Proc. 7th Int'l Symposium on Spatial and Temporal Databases (**SSTD'01**).
Lecture Notes in Computer Science, © Springer Verlag. 2001

Object-Relational Indexing for General Interval Relationships

Hans-Peter Kriegel, Marco Pötke, Thomas Seidl
University of Munich, Institute for Computer Science
Oettingenstr. 67, 80538 Munich, Germany
{kriegel, poetke, seidl}@dbs.informatik.uni-muenchen.de

Abstract. Intervals represent a fundamental data type for temporal, scientific, and spatial databases where time stamps and point data are extended to time spans and range data, respectively. For OLTP and OLAP applications on large amounts of data, not only intersection queries have to be processed efficiently but also general interval relationships including *before*, *meets*, *overlaps*, *starts*, *finishes*, *contains*, *equals*, *during*, *startedBy*, *finishedBy*, *overlappedBy*, *metBy*, and *after*. Our new algorithms use the Relational Interval Tree, a purely SQL-based and object-relationally wrapped index structure. The technique therefore preserves the industrial strength of the underlying RDBMS including stability, transactions, and performance. The efficiency of our approach is demonstrated by an experimental evaluation on a real weblog data set containing one million sessions.

1 Introduction

Modern database applications often manage extended data including time spans for the validity of stored facts [TCG+ 93], tolerance ranges for imprecisely measured values in scientific databases, or approximate values in local caches of distributed databases [OLW 01]. Online analytical processing for data warehouses, for example on the temporal coherence of marketing activities and the sales volume, requires intervals as a basic datatype. Moreover, the practical relevance of intervals has been strongly emphasized by the introduction of the corresponding datatypes and predicates into the new SQL:1999 standard, formerly known as SQL3 [EM 99]. In SQL, an interval comprising a range between two ordered boundaries is termed a "PERIOD" and denotes an anchored duration on the linear time line. Dates can be encoded by a basic temporal datatype or an integer. In addition to plain interval intersection queries, more refined relationships have to be supported for many applications, e.g. sequenced temporal integrity checking [LSD+ 01]. Compulsory predicates on PERIODs include the plain interval intersection as well as a subset of Allen's 13 general interval relationships [All 83] (cf. Figure 1), namely "PRECEDES" (= *before*), "SUCCEDES" (= *after*), "MEETS" (= *meets* $\cup$ *metBy*), and "CONTAINS" (= *contains* or *during*, resp.) [Sno 00].

In order to bring the expressive power of interval predicates and SQL:1999 to life, a robust access method for intervals and their predicates is required. As this component has to be integrated into commercial database servers to complement the interval data model with efficient query execution, a maximal exploitation of the generic functionality of existing RDBMS is essential [JS 99] [TJS 98]. In this paper, we therefore propose a new technique to efficiently evaluate not only the five interval predicates of

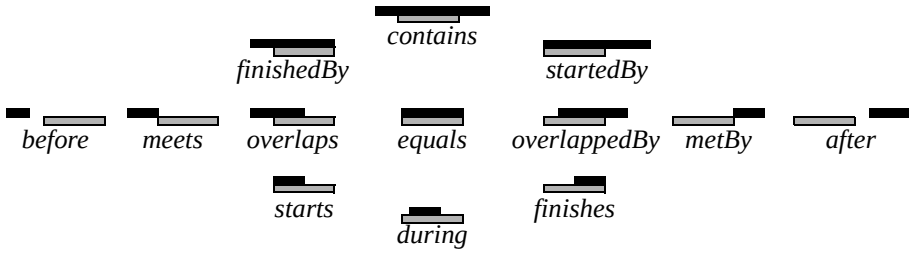


Fig. 1. The 13 general interval relationships according to Allen [All 83]

SQL:1999, but the full set of Allen's general relationships on top of any object-relational database kernel. We follow the layered approach of temporal database systems by transforming the DDL and DML on interval data into conventional statements executed by the underlying DBMS. At the same time, its industrial strength, including stability, transactions, and performance is fully preserved by our proposed method.

Whereas point data has been supported very efficiently for a long time, the problem of managing intervals is not addressed by commercial database systems up to now. Several index structures have been proposed that immediately are built on top of relational database systems. They use the SQL level of an ordinary RDBMS as virtual storage medium, and, therefore, we call them relational storage structures. Among them, the Relational Interval Tree[1] (RI-tree) [KPS 00] provides the optimal complexities of $O(n/b)$ space to store n intervals on disk blocks of size b, $O(\log_b n)$ I/O operations for insertion or deletion of a single interval, and $O(h \cdot \log_b n + r/b)$ I/Os to process an interval intersection query producing r results. The parameter h depends on the extension and the granularity of the data space but not on the number n of intervals. As a competing method, the linear quadtree [Sam 90] as used in Oracle or DB2 for spatial objects maps a main-memory structure onto built-in relational indexes, too, and may be called linear segment tree in the one-dimensional case. Unfortunately, the decomposition of intervals into segments yields a potentially high redundancy in the database in contrast to the RI-tree.

The MAP21 transformation [ND 99] or the H-, V-, or D-order interval spatial transform [GLOT 96] refine the idea to employ a composite index on the interval bounds and order the intervals lexicographically by (*lower, upper*) or (*upper, lower*). Finally, the window list technique [Ram 97] is very efficient but may degenerate for dynamic data sets. An additional broad variety of secondary storage structures for intervals has been proposed in the literature. Since these approaches rely on augmentations of built-in indexes structures, they are not suitable to be used in industrial applications unless they are integrated into the kernel software by the database vendors. A detailed discussion of these aspects is provided in [KPS 00].

What we propose in this paper are extensions of the RI-tree algorithms that efficiently support the general interval relationships of Allen. After recalling the Relational Interval Tree in Section 2, we present our new algorithms in Section 3. We suggest an effective extension of the underlying relational schema that preserves the optimal I/O complexity for the majority of interval relationships. Simple but powerful heuristics

1. Patent pending

minimize the overhead for the remaining interval relationships. In Section 4, we demonstrate the efficiency of our techniques on a real weblog data set of one million intervals.

2 The Relational Interval Tree

The RI-tree [KPS 00] is a relational storage structure for interval data (*lower*, *upper*), built on top of the SQL layer of any RDBS. By design, it follows the concept of Edelsbrunner's main-memory interval tree [Ede 80] and obeys the optimal complexity for storage space and for I/O operations when updating or querying large sets of intervals.

2.1 Relational Storage and Extensible Indexing

The RI-tree strictly follows the paradigm of relational storage structures since its implementation is purely built on (procedural and declarative) SQL but does not assume any lower level interfaces to the database system. In particular, built-in index structures are used as they are, and no intrusive augmentation or modification of the database kernel is required.

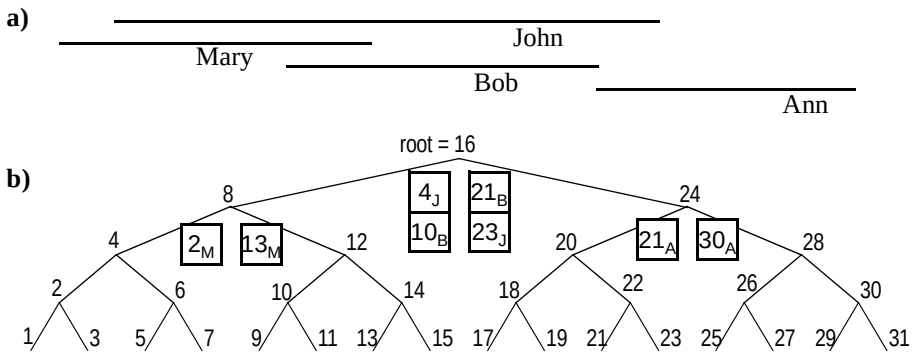
On top of its pure relational implementation, the RI-tree is ready for immediate object-relational wrapping. It fits particularly well to the extensible indexing frameworks, as already proposed in [Sto 86], which enable developers to extend the set of built-in index structures by custom access methods in order to support user-defined datatypes and predicates. They are provided by the latest object-relational database systems, including Oracle8i Server [Ora 99] [SMS+ 00], IBM DB2 Universal Database [IBM 99] [CCF+ 99] or Informix Universal Server [Inf 98] [BSSJ 99].

Although available extensible indexing frameworks provide a gateway to seamlessly integrate user-defined access methods into the standard process of query optimization, they do not facilitate the actual implementation of the access method itself. Modifying or enhancing the database kernel is usually not an option for database developers, as the embedding of block-oriented access methods into concurrency control, recovery services and buffer management causes extensive implementation efforts and maintenance cost [Kor 99], at the risk of weakening the reliability of the entire system. The server stability can be preserved by delegating index scans and maintenance to an external process, but this approach requires a custom concurrency control and induces severe performance bottlenecks due to context switches and inter-process communication. Queries and updates on relational storage structures are processed by pure SQL statements. The robust transaction semantics of the database server is therefore fully preserved.

2.2 Dynamic Data Structure

The structure of an RI-tree consists of a binary tree of height h which makes the range $[0\ldots2^h–1]$ of potential interval bounds accessible. It is called the virtual backbone of the RI-tree since it is not materialized but only the root value $2^{h–1}$ is stored persistently in a metadata table. Traversals of the virtual backbone are performed purely arithmetically by starting at the root value and proceeding in positive or negative steps of decreasing length $2^{h–i}$, thus reaching any desired value of the data space in $O(h)$ CPU time and without causing any I/O operation.

Upon insertion, an interval is registered at the highest node that is contained in the interval. For the relational storage of intervals, the value of that node is used as an arti-



c)

lowerIndex (*node*, *lower*, *id*):	8, 2, Mary	16, 4, John	16, 10, Bob	24, 21, Ann
upperIndex (*node*, *upper*, *id*):	8, 13, Mary	16, 21, Bob	16, 23, John	24, 30, Ann

Fig. 2. Example for an RI-tree. **a)** Four sample intervals. **b)** Virtual backbone and registration positions of the intervals. **c)** Resulting relational indexes *lowerIndex* and *upperIndex*

ficial key. An instance of the RI-tree then consists of two relational indexes which in an extensible indexing environment are preferably managed as index-organized tables. The indexes obey the relational schema *lowerIndex* (*node*, *lower*, *id*) and *upperIndex* (*node*, *upper*, *id*) and store the artificial key value *node*, the bounds *lower* and *upper*, and the *id* of each interval. An interval is represented by exactly one entry in each of the two indexes, and $O(n/b)$ disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the *node* values are determined arithmetically, and updating the indexes requires $O(\log_b n)$ I/O operations per interval.

The illustration in Figure 2 provides an example for the RI-tree. Let us assume the intervals (2,13) for Mary, (4,21) for John, (20,23) for Bob, and (21,30) for Ann (Fig. 2a). The virtual backbone is rooted at 16 and covers the data space from 1 to 31 (Fig. 2b). The intervals are registered at the nodes 8, 16, and 24 which are the highest nodes hit by the intervals. The interval (2,13) for Mary is represented by the entries (8, 2, Mary) in the *lowerIndex* and (8, 13, Mary) in the *upperIndex* since 8 is the registration node, and 2 and 13 are the lower and upper bound, respectively (Fig. 2c).

2.3 Intersection Query Processing

To minimize barrier crossings between the procedural runtime environment and the declarative SQL layer, an interval intersection query (*lower*, *upper*) is processed in two steps. The procedural query preparation step descends the virtual backbone from the root node down to *lower* and to *upper*, respectively. The traversal is performed purely arithmetic without any I/O operation, and the visited nodes are collected in two main-memory tables, **leftQueries** and **rightQueries**, as follows: Nodes left of *lower* may contain intervals which overlap *lower* and are inserted into **leftQueries**. Analogously, nodes right of *upper* may contain intervals which overlap *upper* and are inserted into **right-**

Queries. Whereas these nodes where taken from the paths, the set of all nodes between *lower* and *upper* belongs to the so-called **innerQuery** which needs not to be materialized. All intervals registered at nodes from the **innerQuery** are guaranteed to intersect the query and, therefore, will be reported without any further comparison. The query preparation step is purely based on main memory and requires no I/O operations.

In the subsequent declarative query processing step, the transient tables are joined with the relational indexes *upperIndex* and *lowerIndex* by a single, three-fold SQL statement (Figure 3). The upper bound of each interval registered at nodes in **leftQueries** is compared to *lower*, and the lower bounds of intervals stemming from **rightQueries** are compared to *upper*. The **innerQuery** corresponds to a simple range scan over the intervals with nodes in (*lower*, *upper*). The SQL query requires $O(h \cdot \log_b n + r/b)$ I/Os to report r results from an RI-tree of height h since the output from the relational indexes is fully blocked for each join partner. By the techniques presented in [KPS 00], the height h of the backbone tree is dynamically adjusted to the expansion of the data space and to the minimum length of the intervals. Typically, h is independent of the number n of intervals. For example, when assuming a granularity of one second, a height of 25 is sufficient to address one year (31,536,000 seconds), and a height of 32 makes a century accessible independent of the number of intervals in the database.

```
SELECT id FROM upperIndex i, :leftQueries q
   WHERE i.node = q.node AND i.upper >= :lower
UNION ALL
SELECT id FROM lowerIndex i, :rightQueries q
   WHERE i.node = q.node AND i.lower <= :upper
UNION ALL
SELECT id FROM lowerIndex   /* or upperIndex */
   WHERE node BETWEEN :lower AND :upper;
```

Fig. 3. SQL statement for an intersection query with bind variables leftQueries, rightQueries, lower, and upper

We prefer the implementation for **leftQueries** and **rightQueries** as binding variables for transient tables over the alternative of using set containment predicates 'i.node IN leftQueries' and 'i.node IN rightQueries'. By using transient tables, we save the encoding and decoding of the node values to and from ASCII. Moreover, the entire SQL query is parsed and optimized in advance and reused for multiple queries.

3 Algorithms for General Interval Relationships

In this section, we develop our method for processing general interval relationship queries based on the RI-tree. We first identify and classify the nodes that need to be accessed for the different interval relationships and derive the corresponding algorithms and heuristics for the individual node classes. Finally, we compose the complete queries from the building blocks.

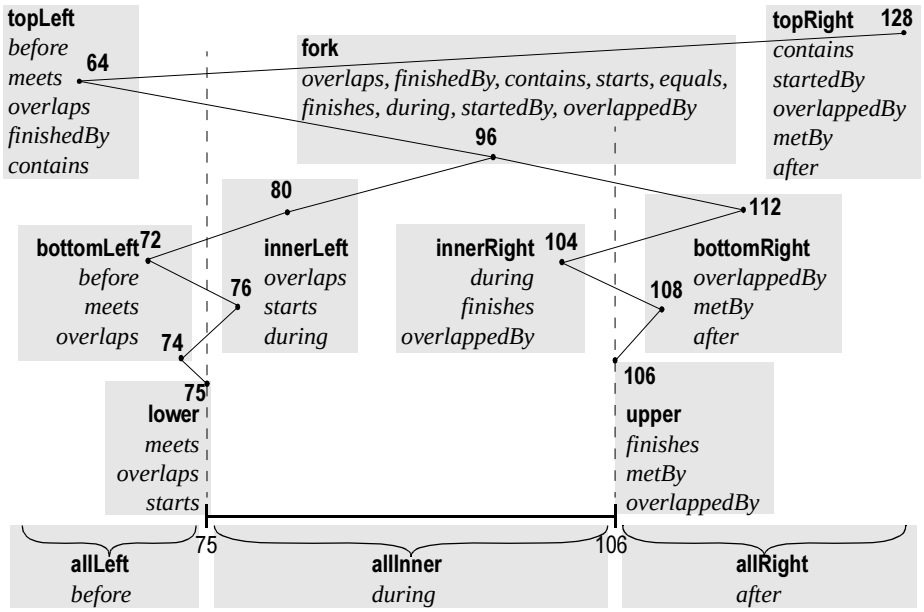


Fig. 4. Node classes generated for a sample query interval (75, 106): The six *traversal classes* **topLeft** (64), **bottomLeft** (72, 74), **innerLeft** (76, 80), **topRight** (128), **innerRight** (104), **bottomRight** (108, 112), the three *singleton classes* **lower** (75), **fork** (96), **upper** (106), and the three *range classes* **allLeft** (nodes < 75), **allInner** (76…105), and **allRight** (nodes > 106). Each class is marked with the corresponding relationships

3.1 Query Generation Schema

For the 13 general interval relationships of Allen [All 83], the plain interval intersection schema is replaced by a more fine grained query generation. While we have distinguished only the three classes *leftQueries*, *rightQueries*, and *innerQuery* for intersection queries, we are now faced with twelve classes of nodes that need to be handled differently. We call these classes **topLeft**, **bottomLeft**, **innerLeft**, **topRight**, **bottomRight**, **innerRight**, **lower**, **fork**, **upper**, **allLeft**, **allInner**, and **allRight**. Figure 4 provides an illustration of the classification for a sample query interval (75, 106). Each class is marked with the corresponding interval relationships. The code in Figure 5 illustrates the computation of the node classes.

The nodes on the path from the root node (included) down to the fork node of the query interval (excluded) fall into the classes **topLeft** or **topRight** depending on whether they are smaller than the lower query bound or greater than the upper query bound. The nodes on the path from the fork node (excluded) down to the lower query bound (excluded) form the classes **bottomLeft** and **innerLeft** depending on their position with respect to the lower query bound. Analogously, the nodes on the path from the fork node (excluded) down to the upper query bound (excluded) form the classes **innerRight** and **bottomRight** depending on their position with respect to the upper query bound. The mentioned node classes are generated by arithmetically traversing the virtual backbone

```
PROCEDURE generalIntervalQuery (string operator, int lower, int upper) {
   List ⟨int⟩ topLeft, bottomLeft, innerLeft;
   List ⟨int⟩ topRight, bottomRight, innerRight;
   int fork;

   //--- Descend from root node down to fork node ---
   int n = root;
   for (int step = n/2; step >= minstep; step = step/2) {
      if (n < lower)          {topLeft ← n; n = n + step;}
      elsif (upper < n)       {topRight ← n; n = n – step;}
      else /* n is fork */    {fork = n; break;}
   }
   //--- Descend from fork node down to lower ---
   if (lower < fork) {
      n = fork – step;
      for (int lstep = step/2; lstep >= minstep; lstep = lstep/2) {
         if (n < lower)          {bottomLeft ← n; n = n + lstep;}
         elsif (lower < n)       {innerLeft ← n; n = n – lstep;}
         else /* n is lower */   {break;}
      }
   }
   //--- Descend from fork node down to upper ---
   if (fork < upper) {
      n = fork + step;
      for (int rstep = step/2; rstep >= minstep; rstep = rstep/2) {
         if (n < upper)          {innerRight ← n; n = n + rstep;}
         elsif (upper < n)       {bottomRight ← n; n = n – rstep;}
         else /* n is upper */   {break;}
      }
   }
   //--- Start the SQL query for operator (see Table 2) ---
}
```

Fig. 5. Computation of the query classes for general interval relationships

and, therefore, we call them the *traversal classes*. As an immediate implication, the cardinality of their union is bound by twice the height of the backbone tree.

The fork node itself as well as the lower and the upper query bound each forms a class of its own, i.e. **fork**, **lower**, and **upper**, and we call them the *singleton classes*. All of the hitherto classes are disjoint except in the special case where **fork** coincides with **lower** or with **upper**. Finally, the union of classes **allLeft**, **allInner**, and **allRight** covers all nodes of the backbone tree except the lower and the upper query bound. We call these node sets the *range classes*. They are not materialized but represented by the predicates 'node < lower' for **allLeft**, 'lower < node AND node < upper' for **allInner**, and 'upper < node' for **allRight**.

3.2 Query Processing for the Traversal Classes

In the following, we assume a query interval q = (lower, upper) to be given. A database interval i registered at a node in **topLeft** is guaranteed to start before the lower query bound, i.e. i.lower < q.lower, but no assumption about the upper bound is valid. In addition to the relationships *before*, *meets*, and *overlaps* even *finishedBy* or *contains* may be valid but no other relationship can hold. For *before* and *meets*, it suffices to test the upper bound against the lower query bound, i.e. i.upper Θ q.lower with Θ denoting < or = for the respective interval relationship. For *finishedBy* and *contains*, the upper bound has to be compared to the upper query bound, i.e. i.upper Θ q.upper with Θ denoting = and >, respectively. For *overlaps*, however, the upper bound has to be compared to both query bounds, resulting in the test q.lower < i.upper < q.upper. In any case, the test is optimally supported by a range scan on the relational upperIndex guaranteeing blocked output:

SELECT id FROM upperIndex i, :**topLeft** q
WHERE i.node = q.node AND i.upper Θ :lower; -- set Θ to < (*before*), = (*meets*)

SELECT id FROM upperIndex i, :**topLeft** q WHERE i.node = q.node
AND i.upper Θ :upper; -- set Θ to = (*finishedBy*), > (*contains*)

SELECT id FROM upperIndex i, :**topLeft** q WHERE i.node = q.node
AND i.upper > :lower AND i.upper < :upper; -- for *overlaps*

Analogous considerations apply to the symmetric class **topRight** and the relationships *after*, *metBy*, *overlappedBy*, *startedBy*, and *contains* which are best processed by scanning the relational lowerIndex:

SELECT id FROM lowerIndex i, :**topRight** q WHERE i.node = q.node
AND i.lower Θ :upper; -- set Θ to > (*after*), = (*metBy*)

SELECT id FROM lowerIndex i, :**topRight** q WHERE i.node = q.node
AND i.lower Θ :lower; -- set Θ to = (*startedBy*), < (*contains*)

SELECT id FROM lowerIndex i, :**topRight** q WHERE i.node = q.node
AND i.lower > :lower AND i.lower < :upper; -- for *overlappedBy*

A database interval i registered at a node in **bottomLeft** is guaranteed to start before the lower query bound, i.e. i.lower < q.lower, and to end before the upper query bound since i.upper < q.fork ≤ q.upper. Only the relationships *before*, *meets*, or *overlaps* may hold, and it suffices to test the upper bound against the lower query bound, i.e. i.upper Θ q.lower where Θ denotes <, =, >, respectively.

Again, the relational upperIndex supports the test while guaranteeing blocked output in any case. As above, analogous considerations apply to intervals from nodes in the symmetric class **bottomRight** and the respective relationships *after*, *metBy*, and *overlappedBy* which are best supported by the relational lowerIndex:

SELECT id FROM upperIndex i, :**bottomLeft** q WHERE i.node = q.node
AND i.upper Θ :lower; -- set Θ to < (*before*), = (*meets*), > (*overlaps*)

```
SELECT id FROM lowerIndex i, :bottomRight q WHERE i.node = q.node
    AND i.lower Θ :upper;     -- set Θ to > (after), = (metBy), < (overlappedBy)
```

A database interval *i* from a node in **innerLeft** ends before the query interval ends since *i*.upper < *q*.fork ≤ *q*.upper but no assumption about the lower bound may be exploited in advance. The only relationships with *q* where *i* may participate are *overlaps*, *starts*, and *during* which are tested by a comparison of the registered lower bound against the lower query bound, i.e. *i*.lower Θ *q*.lower where Θ denotes <, =, > for the respective interval relationship. Optimal support is provided by the relational lowerIndex in this case. Analogously, intervals from **innerRight** may only participate in the relationships *during*, *finishes*, and *overlappedBy* which are efficiently tested by scanning the relational upperIndex:

```
SELECT id FROM lowerIndex i, :innerLeft q WHERE i.node = q.node
    AND i.lower Θ :lower;     -- set Θ to < (overlaps), = (starts), > (during)

SELECT id FROM upperIndex i, :innerRight q WHERE i.node = q.node
    AND i.upper Θ :upper;     -- set Θ to < (during), = (finishes), > (overlappedBy)
```

In any of the previous cases, the existing relational indexes lowerIndex and upperIndex immediately support the required comparisons by efficient range scans guaranteeing blocked output. Unfortunately, this observation does not hold for the singleton classes which we investigate in the following subsection.

3.3 Extended Indexes for the Singleton Classes

For intervals *i* registered at **lower**, we know in advance that they do not start later than the lower query bound, *i*.lower ≤ *q*.lower. As long as **lower** is distinct from **fork** we also know that the interval *i* ends before the upper query bound, *i*.upper < *q*.fork ≤ *q*.upper. Thus, the database interval *i* may participate in the relationships *meets*, *overlaps*, or *starts* but in no other relationships. For *meets*, the upper bound is tested against the lower query bound by scanning the relational upperIndex, and for *starts*, the lower bound of the database interval *i* is compared to the lower query bound by scanning the relational lowerIndex:

```
SELECT id FROM upperIndex i WHERE i.node = :lower
    AND i.upper = :lower;     -- for meets (even if lower = fork)

SELECT id FROM lowerIndex i WHERE i.node = :lower
    AND i.lower = :lower;     -- for starts (if lower ≠ fork)
```

For the relationship *overlaps*, both bounds of the database interval *i* have to be compared with the lower query bound, and the required selection predicate is *i*.lower < *q*.lower < *i*.upper. At this point, the problem occurs that neither the relational lowerIndex nor the relational upperIndex alone suffice to process the query, and a join of the two indexes is performed as a preliminary solution to the existing relational schema:

```
SELECT id FROM lowerIndex l, upperIndex u     -- (preliminary) for overlaps
WHERE l.node = :lower AND l.lower < :lower
    AND u.node = :lower AND u.upper > :lower AND l.id = u.id;
```

This join-based approach yields an expensive execution plan, and the performance drawbacks of join processing fully apply. A solution that avoids these disadvantages while introducing only a small overhead for storage space is to extend each of the two relational indexes by the opposite interval bound. The resulting relational schema consists of two primary indexes, preferably stored as index-organized tables:

```
lowerUpperIndex (node, lower, upper, id)      -- extended lowerIndex
upperLowerIndex (node, upper, lower, id)      -- extended upperIndex
```

Regarding the space required for the node values and the interval bounds (typically 4 bytes each) and for the referring attribute *id* (e.g. 10 bytes for an extended ROWID in Oracle8i), the storage space for the new indexes is only 17% larger than for the previous relational schemata lowerIndex and upperIndex. An important observation is that all of the hitherto investigated range scans are supported by the extended indexes as well. In the following, we use the symbol 'upper(Lower)Index' if we actually scan the upperLowerIndex but the upperIndex would be sufficient. Analogously, we write 'lower(Upper)Index' in the symmetric case. For the relationship *overlaps*, the scan to retrieve the qualifying entries from the node **lower** may now be based on any of the two indexes, lowerUpperIndex or upperLowerIndex, and it has the following form:

```
SELECT id FROM lowerUpperIndex /*upperLowerIndex*/ i WHERE i.node = :lower
    AND i.lower < :lower AND i.upper > :lower;     -- for overlaps
```

The symmetric considerations hold for intervals registered at **upper**. The relationships *finishes* and *metBy* are immediately supported by the respective relational upperIndex or lowerIndex and, following the general observation from above, also by the new extended indexes upperLowerIndex and lowerUpperIndex. For the relationship *overlappedBy*, a join of lowerIndex and upperIndex would be necessary and, again, is avoided by using any one of the extended indexes, lowerUpperIndex or upperLowerIndex:

```
SELECT id FROM upper(Lower)Index i WHERE i.node = :upper
    AND i.upper = :upper;     -- for finishes (if upper ≠ fork)

SELECT id FROM lower(Upper)Index i WHERE i.node = :upper
    AND i.lower = :upper;     -- for metBy (even if upper = fork)

SELECT id FROM lowerUpperIndex /*upperLowerIndex*/ i WHERE i.node = :upper
    AND i.lower < :upper AND i.upper > :upper;     -- for overlappedBy
```

3.4 Heuristics to Scan the Fork Node

The node contributing to the most interval relationships among all query classes is **fork**. Intervals registered at the query's fork node may participate in *overlaps*, *finishedBy*, *contains*, *starts*, *equals*, *finishes*, *during*, *startedBy*, and *overlappedBy* but not in *before* or *after* since the database intervals at **fork** are known to intersect the query interval at least at the fork value. The relationships *meets* and *metBy* can only hold if **fork** coincides with **lower** and **upper**, respectively, and they are already covered by handling those nodes.

In any case, query processing has to consider both bounds of the registered intervals. In order to avoid expensive join processing, we exploit the extended relational indexes.

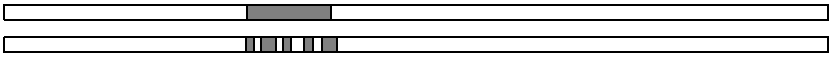


Fig. 6. Blocked and non-blocked index scan for a single range query

The relationships *starts* and *startedBy* are best supported by the lowerUpperIndex, and for the relationships *finishedBy* and *finishes*, the upperLowerIndex is suited best since one of the interval bounds is tested for equality, i.e. lower = :lower or upper = :upper, respectively:

SELECT id FROM lowerUpperIndex i WHERE i.node = :**fork**
AND i.lower = :lower AND i.upper Θ :upper; -- set Θ to < (*starts*), > (*startedBy*)

SELECT id FROM upperLowerIndex i WHERE i.node = :**fork**
AND i.upper = :upper AND i.lower Θ :lower; -- set Θ to < (*finishedBy*), > (*finishes*)

Due to the equality condition for one of the interval bounds, blocked output can be guaranteed which means that no non-qualifying entries have to be discarded from the index range scan. Figure 6 provides an example for a blocked index scan where all of the entries between the first and the last result belong to the actual result set, and an additional example for a non-blocked output where some entries between the first and the last result do not qualify for the result set. This case may occur for the relationships *overlaps*, *contains*, *equals*, *during*, and *overlappedBy* regardless of which of the two relational indexes is scanned.

SELECT id FROM lowerUpperIndex /*upperLowerIndex*/ i WHERE i.node = :**fork**
AND i.lower Θ_1 :lower AND i.upper Θ_2 :upper; -- set Θ_1, Θ_2 to (=, =) (*equals*),
-- (<, <) (*overlaps*), (<, >) (*contains*), (>, <) (*during*), (>, >) (*overlappedBy*)

In the following, we develop heuristics to minimize the overhead of scanning non-qualifying entries in an index in order to choose the best index for **fork** node scans. Our heuristics are based on the observation that the extensions of the intervals from the fork node to the lower bound and from the fork node to the upper bound follow an exponential distribution. We normalize the distances by the step width $step = 2^l$ associated with each level l in the node resulting in a model obeying the same mean value and standard deviation over all entries in the indexes.

The distribution functions that estimate the number of entries registered at the query fork node are given by $\Phi_{left}(lower) = e^{-\beta\,(fork-lower)/step}$ for intervals that start at *lower* or earlier and by $\Phi_{right}(upper) = e^{-\beta\,(upper-fork)/step}$ for intervals that start at *upper* or later. The parameter β is the reciprocal mean value of the normalized differences (*fork* – *lower*)/*step* and (*upper* – *fork*)/*step* and is stored in the RI-tree metadata table. Figure 7 depicts the empirically measured frequencies for our real data set of 1,000,000 intervals from weblog data as well as the approximating density functions Φ_{left} and Φ_{right}.

For the relationship *contains*, the condition 'i.lower < :lower AND i.upper > :upper' has to be evaluated for the entries at the fork node. The fraction of entries scanned when using the *upperLowerIndex* is estimated by $1 - \Phi_{right}(upper)$ whereas the fraction of entries scanned by using the lowerUpperIndex is estimated by $1 - \Phi_{left}(lower)$. Let us em-

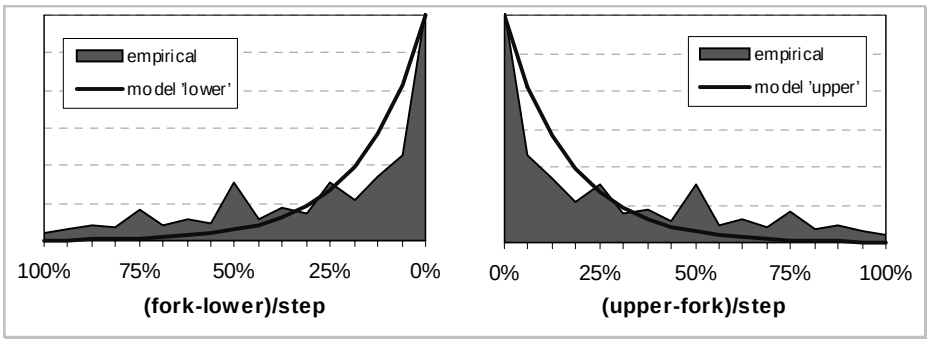


Fig. 7. Empirical frequency and predicted density for the normalized interval extensions from fork to lower (left diagram) and from fork to upper (right diagram)

phasize that the number of qualifying intervals is the same for both ways, and choosing the index scanning the smaller fraction of entries obviously improves the performance of query processing since the overhead to discard non-qualifying entries is minimized. Due to the monotonicity of Φ_{left} and Φ_{right}, the test is simplified to the comparison of (*fork – lower*) and (*upper – fork*) in case of the relationship *contains*. If the latter difference is greater, the *upperLowerIndex* will be employed, and if the prior expression is greater, the *lowerUpperIndex* is chosen.

For the relationship *during*, we compare Φ_{left}(*lower*), i.e. the estimated fraction of entries between lower and fork, and Φ_{right}(*upper*), i.e. the estimated fraction between fork and upper, in order to assess whether the condition 'i.lower > :lower' or the condition 'i.upper < :upper' is more selective. For the relationship *overlaps*, the fraction of entries to the left of lower, i.e. $1 - \Phi_{left}$(*lower*), and the fraction of entries between fork and upper, i.e. Φ_{right}(*upper*), is compared. Finally, the relationship *overlappedBy* requires a comparison of the estimated fraction between lower and fork, i.e. Φ_{left}(*lower*), with the estimated fraction of entries to the right of upper, i.e. $1 - \Phi_{right}$(*upper*). In the experimental section, we will show the effectiveness of this approach which yields only a few percent of overhead for scanning over non-qualifying index entries as opposed to a fixed assignment of the fork node scan to any one of the two relational indexes. The cost for these optimizations are negligible since only a few arithmetic operations are performed.

3.5 Closing Gaps for the Range Classes

A few nodes only or even a single node have to be visited in the relational indexes to completely process the traversal classes and the singleton classes. The range classes, however, span wide ranges of nodes. Most of the entries at nodes from **allLeft**, for instance, participate in the relationship *before*. Only a few intervals belong to other relationships including *meets*, *overlaps*, *finishedBy*, or *contains*. These intervals, if any, are registered in **topLeft** or **bottomLeft** nodes, and they occur as gaps in the range of intervals in the relational upper(Lower)Index as illustrated in Figure 8. The number of gaps is bounded by the height of the backbone tree.

Let us discuss two ways to process the **allLeft** nodes. The first way is to exclude the gaps from the index range scans by "inverting" the **topLeft** and **bottomLeft** lists with

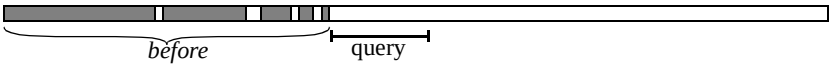


Fig. 8. Index ranges to be scanned for a *before* query (with gaps)

respect to **allLeft** nodes. As a result, a pair-valued list *beforeList* of nodes is generated characterizing the ranges of results between two gaps. The entries in the *beforeList* are of the form (firstNode, lastNode), and their number is bound by the height of the backbone tree. The corresponding SQL statement for the relationship *before* is as follows:

```
SELECT * FROM upperIndex i, :beforeList q     -- (preliminary) for before
WHERE i.node BETWEEN q.firstNode AND q.lastNode AND i.upper < :lower;
```

This statement includes the correct handling of nodes from **topLeft** and **bottomLeft**, so it is complete for the relationship *before*. Whereas the advantage of this solution is the blocked output for the results, there are also some drawbacks. In practice, the gaps are very small and typically occupy only a fraction of a disk block. Thus, scanning the gaps in addition to the results rarely yields any I/O overhead. From a CPU cost point of view, the few additional evaluations are compensated by the saved navigation in the relational index from its root node down to the current leaf nodes. As a minor aspect, the pair-valued list increases the complexity of the query processor since an additional list type needs to be managed. Altogether, scanning all nodes before the lower query bound seems to be a very practical solution. The code for the relationship *before* and, analogously, for the relationship *after* which scans the **allRight** nodes is therefore as follows:

```
SELECT * FROM upper(Lower)Index i WHERE i.node < :lower
    AND i.upper < :lower;     -- for before

SELECT * FROM lower(Upper)Index i WHERE i.node > :upper
    AND i.lower > :upper;     -- for after
```

A similar situation occurs for the node class **allInner** that covers all results of the relationship *during* except for the cases discussed for the node classes **innerLeft**, **fork**, and **innerRight**. Rather than excluding those entries as gaps, we again scan the entire range of **allInner** nodes. For inner nodes to the left of **fork**, the lower bounds of the registered intervals are compared to the lower query bound. Whereas this test is only necessary for the few nodes also in **leftInner**, it does not affect the other nodes since their registered intervals start after **lower** in any case. Analogously, the upper bounds of the inner nodes to the right of **fork** are compared to the upper query bound to report the results while excluding the gaps potentially occurring at **rightInner** nodes:

```
SELECT id FROM lower(Upper)Index i WHERE :lower < i.node AND i.node < :fork
    AND i.lower > :lower     -- left hand part for during

SELECT id FROM upper(Lower)Index i WHERE :fork < i.node AND i.node < :upper
    AND i.upper < :upper     -- right hand part for during
```

What remains to be done for the relationship *during* is to scan the database intervals registered at the **fork** node as discussed above.

3.6 Complete Query Processing

In the preceding subsections, we have derived SQL statements to process general interval relationship queries for the different classes of nodes. For example, the **innerLeft** nodes have to be accessed for *overlaps*, *starts*, or *during* queries. We now switch to the inverse view and collect the classes of nodes that contribute to each interval relationship in order to assemble SQL statements that guarantee completeness for query processing. For instance, a *starts* query has to scan nodes from **innerLeft**, **lower** and **fork**. Table 1 provides a survey of the inverse view.

Table 1: General interval relationships and affected node classes

before	**allLeft** (including **topLeft** and **bottomLeft**)
meets	**topLeft**, **bottomLeft**, **lower**
overlaps	**topLeft**, **bottomLeft**, **innerLeft**, **lower**, **fork**
finishedBy	**topLeft**, **fork**
starts	**innerLeft**, **lower**, **fork**
contains	**topLeft**, **fork**, **topRight**
equals	**fork**
during	**allInner** (including **innerLeft**, **fork**, **innerRight**)
startedBy	**topRight**, **fork**
finishes	**innerRight**, **upper**, **fork**
overlappedBy	**topRight**, **bottomRight**, **innerRight**, **upper**, **fork**
metBy	**topRight**, **bottomRight**, **upper**
after	**allRight** (including **topRight** and **bottomRight**)

The individual SQL statements for the node classes are now combined to the final SQL queries. Since the single SQL statements yield disjoint result sets and, thus, no duplicates are produced, we employ the operator UNION ALL thus saving the effort of eliminating duplicates from the final result. In addition, UNION ALL is a non-blocking operator which fully supports pipelining and fast retrieval of first results. Table 2 comprises the complete SQL statements to process each of Allen's interval relationships based on the two relational indexes *lowerUpperIndex* and *upperLowerIndex*. Rather than to include the heuristic fork node optimization, we present particular instances of the queries in case of the relationships *contains*, *during*, *overlaps*, and *overlappedBy*.

Let us finally analyze the I/O complexity of the algorithms. For *meets*, *finishedBy*, *starts*, *startedBy*, *finishes*, and *metBy*, the bound of $O(h \cdot \log_b n + r/b)$ I/Os is guaranteed since no false hits need to be discarded when scanning the fork node. For *equals*, even $O(\log_b n + r/b)$ I/Os suffice. For *overlaps*, *contains*, *during*, and *overlappedBy*, the proposed heuristics help to reduce the disk accesses but do not affect the worst case complexity of $O(h \cdot \log_b n + r/b + f/b)$ I/Os where f denotes the number of database entries registered at the fork node of the query interval. Whereas $f = O(n/2^h)$ holds in the average

Table 2: Complete SQL statements for interval relationships (w/o fork optimization)

before	SELECT id FROM upperLowerIndex i WHERE i.node < :lower AND i.upper < :lower
meets	SELECT id FROM upperIndex i, :(**topLeft** ∪ **bottomLeft** ∪ **lower**) q WHERE i.node = q.node AND i.upper = :lower
overlaps	SELECT id FROM upperLowerIndex i, :(**topLeft** ∪ **bottomLeft**) q WHERE i.node = q.node AND :lower < i.upper AND i.upper < :upper UNION ALL SELECT id FROM lowerUpperIndex i, :(**innerLeft** ∪ **lower** ∪ **fork**) q WHERE i.node = q.node AND i.lower < :lower AND i.upper < :upper AND i.upper > :lower
finishedBy	SELECT id FROM upperLowerIndex i, :(**topLeft** ∪ **fork**) q WHERE i.node = q.node AND i.upper = :upper AND i.lower < :lower
starts	SELECT id FROM lowerUpperIndex i, :(**innerLeft** ∪ **lower** ∪ **fork**) q WHERE i.node = q.node AND i.lower = :lower AND i.upper < :upper
contains	SELECT id FROM lowerUpperIndex i, :(**topRight** ∪ **fork**) q WHERE i.node = q.node AND i.lower < :lower AND :upper < i.upper UNION ALL SELECT id FROM upperLowerIndex i, :**topLeft** q WHERE i.node = q.node AND :upper < i.upper
equals	SELECT id FROM lowerUpperIndex i /*or upperLowerIndex*/ WHERE i.node = :**fork** AND i.lower = :lower AND i.upper = :upper
during	SELECT id FROM lowerUpperIndex i WHERE i.node > :lower AND i.node <= :**fork** AND i.lower > :lower AND i.upper < :upper UNION ALL SELECT id FROM upperLowerIndex i WHERE i.node > :**fork** AND i.node < :upper AND i.upper < :upper
startedBy	SELECT id FROM lowerUpperIndex i, :(**topRight** ∪ **fork**) q WHERE i.node = q.node AND i.lower = :lower AND :upper < i.upper
finishes	SELECT id FROM upperLowerIndex i, :(**innerRight** ∪ **upper** ∪ **fork**) q WHERE i.node = q.node AND i.upper = :upper AND :lower < i.lower
over-lappedBy	SELECT id FROM lowerUpperIndex i, :(**topRight** ∪ **bottomRight**) q WHERE i.node = q.node AND :lower < i.lower AND i.lower < :upper UNION ALL SELECT id FROM upperLowerIndex i, :(**innerRight** ∪ **upper** ∪ **fork**) q WHERE i.node = q.node AND i.upper > :upper AND i.lower > :lower AND i.lower < :upper
metBy	SELECT id FROM lowerIndex i, :(**topRight** ∪ **bottomRight** ∪ **upper**) q WHERE i.node = q.node AND i.lower = :upper
after	SELECT id FROM lowerUpperIndex i WHERE i.node > :upper AND i.lower > :upper

case, *f* approaches *n* in the degenerate case where that particular fork node is populated significantly above the average. For *before* and *after*, the sequential heuristic algorithms may have to scan *n* entries in degenerate cases but they are expected to share the I/O complexity of the straightforward algorithms which is $O(h \cdot \log_b n + r/b)$ in the average case. A histogram based cost model will help to recognize large fruitless gaps and to exclude them from the range scans.

4 Experimental Evaluation

We implemented the Relational Interval Tree in Oracle8i and extended it by the new algorithms for Allen's interval relationships. For our experiments, we used a real data set of 1,005,447 intervals representing sessions on a frequently accessed web server over a range of two years. The interval bounds range from 31,419 to 66,394,508, resulting in a height of 26 for the virtual backbone tree. From this data set, we randomly selected a sample of 1,000 intervals which we used as query objects. What we demonstrate in the following is the effect of our heuristics to minimize the overhead of scanning non-qualifying index entries which have to be discarded when scanning the node ranges.

The diagrams in Figure 9 illustrate the characteristics of our data set and of the randomly selected queries. From the distribution of the interval lengths depicted in the top diagrams, we can see that there are two peaks, one for intervals lasting some 10 seconds and one for intervals of some 200,000 seconds. The bottom diagrams show the height distribution of the fork nodes, starting with a height of one for the leaf nodes.

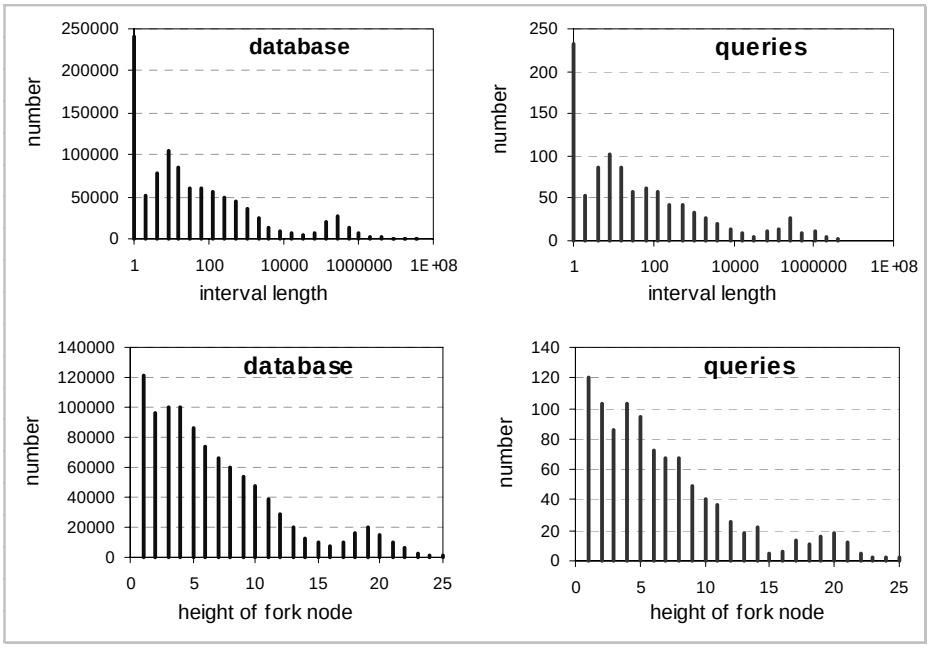


Fig. 9. Characteristics of the real data set and of the query samples: Distribution of the interval lengths (top row) and distribution of the node heights (bottom row)

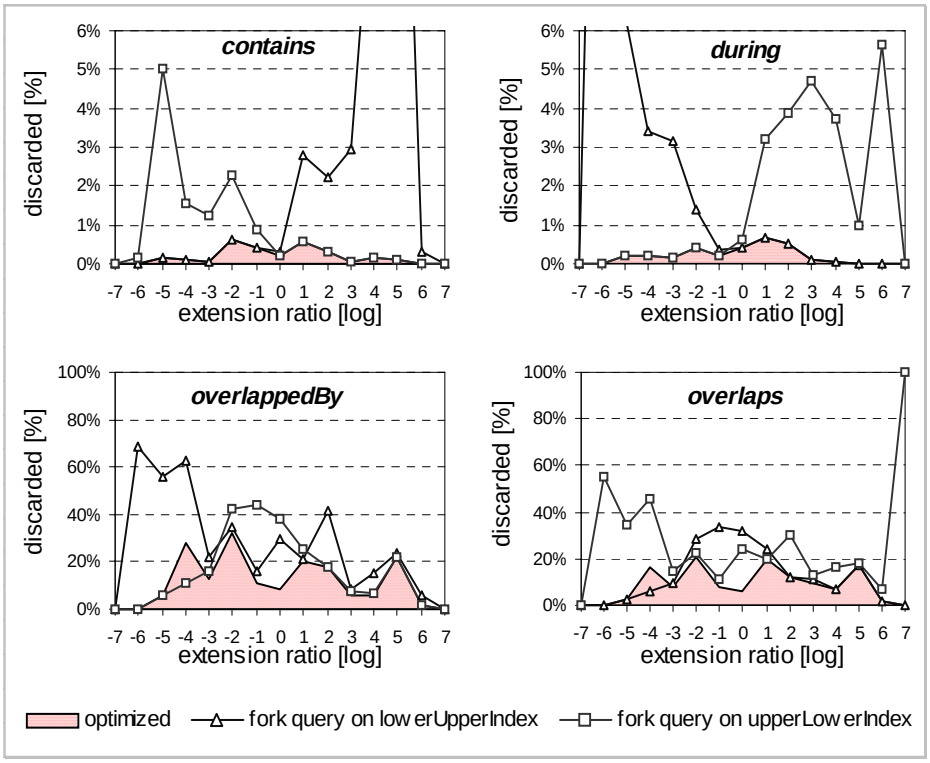


Fig. 10. Percentage of the scanned but discarded index entries at the fork node for the relationships *contains*, *during*, *overlappedBy*, and *overlaps* depending on the ratio of the extensions from the fork node of the query to upper and to lower, respectively. Among the three index usage policies, the optimized version is marked by the shaded area

Figure 10 demonstrates the effect of assigning the index range scan for the fork node to the lowerUpperIndex or to the upperLowerIndex, depending on the chosen heuristics. The abscissa axis ranges over the ratio of the extensions from fork to lower and from fork to upper and is scaled logarithmically. The ordinate axis indicates the percentage of discarded index entries when scanning the fork node. Let us focus on the diagram for *contains*. As expected, running the fork node scan on the *lowerUpperIndex* works well if the fork node of the query is close to the lower query bound (left hand side in the diagram) and is very expensive if fork is close to the upper query bound (the right hand side). Analogously, scanning the *upperLowerIndex* is the better choice if the fork node of the query is closer to *upper* than to *lower*. The effect of the heuristics to choose the index depending on the relative position of the fork value between *lower* and *upper* is depicted at the bottom by the shaded area. For *contains* as well as for *during*, the heuristics actually yield the minimum number of index entries which have to be discarded. A similar observation holds for *overlaps* and *overlappedBy* queries, and the heuristics based on the exponential distribution of interval bound distances to the fork node misses the optimum only in some few cases where the fork node is near to the lower bound.

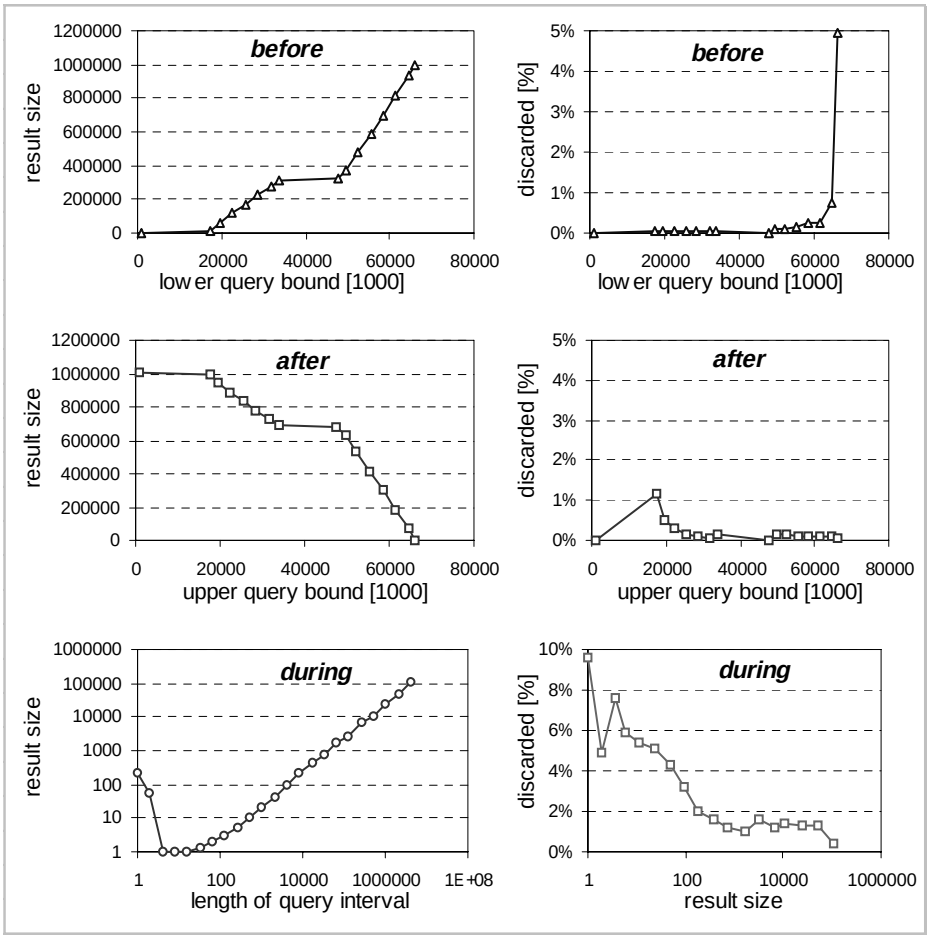


Fig. 11. Percentage of scanned but discarded index entries depending on the height of the query fork node in the virtual backbone tree (relationships *before*, *after*, and *during*) and on the result size (*during* only)

Our next series of experiments addresses the overhead produced by scanning non-qualifying index entries for the interval relationships *before*, *after*, and *during* (see Figure 11). As expected for increasing lower and upper query bounds, the number of results of a *before* query increases and the number of results of an *after* query decreases. Except for a few rare cases, the discarded index entries are significantly below 1%. The selectivity of *during* queries depends on the length of the query interval as indicated by the diagrams in the bottom row. Again, only a few percent of the scanned index entries need to be discarded in most cases. Fractions of 5% to 10% occur only for very small result sets of ten or less answers.

The basic lesson we learn from these observations is that the analysis of the intersection query processing in [KPS 00] also applies to general interval queries. The overhead

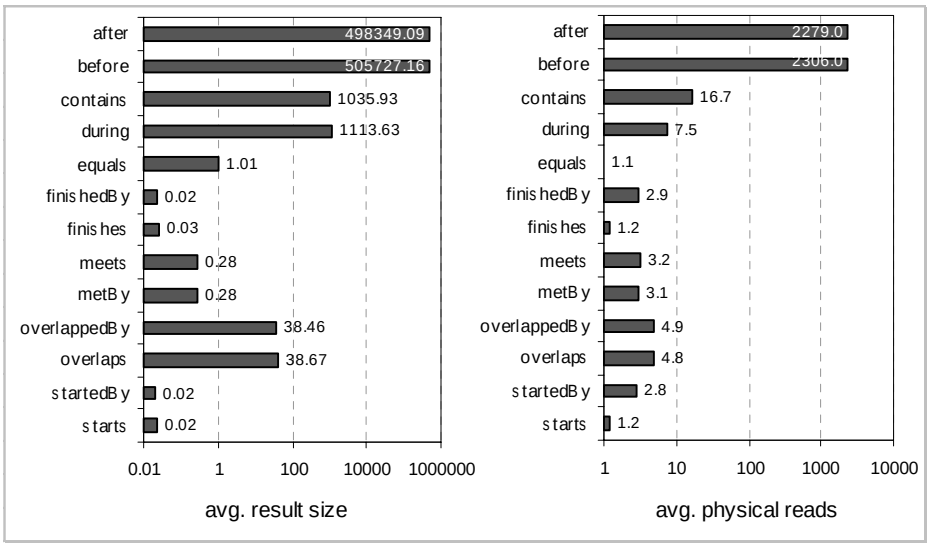


Fig. 12. Result sizes and disk accesses for the individual interval relationships

of discarding non-qualifying index entries is in the range of a few percent, and the number of I/O operations as well as the response time is still dominated by the cardinality of the result set. Figure 12 demonstrates the result sizes and the number of disk accesses averaged over 1000 queries for each of the individual interval relationships. As expected, the number of I/O operations is closely related to the number of results.

5 Conclusions

In this paper, we introduced algorithms for Allen's interval relationships as new query types for the Relational Interval Tree, a purely relational storage structure that manages interval data efficiently on top of SQL. The extension of the previous relational lowerIndex and upperIndex by the opposite query bound, upper and lower, respectively, suffices to support the general interval relationships efficiently. We classified the nodes to be scanned in the indexes into the *traversal*, *singleton*, and *range* classes and investigated the individual handling in each case. Effective heuristics for assigning the fork node scan to one of the two relational indexes are proposed which significantly decrease the number of non-qualifying index entries that are read from the database but do not contribute to the result set. The methodical part is concluded by a simplification for the range node classes thus reducing the query preprocessing effort for *before* and *after* queries. We integrated the algorithms into our Oracle8i domain index implementation of the Relational Interval Tree and observed that the chosen heuristics are very effective such that the good performance measured in [KPS 00] also applies to the general interval relationships.

References

[All 83] Allen J. F.: *Maintaining Knowledge about Temporal Intervals*. Communications of the ACM 26(11): 832-843, 1983

[BSSJ 99] Bliujute R., Saltenis S., Slivinskas G., Jensen C.S.: *Developing a DataBlade for a New Index*. Proc. IEEE 15th Int. Conf. on Data Engineering (ICDE): 314-323, 1999

[CCF+ 99] Chen W., Chow J.-H., Fuh Y.-C., Grandbois J., Jou M., Mattos N., Tran B., Wang Y.: *High Level Indexing of User-Defined Types*. Proc. 25th Int. Conf. on Very Large Databases (VLDB): 554-564, 1999

[Ede 80] Edelsbrunner H.: *Dynamic Rectangle Intersection Searching*. Inst. for Information Processing Report 47, Technical University of Graz, Austria, 1980

[EM 99] Eisenberg A., Melton J.: *SQL:1999, formerly known as SQL3*. ACM SIGMOD Record, 28(1): 131-138, 1999

[GLOT 96] Goh C. H., Lu H., Ooi B. C., Tan K.-L.: *Indexing Temporal Data Using Existing B+-Trees*. Data & Knowledge Engineering, Elsevier, 18(2): 147-165, 1996

[IBM 99] IBM Corp.: *IBM DB2 Universal Database Application Development Guide, Vs. 6.* Armonk, NY, 1999

[Inf 98] Informix Software, Inc.: *DataBlade Developers Kit User's Guide*. Menlo Park, CA, 1998

[JS 99] Jensen C. S., Snodgrass R. T.: *Temporal Data Management*. IEEE Transactions on Knowledge and Data Engineering, 11(1): 36-44, 1999

[Kor 99] Kornacker M.: *High-Performance Extensible Indexing*. Proc. 25th Int. Conf. on Very Large Databases (VLDB): 699-708, 1999

[KPS 00] Kriegel H.-P., Pötke M., Seidl T.: *Managing Intervals Efficiently in Object-Relational Databases*. Proc. 26th Int. Conf. on Very Large Databases (VLDB): 407-418, 2000

[LSD+ 01] Li W., Snodgrass R. T., Deng S., Gattu V. K., Kasthurirangan A.: *Efficient Sequenced Temporal Integrity Checking.* Proc. IEEE 17th Int. Conf. on Data Engineering (ICDE): 131-140, 2001

[ND 99] Nascimento M. A., Dunham M. H.: *Indexing Valid Time Databases via B+-Trees*. IEEE Trans. on Knowledge and Data Engineering (TKDE) 11(6): 929-947, 1999

[OLW 01] Olston C., Loo B. T., Widom J.: *Adaptive Precision Setting for Cached Approximate Values.* Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001

[Ora 99] Oracle Corp.: *Oracle8i Data Cartridge Developer's Guide, Release 2 (8.1.6).* Redwood City, CA, 1999

[Ram 97] Ramaswamy S.: *Efficient Indexing for Constraint and Temporal Databases.* Proc. 6th Int. Conf. on Database Theory (ICDT), LNCS 1186: 419-413, 1997

[Sam 90] Samet H.: *Applications of Spatial Data Structures*. Addison-Wesley, 1990.

[SMS+ 00] Srinivasan J., Murthy R., Sundara S., Agarwal N., DeFazio S.: *Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i*. Proc. IEEE 16th Int. Conf. on Data Engineering (ICDE): 91-100, 2000

[Sno 00] Snodgrass R. T.: *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000

[Sto 86] Stonebraker M.: *Inclusion of New Types in Relational Data Base Systems.* Proc. IEEE 2nd Int. Conf. on Data Engineering (ICDE): 262-269, 1986

[TCG+ 93] Tansel A. U., Clifford J., Gadia S., Jajodia S., Segev A., Snodgrass R.: *Temporal Databases: Theory, Design and Implementation*. Redwood City, CA, 1993.

[TJS 98] Torp K., Jensen C. S., Snodgrass R. T.: *Stratum Approaches to Temporal DBMS Implementation.* Proc. IDEAS Conf.: 4–13, 1998