VALID-TIME INDEXING

Mirella M. Moro
Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
http://www.inf.ufrgs.br/~mirella/

Vassilis J. Tsotras
University of California, Riverside
Riverside, CA 92521, USA
http://www.cs.ucr.edu/~tsotras

SYNONYMS

Valid-Time Access Methods

DEFINITION

A valid-time index is a temporal index that enables fast access to valid-time datasets. In a traditional database, an index is used for selection queries. When accessing valid-time databases such selections also involve the valid-time dimension (the time when a fact becomes valid in reality). The characteristics of the valid-time dimension imply various properties that the temporal index should have in order to be efficient. As traditional indices, the performance of a temporal index is described by three costs: (i) storage cost (i.e., the number of pages the index occupies on the disk), (ii) update cost (the number of pages accessed to perform an update on the index; for example when adding, deleting or updating a record), and (iii) query cost (the number of pages accessed for the index to answer a query).

HISTORICAL BACKGROUND

A valid-time database maintains the entire temporal behavior of an enterprise as best known now [12]. It stores the current knowledge about the enterprise's past, current or even future behavior. If errors are discovered about this temporal behavior, they are corrected by modifying the database. If the knowledge about the enterprise is updated, the new knowledge modifies the existing one. When a correction or an update is applied, previous values are not retained. It is thus not possible to view the database as it was before the correction/update. This is in contrast to a transaction-time database, which maintains the database activity (rather than the real world history) and can thus rollback to a past state. Hence in a valid-time database the past can change, while in a transaction-time database it cannot.

The problem of indexing valid-time databases can be reduced to indexing dynamic collections of intervals, where an interval represents the temporal validity of a record. Note that the term “interval” is used here to mean a “convex subset of the time domain" (and not a "directed duration"). This concept has also been named a “period”; in this discussion however, only the term “interval” is used.

To index a dynamic collection of intervals, one could use R-trees or related dynamic access methods. Relatively fewer approaches have been proposed for indexing valid-time databases. There have been various approaches proposed for the related problem of managing intervals in secondary storage. Given the problem difficulty, the majority of these approaches have focused on achieving good worst case behavior; as a result, they are mainly of theoretical importance. For a more complete discussion the reader is referred to a comprehensive survey [11].

SCIENTIFIC FUNDAMENTALS

The following scenario exemplifies the distinct properties of the valid time dimension. Consider a dynamic collection of “interval-objects”. The term interval-object is used to emphasize that the object carries a valid-time interval to represent the temporal validity of some object property. (In contrast, and to emphasize that transaction-time represents the database activity rather than reality, note that intervals stored in a transaction time database correspond to when a record was inserted/updated in the database.)

The allowable changes in this environment are the addition/deletion/modification of an interval-object. A difference with the transaction-time abstraction (the reader is referred to the Transaction-Time Indexing entry for more details) is that the collection's evolution (past states) is *not* kept. An example of a dynamic collection of interval-objects appears in Figure 1. Assume that collection C_a has been recorded in some erasable medium and a change happens, namely object I_z is deleted. This change is applied on the recorded data and physically deletes object I_z. The medium now stores collection C_b, i.e., collection C_a is

not retained. Note that when considering the valid time dimension, changes do not necessarily come in increasing time order (as is the case in transaction-time databases); rather they can affect *any* object in the collection. This implies that a valid-time database can correct errors in previously recorded data. However, only a single data state is kept, the one resulting after the correction is applied.

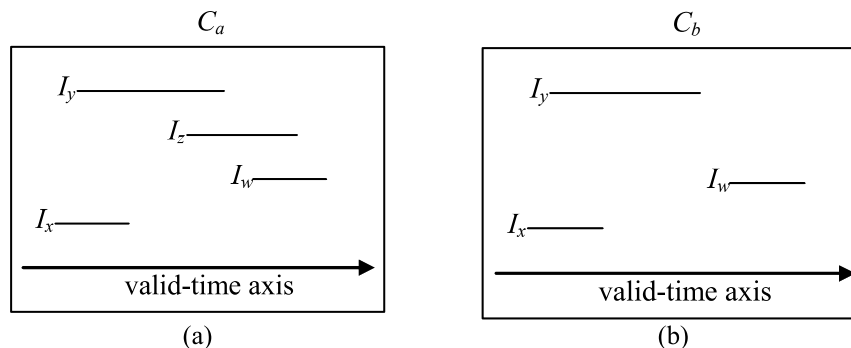

Figure 1. Two valid-time databases.

As a real-life example, consider the collection of contracts in a company. Each contract has an identity (*contract_no*) and an interval representing the contract's duration or validity. In collection C_a, there were four contracts in the company. But assume an error was discovered: contract I_z was never a company contract (maybe it was mistakenly entered). Then, this information is permanently deleted from the collection, which now is collection C_b.

The notion of time is now related to the valid-time axis. Given a valid-time instant, interval-objects can be classified as past, future or current as related to this instant, if their valid-time interval is before, after or contains the given instant. Valid-time databases can be used to correct errors anywhere in the valid-time domain (past, current or future) because the record of any interval-object in the collection can be changed, independently of its position on the valid-time axis. Note that a valid-time database may store records with the same surrogate but with non-intersecting valid-time intervals. For example, another object with identity I_x could be added in the collection at C_b as long as its valid-time interval does not intersect with the valid-time interval of the existing I_x object; the new collection will contain both I_x objects, each representing object I_x at different times in the valid-time dimension.

From the above discussion, an index used for a valid-time database should: (a) store the latest collection of interval-objects, (b) support addition/deletion/modification changes to this collection, and (c) efficiently query the interval-objects contained in the collection when the query is asked. Hence, a valid-time index should manage a *dynamic* collection of intervals.

Related is thus research on the interval management problem. Early work in main memory data-structures has proposed three separate approaches into managing intervals, namely, the Interval Tree, the Segment Tree and the Priority Search Tree [8]. Such approaches are focused on the “stabbing query” [6], i.e., given a collection of intervals, find all intervals that contain a given query point *q*. The stabbing query is one of the simpler queries for a valid-time database environment, since it does not involve any object key attribute (rather, only the object intervals).

Various methods have been proposed for making these main-memory structures disk-resident. Among them are: the External Memory Interval Tree [1], the Binary-Blocked Interval Tree [4], the External Segment Tree [2], the Segment Tree with Path Caching [10] and the External Memory Segment Tree [1], the Metablock Tree [6], the External Priority Search Tree [5], and the Priority Search Tree with Path Caching [10]. Many of these approaches are mainly of theoretical interest as they concentrate on achieving a worst-case optimal external solution to the 1-dimensional stabbing query. For good average behavior, methods based on multi-dimensional, indices should be preferred.

Since intervals are 2-dimensional objects, a dynamic multi-dimensional index like an R-tree may be used. Moreover, the R-tree can also index other attributes of the valid-time objects, thus enabling queries involving non-temporal attributes as well. For example, "find contracts that were active on time *t* and had contract-id in the range (*r1*,*r2*)". While simple, the traditional R-tree approach may not always be very efficient. The R-tree will attempt to cluster intervals according to their length, thus creating pages with possibly large overlapping. It was observed in [7] that for data with non-uniform interval lengths (i.e., a large proportion of "short" intervals and a small proportion of "long" intervals), this overlapping is clearly increased, affecting the query and update performance of the index. This in turn decreases query and update efficiency.

Another straightforward approach is to transform intervals into 2-dimensional points and then use a Point Access Method (quad-tree, grid file, hB-tree, etc.). In Figure 2, interval $I = (x_1,y_1)$ corresponds to a single point in the 2-dimensional space. Since an interval's end-time is always greater or equal than its start-time, all intervals are represented by points above the diagonal $x = y$. Note that an interval (x,y) contains a query time v if and only if its start-time x is less than or equal to v and its end-time y is greater than or equal to v. Then an interval contains query v if and only if its corresponding 2-dimensional point lies inside the box generated by lines $x = 0$, $x = v$, $y = v$, and $y = \infty$ (the shaded area in Figure 2). This approach avoids the overlapping mentioned above. Long intervals will tend to be stored together in pages with other long intervals (similarly, for the short intervals). However, no worst case guarantees for good clustering are possible.

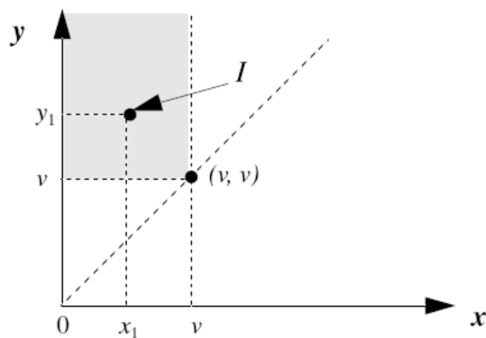

Figure 2. An interval $I = (x_1,y_1)$ corresponds to a point in a 2-dimensional space.

A related approach (MAP21) is proposed in [9]. An interval (l, r) is mapped to a point $z = (l\text{x}10^s) + r$, where s is the maximum number of digits needed to represent any point in the interval range. This is enough to map each interval to a separate point. A regular B^+-tree is then used to index these points. An advantage of this approach is that interval insertions/deletions are easy using the B^+-tree. To answer a stabbing query about q, the point closer but less than q is found among the points indexed in the B^+-tree, and then a sequential search for all intervals before q is performed. At worse, many intervals that do not intersect q can be found (this approach assumes that in practice the maximal interval length is known, which limits how far back the sequential search continues from q).

Another approach is to use a combination of an R-tree with Segment Tree properties. The Segment R-tree (SR-tree) was proposed in [7]. The SR-tree is an R-tree where intervals can be stored in both leaf and non-leaf nodes. An interval I is placed to the highest level node X of the tree such that I spans at least one of the intervals represented by X's child nodes. If I does not span X, it spans at least one of its children but is not fully contained in X, then I is fragmented. Figure 3 shows an example of the SR-tree approach. The top rectangle depicts the R-tree nodes (root, A, B, C, D and E) as well as the stored intervals. Interval L spans the rectangle of node C, but is not contained in node A. It is thus fragmented between nodes A and E.

Using this idea, long intervals will be placed in higher levels of the R-tree. Hence, the SR-tree tends to decrease the overlapping in leaf nodes (in the regular R-tree, a long interval stored in a leaf node will

“elongate” the area of this node thus exacerbating the overlap problem). However, having large numbers of spanning records or fragments of spanning records stored high up in the tree decreases the fan-out of the index as there is less room for pointers to children. It is suggested to vary the size of the nodes in the tree, making higher-up nodes larger. “Varying the size” of a node means that several pages are used for one node. This adds some page accesses to the search cost.

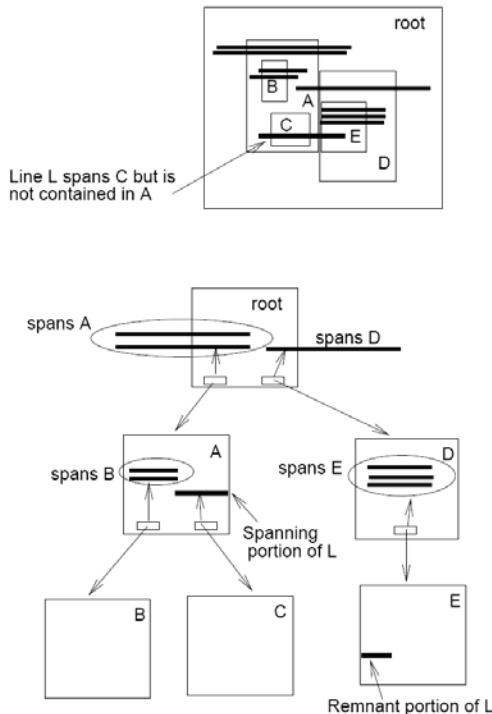

Figure 3. An example of the SR-tree approach.

As with the R-tree, if the interval is inserted at a leaf (because it did not span anything) the boundaries of the space covered by the leaf node in which it is placed may be expanded. Expansions may be needed on all nodes on the path to the leaf, which contains the new record. This may change the spanning relationships since existing intervals may no longer span children, which have been expanded. In this case, such intervals are reinserted in the tree, possibly being demoted to occupants of nodes they previously spanned. Splitting nodes may also cause changes in spanning relationships as they make children smaller - former occupants of a node may be promoted to spanning records in the parent.

In contrast to the traditional R-tree, the space used by the SR-tree is no longer linear. An interval may be stored in more than one non-leaf nodes (in the *spanning* and *remnant* portions of this interval). Due to the use of the segment-tree property, the space can be as much as $O(n \log_B n)$. Inserting an interval still takes logarithmic time. However, due to possible promotions, demotions and fragmentation, insertion is slower than in the R-tree. Even though the segment property tends to reduce the overlapping problem, the (pathological) worst case performance for the deletion and query time remains the same as for the R-tree organization (that is, at worst, the whole R-tree may have to searched for an interval). The average case behavior is however logarithmic. Deletion is a bit more complex, as all the remnants of the deleted interval have to be deleted too. The original SR-tree proposal thus assumed that deletions of intervals are not that frequent.

The SR-tree search algorithm is similar to that of the original R-tree. It descends the index depth-first, descending only those branches that contain the given query point *q*. In addition, at each node encountered during the search, all spanning intervals stored at the node are added to the answer. To improve the performance of the structure, the *Skeleton SR-tree* has also been proposed [7], which is an SR-tree that pre-partitions the entire domain into some number of regions. This pre-partition is based on

some initial assumption on the distribution of data and the number of intervals to be inserted. Then the Skeleton SR-tree is populated with data; if the data distribution is changed, the structure of the Skeleton SR-tree can be changed too.

When indexing valid-time intervals, overlapping may also incur if the valid-time intervals extend to the ever-increasing *now*. One approach could be to use the largest possible valid-time to represent the variable *now*. In [3] the problem of addressing both the *now* and transaction-time Until-Changed (*UC)* variables is addressed by using bounding rectangles/regions that increase as the time proceeds. A variation of the R-tree, the GR-tree is presented. More details appear in [3].

KEY APPLICATIONS

The importance of temporal indexing emanates from the many applications that maintain temporal data. The ever increasing nature of time imposes the need for many applications to store large amounts of temporal data. Specialized indexing techniques are needed to access such data. Temporal indexing has offered many such solutions that enable fast access.

CROSS REFERENCES

Temporal Database, Transaction-time, Valid-time, Transaction-time Indexing, Bi-temporal Indexing, B+-tree, R-tree

RECOMMENDED READING

[1] L. Arge and J. S. Vitter (1996). Optimal Dynamic Interval Management in External Memory. In Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, pages 560-569.

[2] G. Blankenagel and R.H. Gueting (1994). External Segment Trees. Algorithmica, 12(6):498-532.

[3] R. Bliujute, C.S. Jensen, S. Saltenis and G. Slivinskas (1998). R-Tree Based Indexing of Now-Relative Bitemporal Data. VLDB Conference, pp: 345-356.

[4] Y.-J. Chiang and C.T. Silva. (1999). External Memory Techniques for Isosurface Extraction in Scientific Visualization, External Memory Algorithms and Visualization, J. Abello and J.S. Vitter (Eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, vol. 50, pages 247-277.

[5] C. Icking, R. Klein, and T. Ottmann (1988). Priority Search Trees in Secondary Memory. In *Graph Theoretic Concepts in Computer Science*, pages 84-93. Springer Verlag LNCS 314.

[6] P. Kanellakis, S. Ramaswamy, D. Vengroff, and J. S. Vitter (1993). Indexing for Data Models with Constraint and Classes. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems,* pages 233-243.

[7] C. Kolovson and M. Stonebraker (1991). Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 138-147.

[8] K. Mehlhorn (1984). *Data Structures and Efficient Algorithms*, Vol.3: Multi-dimensional Searching and Computational Geometry. Springer Verlag, EATCS Monographs.

[9] M. A. Nascimento and M. H. Dunham (1999). Indexing Valid Time Databases via B^+-Trees. IEEE Transaction on Knowledge Data Engineering, 11(6): 929-947.

[10] S. Ramaswamy and S. Subramanian (1994). Path Caching: a Technique for Optimal External Searching. In *Proceedings of the 13rd ACM Symposium on Principles of Database Systems,* pages 25-35.

[11] B. Salzberg and V. J. Tsotras (1999). A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, 31(2):158-221.

[12] R.T. Snodgrass and I. Ahn (1986). Temporal Databases. IEEE Computer 19(9):35-42.