[4] B. Chazelle, "Filtering Search: A New Approach to Query-Answering," *Siam J. of Computing* 15(3) (1986), 703–724.

[5] E. F. Codd, "A Relational Model for Large Shared Data Banks," *CACM* 13(6) (1970), 377–387.

[6] D. Comer, "The Ubiquitous B-tree," *Computing Surveys* 11(2) (1979), 121–137.

[7] R. Elmasri, Y. Kim & G. Wuu, "Efficient Implementation Techniques for the Time Index," *Proc. of the 7th IEEE Intl. Data Engineering Conference* (1991).

[8] R. Elmasri, G. Wuu & Y. Kim, "The Time Index: An Access Structure for Temporal Data," *Proc. 16th Conference on Very Large Databases* (1990).

[9] Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM-SIGMOD Conference on Management of Data* (1985).

[10] J. Jaffar & J. L. Lassez, "Constraint Logic Programming," *Proc. 14th ACM POPL* (1987).

[11] P. C. Kanellakis, G. M. Kuper & P. Z. Revesz, "Constraint Query Languages," *Proc. 9th ACM PODS* (1990), invited to the special issue of *JCSS* on Principles of Database Systems (to appear). A complete version of the paper appears as Technical Report TR-CS-90-31, Brown University.

[12] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff & J. S. Vitter, "Indexing for Data Models with Constraints and Classes," *Proc. 12th ACM PODS* (1993), invited to the special issue of *JCSS* on Principles of Database Systems (to appear). A complete version of the paper appears as Technical Report TR-CS-93-21, Brown University.

[13] N. Kangelaris & V. J. Tsotras, "The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries," CATT-Tech. Report 93-68, Polytechnic University, Dec. 1993. Accepted for publication, Information Systems, An International Journal, 1994.

[14] S. Ramaswamy & S. Subramanian, "Path Caching: A Technique for Optimal External Searching," *Proc. 13th ACM PODS* (1994).

[15] B. Salzberg & V. J. Tsotras, "A Comparison of Access Methods for Time Evolving Data," To appear in the ACM Computing Surveys. Available as Technical Report CATT-TR-94-81, Polytechnic Univ. or NU-CCS-94-21, Northeastern Univ.

[16] T. Sellis, N. Roussopoulos & C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 13th VLDB Conference* (1987).

[17] S. Subramanian & S. Ramaswamy, "The P-range tree: A new data structure for range searching in secondary memory," *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (1995).

[18] A. U. Tanzel, J. Clifford, S. Gadia, S. Jajodia, A. Segev & R. Snodgrass, eds., *Temporal Databases: Theory, Design and Implementation*, The Benjamin/Cummings Publishing Company Inc., 1993.

This article was processed using the LaTeX macro package with LLNCS style

in the output of the pure-timeslice query corresponding to the range-timeslice query, and B is the disk blocksize.

Temporal objects can be "inserted" into or "deleted" from this data structure in $O(\log_B n)$ I/O's amortized.

Even though we do not explore the idea here, we believe that it is possible to migrate "older" parts of the temporal data structure to tertiary storage without re-organizing the entire index. Also, as was the case with constraint indexing, it is possible to chase down individual temporal objects and completely delete them from the temporal index or add arbitrary temporal objects (thus implementing what are called "valid time" temporal indexes). As in constraint indexing, those operations are unbounded and can also affect the query performance of the data structure.

4 Conclusions and Open Problems

We have presented new algorithms for constraint and temporal indexing using B^+-trees. The constraint indexing solution is optimal for querying, but does not have good worst-case bounds for updating. Therefore, the search for a good, simple data structure for interval intersection that can also handle updates in optimal worst-case time continues.

We believe that both algorithms that we have suggested in this paper are of practical significance. Both the algorithms have very small constants, which we believe will make them suitable for implementation. We are currently exploring the empirical performance of these data structures. In particular, we are comparing their average case performance against the R*-tree [3], the efficient data structure from the R-tree family.

Acknowledgments: This paper is dedicated to the memory of my beloved advisor Prof. Paris Kanellakis.

Jan Chomicki read initial drafts of this paper and suggested many improvements. The comments of the anonymous referees significantly improved the presentation of the paper and also helped fix some bugs. Their help is gratefully acknowledged.

References

[1] L. Arge & J. S. Vitter, "Optimal Interval Management in External Memory," 1995, Private Communication.

[2] R. Bayer & E. McCreight, "Organization of Large Ordered Indexes," *Acta Informatica* 1 (1972), 173–189.

[3] N. Beckmann, H. Kriegel, R. Schneider & B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. 1990 ACM-SIGMOD Conference on Management of Data* (1990).

The real power of the composite index is used for answering range-timeslice and pure-key-timeslice queries. In order to answer a range-timeslice query, say on a temporal key range (k_1, k_2), at time instant e, we first identify w, the starting point of the window into which e falls as before. We then perform a range query to retrieve all objects in the range $[(w, k_1), (w, k_2)]$. However, the size of the output of this query can be as large as the number of objects in w's window, which is not optimal.

The pure-key-timeslice query at temporal key value k and time instant e, which is a special case of the range-timeslice query can be answered by retrieving all objects whose composite key equals (w, k), where w is the starting point of the window into which e falls.

Let us now consider the dynamic version of the problem. The key observation behind making the data structure dynamic is that Procedure *construct-wlist* needs only the values of the variables *total*, *low*, and *cur* to process intervals in an online fashion. The crucial assumption is, of course, that endpoints are given to Procedure *construct-wlist* in sorted order. If endpoints represent time instants, this is clearly true. We can get a dynamic procedure for processing endpoints by replacing the idea of adding an object to a window-list by adding the object (with its composite key of *(starting-point-of-window, temporal-key)*) into a B^+-tree.

The one catch is that the endpoints of freshly inserted temporal objects are not known in advance. (The algorithm assumes that both the endpoints are known even when a starting point is processed.) It is possible to handle this, however, by marking each object's ending point as "open" when it is first inserted into the B^+-tree. When an object is "deleted", we change the ending point of the object in the current window to the correct endpoint. *Note that it is crucial that we only modify an object in the current window.* An object can be present in an unbounded number of windows. In every window that it is present in except the last one, its ending point will be marked as "open", which simply implies that it covers the whole width from the starting point to the end of that window.

Finally, the transition from one window to another in *construct-wlist* is not smooth. (W_{j+1} is constructed from W_j by taking all intervals in W_j that are "open".) This implies, unfortunately, that our bounds for updating have to be amortized. It is clear that inserting n objects into a temporal index results in no more $\frac{2\delta}{\delta-1}n$ additions to the B^+-tree, but individual operations don't have nontrivial bounds. We put everything together in the following theorem:

Theorem 3.2 *There exists a data structure that uses only B^+-trees and occupies $O(n/B)$ disk blocks, that can:*

- *answer pure-timeslice queries in optimal $O(\log_B n + t/B)$ I/O's;*
- *answer range-timeslice queries in sub-optimal $O(\log_B n + s/B)$ I/O's;*
- *answer pure-key-timeslice queries in optimal $O(\log_B n)$ I/O's;*

Here, n is the number of objects in the data structure, t, the number of objects that belong in the output of the timeslice query, s the number of objects that belong

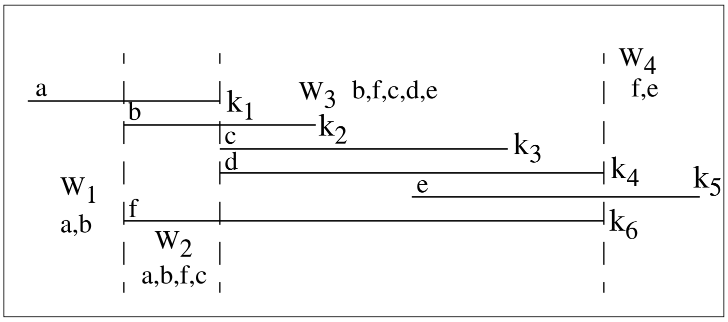

Fig. 3. Indexing a set of temporal objects: Each object is identified by a time interval and a key value. In this example also, $\delta = 2$.

construct window-lists based on objects' time intervals. Now, these window-lists will contain temporal objects instead of intervals.

While indexing constraints, we indexed each window-list as a whole based on starting point of the window's interval. For temporal indexing, we index each temporal object in the window-lists separately. The key for each object in a window-list is the pair *(starting-point-of-window, key-of-temporal-object)*. That is, we use a composite key consisting of the window's starting point and the temporal object's key. In this fashion, we insert all the objects in all the window-lists into a B^+-tree.

Example 3.1 *Consider the temporal objects represented in Figure 3. The time intervals corresponding to these objects are the same as the intervals in Figure 2. Each object, in addition, is also identified with a key value. Object a is associated with key k_1, b with key k_2, etc.*

As in the constraint indexing case, window-lists are created using the interval endpoints. Let x_1 denote the starting point of the time interval of object x, and x_2 its ending point. From window W_1, we insert (a_1, k_1, a), and (a_1, k_2, b) into the temporal index, where (a_1, k_1) is the composite key for object a, and (a_1, k_2) is the composite key for object b. We then insert (b_1, k_1, a), (b_1, k_2, b), (b_1, k_6, f) and (b_1, k_3, c) from window W_2 into the B^+-tree , and so on for all the windows. (Objects can be inserted multiple times in the index.)$\square$

Answering a pure-timeslice query in this environment is no different from answering a stabbing query in the constraint environment. Given a timeslice query at time e, we take the following steps:

- Locate the window into which time instant e falls. (We look for the first object whose key value is less than $(e, -\infty)$ in the index. Using this, we can identify the window.) Let that window's key be w.
- Perform a range query to retrieve all objects in the range $[(w, -\infty), (w, +\infty)]$.

It is easy to show that this search is optimal because at least $1/\delta$ fraction of the objects in w's window will belong in the output.

window list whose key is immediately to the left of the stabbing query. Clearly, this can be done in $O(\log_B n)$ disk accesses. Therefore the whole output can be reported in $O(\log_B n + t/B)$ I/O's.

The final part is also easy. The major portion of the time to build the data structure comes from sorting the $2n$ endpoints, which takes $O((\frac{n}{B})\log_B n)$ I/O's. Once that is done, we insert no more than $O(n)$ entries into a B$^+$-tree, which can also be done in $O((\frac{n}{B})\log_B n)$ I/O's. The lemma then follows.□

Using the result from [12] (Proposition 2.2) that reduces interval management to stabbing queries, we get an optimal static solution for the interval management problem. (This reduction divides interval management into the problems of answering stabbing queries and performing range searches on B$^+$-trees.)

Theorem 2.2 *There exists an optimal algorithm for the interval management problem that can answer interval intersection queries in optimal $O(\log_B n + t/B)$ I/O's, using optimal storage $O(n/B)$. Here, n is the number of input intervals, B the block size on disk and t, the number of intervals in the output to the query.*

Unfortunately, it does not seem easy to get good worst-case bounds on the update performance of this data structure. It is easy to add (delete) intervals into (from) the data structure (one simply adds/deletes the interval to/from all the windows that intersect it), but the amount of I/O that needs to be performed does not seem to have non-trivial upper bounds. Adding and deleting arbitrary intervals can also ruin the querying efficiency of this data structure.

In spite of this disadvantage, we believe that this is a practical data structure for the problem of constraint indexing. Occasional reorganizations can take care of imbalances introduced by insertions and deletions. Furthermore, the fact that the data structure only standard B$^+$-trees opens the possibility that constraint databases can be implemented on top of relational and object-oriented backends. The backends can provide the physical layer support and the concurrency control and recovery support, while the other desirable constraint linguistic features are implemented on top.

3 Efficient Temporal Indexing

In this section, we will modify the constraint indexing technique from the previous section to obtain an efficient index for many temporal indexing problems. The bounds we get are similar to the ones obtained in [13], but the resulting data structure is extremely simple and uses only a single B$^+$-tree. We will first consider the static version of the problem, where all the temporal objects are available and then consider the dynamic version, where objects are added in increasing time order.

Recall that in our temporal indexing model, each object has associated with it, a key, a time interval and on oid. In order to index these objects, we first

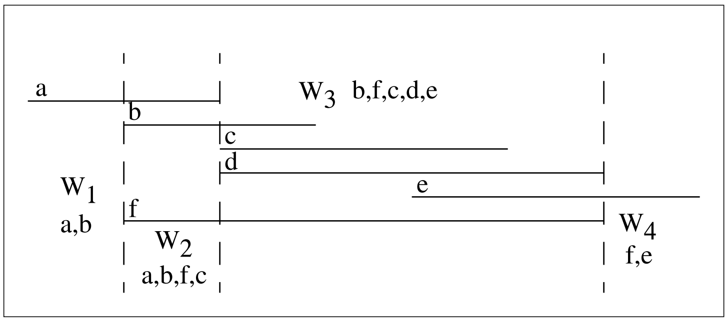

Fig. 2. Windows produced by procedure *construct-wlist* on a set of intervals. In this example, $\delta = 2$

2. We are scanning the ending point of an interval and $\delta \times (\text{low}_j - 1) < |W_j|$.

   In either case, for $\delta > 1$, $\delta \times \text{low}_j < |W_j| + \delta$, or, $\delta \times \text{low}_j < A_j + B_j + \delta$. It is also clear that for all $j < p$, $\delta A_j \leq \delta \times \text{low}_j$.
   Therefore, for $j < p$, $\delta A_j < A_j + B_j + \delta$ or $A_j < \frac{B_j+\delta}{\delta-1}$. This implies that

$$\sum_{1 \leq j \leq p} (A_j + B_j) < \sum_{1 \leq j < p} \left(\frac{B_j + \delta}{\delta - 1} + B_j \right) + A_p + B_p$$

$$= \sum_{1 \leq j < p} \left(\frac{\delta B_j}{\delta - 1} + \frac{\delta}{\delta - 1} \right) + A_p + B_p \tag{2}$$

Each endpoint of an input interval (totally, there are $2n$ of them) can give rise to exactly one B-part. In addition, $p+1$ of the endpoints cannot give rise to any B-parts because they fall on window boundaries. Hence, $\sum_{1 \leq j \leq p} B_j \leq (2n - p - 1)$. Substituting this into Inequality 2, we get

$$\sum_{1 \leq j \leq p} (A_j + B_j) < \frac{\delta(2n - p - 1 - B_p) + \delta(p - 1) + (\delta - 1)B_p}{\delta - 1} + A_p$$

$$< \frac{2\delta}{\delta - 1} n \tag{3}$$

Combining $\sum_{1 \leq j \leq p} |W_j| = \sum_{1 \leq j \leq p} (A_j + B_j)$ with Inequality 3, we get Inequality 1. In order to prove the same inequality without the assumption of non-coincident endpoints, we simply observe that exactly the same window-lists are produced by Procedure *construct-wlist* if the endpoints are replaced by their rankings.

The second part of the proof is easy. We will index every window-list using its key, which is the endpoint at which it was started. The construction of the window lists using Procedure *construct-wlist* guarantees that in any window, the inequality $|W_j| < \delta \times low$ is always true. This means that if t intervals are contained in a window list, at least t/δ of them will belong in the output. Given a stabbing query, we can locate the window into which it falls by looking for the

```
/* Input to this routine is a sorted list of (endpoint, interval) pairs. This routine in-
serts p window-lists into a B+-tree. Each window-list has two components, a key that
identifies the beginning of its interval, and a list of intervals associated with it.*/
Procedure construct-wlist
Initialize an empty B+-tree BT
win = 1; W_win.key = e_1;/* win is the number of the "current" window */
total = 0; /* total is the total number of intervals in the current window */
low = 1; /* low is the minimum number of intervals that any stabbing query in the
current window would intersect */
cur = 0; /* cur is the number of intervals that a stabbing query at the current point
would intersect with */
for (i=1; i <= 2n; i++)
    if (e_i is a starting endpoint)
        cur++;
        total++;
        if (δ × low < total) then /* current window is finished */
            Insert W_win into BT
            win++;
            W_win.intervals = {Intervals in W_win-1 whose ending point > e_i} ⋃ current
            interval
            W_win.key = e_i;
            low = total = cur;
        else /* add interval to current window */
            Add current interval to W_win.intervals
    else /* e_i is an ending point */
        cur = cur - 1;
        low = min(low, cur);
        if (δ × low < total) then /* current window is finished */
            Insert W_win into BT
            win++;
            W_win.intervals = {Intervals in W_win-1 whose ending point > e_i}
            W_win.key = e_i;
            low = max(cur,1); total = cur;
```

Fig. 1. Procedure *construct-wlist* (adapted from [4]) used in the construction of the window lists used to prove Theorem 2.2

fully coincide with a window's W_j aperture (or width) $[w_j, w_{j+1}]$ are called its A-parts, and those parts that partially coincide with a window are called its B-parts. Let A_j (B_j resp.) represent the total number of A-parts (B-parts resp.) in window W_j. Our non-coinciding endpoint assumption assures us now that $A_p \leq 1$.

Let us use O_x to indicate the set of intervals that intersect with query point x and $|O_x|$ to indicate the cardinality of O_x. Let low_j stand for $\min_{x \in [w_j, w_{j+1}]}(|O_x|)$, the minimum number of intervals that intersect *any* point in window W_j.

A new window W_{j+1}, for $j < p$ is started only under the following two conditions:

1. We are scanning the starting point of an interval and $\delta \times \mathrm{low}_j < |W_j| + 1$.

Once these two conditions are satisfied, it is easy to show that stabbing queries can be answered efficiently. Searching for the appropriate window clearly only takes $O(\log_B n)$ disk accesses. If the window-list contains t intervals, Condition 1 guarantees that at least t/δ of them belong in the query. Therefore, the query time is $O(\log_B n + t/B)$ disk accesses, which is optimal. Figure 1 presents an algorithm for creating the data structure. (The algorithm first creates a window-list and then inserts the whole list into the B$^+$-tree. This is to make the proof easier. In a more practical setting, one would insert intervals as they were processed. That is, operations that manipulate the *intervals* member of the W_{win} structure in Figure 1 would directly insert intervals into the B$^+$-tree.)

Procedure *construct-wlist* in Figure 1 takes as input the sorted set of endpoints, with ties broken arbitrarily. It keeps adding intervals to the "current" window as long as the total number of intervals in the window does not exceed $\delta \times low$, where low is the minimum number of intervals that any query falling in the current window will intersect with. When this condition is no longer satisfied, the current window is inserted into the B$^+$-tree, and a new window started with those intervals in the current set whose ending points are greater than the current endpoint. If we have many intervals with coincident endpoints, it is possible that *construct-wlist* will produce windows with zero width. The simplest way to handle that problem is to discard all windows of zero width and store only the first and last windows produced at the coincident endpoint. Since these two windows will list all intervals at the coincident endpoints, we will not miss any endpoints. See Figure 2 for an example of windows produced by procedure *construct-wlist*.

Lemma 2.1 *There exists an algorithm for answering stabbing queries optimally in $O(\log_B n + t/B)$ disk accesses, using $O(n/B)$ blocks on disk. It can be constructed in $O((\frac{n}{B}) \log_B n)$ I/O's.*

Proof. We will prove this lemma in three parts. We will first prove that the window lists produced by Procedure *construct-wlist* in Figure 1 use only linear space. That done, we will show how to use a B$^+$-tree to index the window lists and answer stabbing queries optimally. Finally, we will show that the data structure takes only $O((\frac{n}{B}) \log_B n)$ I/O's to build. To show the first part, we will prove the following inequality:

$$\sum_{1 \le j \le p} |W_j| < \frac{2\delta}{\delta - 1} n \tag{1}$$

Initially, we will make the assumption that input intervals do not share endpoints. We will show how to remove this restriction later.

In order to prove the inequality above, we observe that an input interval $[a_i, b_i]$ can be situated in many windows. In order to establish an upper bound on the space we use for the window lists, we need to count the total number of times input intervals are stored in different windows. In order to do that, we divide an input interval into A-parts and B-parts. Parts of an interval that of all intervals that intersect that endpoint. We then index these sets, using a B^+-tree, with each set keyed by the endpoint that it is associated with. In order to answer a stabbing query, we find the endpoint nearest to the query point, on the right, and examine the intervals in the set corresponding to the endpoint, and report the intervals that intersect with the query point as the answer to the query. (Only those intervals that actually begin at that nearest endpoint will not belong in the answer to the query.) In fact, what has been described here is the essence behind the Time Index of [7,8]. It can easily be seen that this method is fairly wasteful of space, storing upto $O(n^2)$ intervals in the worst case in the B^+-tree. (Inserting and deleting intervals is also fairly hard, taking upto $O(n)$ insertions/deletions to/from the B^+-tree.) Let us call this solution the *all-points* method for easy reference.

The key intuition behind obtaining a more efficient data structure is: *On the average, adjacent sets in the* all-points *method do not differ greatly from each other.* By choosing the sets that we decide to index carefully, it is possible to reduce the storage needed from $O(\frac{n^2}{B})$ to $O(n/B)$.

First, let us sort the input endpoints, $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, to give $e_1, e_2, ..., e_{2n}$. Instead of choosing the set of all endpoints and indexing them, we will choose a set of *windows*, $W_1, W_2, \ldots, W_p$, (The algorithm that we devise will choose this $p < 2n$ appropriately.) over endpoints $w_1, w_2, \ldots, w_{p+1}$. These endpoints are chosen from the input set of endpoints such that $w_1 = e_1$, $w_{p+1} = e_{2n}$, $w_j \leq w_{j+1}$ and $W_j = [w_j, w_{j+1}]$. In words, the windows represent a partition of the interval between e_1 and e_{2n} into p contiguous intervals. Associated with each window, is its *window-list*, the set of all input intervals that intersect the window's interval W_j. We will use $|W_j|$ to indicate the number of elements in window W_j's window-list.

We store the window-lists in a B^+-tree, indexing them by the starting point of the corresponding window. In order to answer a stabbing query, we perform a search for the query point and retrieve the window-list of the window that it falls into. (If it falls exactly on the boundary of two windows, we look at both the window-lists in the B^+-tree .) We then examine every interval in the window-list(s) and report all the intervals that intersect with the query interval.

We enforce two key conditions on the windows to make search efficient and space needed to store the window-lists low:

1. For each query point q that intersects with a window W_j, the number of input intervals that belong in the output set O_q satisfies the condition $0 < |W_j| \leq \delta \max(1, |O_q|)$, where $\delta > 1$ is a slack parameter that provides a tradeoff between space and query time, and $|O_q|$ the size of the output O_q. Essentially, we guarantee that searching a window-list to answer a query that intersects with its window is never inefficient.
2. $\sum_{1 \leq j \leq p} |W_j| < \frac{2\delta}{\delta - 1} n$ That is, the window-lists need only linear storage.

---

points of input intervals. Where necessary, we will use the phrase "ending point" explicitly.

We believe that these are good representatives of temporal indexing problems. Some more general problems are also considered by [15]. E.g., a generalization of the range-timeslice query, where a time interval is used instead of a time instant. Clearly, this problem is a generalization of two dimensional search which is known to be hard to solve with good worst-case bounds [17]. (It is known, for example, that any algorithm that achieves poly-log query bounds has to use almost logarithmic storage replication.) Some simpler problems are also considered by [15]. For example, given a key range, find the history of the keys in the range. This can be easily be solved by using a secondary B$^+$-tree index on the key value.

The data structure in [13] achieves the optimal bound of $O(\log_B n + t/B)$ I/O's for pure-timeslice queries, the sub-optimal bound of $O(\log_B n + s/B)$ I/O's for range-timeslice queries (s is the number of objects that belong in the output of the pure-timeslice query corresponding to the range-timeslice query), and the optimal bound of $O(\log_B n)$ I/O's for pure-key-timeslice queries. This data structure is fairly involved. We present a solution based on the constraint indexing technique that achieves the same bounds using a single B$^+$-tree. Updates can be made to this structure, but the optimal bound obtained ($O(\log_B n)$) is amortized.

Orthogonal to these approaches, constraint indexing can be performed using members from the R-tree family [3,9,16]. These data structures do not guarantee good worst-case bounds, but have been empirically determined to provide good average performance for interval indexing and other problems.

In summary, we present solutions for constraint and temporal indexing that are considerably simpler than previous solutions. Their practical significance lies in the fact that they use only B$^+$-trees, raising the possibility that these data models can be implemented on top of existing relational and object-oriented backends.

2 An Optimal Solution to the Interval Intersection Problem on Disk

In this section, we present a new, simple solution to the interval intersection problem. It is shown in [11,12] that indexing for a large class of constraint data models reduces to this problem. Our solution is adapted from Chazelle's algorithm for interval overlap in main memory [4].

We first consider the slightly simpler case of what has been referred to as the "stabbing" query in [12]. It has been shown in [12] that interval intersection can easily be reduced to this problem. In the stabbing query problem, the input is a set of n intervals $[a_i, b_i]$, $1 \leq i \leq n$. (The a_i's and b_i's can come from any ordered domain.) Given a query point q, we are asked to find all the input intervals that intersect this point.

A straightforward method to handle this problem is to identify the set of all unique endpoints[2] in the input, and for each such endpoint, associate the set

[2] We will use the term *endpoint* to generically refer to both the starting and ending

range searching. Typically, these approaches achieve the optimal query time of $O(\log_B n + t/B)$ I/O's using optimal $O(n/B)$ disk blocks of storage. Update processing seems much harder to do and some of these methods achieve $O(\log_B n)$ I/O's for updates, typically amortized. In this paper, we present a new algorithm for this problem based on a radically different approach using a query-time/space tradeoff. Our approach is based on an algorithm developed by Chazelle in [4] for interval intersection in main memory. Our approach uses only B$^+$-trees, achieves the optimal query time of $O(\log_B n)$ I/O's using linear space $O(n/B)$. It can handle updates, but it does not seem possible to obtain non-trivial bounds for update processing.

Indexing Temporal Objects: The temporal model we use is a tuple-versioning model [15,18]. ([15] is a comprehensive survey of the field of temporal indexing.) In our model of the temporal indexing problem, each record to be indexed is assumed to have three fields: (1) a time invariant *key* attribute, (2) a time variant *interval* attribute, where each interval consists of a *start-time* and an *end-time*, and (3) an object *id*. Further, we also assume that the underlying database system only supports the notion of *transaction time* [15,18]. Essentially, this means the following:

- When an object is *added* to the database at time e_1, its time interval is (e_1, now) where *now* is a reference to the "current" time. The intuition is that this object is activated at time e_1 and will continue to exist until it is deleted.
- When a previously inserted object is *deleted* at time instant e_2, its time interval, which was originally (e_1, now) is changed to (e_1, e_2).
- No other operations are allowed on objects that already exist in the database. For example, it is illegal to change the time interval of an object from (e_1, e_2) to (e'_1, e'_2) or add an object that references time in the "past".

Essentially, these conditions make the temporal indexing problem an *append only* problem. We will use a derivative of the constraint index for the temporal indexing problem. Even though the constraint indexing technique cannot give reasonable bounds for arbitrary updates, it can be made to give good amortized bounds for the append only case and that will be the crucial property that we will use for the temporal index.

It is possible to arrive at a number of different ways to query a temporal index. The three main problems that we will be concerned with are:

- Given a time instant e, find every object whose time interval contains the query instant. This is called the *pure-timeslice* query in [15].
- Given a time instant and a key range, find each object whose time interval contains the query instant, and whose key value falls in the query key range. This is called the *range-timeslice* query in [15].
- Given a time instant and a key value, find the object whose time interval contains the query instant, and whose key value equals the query key value. We will call this the *pure-key-timeslice* query.

In this paper, we study indexing in the context of constraint and temporal data models. We do not better previously known worst-case bounds for these problems (many of them were optimal to begin with!), but using a query-time/space tradeoff, we present algorithms that are considerably simpler than previously known algorithms. Since the algorithms we present use only B^+-trees, which are standard on all database systems, we believe that our algorithms are also of practical significance.

We first introduce our model for secondary storage algorithms and then look at the performance of B-trees for 1-dimensional range searching, since we will use B^+-trees as our "benchmark". We make the standard assumption that each secondary memory access transmits one page or B units of data, and we count this as one I/O. The efficiency of our algorithms is measured in terms of the number of I/O operations that they perform. Let R be a relation with n tuples and let the output of a query on R have t tuples. Our I/O bounds are expressed in terms of n, t and B and all constants are independent of these three parameters. We will also use the symbol n to indicate the number of intervals to be indexed in the constraint indexing problem, and the number of temporal objects to be indexed in the temporal indexing problem. A B^+-tree on attribute x of the n-tuple relation R uses $O(n/B)$ pages of secondary storage. The following operations define the problem of *external dynamic 1-dimensional range searching* on relational database attribute x, with the corresponding I/O time performance bounds using the B^+-tree on x: (1) Find all tuples such that for their x attribute $(a_1 \leq x \leq a_2)$. If the output size is t tuples, then this range searching is in worst-case $O(\log_B n + t/B)$ secondary memory accesses. If $a_1 = a_2$ and x is a key, i.e., it uniquely identifies the tuple, then this is key-based searching. (2) Insert or delete a given tuple are in worst-case $O(\log_B n)$ secondary memory accesses. In this paper, we will be concerned with indexing for constraint and temporal data models. We will aim for algorithms with provably good worst-case I/O bounds.

Indexing Constraints: Constraint programming paradigms are inherently "declarative", since they describe computations by specifying how these computations are constrained. A general constraint programming framework for database query languages called Constraint Query Languages or CQLs was presented in [11]. This framework adapts ideas of Constraint Logic Programming, e.g., from [10], to databases, provides a calculus and algebra, guarantees low data complexity, and is applicable to managing spatial data.

It is, of course, important to index constraints and thus support these new language features with efficient secondary storage manipulation Fortunately, it is possible to do this by reducing the problem of indexing constraints, for a fairly general class of constraints, to dynamic interval management on secondary storage. Given a set of input intervals, dynamic interval management involves being able to perform the following operations efficiently: (1) Answer interval intersection queries. That is, to report all the input intervals that intersect a query interval. (2) Delete or insert intervals from the interval collection.

Previous approaches [1,12,14,17] are fairly complex and rely mostly on reducing the interval intersection problem to special cases of two-dimensional

Efficient Indexing for Constraint and Temporal Databases

Sridhar Ramaswamy*

Bell Labs, Room 2D144, 600 Mountain Ave, Murray Hill, NJ 07974, USA.
Email: sridhar@bell-labs.com

Abstract. We examine new I/O-efficient techniques for indexing problems in constraint and temporal data models. We present algorithms for these problems that are considerably simpler than previous solutions. Our solutions are unique in the sense that they only use B^+-trees rather than special-purpose data structures. Indexing for many general constraint data models can be reduced to interval intersection. We present a new algorithm for this problem using a query-time/space tradeoff, which achieves the optimal query time $O(\log_B n + t/B)$ I/O's in linear space $O(n/B)$ using B^+-trees. (Here, n is the number of intervals, t the number of intervals in the output of a query, and B the disk block size.) It is easy to update this data structure, but small worst-case bounds do not seem possible. Previous approaches have achieved these bounds but are fairly complex and rely mostly on reducing the interval intersection problem to special cases of two-dimensional search. Some of them can also handle updates in $O(\log_B n)$ I/O's amortized. Indexing in many temporal models becomes a generalization of interval management, where each temporal object is characterized by an interval *and a key*. There are many different ways of querying these objects, and we achieve optimal bounds for many of these queries. These bounds are achieved using a modification of the technique used for the constraint indexing problem. Our technique is much simpler than other techniques that have been used for achieving similar bounds.

1 Introduction

The successful realization of any data model in a large-scale database requires supporting its language features with efficient secondary storage manipulation. Consider the relational data model of [5]. While the declarative programming features (relational calculus and algebra) of the model are important, it is crucial to support these features by data structures for searching and updating that make optimal use of secondary storage. B-trees and their variants B^+-trees [2, 6] are examples of such data structures. They have been an unqualified success in supporting external dynamic 1-dimensional range searching in relational database systems.

* This work was done while the author was at Bell Communications Research, Morristown, NJ 07960, USA.