Relational Implementation of Multi-dimensional Indexes for Time Series

Tomasz Nykiel[1] and Parke Godfrey[2]

[1] University of Toronto, Toronto ON
tnykiel@cs.toronto.edu
[2] York University, Toronto ON
godfrey@cse.yorku.ca

Abstract. Similarity search over time-series data is a useful, but expensive, application. Sequence data can be transformed via an orthonormal transformation, which is then dimensionally reduced, to be indexed. R-trees and R*-trees have been used for this purpose. These do not scale well, however, for very large datasets.
We propose a new indexing data-structure within relational tables, indexed with B+-tree indexes. We employ SQL in the retrieval process, and achieve sequential, instead of random I/O, access. We perform comprehensive experiments to test and compare our proposed technique against existing solutions, and demonstrate it to be scalable and more efficient for very large datasets.

1 Introduction

Time-series data record values of a measure at sequential points in time. The need for search over such data arises in many real-world applications. Mining astronomical [1, 2] and biomedical data [3] are large challenges, due to sizes of the datasets involved. Time-series similarity matching is used as a subroutine in many applications in *knowledge discovery in databases* (KDD), such as clustering, classification, mining association rules, and future trend discovery.

Most techniques perform dimensionality reduction, and use spatial access methods to index the data in the transformed space. The indexes can be used to store and index all the (contiguous) sub-sequences of a given window size. The R-tree has been the standard approach for the indexing. Techniques that have been used for dimensionality reduction include, among others, *DFT* [4] *DWT* [5] *SVD* [6, 7], and *PAA* [8, 9].

Keogh and Kasetty [10] presented a comprehensive overview of the problems with existing solutions. First, the indexing approaches do not scale. It is intended they reside in main memory. They do not perform well as *external* indexes—that is, when stored on, and used from, disk—because inherently random access patterns. Second, the indexes grow exponentially with respect to the dimensionality. Our research concentrates on developing easy-to-scale indexing techniques that can be built within, and tuned for, standard relational platforms, to enable efficient similarity search for very large datasets.

We have developed a multi-dimensional indexing data structure called the *SQL "R" tree* (SQLRtree). It does not characterize all the feature points, but groups them into regions called *minimum bounding rectangles* (MBRs). These are then stored in an index tree. The structure is *multi-resolutional* with respect to different window sizes. The index tree is mapped as a table which is indexed with B+-tree indexes. We achieve sequential access patterns in the retrieval process by judicious design of the underlying indexes and retrieval queries.

We implement two SQL-based retrieval procedures: *PrefixSearch* and *MultiMatching*. The first does not exploit the index fully, but relies on faster pruning. The second combines information from different window sizes—or multiple instances of the same window size—to match more precisely the query sequence. We investigate the trade-offs between these two approaches. The procedures are carefully designed and implemented so that the query plans issued by the query optimizer support fast retrieval and efficient disk access patterns. Our analysis and experiments confirm this, and demonstrate that our approach is highly scalable, and outperforms the standard R-tree indexes.

Ultimately, one might extend relational systems to handle time-series data naturally, and to embed needed techniques into the platform to handle similarity search efficiently. An intermediate approach—the one that we take—is to build indexes for time-series data on top of the existing relational platform, and to handle similarity queries via SQL. A disadvantage is that one may pay various overhead, which one could potentially avoid if one built from scratch. The advantage is that one can leverage the technology of the relational system, such as buffer pool management, query optimization, and standard, external indexes.

In §2, we review the relevant work to multidimensional indexing in time-series searches. In §3, we present the design of the SQLRtree. In §4, we present a performance evaluation and analyze the results. In §5, we summarize our contributions, and offer conclusions.

2 Background and Related Work

Indexes can improve greatly the performance for querying time series. This has two orthogonal aspects: (1) a *dimensionality* reduction to a small, fixed number of dimensions (coefficients) to index; and (2) the index data-structure itself.

A time-series sequence of length n can be considered as a point in n-dimensional space. This n is usually too large—and likely not fixed—for indexing. Commonly, eight to twelve (k) coefficients are used. Retrieval is infeasible with more coefficients, as the index's size grows rapidly with the number of dimensions. Once a viable dimensionality reduction is in place, one still has to index these k-dimensional points efficiently. Our work is focused on this second task.

One cannot reasonably restrict focus to just sequences that *exactly* match the query, as these would be exceedingly rare. Instead, one wants to find sequences that are within a given *distance* ϵ. Thus, the index structure must support range queries. The dimensionality reduction technique must preserve distances, so not to introduce any *false dismissals*. That is, all sequences that ultimately match to within ϵ are likewise found in the reduced space. This may, however, introduce *false alarms*. Any such distance-preserving reduction is called *orthonormal*.

Agrawal et al. [4] proved that using k initial coefficients of the *discrete Fourier transform* (DFT) is orthonormal. They called this the *F-index*. Chan and Fu [5] proved likewise for the *discrete wavelet transform* (DWT). Other reduction techniques have been proved to be distance preserving too, making them suitable for the task. The choice of which reduction is best for time-series queries is still quite debated. For our work, we use the *discrete wavelet transform* (DWT).[3] Once the candidate matches are found by the index, it is necessary in a *post-processing step* to scan and test the original data, to dismiss the false alarms.

In the literature, first to be considered was *whole matching*; the query sequence and the target sequences must have the same length. Sub-sequence matching generalizes this by allowing the query to be shorter than the target; the task then is to find best matching sub-sequences among the targets. Some work fixed the size of the query sequence allowed, as a simplification. Permitting variable-length queries is, of course, much more useful.

Faloutsos et al. [11] improved on the F-index method for whole-matching queries by using MBRs. This *Gemini Framework* is the basis for much of the subsequent work. One sets a minimum query length of w. Target sequences may be of varying length. A sliding window of size w is passed over the sequences to index their contiguous sub-sequences (of length w). For each window placement, the indexable features are extracted (using one's choice of orthonormal transform). A sequence of length n is thus mapped to a curve, or *trail*, in the feature space (of the index of k dimensions). This trail consists of $n - w + 1$ points.

The trail of a given sequence is divided into sub-trails. Each sub-trail is then represented by its MBR. (An MBR can be represented by its lower left and upper right corners, thus by two vectors of length k.) Only the MBRs are then indexed. This still preserves the property of no false dismissals.

When the query length is w, the algorithm is similar to whole matching. The query sequence is first mapped into the feature space. Given this query "point", all MBRs that intersect with the *query rectangle*—the query's point in the projected, transformed space, plus or minus ϵ—are fetched. A post-processing step examines the retrieved sub-sequences to discard all false alarms.

The case of queries longer than w is more complex, since the index contains information only about sub-sequences of length w. The authors introduced two methods for handling this: *PrefixSearch* and *MultiPiece*. *PrefixSearch* simply queries with the query's prefix sequence (Q) of length w. Of course, this does not take full advantage of the index and query. *MultiPiece* splits the query sequence in p pieces of length w. Each sub-query of length w is then processed, and the resulting sets of candidate matches are intersected. This takes fuller advantage of the index to introduce fewer false alarms, but pays extra computational cost.

[3] The distance between two sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ of equal length k can be measured in different ways. The distance preserving reductions preserve with respect to *any* proper distance metric. Which metric is best suited to time-series is also still debated. We use Euclidean distance, a common choice.

The matching task is complicated when the lengths of the data and query sequences are variable. Kahveci and Singh [12] introduced indexing with multiple window sizes. Let l be the longest sequence in the database. For some integer b, $2^b \leq |l| \leq 2^{b+1}$. Let the minimum length of a query be 2^a, and assume $a \leq b$ (which is reasonable). Let $s_1, s_2, \ldots, s_n$ be the sequences in the database. One must then store a grid of indexes $T_{i,j}$; i ranges from a to b, and j from 1 to n.

Search works as follows. Given some query of size $x2^a$, and a range ϵ, it is partitioned as by "binary representation" into non overlapping sub-queries. Each sub-query is then a different window size. That is, the query Q is partitioned to $Q_1, Q_2, \ldots, Q_t$ with $|q_i| = 2^{c_i}$ and $a \leq c_1 < \ldots < c_i < \ldots < c_t \leq b$. A search for Q_1 using the first row in the index is performed, using the corresponding window size. This repeats for the remaining segments of Q. An advantage of the approach over *MultiPiece* is that the number of sub-queries is smaller compared The drawback is that it uses more storage for the different window sizes.[4]

Most work has chosen for its index data structure to use R-trees (or R*-trees). The *R-tree* spatial index was introduced by Guttman [13]. It is similar to the *B-tree*, but better suited for multi-dimensional data. It splits the space into hierarchically nested *minimum bounding rectangles* (MBRs). These MBRs are permitted to overlap. An *R-tree* node may have a variable number of entries, with a pre-specified minimum and maximum number of entries to store. As with a B-tree, each node must be filled to at least half capacity, with the exception of the root node. A non-leaf node records the MBR that encloses its children, and identifiers for them. A leaf entry stores the MBR of the data elements belonging to it, and identifiers for them. While R-trees provide no guarantees on worst-case performance, they have been found to work well in many real-world applications.[5]

In [15], the authors pursue a goal similar to ours. They simulate R-trees in relational tables with B+-tree indexes to handle multi-dimensional data. Our technique indexes the MBRs in a different way, exploiting the locality of patterns in time series, enabling sequential scans.

3 Design of the SQLRtree

Our key concern is to design the physical database and access methods (SQL queries) to ensure access is primarily via sequential reads. We err on the side of simplicity in our design, to ensure scalability. We build the SQLRtree "index" in tables. We use B+-tree indexes to support fast retrieval. The SQLRtree building and retrieval procedures are then implemented via recursive SQL, in a simple, declarative manner. We assure the procedures are translated by the query optimizer into efficient query plans that exploit the indexes as we intend.

[4] Methods for compressing their index were proposed in [12].

[5] A number of variants of this basic idea have followed, including the *R*-tree*, introduced by Beckmann et al. [14]. This uses a revised node-split algorithm to force reinsertions to achieve better balance. The overhead of this reinsertion algorithm, however, is significant, and would be infeasible for very large datasets.

SQLRtree. The SQLRtree data-structure is similar to the R-tree. It is a tree of hierarchically nested, k-dimensional bounding boxes. Each node has a variable number of children, not to exceed some fixed ceiling. Each non-leaf node stores the MBR of the children nodes, and identifiers for the children. Each leaf-node stores the MBR of its underlying feature points, and the information needed to identify them. Feature points are grouped according to (identifiers of) their sequence positions. Thus, the only information needed to identify the children of a node is the identifier of the first child and the number of children. Each tree indexes one given sequence at one given window size.[6] To index all sequences, a forest of trees is constructed for each window size, and for each sequence.

Functionality. The following operations on the SQLRtree are supported.

Bulk-build. This is used to build initially the tree, and capitalizes on sequential I/O. Feature points are placed in the same order as in their sequences. Each sequence can be scanned with a sliding window to group the feature points into *level 0* boxes. Once the *level 0* nodes are finished, the *level 1* nodes are created with the same strategy. Each node contains a fixed number of children (the fan-out parameter). The bulk build continues until just a single node is created. Here, MBRs are grouped geographically, which is a distinct departure from the R-trees. This will not result in optimal box grouping. However, it is possible to bulk-build the tree (critical for very large datasets). Due to data locality in the actual data (adjacent data points in the time series do not change dramatically, on average), our MBRs should still be compact and filter well.

Insertions. For time-series data, insertions take place at the end of the sequences. As feature-point placement is the same as in the sequence, new points fall to the right shoulder of the SQLRtree. No redistribution is necessary.

Deletions. Deletions usually take place at the beginning of the sequences, and affect only the left shoulder of the tree. No redistribution is necessary.

Retrieval. The input of a search is a query box. The search begins with the root node. For each considered node, it is determined whether the query box intersects with the node's MBR. If not, the node's sub-tree is pruned. If so, the process continues recursively down the sub-tree.

Relational Schema. The SQLRtree and the original data are stored in tables. Thus, they do not rely on main-memory structures. The database system collects detailed statistics on the tables on each column. The schema is as follows.

SEQVAL stores the original data. Each row represents a data sample via a sequence identifier, an offset, and the value itself (a double precision float).

BRECT stores the information about the nodes in the SQLRtree: the sequence id, the window size, the node's level in the tree, and its offset within its level. The row also includes the two vectors describing the bounding rectangle. Finally, included are the identifier of the first child and the number of children. For indexing eight dimensional (8d) space, its columns are SEQ_ID, WIND_SIZE, LEVEL, LEVEL_ID, N0, ..., N7, X0, ..., X7, FIRST_ID, and CNT. The N's and the X's are doubles, representing MBR's "corners"; the rest are integers.

[6] We apply the multi-resolution approach of [12] for more efficient retrieval.

```
with Tree (seq_id, wind_size, level,
        level_id, first_id,cnt) as (
  select R.seq_id, R.wind_size, R.level,
         R.level_id, R.first_id, R.cnt
  from BRECT R
  where R.seq_id = 0 and R.wind_size = 32
    and R.level = 5
    and R.n0 < 84.82654
        ⋮
    and R.x7 > 0.00832
  union all
  select C.seq_id, C.wind_size, C.level,
         C.level_id, C.first_id, C.cnt
  from TREE Parent, BRECT C
  where C.level_id ≥ P.first_id
    and C.level_id < P.first_id + P.cnt
    and C.seq_id = 0
    and C.wind_size = 32
    and C.level = P.level - 1
    and C.n0 < 84.82654
        ⋮
    and C.x7 > 0.008322
)
select  T.seq_id, T.level_id, T.first_id, T.cnt
from    Tree T
where   T.level = 0;
```

(a) PrefixSearch

```
    ⋮
select T.seq_id, T.level_id, T.first_id, T.cnt
from Tree T
where T.level = 0
  and exists (
    select * from BRECT B
    where B.seq_id = 0 and B.wind_size = 32
      and B.level = 0
      and B.level_id = T.level_id + 1
      and B.n0 < 84.88041
          ⋮
      and B.x7 > -0.02939 )
  and exists (
    select * from BRECT B
    where
    B.seq_id = 0 and B.wind_size = 32
      and B.level = 0
      and B.level_id = T.level_id + 2
      and B.n0 < 85.09579
          ⋮
      and B.x7 > -0.02047 )
  and
      ⋮
```

(b) MultiMatching (filtering)

Fig. 1. R-trees: 8d & 16d.

In addition, we use the database's indexing facility to enable fast access. A *unique*, clustered, B+-tree index is declared on each of the two fact tables.
ICLSEQVAL indexes SEQVAL by sequence identifier and offset. Its key is SEQ_ID asc, OFFESET asc. VALUE is an *included* column in the index.
ICLBRECT indexes BRECT by sequence identifier, window size, level, and the identifier within the level. Its key is SEQ_ID asc, WIND_SIZE asc, LEVEL asc, LEVEL_ID asc.[7] As ICLBRECT is clustered, the records are sorted by the index key. As all the children of a qualifying node need to be fetched, this can be done now by sequential reads.

Retrieval Procedures. It cannot be assumed, of course, that the SQLRtree will contain a window size that matches the length of the query. We adapt two retrieval procedures. The first, *PrefixSearch*, uses only a prefix of the query of a length equal to one of the stored window sizes (as in [11]). The second, *MultiMatching*, uses the subsequent parts as well (similar to [11, 12]).

Our *PrefixSearch* procedure is implemented by a recursive SQL query. Fig. 1(a) shows a generated SQL query for this based on a given query and threshold that traverses the SQLRtree to fetch all matching leaf node identifiers. The example SQL query uses a window size of 32. The recursive base fetches the root of the tree. The query box is tested against the MBR, represented by columns $N_0, ..., N_7$ and $X_0, ..., X_7$, which represents the lower "left" and upper "right" of

[7] The feature vectors themselves cannot be included in the index without incurring significant space overhead.

the eight-dimensional MBR. The numbers are representing the query point plus or minus ϵ; they are derived on the fly from user query.

The recursive part iterates level by level over the previous level's qualifying nodes. The join is on the identifier of the first child, and the number of children, as described above. The recursion continues until either no remaining nodes qualify, or it reaches *level 0* of the SQLRtree, the leaf nodes. Since the feature points underlying the leaf nodes are clustered positionally, the index record suffices to determine the offsets to the data records to examine. The SQL query itself does not dictate how this will be executed. The query optimizer determines that.

MultiMatching is executed as follows. First, the index is searched using the prefix of the query point, as in *PrefixSearch*. Then, for each matching leaf node, the subsequent segments of the query are checked. These leaf nodes can be located based on the offsets in the query and on the identifier of the leaf node that matched the prefix. Thus, the leaf nodes corresponding to subsequent segments of the query can be fetched directly from the BRECT table.

We have choices for how we match the subsequent segments. The segments can be equal lengths. The longest possible prefix can be used, and then the subsequent parts use smaller window sizes. Fig. 1(b) shows the subsequent filtering step after fetching leaf nodes that match the query point.[8] In the example, the leaf occupancy is 32 and the window size is 32, as well. The subsequent filtering process involves only qualifying leaf nodes, from *level 0*. These are accessed directly from the BRECT table. Regardless of the technique used, there is always the need to scan the original data to prune the false alarms, which is further discussed in [16].

4 Experiments

4.1 The Setup

Environment. The machine used in the evaluation is a server with two quad-core *Intel Xeon* 3.20 GHz processors. The storage consists of 3.2Tb RAID system. The main memory size is 4Gb. The RDBMS used is IBM DB2 v 9.1.3. The buffer pool size in each experiment was set conservatively at 40Mb.

Datasets. Three synthetic datasets were used, due to the lack of publicly available very large time-series. We used Brownian noise data [10]. Dataset 1 and 2 were approximately 30Gb, each holding a sequence of length 1,000,000,000. Dataset 3 was 120Gb, holding a sequence of 4,000,000,000 data samples. The definition of Brownian noise is $x_{t+1} = x_t + \delta$ where $\delta \in [-step, +step]$. The delta was chosen uniformly at random within plus or minus step. The parameters were set to $x_0 = 1.5$ and $step = 0.001$ [4]. A number of control probes (of length 270) were inserted uniformly at random into the data, enabling us to determine in advance the query selectivity. The query selectivities were 0.1% for Datasets 1 and 3, and 0.001% for Dataset 2. For each dataset, indexes were built for window

[8] The *PrefixSearch* step is omitted for brevity.

sizes varying from 32 to 128, for dimensionalities varying from 8 to 32. Extensive experiments can be found in [16].

The only research that we are aware of for very large time-series is [17]; however, this is for a very large number of sequences of fixed length and only for *whole matching*. Our index is to support sub-sequence matching and accommodates variable length queries and arbitrary sequence lengths.

Scenarios. We tested our technique under multiple scenarios. For evaluating *PrefixSearch*, three *prefix* test scenarios are considered, each using a different window size (w) to match the query prefix: *32*, *64*, and *128*. The prefix is matched with the same distance ϵ as for the whole query. For evaluating *MultiMatching*, six *multi* scenarios are evaluated. Five of these test the technique using a single window size (as in [11]): *2*128*, *2*64*, *4*64*, *2*32*, and *8*32*. Three of these—*2*128*, *4*64*, and *8*32*—use as much of the query sequence as possible. Two of these—*2*64* and *2*32*—do not, but allow us to evaluate the overhead introduced by the *MultiMatching* filtering (in multiple stages) versus the benefit of increased selectivity. The last *multi* test, *128+64+32*, uses the multi-resolutional approach. First, *PrefixSearch* is applied with a 128 length prefix. Then, window sizes 64 and 32 are used in two filtering steps. In the scenarios with multi-stage matching, the distance threshold ϵ is applied to each segment.

We evaluated the standard R-tree approach.[9] The indexes were built at only window size 32 due to the requisite preprocessing time. This one window size suffices to show the infeasibility of the standard approach for large datasets. For the standard approach, we tested *prefix 32*, *multi 2*32*, and *multi 8*32*.

The query used for testing is the same as the probe inserted in the data in the generation phase. The distance ϵ is set to a value which allows more sub-sequences match than just the number of probes inserted. We compare the indexing techniques only, without checking candidates against actual data.

4.2 Query Plans

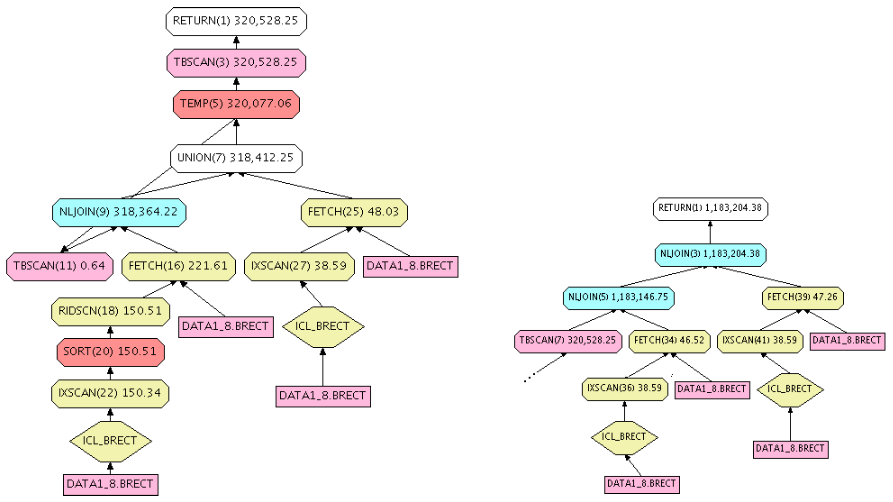
We next consider the query plans generated by the query optimizer to traverse the SQLRtree. We show that the query plans behave as we designed.

PrefixSearch. Fig. 2(a) shows the plan automatically generated by DB2 for *PrefixSearch*. The plan first identifies the original data offsets that are candidates. All windows starting at these offsets must be examined.

Base Case. The right side of the tree, operators (25) and (27), access the root node of the SQLRtree in BRECT. The access path uses the index ICLBRECT, as expected. The root record is fetched, and tested against the query rectangle.

Recursion. Operator (11) scans the qualifying nodes from the previous iteration. The nodes are joined with their children via nested-loop join (9). Operator (16) fetches the children, accessing the BRECT table—indirectly, via an index-only plan—using the clustered B+-tree index ICLBRECT (22). For each ICLBRECT index entry, the record is fetched from the BRECT table and checked against the query box. Since the ICLBRECT index is clustered on appropriate columns,

[9] We did not consider R*-trees here as the time to construct them would be prohibitive.



(a) *PrefixSearch.* (b) *MultiMatching.*

Fig. 2. Query plans.

fetched children nodes are on sequential pages. This exploits data locality, as many pages that are needed are already in the buffer pool. Once the current level is evaluated, the recursion iterates for the next level.

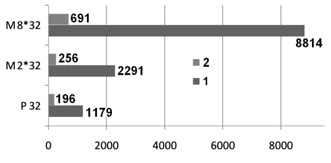
MultiMatching. Fig. 2(b) shows an example of the *MultiMatching* plan.
Prefix. The omitted part on the left is the same as in the previous example. It evaluates the candidate leaf nodes that match the prefix of the query.
Subsequent Segments. Once a leaf node has qualified, the two subsequent segments are checked. Two nested loop joins (3) and (5) fetch the leaf nodes that contain the subsequent segments of the qualifying sub-sequences. The logical SQLRtree is not traversed; rather, these leaf nodes are fetched directly, (36) and (41), using the ICLBRECT index. Once a subsequent-segment leaf node is fetched, the BRECT table is accessed to retrieve its MBR.

When the segments are all the same window size, the leaf nodes fetched by (34) and (39) are likely to be already in the buffer pool. This also true of the index pages when the ICLBRECT index is accessed in (36) and (41), because the SQLRtree nodes are clustered by level identifier. However, when the segments are of different sizes, different pages have to be accessed, which increases the I/O load. When more segments of the query are used, the query plan is analogous.

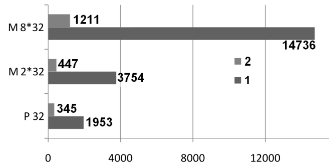
4.3 Time Results

We present a performance analysis. First, we demonstrate that R-trees are not scalable to very large datasets. Next, we present results for the SQLRtree.

Dataset 1 is generated to have *low selectivity* with the probe query; that is, there are a high number of matches. Dataset 2 is reasonably highly selective, which allows us to measure likely performance of our method in real applications.

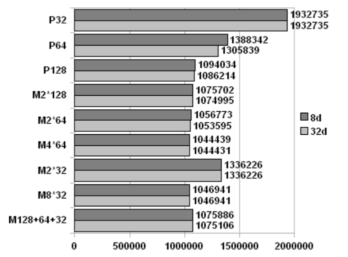


(a) 8d

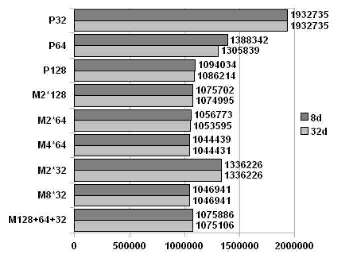


(b) 16d

Fig. 3. R-trees: 8d & 16d.



(a) Number of fetched MBRs

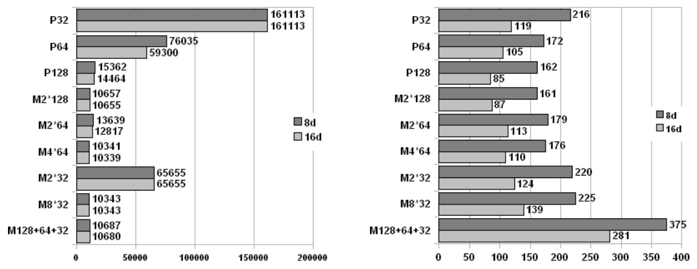


(b) Time in (s)

Fig. 4. Dataset 1.

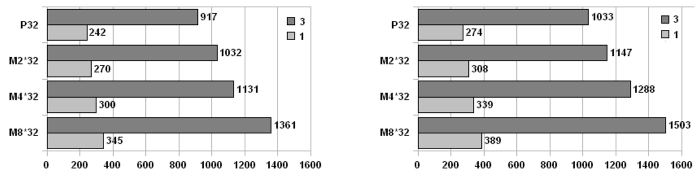
For Datasets 1 and 3, the probe is inserted, on average, at every 1000 offsets. The fan-out of the SQLRtree is set to 32. This means that each *level 0* node covers 32 offsets in the original sequence; so each *level 1* node covers 1024 points. Since the probe distribution is uniform, *all level 1* nodes are likely to intersect with the query rectangle. This implies that all *level 0* nodes have to be examined (worst-case scenario). In the case of the R-tree approach, this is not the case. As bounding boxes are clustered based on their neighborhoods, the whole structure does not likely have to be traversed. Our results show, however, that SQLRtree outperforms R-trees in this scenario. For Dataset 2, the probe is inserted, on average, at every 100,000 offsets. The datasets each consist of one sequence. However, note that multiple sequences are easily accommodated in our schema.
R-Tree. For the standard approach, Fig. 3(a) reports the execution times for the two datasets for each of the scenarios described above. The execution times for Dataset 1 are high (up to 20 minutes), even though the number of bounding boxes fetched in *prefix 32* did not exceed 6% of the total. It has been claimed that R-trees work efficiently for queries with selectivities of up to 10%.

The time to perform *multi 2*32* and *multi 8*32* for Dataset 1 were longer—two and eight times (2 hours), respectively—than for *prefix 32*, which confirms minimal page reuse. For Dataset 2, the time needed for multiple matching did not grow linearly with the number of segments. This suggests that many pages are reused subsequently. However, the times are still up to 11 minutes. Fig. 3(b) presents the results for the test that used 16 coefficients for indexing. The retrieval time went dramatically up (up to 4 hours). The standard R-tree indexing technique is not efficient for large datasets or high-dimensional indexing.
SQLRtree. We report the performance of the SQLRtree under the different scenarios for Dataset 1, for 8d and 32d trees. Fig. 4 reports the number of fetched



(a) Number of fetched MBRs (b) Time in (s)

Fig. 5. Dataset 2.



(a) 8d (b) 16d

Fig. 6. Comparison of Datasets 1 & 3.

boxes (a) and the time spent (b). For the 8d structure in multi-matching, the time needed for 2-stage matching is 15% higher than for the prefix search only. For 4-stage matching, it is 25% more, and for 8-stage, 45% higher. This demonstrates beneficial locality in the buffer pool. For *multi 128+64+32* involving different window sizes, the performance degrades significantly, since the locality effect enjoyed by single-size filtering is lost. For the 32d index, the time is never more than 50% higher than for the 8d indexing. The 8-stage matching for this took only 389 seconds. The R-tree using 16 coefficients took more than 4 hours.

We next consider Dataset 2 (higher selectivity). Fig. 5 reports the results for the 8d and 16d trees. Only for case of *prefix 32* for 8 dimensions the R-tree yields similar performance as our technique. In other cases the use of SQLRtree introduces remarkable savings. Interestingly, the 16d index performs better than the 8d one. Its greater precision allows for more aggressive pruning.

We evaluated the SQLRtree on Dataset 3 of four billion data samples (120 Gb), generated with the same parameters as Dataset 1, yielding the same selectivity with respect to the query. The index was only built for a window size of 32—the scenario closest to worst case—as the selectivity of the query prefix of length 32 is the lowest. We tested both *PrefixSearch* and *MultiMatching*. The resulting SQLRtree is four times the size of the index for Dataset 1, and so sized up linearly. Fig. 6 reports the comparison of execution times for 8d and 16d between the test cases for Datasets 1 and 3. The times scale linearly also.

The experiments demonstrate the performance of the SQLRtree does not degrade significantly with query selectivity. Even in the worst case scenario, the appropriate data alignment yields significant speed-up. Our technique scales linearly with respect to the dimensionality *and* scales to high dimensionality.

5 Conclusions

There has been little research devoted to efficient indexing for similarity search that scales to datasets of tens or hundreds of gigabytes. We have proposed a novel index data structure called the SQLRtree, mapped onto standard relational structures, together with the retrieval procedures. It uses B+-tree indexes to ensure access optimization. The design handles efficient retrieval for range queries with low selectivity, and can use any orthonormal dimensionality reduction technique and any distance metric. Our technique has the advantage that it is realized in external storage, which makes it easily scaled. We establish via performance evaluation that our technique scales to very large datasets efficiently.

References

1. Subba Rao, T., Priestley, M.B., Lessi, O.: Applications of time series analysis in astronomy and meteorology. London Chapman and Hall (1997)
2. : MACHO project. `http://wwwmacho.anu.edu.au/`
3. Zeger, S., Irizarry, R., Peng, R.: On time series analysis of public health and biomedical data. Technical Report 1054, Johns Hopkins University (2004)
4. Agrawal, R., Faloutsos, C., Swami, A.N.: Efficient similarity search in sequence databases. In: FODO '93. (1993) 69–84
5. K.Chan, A.W.Fu: Efficient time series matching by wavelets. In: ICDE '99. (1999) 126–133
6. Kanth, K.V.R., Agrawal, D., Singh, A.: Dimensionality reduction for similarity searching in dynamic databases. SIGMOD Rec. **27**(2) (1998) 166–176
7. Korn, F., Jagadish, H.V., Faloutsos, C.: Efficiently supporting ad hoc queries in large datasets of time sequences. SIGMOD Rec. **26**(2) (1997) 289–300
8. Keogh, E.J., Pazzani, M.J.: A simple dimensionality reduction technique for fast similarity search in large time series databases. In: PAKDD '00. (2000) 122–133
9. Yi, B.K., Faloutsos, C.: Fast time sequence indexing for arbitrary lp norms. In: VLDB '00. (2000) 385–394
10. Keogh, E., Kasetty, S.: On the need for time series data mining benchmarks: A survey and empirical demonstration. In: KDD. (2002) 102–111
11. Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: Fast subsequence matching in time-series databases. SIGMOD Rec. **23**(2) (1994) 419–429
12. Kahveci, T., Singh, A.K.: Variable length queries for time series data. In: ICDE '01. (2001) 273–282
13. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD '84. (1984) 47–57
14. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: SIGMOD '90. (1990) 322–331
15. Böhm, C., Berchtold, S., Kriegel, H.P., Michel, U.: Multidimensional index structures in relational databases. JIIS **15**(1) (2000) 51–70
16. Nykiel, T.: Indexing in similarity searches in time series data. Master's thesis, York University (2008)
17. Shieh, J., Keogh, E.: iSAX: indexing and mining terabyte sized time series. In: KDD '08. (2008) 623–631