In case the fixed range cannot physically be reached (e.g. in extreme cases like $p < 1\%$ or

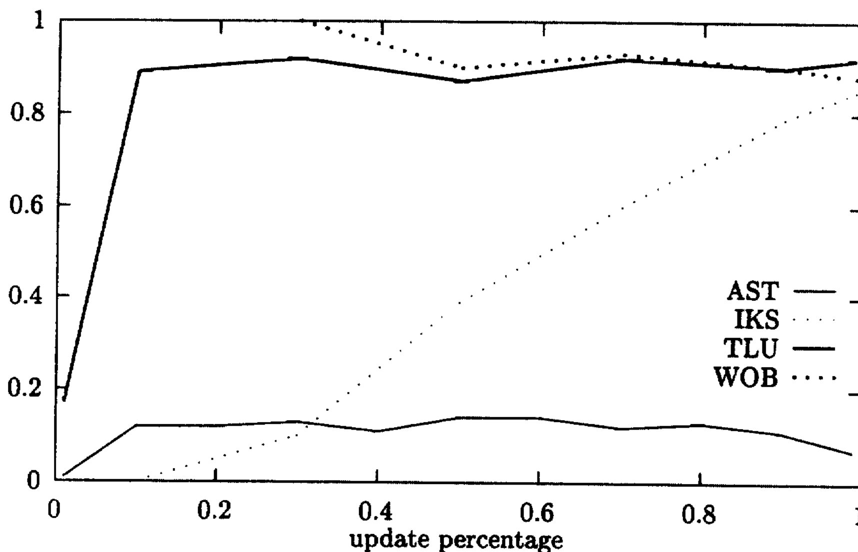


Fig. 2. Fraction of redundant records (F_{red}).

$p > 99\%$), the AST policy produces the feasible redundancy closest to bounds of the user-defined range.

4.3. Secondary memory requirements

The total storage space needed for data in terms of number of nodes is another very important parameter, on which redundancy has a direct impact. Assuming each node to be stored in one disk block, Fig. 3 represents the disk space required by the TSB-tree under the different split policies. It can be observed from the figure that for small values of p the IKS policy and the adaptive split policy have almost the same behaviour, whereas other policies require much more space. When p exceeds 50%, the figure shows how also the IKS solution behaves worse than the adaptive one. Obviously, this is due to the high redundancy that can be found in a 'traditional' TSB-tree when the percentage of updates increases: the many more records (copies) that have to be arranged require a larger number of nodes to be created. Notice how about one half of the space allocated on disk by the TSB-tree with the WOB and TLU (when $p > 0.1$) policies is due to redundant data storage. The TSB-tree with the AST policy appears to be almost less expensive in memory requirements, with quite a uniform behaviour when p is varied.

4.4. Single version current utilization

Single version current utilization shows how effective a split policy is in minimizing the space for current data. It is easy to understand how this merit figure is even more important in an access structure where the current data are privileged and stored on more expensive media. This parameter (called U_{svc}) is given by:

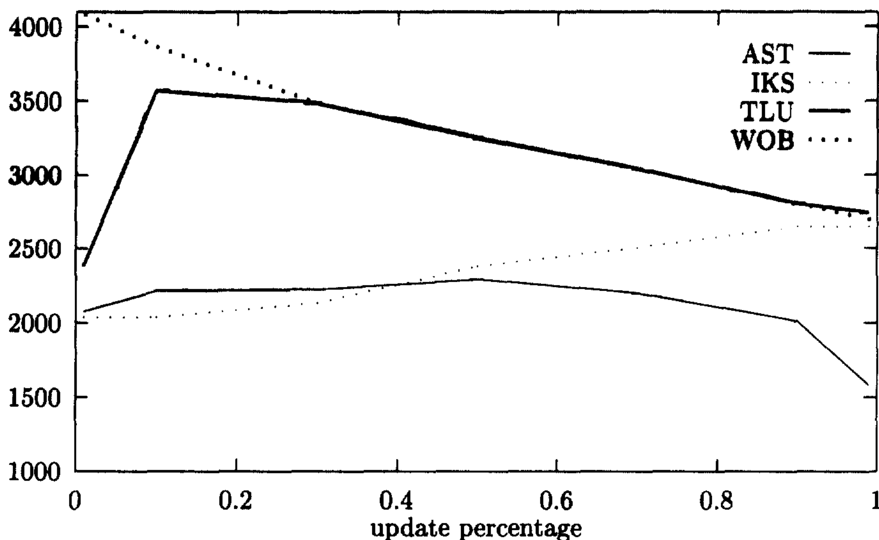


Fig. 3. Total number of nodes (N).

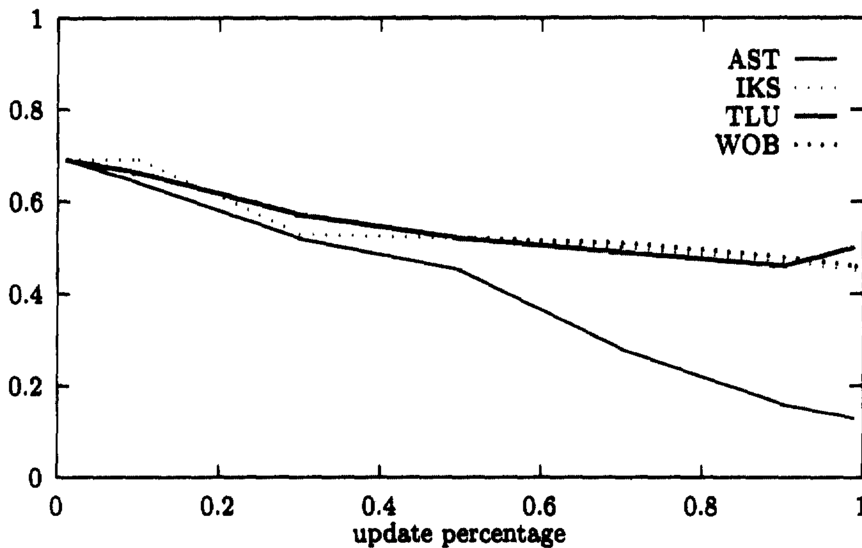


Fig. 4. Single version current utilization (U_{svc}).

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

From the graph in Fig. 4 we can observe that, in every case, U_{svc} decreases when p increases, because the number of different records becomes smaller. With the 'traditional' policies this decrease is not as great as with the AST policy because more T-splits are done, keeping the number of current nodes lower. When only insertions are effected, the TSB-tree behave like a regular B-tree, keeping U_{svc} at almost 0.69. Obviously, the performance under this aspect could be improved by accepting a higher value for the redundancy.

4.5. Single version total utilization

Single version total utilization shows the storage needed for single version data, relating it to the total space required by all record versions (redundant or not, past or current). Single version total utilization is given by:

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

and its behaviour may be seen in Fig. 5. There is no apparent difference between the AST and the IKS policy: when almost only insertions are performed, they both behave like a B-tree because of the small size of the historical data. If the update probability increases, then U_{svt} tends to zero in all cases, because the current data becomes an even smaller part of the total storage space. With TLU or WOB policies, U_{svt} values are smaller because more nodes are necessary to store all the data owing to the higher duplication.

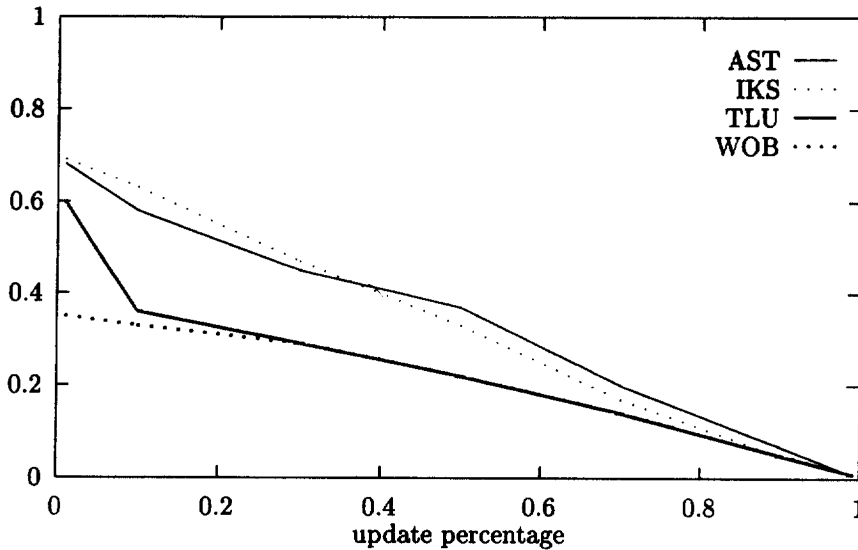


Fig. 5. Single version total utilization (U_{svt}).

4.6. Multiple version utilization

Multiple version utilization is a very important parameter because it measures how efficiently a storage structure supports multiversion data indexing. It is defined as:

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

and shows the amount of memory space used to store the different records inserted with respect to the space cost paid for storing the redundant records created by time splits. As shown in Fig. 6, the behaviour of the AST policy is always better than TLU or WOB policy for any p value. The TSB-tree with the AST policy also behaves in quite a similar way to the TSB-tree with IKS policy while p is under 50%. For higher update percentages, the new performance improves because of the higher fraction of redundant records which lowers the memory utilization also for the IKS policy.

4.7. File expansion cost

File expansion cost takes into account the performance of the structure concerning the addition of new records.

In order to be consistent with the former TSB-tree evaluation [4], we exclude from the analysis the cost of (re-)writing the node receiving the new record (version). Therefore, assuming a separate medium for historical nodes, the expansion cost due to the AST policy can be expressed as for the TLU policy:

$$\text{expand[AST]} = (2 \times \text{T-split} + 2 \times \text{K-split} + 3 \times \text{KT-split})/R$$

since all the three kinds of split are possible, the original full node and the parent index node need to

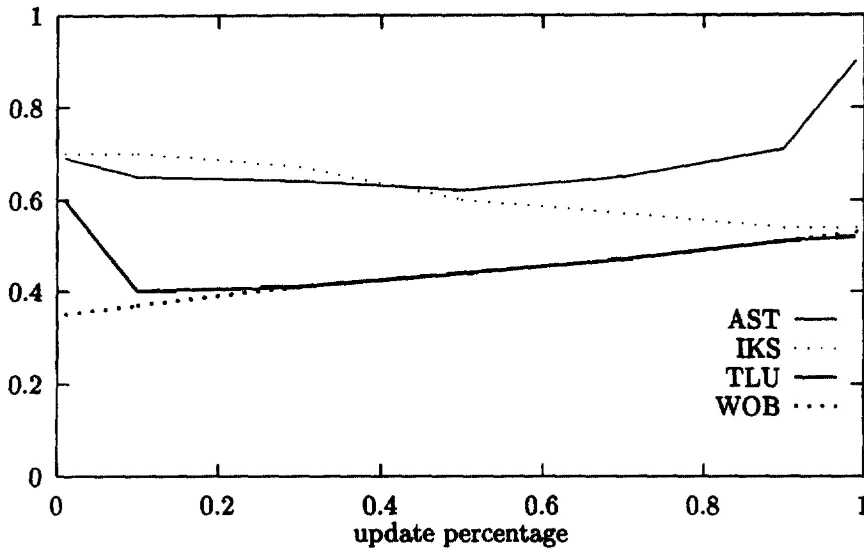


Fig. 6. Multiple version utilization (U_{mv}).

be rewritten in every case, and a second new node is created in the case of KT-splits. Experimental costs are displayed in Fig. 7. As can be seen, expansion costs with the AST policy are very uniform and very good also for high update percentages. They are always lower than those paid by the IKS and TLU policies, due to the lower number of T-splits effected. Only the WOB policy performs better, since historical nodes are not migrated on a different media.

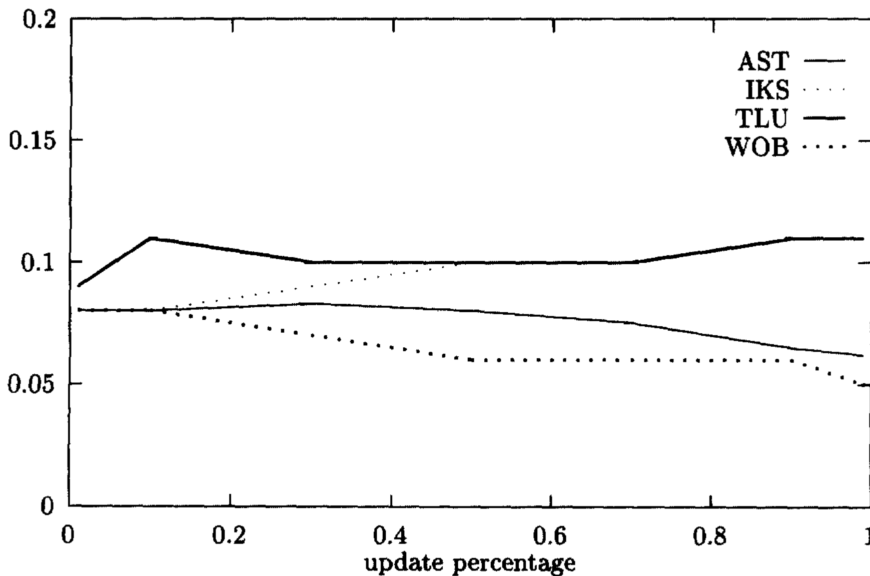


Fig. 7. File expansion cost (expand).

5. Conclusions

In this paper we have presented an improvement of Lomet and Salzberg's Time Split B-tree, which is an index for transaction-time databases. The improvement consists of a new split policy that allows the organization to limit data duplication. The policy is based on moving split thresholds which are dynamically changed through an adaptive mechanism. The policy has been shown to be effective in maintaining redundancy within given bounds.

The total memory occupation with the AST policy is not only very stable, but also assumes a small value because of the low level of redundancy obtained. In addition, the percentage of memory utilization is higher than in the 'traditional' TSB-tree. Also file expansion costs—for a TSB-tree maintaining current records on magnetic disk and migrating historical data on a different media—are optimal for the AST policy.

The only merit figure that is not improved by the AST policy is the single version current utilization (U_{svc}). However, it should be taken into account that, with the 'traditional' split policies, a large fraction of current records is made of redundant copies. If the single version current utilization is considered of primary importance for special application aims, the AST policy can still be adopted for its optimization by fixing, for instance, a duplication range equal to 150–200%. Obviously, in such a case, all the other merit figures of the index structure could drop to unacceptable levels, due to the huge amount of forced duplication.

Future work will be devoted to a more concrete evaluation of the impact of the different split policies on query costs. Although time-based queries are unlikely to play a significant role in the complete workload of a real transaction-time database, the impact of a reduced redundancy on average response times must be better characterized in a qualitative and quantitative way. Moreover, beyond the choice of a 'low' redundancy range fixed as 10–15% in this paper, clear design rules should be given to the database user/administrator for the best choice of the target redundancy, taking into account its effects on storage and query costs.

Finally, it can also be noticed that the new split policy could also be applied to different data structures based on the same kind of partitioning of the key-time space as the TSB-tree (as a matter of fact, it was originally developed for another access structure for transaction-time databases, called KT-Index [1]).

Acknowledgments

The authors would like to thank Vassilis J. Tsotras and Bernard Seeger (also for their patience during a night presentation of this work at the Dagstuhl Seminar on Temporal Databases in 1997) and the anonymous referees for their relevant comments which, in particular, will greatly help our future research.

References

[1] L. Amadesi, F. Grandi, The KT-Index: An Adaptive Access Method for Transaction-time Databases, *CIOC-CNR Tech. Rep.*, Bologna, Italy, July 1995.

[2] C. Dyreson, F. Grandi, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N.L. Sarda, M.R. Scalas, A. Segev, R.T. Snodgrass, M.D. Soo, A. Tansel, P. Tiberio, G. Wiederhold, A consensus glossary of temporal database concepts. In: C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gadia, P. Hayes, S. Jajodia (Eds.), *ACM SIGMOD Record* 23(1) (March 1994) pp. 52–64.

[3] D. Lomet, B. Salzberg, Access methods for multiversion data, *Proc. ACM SIGMOD Conf.*, Portland, OR (1989) pp. 315–324.

[4] D. Lomet, B. Salzberg, The performance of a multiversion access method, *Proc. ACM sigmod CONF.*, Atlantic City, NJ (1990) pp. 353–363.

[5] D. Lomet, B. Salzberg, Transaction-time databases, Chap. 16, in: A. Tansel, R. Snodgrass, J. Clifford, S. Gadia, S. Jajodia, A. Segev (Eds.), *Temporal Databases: Theory, Design and Implementation*, The Benjamin/Cummings Publishing Company, Redwood City, CA (1993).

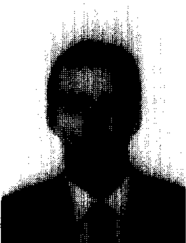
[6] J.A. Orenstein, Redundancy in spatial databases, *Proc. ACM SIGMOD Conf.* Portland, OR (1989) pp. 294–305.

[7] B. Salzberg, V.J. Tsotras, A Comparison of Access Methods for Temporal Data, *TimeCenter Tech. Rep. TR-18* (June 1997).

[8] A. Tansel, R. Snodgrass, J. Clifford, S. Gadia, S. Jajodia, A. Segev (Eds.), *Temporal Databases: Theory, Design and Implementation*, The Benjamin/Cummings Publishing Company, Redwood City, CA (1993).

[9] G.K. Zipf, *Human Behavior and the Principle of the Least Effort*, Addison-Wesley, Reading, MA (1949).

Laura Amadesi received the Laurea in Electronic Engineering from the University of Bologna, Italy, in 1994, and the Master degree in Business Administration from the ProFinGest Consortium, Bologna, in 1995. Since 1995 she has been with Bain & Company, Italy, working as a strategy consultant.



Fabio Grandi received the Laurea in Electronic Engineering from the University of Bologna, Italy, in 1988, and the Ph.D. in Electronics Engineering and Computer Science, in 1994. Since 1989 he has worked at the CIOC center (currently CSITE) of the Italian National Research Council (CNR) in Bologna, supported by a fellowship from the CNR, in the field of neural networks and temporal databases. In 1993 and 1994 he was an Adjunct Professor at the Universities of Ferrara, Italy and Bologna. He is currently with the Dept. of Electronics, Computer Science and Systems of the University of Bologna as a Research Associate. His areas of interest are temporal databases, storage and access structures, access cost models and information retrieval systems. He is a member of the TSQL2 language design committee.