Efficient and Scalable Management of Interval Data

George Christodoulou,
supervised by Nikos Mamoulis, Panagiotis Bouros

University of Ioannina, Greece

Abstract

The management of intervals has been an active research area since databases were invented. A popular direction of research is the indexing and retrieval of intervals, finding a wide range of applications. Emerging and widely used systems are built dependent on temporal and uncertain data. Many algorithms and indices have been proposed, concentrated on a variety of queries. Most algorithms are either suboptimal in space consumption or perform well only for specific query types. We need novel and efficient in-memory indices for intervals, which can execute queries with high performance. In this PhD research, we aim to explore partitioning approaches, which are versatile, have low space requirements and provide high query performance.

Keywords

Interval Data, Main Memory, Indexing, Query processing

1. Introduction

Intervals are representations of value ranges. Quite often, these ranges represent periods of time described as a tuple $[start, end]$. In a temporal database, an interval-based data model can timestamp each tuple or attribute value with a validity time interval [1, 2]. Along with valid time, an interval-based model can timestamp transaction time, which captures when a tuple is inserted and deleted from the database.

In statistical and probabilistic databases [3], uncertain values are often approximated by confidence intervals. Real-world examples of uncertain values include temperature values obtained from IoT devices or recorded seismic waves. For such cases, it would be more appropriate to record an observation using an interval range $[x, y]$ rather than a single value.

In data anonymization [4] attributes can be generalized to intervals. Stored values can be replaced with semantically consistent but less precise alternatives in the form of intervals. In this way, information from a private table, like the identity of any individual to whom the released data refer cannot be recognized. In XML data indexing techniques [5], the scope of an XML element can be modeled as an interval defined by the positions of the starting and closing tag of the element.

Intervals are indexed by data structures in order to efficiently evaluate different types of queries. There are several query types over intervals, so different data structures may be needed for their efficient evaluation. These query types include:

Stabbing queries (or snapshot queries in the context of temporal databases) ask for the intervals in the database (or the objects associated with them), which include a query value x. For example, interval $[6, 9]$ is a result for the query value $x = 7$.

Interval range queries retrieve intervals in a collection of intervals, which overlap (i.e., have at least one common value) with a given query interval x. For example, interval $[6, 9]$ is a result for an interval range query with $x = [3, 7]$.

Assuming that the intervals model time periods, during which a tuple in a temporal database is valid, *temporal aggregation* is the computation of an aggregation over all the tuples which are valid during a time window. For example: *find the total funding amount from all active projects from* 1/1/2021 *until* 1/1/2022.

Interval Join takes as input two collections of intervals R and S and the objective is to find a subset of their cross product $R \times S$, such that every pair of intervals in the result satisfy a temporal relationship (e.g. overlaps).

The topic of this dissertation is to study the problem of indexing and querying a large collection of records, based on an interval attribute that characterizes each object. The collection can be known before indexing or evolve over time, which is common in temporal databases or streaming data. Furthermore, the collection can be distributed at different nodes of a DBMS. The challenge is to find solutions which take advantage of modern hardware such as multi-core systems and large main memories,

Published in the Workshop Proceedings of the EDBT/ICDT 2023 Joint Conference (March 28-March 31, 2023, Ioannina, Greece)

gchristodoulou@cse.uoi.gr (G. Christodoulou)

can handle traditional and on demand indexing of intervals, and provide high performance for a wide variety of query types and predicates.

The rest of the report is organized as follows. Section 2 reviews the state of the art data structures for intervals, for different types of interval queries. Section 3 presents our work so far on interval data management. In Section 4, we present our plans for our next steps.

2. Related Work

There have been numerous studies that investigate the problem of interval data management. This section presents the state-of-the-art on areas related to this research, including data structures for interval data, algorithms for temporal aggregation and algorithms for interval joins.

A simple and practical data structure for intervals is a 1D-grid, which divides the domain into pairwise disjoint partitions. Each interval is assigned to all partitions that it overlaps with. Given a range query q, the results can be obtained by accessing each partition that overlaps with q. If the interval of a range query q overlaps with multiple partitions, duplicate results may be produced. Thus, duplicate elimination is needed. Finally, if the collection contains many long intervals, the index may grow large in size due to excessive replication.

The period index [9] is a domain-partitioning structure, specialized for *range* and *duration* queries. The time domain is split into coarse partitions and each of them is divided hierarchically to levels. Each level corresponds to a duration length and each interval is assigned to the level corresponding to its duration. During query evaluation, only the divisions that overlap the query range are accessed and only at the levels which are inside the query duration limits. The main drawback of this approach is that the hierarchy of duration sizes helps only at the upper levels. At the bottom level, an interval can expand through the whole domain and thus be stored into every partition it overlaps.

One of the most popular data structures for intervals is Edelsbrunner's *interval tree* [6]. The tree divides the domain hierarchically and places intervals recursively at the first node they overlap. The intervals assigned to each node are sorted in two lists based on their starting and ending values, respectively. Interval trees are used to answer *stabbing* and interval (i.e., *range*) queries. The main drawbacks are the redundant comparisons needed for the query result and the unnecessary visits of nodes which may not contain results. A relational interval tree for disk-resident data was proposed in [7].

Indexing intervals has regained interest with the advent of temporal databases [2]. A number of indices are proposed for secondary memory, mainly for effective versioning and compression [11, 12]. The timeline index [8] is a general-purpose access method for temporal (versioned) data, implemented in SAP-HANA. The basic idea is that, periodically at certain timestamps, *checkpoints* are kept with the alive intervals. The evaluation of some query types in the Timeline Index are suboptimal. The index also requires a lot of extra space to store the active sets of the checkpoints. The timeline index can be directly used for temporal aggregation. Piatov et al. [20] present a collection of plane-sweep algorithms that extend the timeline index to support aggregation over fixed intervals, sliding window aggregates, and MIN/MAX aggregates. The timeline index was later adapted for interval overlap joins [16]. A *domain partitioning* technique for parallel processing of interval joins was proposed in [17]. Additional research on indexing intervals addresses operations such as *temporal aggregation* [13, 14] and *interval joins* [15, 16, 17, 18, 19].

3. Current Work

In this section, we show the body of work that has been done or is currently the focus of this PhD research. Our approach aims on limiting the space requirements like long intervals handling and keeping only necessary information for each interval. Also, evaluating different query types with the minimum amount of comparisons needed in order to minimize the computational cost.

Hierarchical Index For Selection Queries

The basic idea introduced in [21], is a hierarchical index (HINT) with binary representations of intervals. A regular hierarchical decomposition of the domain into partitions is defined, where at each level ℓ from 0 to m, there are 2^ℓ partitions. The number of bits used for the interval representations is m and therefore $m+1$ levels are created. Figure 1 illustrates the hierarchical domain partitioning for $m = 4$. Next, each interval s is assigned to the *smallest set of partitions* from all levels which collectively define s. We assign longer interval parts in higher levels so that we avoid extensive replication. For example, in Figure 1, interval $[5, 9]$ is assigned to one partition at level $\ell = 4$ and two partitions at level $\ell = 3$.

The number of bits m is critical for the performance of our index. We can model our data collection with m equal to the number of bits needed for

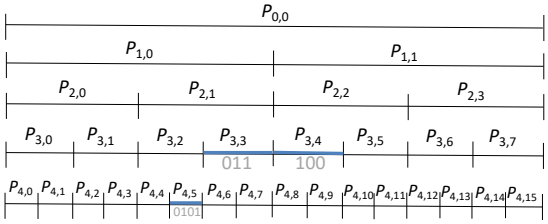

Fig. 1: Hierarchical partitioning and assignment of $[5, 9]$

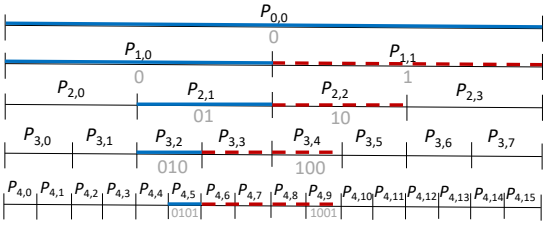

Fig. 2: Accessed partitions for range query $[5, 9]$

representing the biggest endpoint in the collection. In this case, intervals are assigned to partitions which they completely cover. When evaluating a range query, each relevant partition contains intervals which surely overlap with the query interval. Thus, there is no need for comparisons, and without comparisons there is also no need for storing the endpoints of intervals. So, for each interval we keep only an identifier. The drawback of this approach is the space requirement for covering large domains.

We can reduce memory consumption by using a smaller m and rescale all the intervals using only the prefix with the m most significant bits of their binary representations. The query intervals are also rescaled using the same amount of bits. In that way, fewer levels are created but we may need to perform comparisons at the first partition and the last partition of each level that the query interval overlaps. This holds because the intervals do not necessarily contain the whole partition that they are assigned to, as before, because of rescaling. Although, comparisons can be avoided after a certain level. For the range query q evaluation, we need to find partitions that overlap with q at each level.

The division of intervals in each partition into groups, *originals* P^O and *replicas* P^R, helps avoiding the production of duplicate query results and minimizes the number of intervals that have to be accessed in each query. For example, in Figure 2 we can see the accessed partitions for the query interval $[5, 9]$. The binary representations of $q.st$ and $q.end$ are 0101 and 1001, respectively. The relevant partitions at each level are shown in bold (blue) and dashed (red) lines and can be determined by the corresponding prefixes of 0101 and 1001. At each level, we report both originals and replicas in the first partitions while in the subsequent partitions, we report only the original intervals, so we avoid duplicate results.

We evaluated our method against state-of-the-art indices. HINT has low space complexity and minimizes the number of data accesses and comparisons during query evaluation. Our experimental analysis shows that HINT outperforms previous work by one order of magnitude in a wide variety of data and query distributions.

Queries With Allen's Predicates

Currently, we focus on the performance of our index on more specific queries. Intervals may satisfy more sophisticated relations, which are based on Allen's relationships [22](e.g., find all intervals that are *covered by* the query interval). The principles of HINT are useful for the retrieval of data intervals based on Allen's relationships. The hierarchical partitioning applies independently of the query type. Although, multiple challenges arise by this functionality extension on HINT. The main challenge is to optimize the index for all the query predicates, which access differently the relevant partitions. Another challenge emerges from the increased information we need to keep for each interval, because different endpoints are useful for answering different query predicates. Our index is evaluated against the state-of-the-art solutions with multiple real datasets. Our first experiments show a small increase in storage consumption but also a consistent lead in query throughput.

4. Future Work

For future work, there are several directions. First, we plan to investigate extensions of our index that could support queries that combine temporal selections and selections on additional object attributes (e.g., find all people employed during February 2021, whose wages were at least $5000).

Indexing Temporal Data. Our envisioned method will be based on HINT with an adaptation of temporal data specifics. Indexing temporal data differentiates from indexing a known collection of data, mainly because data evolve. The index will handle closed time intervals, but multiple objects may have alive time intervals. These time intervals will be inserted in the lowest level, but will end up in partitions of higher levels. Furthermore, the number of bits/levels used for the interval representations will be dependent of the time granularity of the

database and needs separate investigation for an optimal setup.

Multiple Temporal Operators. Most proposed indices are specialized on one temporal operator. Keeping a different index for each type of query is not affordable for a DBMS in terms of tuning, maintenance and storage overhead. Different indices will have different beneficial sortings, will possibly cause data replication and different optimizations in general. All the commonly used operators, temporal aggregation, time travel and temporal join should be supported by one index. Eventually, we want to create a versatile index for answering different types of queries and capable of temporal database system integration.

Additional Attributes. Another challenge consists in managing dimensions added from additional object attributes. Other attributes may lead to different query evaluation strategies, different optimization techniques or even specific indexing/sorting that will improve throughput and storage consumption.

Distributed Computation. Moreover, the data may be distributed among multiple physical locations. Splitting the data must be done efficiently, with low storage consumption and the query algorithm will contain multiple tasks contributing to the result. In each single node, the tasks will be assigned to different cores for parallel processing of the queries.

Acknowledgments

Partially supported by Greek national funds, under the Research-Create-Innovate call (project T2EDK-02848).

References

[1] R. T. Snodgrass, I. Ahn, Temporal databases, Computer 19 (1986) 35–42.

[2] M. H. Böhlen, A. Dignös, J. Gamper, C. S. Jensen, Temporal data management - an overview, in: eBISS, 2017, pp. 51–83.

[3] N. N. Dalvi, D. Suciu, Efficient query evaluation on probabilistic databases, in: VLDB, 2004, pp. 864–875.

[4] P. Samarati, L. Sweeney, Generalizing data to provide anonymity when disclosing information (abstract), in: ACM PODS, 1998, p. 188.

[5] J. Min, M. Park, C. Chung, XPRESS: A queriable compression for XML data, in: ACM SIGMOD, 2003, pp. 122–133.

[6] H. Edelsbrunner, Dynamic Rectangle Intersection Searching, Technical Report 47, Institute for Information Processing, Technical University of Graz, Austria, 1980.

[7] H. Kriegel, M. Pötke, T. Seidl, Managing intervals efficiently in object-relational databases, in: VLDB, 2000, pp. 407–418.

[8] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, N. May, Timeline index: a unified data structure for processing queries on temporal data in SAP HANA, in: ACM SIGMOD, 2013, pp. 1173–1184.

[9] A. Behrend, A. Dignös, J. Gamper, P. Schmiegelt, H. Voigt, M. Rottmann, K. Kahl, Period index: A learned 2d hash index for range and duration queries, in: SSTD, 2019, pp. 100–109.

[10] M. de Berg, O. Cheong, M. J. van Kreveld, M. H. Overmars, Computational geometry: algorithms and applications, 3rd Edition, Springer, 2008.

[11] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, An asymptotically optimal multiversion b-tree, VLDB J. 5 (1996) 264–275.

[12] D. B. Lomet, M. Hong, R. V. Nehme, R. Zhang, Transaction time indexing with version compression, Proc. VLDB Endow. 1 (2008) 870–881.

[13] N. Kline, R. T. Snodgrass, Computing temporal aggregates, in: IEEE ICDE, 1995, pp. 222–231.

[14] B. Moon, I. F. V. López, V. Immanuel, Efficient algorithms for large-scale temporal aggregation, IEEE TKDE 15 (2003) 744–759.

[15] A. Dignös, M. H. Böhlen, J. Gamper, Overlap interval partition join, in: ACM SIGMOD, 2014, pp. 1459–1470.

[16] D. Piatov, S. Helmer, A. Dignös, An interval join optimized for modern hardware, in: IEEE ICDE, 2016, pp. 1098–1109.

[17] P. Bouros, N. Mamoulis, A forward scan based plane sweep algorithm for parallel interval joins, Proc. VLDB Endow. 10 (2017) 1346–1357.

[18] D. Piatov, S. Helmer, A. Dignös, F. Persia, Cache-efficient sweeping-based interval joins for extended allen relation predicates, VLDB J. 30 (2021) 379–402.

[19] F. Cafagna, M. H. Böhlen, Disjoint interval partitioning, VLDB J. 26 (2017) 447–466.

[20] D. Piatov, S. Helmer, Sweeping-based temporal aggregation, in: SSTD, 2017, pp. 125–144.

[21] G. Christodoulou, P. Bouros, N. Mamoulis, HINT: A hierarchical index for intervals in main memory, in: ACM SIGMOD, 2022, p. 1257–1270.

[22] J. F. Allen, An interval-based representation of temporal knowledge, in: IJCAI, 1981, pp. 221–226.