HINT on Steroids

Batch Query Processing for Interval Data

Bouros, Panagiotis; Titkov, Artur; Christodoulou, George; Rauch, Christian; Mamoulis, Nikos

DOI
10.48786/edbt.2024.38

Publication date
2024

Document Version
Final published version

Published in
Proceedings of the 27th International Conference on Extending Database Technology, EDBT 2024

Citation (APA)
Bouros, P., Titkov, A., Christodoulou, G., Rauch, C., & Mamoulis, N. (2024). HINT on Steroids: Batch Query Processing for Interval Data. In *Proceedings of the 27th International Conference on Extending Database Technology, EDBT 2024* (3 ed., pp. 440-446). (Advances in Database Technology - EDBT; Vol. 27, No. 3). OpenProceedings.org. https://doi.org/10.48786/edbt.2024.38

Short Paper

HINT on Steroids: Batch Query Processing for Interval Data

Panagiotis Bouros
Johannes Gutenberg University
Mainz
Germany
bouros@uni-mainz.de

Artur Titkov
Johannes Gutenberg University
Mainz
Germany
artitkov@uni-mainz.de

George Christodoulou
Delft University of Technology
Netherlands
g.c.christodoulou@tudelft.nl

Christian Rauch
Johannes Gutenberg University
Mainz
Germany
crauch@uni-mainz.de

Nikos Mamoulis
University of Ioannina
Greece
nikos@cse.uoi.gr

ABSTRACT

A wide range of applications manage interval data. HINT was recently proposed to hierarchically index intervals in main memory. The index outperforms competitive structures by a wide margin, but under its current setup, HINT is able to service only single query requests. In practice however, real systems receive a large number of queries at the same time and so, our focus in this paper is on batch query processing. We propose two novel evaluation strategies termed level-based and partition-based, which both work in a per-level fashion, i.e., all queries for an index level are computed before moving to the next level. The new strategies operate in a cache-aware fashion to reduce the cache misses caused by climbing the index hierarchy or accessing multiple partitions per level, and to decrease the total execution time for a query batch. Our experimental analysis with both real and synthetic datasets showed that our batch processing strategies always outperform a baseline that executes queries in a serial fashion, and that partition-based is overall the most efficient strategy.

1 INTRODUCTION

Given a discrete or continuous 1D space, an interval is defined by a starting and an ending point in this domain. For instance, in the space of all non-negative integers $\mathbb{N}$, an interval $[st, end]$ with st, $end \in \mathbb{N}$ and $st \leq end$, is the subset of $\mathbb{N}$, which includes all integers x with $st \leq x \leq end$.[1] Collections of such intervals are found in a wide range of applications; for example, in temporal databases [4, 29], where each tuple has a *validity interval* to capture the period of time that the tuple is valid. In statistics and probabilistic databases [12], *uncertain* values are often approximated by (confidence or uncertainty) intervals. In data anonymization [27], attribute values are often generalized to value ranges. Several computational geometry problems [13] (e.g., windowing) use interval search as a module. The internal states of window queries in Stream processors (e.g. Flink/Kafka) can be modeled as intervals [2].

We target the case of range (selection) queries as a fundamental retrieval task on intervals. Let $\mathcal{S}$ be a set of objects, all carrying an interval attribute. We model each object $s \in \mathcal{S}$ as a $\langle id, st, end \rangle$ triple, where $s.id$ is the object's identifier, used to access all other attributes of the object. Given a query interval $q = [q.st, q.end]$, the goal is to find the ids of all objects $s \in \mathcal{S}$, whose intervals overlap with q, i.e., with $s.st \leq q.st \leq s.end$ or $q.st \leq s.st \leq q.end$. Such selection queries are known as *time travel* or *timeslice* queries in temporal databases [19, 26], e.g., to find the employees of a company, employed sometime in [1/1/2021, 2/28/2021].

A plethora of data structures have been proposed for managing intervals. The *interval tree* [16] divides the domain hierarchically by placing all intervals strictly before (after) the domain's center to the left (right) subtree and all intervals that overlap with the domain's center at the root. The *timeline index* [18] is a general-purpose access method for temporal (versioned) data, implemented in SAP-HANA. It maintains the endpoints of all intervals in a dedicated table, called the event list. Another simple and practical data structure is a *1D-grid*, which divides the domain into k pairwise disjoint in terms of their interval span, partitions $P_1, P_2, \ldots, P_k$; the partitions collectively cover the entire data domain. Each interval object is assigned to all partitions that it overlaps. The *period index* [3] is a self-adaptive structure which splits the domain into coarse partitions as in a 1D-grid, and then further divides each partition hierarchically, in order to organize the contained intervals based on their positions and durations. HINT [10, 11] applies a hierarchical partitioning approach. Similar to 1D-grid, an interval is assigned to all partitions that it overlaps, but to at most two partitions per level and so, HINT has controlled space requirements. To enhance query processing, the contents inside every partition are further split based on whether they begin inside or before the partition boundaries.

Motivation. The above indexing structures are all optimized to service single-query requests. However, modern transactional databases, OLTP systems and cloud services (e.g., maintained by Amazon and Google) must deal with query-heavy workflows with thousands or millions of queries received per second. For instance, according to the official statistics, Amazon S3 receives 1M requests per second.[2] Under such a setting, systems opt for processing the queries in *batches* to save resources and reduce the overall time. AWS for instance allows users to run batch processing jobs on their analytical services, e.g., Amazon RedShift.[3] Given a *batch* of selection queries Q, a straightforward approach is to execute these queries in a serial fashion by probing an interval index. Such a *query-based* approach though operates in a *cache agnostic* fashion; consequently, cache misses will affect the

[1] Note that the intervals in this paper are closed. Yet, our techniques and discussions apply on generic intervals where the *start* and *end* sides are either open or closed.

[2] https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/

[3] https://docs.aws.amazon.com/wellarchitected/latest/analytics-lens/ batch-data-processing.html

total execution time. Alternatively, we could treat batch Q as a second input, and then compute the $Q \bowtie S$ interval join, using the state-of-the-art optFS method from [5, 6]. Compared to query-based, this *join-based* evaluation allows for sharing computations and comparisons among objects, but as the size of batch Q is typically smaller than the cardinality of the input collection S, the strategy is expected to be slower than query-based, as shown in [11]. In contrast, *multi* or *batch* query processing relies on specialized computation sharing techniques to reduce the total execution time of a batch. Such processing techniques have been widely used e.g., for traditional relational data [17, 28, 30], spatial data [8, 9, 24] and graphs [20, 21, 31], and in IR systems [15, 22].

Contributions. To the best of our knowledge, this is the first work that investigates batch processing for selection queries on intervals. For this purpose, we build on the state-of-the-art HINT index (Section 2), which is shown to be typically an order of magnitude faster than the competition, because it minimizes the number of data accesses and comparisons. In addition, HINT also exhibits the lowest space complexity, while offering a competitive building time. We devise two novel strategies for batch processing (Section 3), termed the *level-based* and the *partition-based* strategy. Both strategies capitalize on HINT's structure and how the intervals are organized in memory. They operate on a per-level fashion, i.e., they first evaluate all queries for an index level before moving to the next. Some of our ideas can be applied to other interval indices; for example, 1D-grid can successfully adopt a partition-based batch processing approach. Our experimental analysis (Section 4) with both real and synthetic datasets show that our advanced strategies always outperform the query-based baseline and an 1D-grid enhanced by the partition-based strategy, and that partition-based is overall the most efficient strategy. For inputs with long intervals, partition-based achieves a 33% average decrease on the total execution time over query-based, while for datasets with short intervals, a 50% drop is observed.

2 INDEXING INTERVALS WITH HINT

HINT [10], is a hierarchical index for intervals, utilizing their binary representation. Parameter m indicates the number of bits for representing intervals, resulting in the establishment of $m+1$ levels. Figure 1 exemplifies the case of $m = 4$ and a hierarchy of 5 index levels. Each level ℓ ($0 \leq \ell \leq m$), uniformly divides the domain into 2^ℓ partitions. As we ascend the HINT hierarchy, each level ℓ corresponds to a more significant bit of the binary representation. Consequently, the number of partitions in each level decreases by a factor of 2 while covering double the size. During the insertion process, every interval s undergoes normalization and discretization within the $[0, 2^m - 1]$ domain and is inserted to at most 2 partitions per level. If a given interval spans more than 2 partitions at a specific level, it is assigned to an upper level, where partitions cover a larger part of the domain. Overall, the assignment principle is based on selecting the *smallest set* of partitions across all levels that collectively cover an interval s. Furthermore, intervals in each partition P are divided into two classes: those that start *inside* P (called *originals*), denoted by P^O, and those that start *before* P (called *replicas*), denoted by P^R.

Given a selection query $q = [q.st, q.end]$, at each index level ℓ only the sequence of partitions $P_{\ell,i}$ that intersect q are accessed; we call these, *relevant* partitions. For example, consider query q_2 in Figure 2; the relevant partitions at the bottom level of the index are $\{P_{4,10}, \ldots, P_{4,13}\}$. To avoid duplicated results, originals and replicas classes are only accessed for the first relevant partition at each level ℓ, while for the remaining partitions only originals are considered. Finally, the endpoints of an interval s are compared to query q only for the first and the last relevant partition at a level; for every (original) interval s inside the rest, intermediate partitions $q.st < s.st < q.end$ holds, by construction of the index.

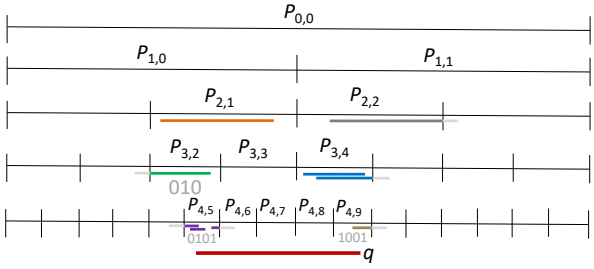

Figure 1: Benefits of the bottom-up traversal

ALGORITHM 1: Selection query on HINT

```
Input  : HINT index H, selection query q
Output : set of all intervals that overlap with q
1  compfirst ← TRUE;
2  complast ← TRUE;
3  foreach level ℓ = m to 0 do                                  ▷ bottom-up fashion
4      f ← prefix(ℓ, q.st);                                     ▷ first overlapping partition
5      l ← prefix(ℓ, q.end);                                    ▷ last overlapping partition
6      foreach partition i = f to l do
7          if i = f then
8              if i = l and compfirst and complast then
9                  output {s.id|s ∈ H.P^O_{ℓ,i}, q.st ≤ s.end ∧ s.st ≤ q.end};
10                 output {s.id|s ∈ H.P^R_{ℓ,i}, q.st ≤ s.end};
11             else if i = l and complast then
12                 output {s.id|s ∈ H.P^O_{ℓ,i}, s.st ≤ q.end};
13                 output {s.id|s ∈ H.P^R};
14             else if compfirst then
15                 output {s.id|s ∈ H.P^O_{ℓ,i} ⋃ H.P^R_{ℓ,i}, q.st ≤ s.end};
16             else
17                 output {s.id|s ∈ H.P^O_{ℓ,i} ⋃ H.P^R_{ℓ,i}};
18         else if i = l and complast then                      ▷ l > f
19             output {s.id|s ∈ H.P^O_{ℓ,i}, s.st ≤ q.end};
20         else     ▷ in-between or last (l > f), no comparisons
21             output {s.id|s ∈ H.P^O_{ℓ,i}};
22     if f mod 2 = 0 then                                      ▷ last bit of f is 0
23         compfirst ← FALSE;
24     if l mod 2 = 1 then                                      ▷ last bit of l is 1
25         complast ← FALSE;
```

Bottom-up traversal. We further reduce the number of partitions where comparisons are required by traversing HINT in a *bottom-up* fashion, instead of a conventional *top-down*. Under the bottom-up traversal, the expected number of partitions requiring comparisons is 4, according to [10]. Consider again Figure 1. For query q, no comparisons are needed in partition $P_{3,4}$, because all intervals assigned to $P_{3,4}$ should overlap with $P_{4,8}$ and the extent of $P_{4,8}$ is covered by q. Hence, the start of all intervals in $P_{3,4}$ is guaranteed to be before $q.end$ (which is inside $P_{4,9}$).

Algorithm 1 illustrates how HINT evaluates a selection query, in a bottom-up fashion. The algorithm uses two auxiliary flags, *compfirst* and *complast* to mark if it is necessary to perform comparisons at the current level (and all levels above it), for the first and the last relevant partition, respectively, At each level ℓ, the sequence of relevant partitions to the query is identified

in Lines 4–5, based on the ℓ-prefixes of $q.st$ and $q.end$, denoted by f and l, respectively. Every relevant partition $P_{\ell,i}$ is then processed in Lines 6–21. For the first relevant partition $P_{\ell,f}$ both originals $P^O_{\ell,f}$ and replicas $P^R_{\ell,f}$ are accessed. If $f = l$, i.e., the first and the last relevant partitions coincide, and both *compfirst*, *complast* are set, then comparisons are needed for both $P^O_{\ell,f}$ and $P^R_{\ell,f}$. Otherwise, if only *complast* is set, the algorithm safely skips the $q.st \leq s.end$ comparisons, while if only *compfist* is set, regardless whether $f = l$, we only perform $q.st \leq s.end$ comparisons to both $P^O_{\ell,f}$ and $P^R_{\ell,f}$. If neither flag is set, then all intervals in the first relevant partition are simply reported as results. When the last partition $P_{\ell,l}$ is examined and $l > f$ (Line 17) the algorithm considers $P^O_{\ell,l}$ and applies only the $s.st \leq q.end$ test for each interval there. Finally, for every partition in-between the first and the last one, all original intervals are simply reported.

Optimizations. A series of optimizations were proposed in [10] to boost the query processing on HINT. First, the number of performed comparisons are reduced by further dividing the P^O and P^R classes of a partition P. Specifically, P^O is split into *subdivisions* $P^{O_{in}}$ and $P^{O_{aft}}$, so that $P^{O_{in}}$ ($P^{O_{aft}}$) holds the intervals from P^O that *end* inside (resp. after) P. Similarly, each P^R is divided into $P^{R_{in}}$ and $P^{R_{aft}}$. Second, the *storage* optimization reduces the memory footprint of the index. So far, each interval s is stored as a $\langle s.id, s.st, s.end\rangle$ triplet. But, only the $P^{O_{in}}$ subdivisions require both endpoints. For $P^{O_{aft}}$ and $P^{R_{in}}$, $s.st$ and $s.end$ are only needed, respectively, while for $P^{R_{aft}}$, none of the endpoints are required, as no comparisons are performed. Another optimization to save on comparisons is to keep the subdivisions sorted; each using its own *beneficial sorting*. Due to *data skewness & sparsity*, many partitions may be empty, especially at the lowest levels. To deal with this, HINT merges the contents of all P^O divisions at the same level ℓ into a single table T^O_ℓ and builds an auxiliary index which is used to access non-empty divisions upon querying. The last optimization deals with potential *cache misses* while traversing the index. As no comparisons are needed at most of the levels, HINT stores the *id* and the *endpoints* of an interval separately. When no comparisons are needed, the index directly reports results from the *id* array.

3 BATCH PROCESSING STRATEGIES

Given a collection of intervals $\mathcal{S}$ indexed by HINT, and a batch of selection queries Q, we next discuss three evaluation strategies. Without loss of generality and for illustration purposes, we describe the strategies using an unoptimized HINT. As a running example, we use the index and the $Q = \{q_1, q_2, q_3\}$ batch in Figure 2. For each query q, we highlight its relevant (i.e., overlapping) partitions on each level according to its $[q.st, q.end]$ range.

3.1 Query-based

A straightforward approach for processing Q is to sequentially and independently compute each query, using Algorithm 1. We call this strategy, *query-based*, and show its pseudocode in Algorithm 2. Despite its simplicity, the main shortcoming of query-based is that the strategy operates in a cache agnostic fashion. As every issued query q typically overlaps multiple partitions from different levels of the index, the computation of all queries in Q requires accessing data in different parts of the main memory. Consequently, the memory access pattern is prone to cache misses.

ALGORITHM 2: Query-based strategy

```
Input  : HINT index H, batch of queries Q
Output : set of all overlapping intervals, for each q ∈ Q
1 foreach query q ∈ Q do
2     SelectionQuery(H, q);                          ▷ Using [10, 11]
```

ALGORITHM 3: Level-based strategy

```
Input  : HINT index H, batch of queries Q
Output : set of all overlapping intervals, for each q ∈ Q
1  foreach query q ∈ Q do                                  ▷ Initialization
2      compfirst[q] ← TRUE;
3      complast[q] ← TRUE;
4  foreach level ℓ = m to 0 do                             ▷ bottom-up fashion
5      foreach query q ∈ Q do
6          f ← prefix(ℓ, q.st);          ▷ first overlapping partition
7          l ← prefix(ℓ, q.end);         ▷ last overlapping partition
           ...
           Lines 6-21 in Algorithm 1
           ...
24         if f mod 2 = 0 then                          ▷ last bit of f is 0
25             compfirst[q] ← FALSE;
26         if l mod 2 = 1 then                          ▷ last bit of l is 1
27             complast[q] ← FALSE;
```

Consider our running example in Figure 2. Assuming that the query-based strategy will execute the queries in the order implied by their subscript, the first row in Table 1 illustrates the occurred access pattern, i.e., the order in which the partitions of the index will be accessed. First, all relevant partitions for q_1 are accessed on each level of the index (highlighted in blue), following the bottom-up approach proposed in [10]; similarly, the relevant partitions for q_2 (highlighted in gray) are accessed next. The two sets of partitions are located on opposite sides of the index, which causes several "jumps" to different parts of the memory; we refer to these jumps as *horizontal*. Finally, for query q_3, we need to "jump" back to the front part of the index to access the partitions highlighted in red.

In an effort to improve the above access pattern, one solution is to execute the queries according to their starting endpoint $q.st$. In the example of Figure 2, the query-based strategy will now execute first q_1, followed by q_3 and lastly, q_2. The modified access pattern is depicted in the second row of Table 1. This new pattern enables us to finish first with the queries accessing partitions in the front part of the index, before moving to the back.

3.2 Level-based

Sorting the queries by their start will reduce cache misses caused by horizontal jumps and therefore, will enhance the query-based strategy. However, the bottom-up approach employed for each query will still incur cache misses because of the *vertical* jumps in the index. For example in case of the adjacent q_1 and q_3 queries in Figure 2, we have to first climb all levels of the index to compute q_1 and then, start over from the bottom level for q_3.

To deal with these vertical jumps, we propose a different strategy which capitalizes on the fact that partitions in HINT are physically organized in a level-based fashion. The *level-based* strategy still builds upon the bottom-up approach but the evaluation process proceeds to the next level of the index only after the relevant partitions for all queries in the batch Q are already accessed and processed to report potential results. Algorithm 3 shows the pseudocode of this strategy. Algorithm 3 extends Algorithm 1

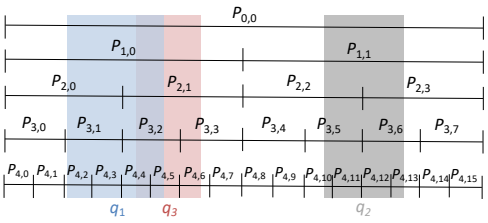

Figure 2: Running example

Table 1: Access patterns for the queries in Figure 2

Strategy	Accessed partitions
Query-based	$P_{4,2} \rightarrow P_{4,3} \rightarrow P_{4,4} \rightarrow P_{4,5} \rightarrow P_{3,1} \rightarrow P_{3,2} \rightarrow P_{2,0} \rightarrow P_{2,1} \rightarrow P_{1,0} \rightarrow P_{0,0} \rightarrow$ $P_{4,10} \rightarrow P_{4,11} \rightarrow P_{4,12} \rightarrow P_{4,13} \rightarrow P_{3,5} \rightarrow P_{3,6} \rightarrow P_{2,2} \rightarrow P_{2,3} \rightarrow P_{1,1} \rightarrow P_{0,0} \rightarrow$ $P_{4,4} \rightarrow P_{4,5} \rightarrow P_{4,6} \rightarrow P_{3,2} \rightarrow P_{3,3} \rightarrow P_{2,1} \rightarrow P_{1,0} \rightarrow P_{0,0}$
Query-based with sorting	$P_{4,2} \rightarrow P_{4,3} \rightarrow P_{4,4} \rightarrow P_{4,5} \rightarrow P_{3,1} \rightarrow P_{3,2} \rightarrow P_{2,0} \rightarrow P_{2,1} \rightarrow P_{1,0} \rightarrow P_{0,0} \rightarrow$ $P_{4,4} \rightarrow P_{4,5} \rightarrow P_{4,6} \rightarrow P_{3,2} \rightarrow P_{3,3} \rightarrow P_{2,1} \rightarrow P_{1,0} \rightarrow P_{0,0} \rightarrow$ $P_{4,10} \rightarrow P_{4,11} \rightarrow P_{4,12} \rightarrow P_{4,13} \rightarrow P_{3,5} \rightarrow P_{3,6} \rightarrow P_{2,2} \rightarrow P_{2,3} \rightarrow P_{1,1} \rightarrow P_{0,0}$
Level-based with sorting	$P_{4,2} \rightarrow P_{4,3} \rightarrow P_{4,4} \rightarrow P_{4,5} \rightarrow P_{4,4} \rightarrow P_{4,5} \rightarrow P_{4,6} \rightarrow P_{4,10} \rightarrow P_{4,11} \rightarrow P_{4,12} \rightarrow P_{4,13} \rightarrow$ $P_{3,1} \rightarrow P_{3,2} \rightarrow P_{3,2} \rightarrow P_{3,3} \rightarrow P_{3,5} \rightarrow P_{3,6} \rightarrow$ $P_{2,0} \rightarrow P_{2,1} \rightarrow P_{2,1} \rightarrow P_{2,2} \rightarrow P_{2,3} \rightarrow$ $P_{1,0} \rightarrow P_{1,0} \rightarrow P_{1,1} \rightarrow$ $P_{0,0} \rightarrow P_{0,0} \rightarrow P_{0,0}$
Partition-based with sorting	$P_{4,2} \rightarrow P_{4,3} \rightarrow P_{4,4} \rightarrow P_{4,4} \rightarrow P_{4,5} \rightarrow P_{4,5} \rightarrow P_{4,6} \rightarrow P_{4,10} \rightarrow P_{4,11} \rightarrow P_{4,12} \rightarrow P_{4,13} \rightarrow$ $P_{3,1} \rightarrow P_{3,2} \rightarrow P_{3,2} \rightarrow P_{3,3} \rightarrow P_{3,5} \rightarrow P_{3,6} \rightarrow$ $P_{2,0} \rightarrow P_{2,1} \rightarrow P_{2,1} \rightarrow P_{2,2} \rightarrow P_{2,3} \rightarrow$ $P_{1,0} \rightarrow P_{1,0} \rightarrow P_{1,1} \rightarrow$ $P_{0,0} \rightarrow P_{0,0} \rightarrow P_{0,0}$

with two modifications. First, we maintain a $compfirst[q]$ and a $complast[q]$ flag for each query q in batch Q, which are initialized in Lines 2–3 and updated in Lines 24–27 at each level, according to the last bits of the first relevant partition f and the last l. Second, we introduce in Line 5, a new `for-loop` to iterate over all queries in the batch, at current level ℓ. Each query q is then processed as in Lines 6–21 of Algorithm 1.

Similarly to the query-based strategy, level-based can also benefit from sorting the queries by their start, avoiding the horizontal jumps when accessing the relevant partitions at each level. Going back to our running example, the third row in Table 1 depicts the access pattern for the the level-based strategy, with sorting activated. To better illustrate the effect of the strategy, we write the sequence of accessed partitions in five lines, one for each level of the index. Notice how on each line (index level), the evaluation switches from the relevant partitions of q_1, to the ones of q_3 and finally, to q_2, before moving to the next level. Under this premise, we avoid the vertical jumps incurred by independently applying the bottom-up approach in the query-based strategy.

ALGORITHM 4: Partition-based strategy

```
   Input  : HINT index H, batch of queries Q
   Output : set of all overlapping intervals, for each q ∈ Q
1  foreach query q ∈ Q do                                   ▷ Initialization
2      compfirst[q] ← TRUE;
3      complast[q] ← TRUE;
4  foreach level ℓ = m to 0 do                              ▷ bottom-up fashion
5      foreach partition i in level ℓ do
6          foreach relevant query q ∈ Q to partition i do
7              f ← prefix(ℓ, q.st); ▷ first overlapping partition
8              l ← prefix(ℓ, q.end); ▷ last overlapping partition
               ...
               Lines 7-21 in Algorithm 1
               ...
24     foreach q ∈ Q do
25         if f mod 2 = 0 then                              ▷ last bit of f is 0
26             compfirst[q] ← FALSE;
27         if l mod 2 = 1 then                              ▷ last bit of l is 1
28             complast[q] ← FALSE;
```

3.3 Partition-based

Despite evaluating queries on a per-level basis and examining the queries by their start, jumps can still occur in the level-based strategy. Consider again the access pattern of level-based in Table 1; specifically, the first line which corresponds to the bottom level of the index. The strategy will access partitions $P_{4,4}$ and $P_{4,5}$ first for q_1 and then again for q_2, in the exact same order. To deal with this type of horizontal jumps, we next introduce the *partition-based* strategy. Similar to the level-based, the partition-based strategy adopts the per-level evaluation and can benefit from sorting the queries, but it processes independently every partition. Intuitively, in order to proceed to the next partition in a level, all queries relevant to the current partition must be first evaluated. Algorithm 4 illustrates the pseudocode of the strategy. As the key difference to Algorithm 3, the partition-based strategy introduces a new `for-loop` to iterate over all partitions on current level ℓ, in Line 5. Notice how Algorithm 3 iterates over each query in batch Q for current level ℓ, while Algorithm 4 iterates over all relevant queries in the batch Q (Line 6) for current partition i, i.e., all queries whose range overlaps with i, on the current level. These relevant queries are then executed similar to Lines 7–21 in Algorihtm 1.

The fourth row in Table 1 shows the access pattern for the partition-based strategy. If we compare this pattern to the level-based, we observe that when processing the bottom level of the index, the partition-based strategy will first finish with partition $P_{4,4}$ for both queries q_1 and q_3, then access $P_{4,5}$, for the same queries and finally, move on to partition $P_{4,10}$. Note that despite applying a partition-based evaluation at each level, the contents of $P_{4,7}$, $P_{4,8}$, $P_{4,9}$ and $P_{3,4}$ will be never scanned as no query overlaps with them.

Last, we elaborate on Line 6 of Algorithm 4 and the fast computation of the relevant queries in Q for current partition i. A straightforward approach for this purpose would compare every query in Q to partition i, incurring extra computational costs. Instead, we rely on the cheap bitwise operations, used to determine the first and the last relevant partitions of a query. Specifically, we define for every partition i, a range of relevant queries; for this purpose, we require the queries to be examined in increasing order of their start endpoint. The range of i's relevant queries starts from the first query q for which $prefix(\ell, q.st) = i$, to the last query with $prefix(\ell, q.end) = i$.

4 EXPERIMENTAL ANALYSIS

Finally, we present our experimental analysis. We implemented all strategies in `C++`, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native` activated. [4] Our experiments ran on an Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20GHz with 384GBs of RAM, running CentOS Linux.

[4] Source code available in https://github.com/pbour/batch_hint.

Table 2: Characteristics of real datasets

	BOOKS [5]	WEBKIT [5, 14, 25]	TAXIS [6]	GREEND [7, 23]
Cardinality	2,312,602	2,347,346	172,668,003	110,115,441
Size [MBs]	27.8	28.2	2072	1321
Domain [sec]	31,507,200	461,829,284	31,768,287	283,356,410
Min duration [sec]	1	1	1	1
Max duration [sec]	31,406,400	461,815,512	2,148,385	59,468,008
Avg. duration [sec]	2,201,320	33,206,300	758	15
Avg. duration [%]	6.98	7.19	0.0024	0.000005

Table 3: Parameters of synthetic datasets

parameter	**values** (defaults in **bold**)
Domain length	32M, 64M,**128M**, 256M, 512M
Cardinality	**10M**, 50M, 100M, 500M, 1B
α (interval length)	1.01, 1.1, **1.2**, 1.4, 1.8
σ (interval position)	10K, 100K, **1M**, 5M, 10M

Setup. All implemented strategies were developed on top of the *subs+sort* version of HINT/HINT^m [10], which employs the *subdivisions* and *sorting* optimizations. We also activated the *skewness & sparsity* and the *cache misses* optimizations, but not the *storage* one. According to [11], this HINT version exhibits the best performance on selection queries for all basic relationships in Allen's Algebra [1]; without loss of generality, we tested only the widely adopted `G-OVERLAPS` relationship for the rest of our analysis. Similar to most of the previous works, we assume that both the index and the queries fit in main memory.

We experimented with 4 collections of real intervals, which have also been used in previous works; Table 2 summarizes their characteristics. BOOKS contains the periods of time in 2013 when books were lent out by Aarhus libraries (https://www.odaa.dk). WEBKIT records the file history in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org); the intervals indicate the periods during which a file did not change. TAXIS stores the time periods of taxi trips (pick-up and drop-off timestamps) from NY City in 2013 (https://www1.nyc.gov/site/tlc/index.page). GREEND records time periods of power usage from households in Austria and Italy from January 2010 to October 2014. Collections BOOKS and WEBKIT contain around 2M, long on average, intervals each; TAXIS and GREEND have over 100M short intervals. For each dataset, we set parameter m using the cost model and the analysis in [10], i.e., 10 for BOOKS, 12 for WEBKIT and 17 for TAXIS, GREEND.

We also generated synthetic collections to simulate different cases for the lengths and the skewness of the input intervals, following the approach in [10]. Table 3 summarizes the construction parameters and their default values. The domain of the datasets ranges from 32M to 512M while their cardinality ranges from 10M to 1B. The lengths of the intervals follow a zipfian distribution, controlled by parameter α. A small value of α results in most intervals being relatively long, while a large value results in the great majority of intervals having length 1. The middle point of every interval is positioned according to a normal distribution centered at the middle point of the domain. We control this position using the deviation parameter σ; the greater the value of σ, the more spread the intervals are in the domain.

We measured the total execution time incurred by each strategy for the entire query batch, which includes the sorting costs when employed.[5] On the real collections, we ran queries uniformly distributed in the domain, while on the synthetic, the positions of the queries follow the distribution of the data. For both collection types, we vary the selectivity of the queries, in terms of their extent as a percentage of the domain inside the $\{0.01\%, 0.05\%, 0.1\%, 0.5\%, 1\%\}$ range, and the size of the query batch inside $\{1K, 5K, 10K, 50K, 100K\}$. In each test, we vary one of the above parameters while fixing the other to its default value; 0.1% of the domain, for the query extend; 10K in the real datasets and 1K in the synthetic, for the batch size.

Results. Figure 3 reports the total execution time for each strategy on the real datasets. In the first row of plots, we vary the selectivity of the queries, and in the second, the size of the query batch. We consider the query-based strategy without sorting as our baseline method. As the first observation, the tests confirm our intuition in Section 3.1; examining the queries in the batch sorted by their start will enhance the performance, due to reducing the number of horizontal jumps. Indeed, the query-based strategy with this sorting clearly outperforms the baseline query-based without, in all cases. The performance gain is more pronounced in TAXIS and GREEND, where intervals are typically stored at the bottom levels, rendering the horizontal jumps more impactful. Under this prism, we test level-based and partition-based with the query sorting always activated.

The experiments also show the benefits of batch processing and the advantage of our proposed level-based and partition-based advanced strategies over query-based with sorting. We observe that the performance gain is in practice larger in case of BOOKS and WEBKIT, compared to TAXIS and GREEND, because of the length of the contained intervals (see Table 2). The intervals in BOOKS and WEBKIT are stored at the higher levels of the index due to their significantly large length. Consequently, the impact of the vertical jumps is more pronounced in these datasets. As a result, level-based has almost identical total times to query-based with sorting on TAXIS and GREEND, while partition-based is always faster, because additional horizontal jumps are avoided by depleting all queries relevant to a partition before moving to the next, at the current level. To highlight the impact of computation sharing, Table 4 lists the percentage of the queries inside batch Q that would have been executed in a serial fashion, within the total time of each strategy; under this, the lower the percentage, the largest the number of queries that are positively affected by batch processing. The table shows both the benefit of batch processing selection queries over a serial execution (with or without sorting) and the advantage of the partition-based strategy.

Overall, we observe that partition-based is the most efficient strategy, for all datasets and in all conducted tests. Regarding the impact of the experimental parameters, all strategies are slowed down (1) when increasing the query extent as the queries become less selective and so, more time-consuming with larger result sets, and (2) when increasing the batch size, as more queries are evaluated. Nevertheless, partition-based is consistently the faster strategy.

Figure 4 reports on the synthetic datasets. Parameter m is again set to the best value on each dataset, using the model in [10]. The plots follow a similar trend to Figure 3. As expected the domain size, the dataset cardinality, the query extent and the batch size, all negatively affect the performance of the strategies. Increasing the domain size under a fixed query extent, affects the performance similar to increasing the query extent, i.e., the queries become longer and less selective, including more results. In contrast, when α grows, the intervals become shorter, so the

[5]To deal with latency, systems employ a waiting timeout for defining a batch. When the waiting time exceeds this threshold, the batch is executed regardless its size. Without loss of generality, we ignore this waiting time in our experiments.

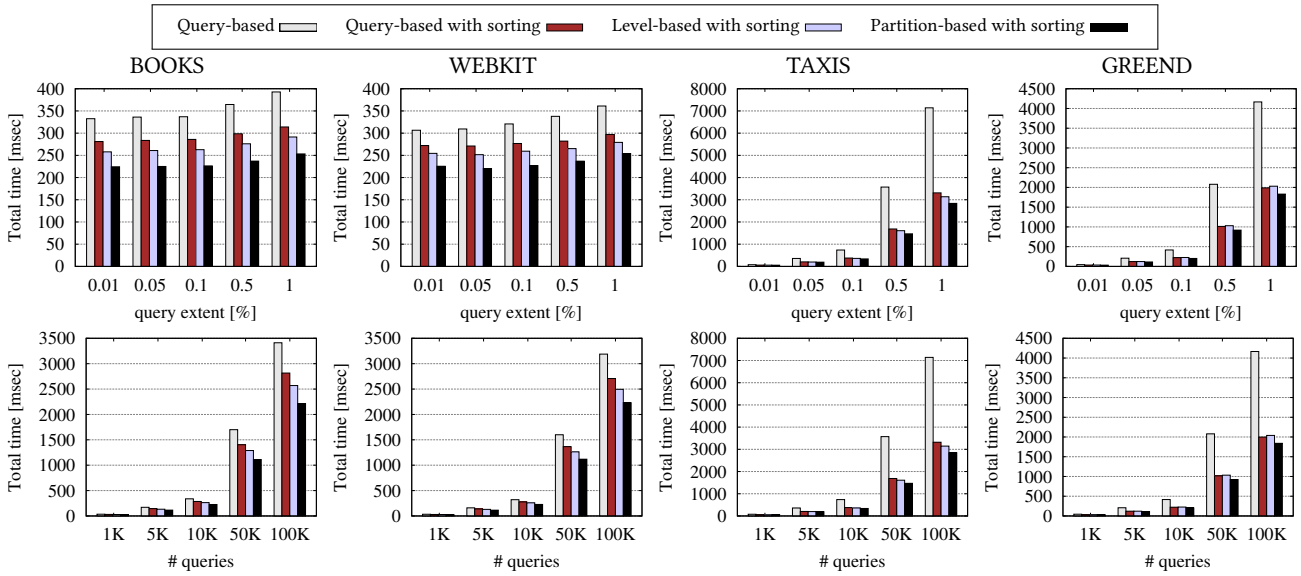

Figure 3: Comparison: real datasets

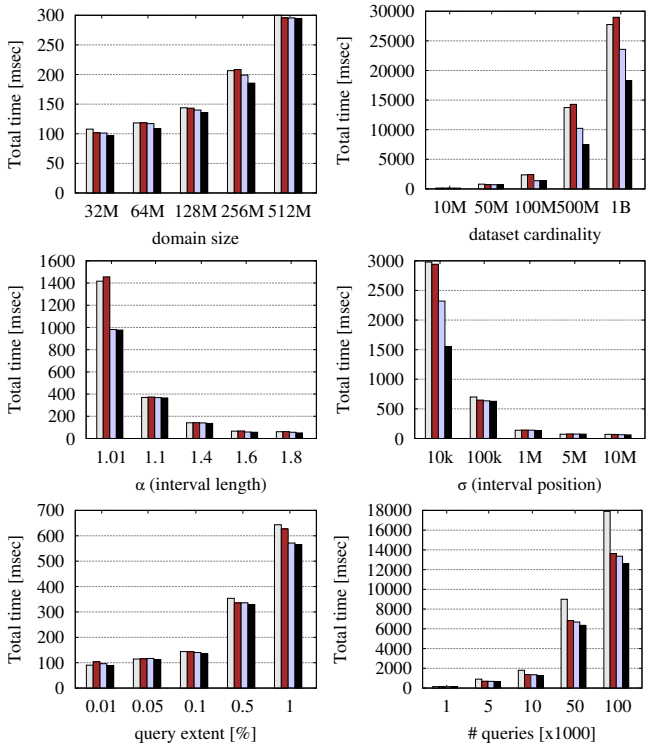

Figure 4: Comparison: synthetic datasets

performance of all strategies improves. Similarly, when increasing σ the intervals are more widespread, meaning that the queries are expected to retrieve fewer results, and the query cost drops accordingly.

Lastly, we study the applicability of the partition-based strategy to an alternative interval index. Table 5 shows that 1D-grid benefits from a partition-based batch processing but its performance still remains typically an order of magnitude inferior (in 3 outs of 4 datasets) to the partition-based HINT; This result is in line with the single-query case in [10, 11].

Table 4: Impact of computation sharing - lower numbers better; default query extent 0.1% and 10K query batch

strategy	BOOKS	WEBKIT	TAXIS	GREEND
Query-based with sorting	85%	86%	51%	53%
Level-based with sorting	78%	81%	49%	54%
Partition-based with sorting	67%	71%	46%	48%

Table 5: Applicability of partition-based strategy, total time [secs]; default query extent 0.1% and 10K query batch

strategy	BOOKS	WEBKIT	TAXIS	GREEND
1D-grid query-based	2.336	2.565	4.398	1.231
1D-grid partition-based with sorting	1.566	1.627	3.629	0.679
HINT partition-based with sorting	0.223	0.226	0.337	0.201

5 CONCLUSIONS

We studied the batch processing of selection queries on intervals. For this purpose, we built upon the state-of-the-art main-memory index on intervals, HINT. Under its current setup, HINT can only employ a query-based evaluation strategy where every query in the given batch is computed independently to the rest. Such a strategy however, is cache-agnostic and prone to cache misses while traversing the index. Instead, we proposed the level-based and partition-based strategies, which both operate in per-level fashion, i.e., they first evaluate all queries for a level of the index before moving to the next. Partition-based strategy in particular, proceeds to the next partition on a level after all queries relevant to the current one are computed. Our experiments showed that both strategies always outperform the query-based baseline, and that the partition-based strategy is overall the most efficient. In the future, we plan to investigate the parallel processing of query batches in multi-core CPUs and under a distributed setting.

ACKNOWLEDGMENTS

Partially funded by the Hellenic Foundation for Research and Innovation (HFRI) under the "2nd Call for HFRI Research Projects to support Faculty Members & Researchers" (Project No. 2757).

REFERENCES

[1] James F. Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (1983), 832–843. https://doi.org/10.1145/182.358434

[2] Ahmed Awad, Riccardo Tommasini, Samuele Langhi, Mahmoud Kamel, Emanuele Della Valle, and Sherif Sakr. 2022. D^2IA: User-defined interval analytics on distributed streams. *Inf. Syst.* 104 (2022), 101679. https://doi.org/10.1016/J.IS.2020.101679

[3] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period Index: A Learned 2D Hash Index for Range and Duration Queries. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019, Vienna, Austria, August 19-21, 2019*. ACM, 100–109. https://doi.org/10.1145/3340964.3340965

[4] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. 2017. Temporal Data Management - An Overview. In *Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Bruxelles, Belgium, July 2-7, 2017, Tutorial Lectures (Lecture Notes in Business Information Processing)*, Vol. 324. Springer, 51–83. https://doi.org/10.1007/978-3-319-96655-7_3

[5] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *Proc. VLDB Endow.* 10, 11 (2017), 1346–1357. https://doi.org/10.14778/3137628.3137644

[6] Panagiotis Bouros, Nikos Mamoulis, Dimitrios Tsitsigkos, and Manolis Terrovitis. 2021. In-Memory Interval Joins. *VLDB J.* 30, 4 (2021), 667–691. https://doi.org/10.1007/S00778-020-00639-0

[7] Francesco Cafagna and Michael H. Böhlen. 2017. Disjoint interval partitioning. *VLDB J.* 26, 3 (2017), 447–466. https://doi.org/10.1007/S00778-017-0456-7

[8] Yun Chen and Jignesh M. Patel. 2007. Efficient Evaluation of All-Nearest-Neighbor Queries. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. IEEE Computer Society, 1056–1065. https://doi.org/10.1109/ICDE.2007.368964

[9] Farhana Murtaza Choudhury, J. Shane Culpepper, Zhifeng Bao, and Timos Sellis. 2018. Batch Processing of Top-k Spatial-Textual Queries. *ACM Trans. Spatial Algorithms Syst.* 3, 4 (2018), 13:1–13:40. https://doi.org/10.1145/3196155

[10] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1257–1270. https://doi.org/10.1145/3514221.3517873

[11] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2023. HINT: A Hierarchical Interval Index for Allen Relationships. *VLDB J.* (2023). https://doi.org/10.1007/s00778-023-00798-w

[12] Nilesh N. Dalvi and Dan Suciu. 2004. Efficient Query Evaluation on Probabilistic Databases. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 864–875. https://doi.org/10.1016/B978-012088469-8.50076-0

[13] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer.

[14] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 1459–1470. https://doi.org/10.1145/2588555.2612175

[15] Shuai Ding, Josh Attenberg, Ricardo Baeza-Yates, and Torsten Suel. 2011. Batch query processing for web search engines. In *Proceedings of the Forth International Conference on Web Search and Web Data Mining, WSDM 2011, Hong Kong, China, February 9-12, 2011*. ACM, 137–146. https://doi.org/10.1145/1935826.1935858

[16] Herbert Edelsbrunner. 1980. *Dynamic Rectangle Intersection Searching*. Technical Report 47. Institute for Information Processing, TU Graz, Austria.

[17] Mehrad Eslami, Vahid Mahmoodian, Iman Dayarian, Hadi Charkhgard, and Yicheng Tu. 2020. Query batching optimization in database systems. *Comput. Oper. Res.* 121 (2020), 104983. https://doi.org/10.1016/J.COR.2020.104983

[18] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1173–1184. https://doi.org/10.1145/2463676.2465293

[19] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Rec.* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[20] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable Multi-query Optimization for SPARQL. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. IEEE Computer Society, 666–677. https://doi.org/10.1109/ICDE.2012.37

[21] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2020. Fast Query Decomposition for Batch Shortest Path Processing in Road Networks. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1189–1200. https://doi.org/10.1109/ICDE48307.2020.00107

[22] Joel Mackenzie and Alistair Moffat. 2023. Index-Based Batch Query Processing Revisited. In *Advances in Information Retrieval - 45th European Conference on Information Retrieval, ECIR 2023, Dublin, Ireland, April 2-6, 2023, Proceedings, Part III (Lecture Notes in Computer Science)*, Vol. 13982. Springer, 86–100. https://doi.org/10.1007/978-3-031-28241-6_6

[23] Andrea Monacchi, Dominik Egarter, Wilfried Elmenreich, Salvatore D'Alessandro, and Andrea M. Tonello. 2014. GREEND: An energy consumption dataset of households in Italy and Austria. In *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014*. IEEE, 511–516. https://doi.org/10.1109/SMARTGRIDCOMM.2014.7007698

[24] Apostolos Papadopoulos and Yannis Manolopoulos. 1998. Multiple Range Query Optimization in Spatial Databases. In *Advances in Databases and Information Systems, Second East European Symposium, ADBIS'98, Poznan, Poland, Spetember 7-10, 1998, Proceedings (Lecture Notes in Computer Science)*, Vol. 1475. Springer, 71–82. https://doi.org/10.1007/BFB0057718

[25] Danila Piatov and Sven Helmer. 2017. Sweeping-Based Temporal Aggregation. In *Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings (Lecture Notes in Computer Science)*, Vol. 10411. Springer, 125–144. https://doi.org/10.1007/978-3-319-64367-0_7

[26] Betty Salzberg and Vassilis J. Tsotras. 1999. Comparison of Access Methods for Time-Evolving Data. *ACM Comput. Surv.* 31, 2 (1999), 158–221. https://doi.org/10.1145/319806.319816

[27] Pierangela Samarati and Latanya Sweeney. 1998. Generalizing Data to Provide Anonymity when Disclosing Information (Abstract). In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*. ACM Press, 188. https://doi.org/10.1145/275487.275508

[28] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (1988), 23–52. https://doi.org/10.1145/42201.42203

[29] Richard T. Snodgrass and Ilsoo Ahn. 1986. Temporal Databases. *Computer* 19, 9 (1986), 35–42. https://doi.org/10.1109/MC.1986.1663327

[30] Yicheng Tu, Mehrad Eslami, Zichen Xu, and Hadi Charkhgard. 2022. Multi-Query Optimization Revisited: A Full-Query Algebraic Method. In *IEEE International Conference on Big Data, Big Data 2022, Osaka, Japan, December 17-20, 2022*. IEEE, 252–261. https://doi.org/10.1109/BIGDATA55660.2022.10020338

[31] Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, and Katja Hose. 2020. Efficient Continuous Multi-Query Processing over Graph Streams. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. OpenProceedings.org, 13–24. https://doi.org/10.5441/002/EDBT.2020.03