Fast Indexing for Temporal Information Retrieval

Christian Rauch
Institute of Computer Science
Johannes Gutenberg University Mainz
Mainz, Germany
crauch@uni-mainz.de

Panagiotis Bouros
Institute of Computer Science
Johannes Gutenberg University Mainz
Mainz, Germany
bouros@uni-mainz.de

Abstract

Temporal aspects have received tons of interest in Information Retrieval (IR) and related fields, including database search. The focus of *temporal* IR is on improving the effectiveness of search by exploiting temporal information in objects and queries. In this work, we study efficient indexing for the fundamental time-travel IR query. Given such a query q with a time interval of interest and a set of descriptive elements (e.g., keywords), the goal is to retrieve all data objects (e.g., documents) whose time interval overlaps with query's and their description contains the elements in q. Existing methods extend the inverted index to answer time-travel IR queries, with simple but ineffective temporal indexing. We propose new methods which capitalize on the state-of-the-art interval index HINT in two ways; either by extending again the inverted index or adopting the time-first irHINT approach which directly builds on HINT. Our experiments showed that irHINT outperforms all IR-first methods, while exhibiting good indexing and updating costs.

CCS Concepts

• **Information systems** → **Information retrieval query processing**; **Temporal data**; **Database query processing**.

Keywords

Temporal data, interval data, information retrieval, query processing, indexing

ACM Reference Format:
Christian Rauch and Panagiotis Bouros. 2026. Fast Indexing for Temporal Information Retrieval. In *Proceedings of Proceedings of the 2026 International Conference on Management of Data (SIGMOD '26).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Information Retrieval (IR) focuses on retrieving the most relevant objects from an existing collection; these objects are typically unstructured or semi-structured documents, e.g., web pages, but a similar process also applies for text-search in database systems and multimedia search, e.g., for images. Object relevance is defined in comparison to a user-defined query object, e.g., a set of keywords (most popular format), a document or a query image. To incorporate temporal dynamics, *temporal* IR has emerged as an interesting subfield of research. Prior work in temporal IR includes topics such as query analysis to understand the intent behind user search, e.g., [39, 40, 46, 50, 57, 63], indexing and query processing, e.g., [3, 4, 7, 53, 62], ranking e.g., [6, 12, 17, 27, 37, 42, 58] and clustering e.g., [1, 2, 10, 65]. An extensive overview of the temporal IR can be found in [11, 24, 41] and its Wikipedia article.[1]

Our focus is on the indexing and query processing challenge. We study the problem of efficiently processing *time-travel* IR containment queries which are among the most fundamental and important instances of temporal IR search. [2] We assume that every data object o (e.g., a document version in a archive) is associated with a *time interval* $[o.t_{st}, o.t_{end}]$ which models its lifespan (e.g., the time period during which the version was valid), and a set of descriptive *elements* $o.d$ (e.g., the terms included in the version). Given a collection of such objects and a query object q, a time-travel IR query retrieves all data objects whose time interval overlaps with the $[q.t_{st}, q.t_{end}]$ query interval and their description contains the elements from $q.d$. Such a query finds application in several scenarios. In archive search, a temporal IR query could retrieve all versions (or revisions) of articles in Wikipedia from January 1, 1980 until December 31, 2000, relevant to the US elections; in this example, the query contains the "US", "elections" keywords and time interval [1980-01-01, 2000-12-31]. As an another example consider the field of Music IR and the collection of streaming user sessions maintained by Spotify [9]; each session spans a specific time period and its description holds the *id*s of all streamed tracks. A time-travel IR query could request the sessions where users listened to Beethoven's "Ode to Joy" and "Für Elise" from January 1 until January 31, 2024. Time-travel IR queries can be also used in market analysis where basket data contain the products bought by customers within a specific time period, i.e., their visit to the store. For instance, we might be interested in finding all last month sessions where a copy of the "Shining", the "It" and the "Misery" novels on Amazon were purchased.

A straightforward approach for time-travel IR queries is to directly utilize an IR index; the most dominant is the *inverted index* [67], which associates every element e with a postings list $\mathcal{I}[e]$ of the objects that contain e. To incorporate the time dimension, state-of-the-art methods extend the classic inverted index so that every posting also includes the time interval of the corresponding object. Despite its simplicity, this approach fails to filter out candidates based on their time intervals as no temporal indexing is in place. For this purpose, Berberich at al. [7] proposed a *vertical* partitioning of every postings list by first dividing the time domain

SIGMOD '26, Bengaluru, India

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

[1] https://en.wikipedia.org/wiki/Temporal_information_retrieval
[2] Containment and relevance ranking queries are the most fundamental types of IR search; in the future, we plan to also study relevance-based temporal IR search.

into disjoint *slices*. Later, Anand et al. [4] presented a *horizontal* partitioning, where the contents of each postings list are grouped into *shards* according to their $o.t_{st}$ timestamp.

Motivation. Although slicing and sharding do accelerate temporal IR search, they share two key weaknesses. First, they both fail to capitalize on efficient temporal indexing. Indexing intervals has been extensively studied in the past for temporal indexing, amongst other scenarios. Proposed structures include the segment [22] and the interval tree [26], 1D partitioning such a 1D-grid, the period index [5] and its follow-up RD-index [13, 14], the timeline index [43] and HINT [19, 20]. The slicing technique essentially builds upon the 1D-grid partitioning. However, we focus on HINT as the two independent studies in [19, 20] and [13, 14] showed this is the state-of-the-art for indexing intervals outperforming by at least one order of magnitude, all competitive indices, including a 1D-grid.

As their second weakness, both slicing and sharding follow an IR-first approach, i.e., they primarily index the description of an object and secondarily, its valid time. This approach will be efficient unless the query elements appear frequently in the collection, in which case, a large number of comparisons are required to determine the temporally qualifying objects. Such a scenario is in fact common as in many cases, the probability of an element to appear in an query follows the element frequency distribution in the collection.

Surprisingly, no previous indexing method in temporal IR has investigated the option to build on top of interval (temporal) indexing instead of the inverted index; such an approach will be able to filter out candidates first by time. Under this premise, our study shares common ground with spatio-textual querying; see [16] for the most recent survey. However, there are two differences; (1) the majority of these works consider objects without extent, i.e., points in space, and (2) their spatial/multi-dimensional indexing is not suitable for intervals. For instance, the authors in [21] showed that HINT outperforms an off-the-shelf spatial index [22, 31], after mapping the interval timestamps to a $(start, end)$ 2D space [48, 62].

Contributions. In view of the above weaknesses, we investigate how to enhance the computation of time-travel IR queries by utilizing the state-of-the-art HINT to index the object's time interval. For this purpose, we explore two directions designing solutions that adopt the IR-first approach and a completely novel time-first. Specifically, the key contributions of this paper are as follows:

- We devise an IR-first method in two variants which build upon the temporal inverted index, similar to [4, 7]. But, instead of slicing or sharding the postings lists, we organize every list as a HINT. Such design takes advantage of HINT's efficiency in determining the objects that qualify temporally. Also, it can lower the storage requirements compared to slicing as HINT is more space efficient than the 1D-grid.
- We also design a hybrid that pairs the advantages of HINT with slicing. By employing a dual-copy for each postings list, this method combines HINT's fast range (time-travel) queries on intervals with the efficiency of slicing for the list intersections.
- We propose a novel time-first indexing scheme called irHINT which directly builds upon the temporal indexing of HINT. The irHINT injects every partition of HINT's hierarchy with inverted indexing. We present two variants of irHINT; the first focuses on enhancing the query performance, while the second, on reducing the index size.
- We compare our methods against the temporal inverted index with slicing or sharding. The tests show that our IR-first methods are competitive to the state-of-the-art; sometimes even faster. More importantly though, the tests showed the advantage of time-first indexing; performance irHINT can be up to 2 times faster than the fastest IR-first competitor with lower space requirements; the size variant is also faster than the competition while having the lowest index size.

Table 1: Notation summary

notation	description
$i = [t_{st}, t_{end}]$	time interval
$e \in \mathcal{D}$	descriptive element from global dictionary $\mathcal{D}$
$o \in O$	data object in collection O
$\langle o.id, [o.t_{st}, o.t_{end}], o.d \rangle$	object's id, time interval and description
$\mathcal{I}$	(temporal) inverted index
$\mathcal{I}[e]$	postings list for element e
$\mathcal{H}$	HINT index
$\mathcal{H}[e]$	postings HINT for element e
$P_{\ell,j}$	j-th partition at level ℓ of HINT
$P_{\ell,f}$ ($P_{\ell,l}$)	first (last) relevant partition at level ℓ
$P^O_{\ell,j}$ ($P^R_{\ell,j}$)	division of $P_{\ell,j}$ with originals (replicas)
$\mathcal{I}^O_{\ell,j}$ ($\mathcal{I}^R_{\ell,j}$)	inverted index for original (replicas) division of $P_{\ell,j}$
$q = \langle [q.t_{st}, q.t_{end}], q.d \rangle$	time-travel IR query

Outline. The rest of the paper is as follows. Section 2 formally defines time-travel IR queries and provides the necessary background on indexing. Section 3 presents our IR-first solutions, while Section 4 details irHINT. Section 5 reports our experiments. Last, Section 6 discusses related work and Section 7 concludes our study.

2 Preliminaries

We introduce notation and formally define the problem of temporal IR search. Then, we briefly describe the state-of-the-art indices for time-travel IR queries and also revisit the state-of-the-art interval index. Table 1 summarizes the notation used throughout the text.

2.1 Notation and problem definition

A *time interval* or simply an *interval* in the context of this paper, is defined by its starting and ending point in the time domain. Formally, an interval $i = [t_{st}, t_{end}]$ with $t_{st} \leq t_{end}$ includes all time points t with $t_{st} \leq t \leq t_{end}$. We say that two intervals i_1, i_2 *overlap* if they share at least one time point; formally:

$$Overlap(i_1, i_2) = \begin{cases} TRUE, & \textbf{if } i_2.t_{st} \leq i_1.t_{st} \leq i_2.t_{end} \textbf{ or} \\ & \quad i_1.t_{st} \leq i_2.t_{st} \leq i_1.t_{end} \\ FALSE, & \textbf{otherwise} \end{cases}$$

We model a data object o as an $\langle id, [t_{st}, t_{end}], d \rangle$ triple, where $o.id$ is the object's identifier (used to access other types of information potentially attached to the object), $[o.t_{st}, o.t_{end}]$ is the time interval associated with o representing its lifespan, and d is a set of elements drawn from a global dictionary $\mathcal{D}$ which describe the object. [3] For example, if o is a document (or a version of a document) then set d includes the terms contained in o.

[3]For now, we only assume *set* semantics for $o.d$; *bag* semantics and language models, where $o.d$ can contain an element e multiple times are left for future work.

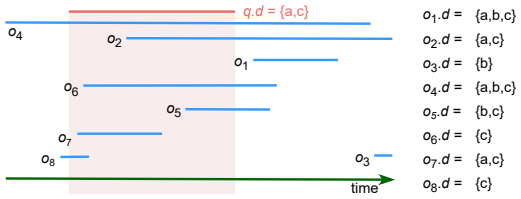

Figure 1: Our running example

ALGORITHM 1: Time-travel IR query on *t*IF

Input : *t*IF index $\mathcal{I}$, query $q = \langle [t_{st}, t_{end}], d \rangle$
Output : set of object ids

```
1  C ← ∅;                                              ▷ initialize result set
2  sort q.d by element frequency;                      ▷ in increasing order
3  e* ← least frequent element in q.d;
4  foreach entry ⟨o.id, [o.t_st, o.t_end]⟩ ∈ I[e*] do
5      if q.t_st ≤ o.t_end and o.t_st ≤ q.t_end then   ▷ check temporal predicate
6          C ← C ⋃ {o.id};                             ▷ update candidates
7  foreach element e ∈ q.d \ {e*} do
8      C ← C ⋂ I[e];                                   ▷ update candidates via list intersection
9  return C;
```

Given a collection of objects O, we next define *temporal* IR search, which blends its classic containment search counterpart with time-travel search; the latter is defined as a range query on intervals.

Definition 2.1 (Time-travel IR query). Let $[q.t_{st}, q.t_{end}]$ be a query time interval and $q.d$ be a set of query elements, a *time-travel IR* query returns all objects in O whose interval overlaps with $[q.t_{st}, q.t_{end}]$ and their description contains all elements in $q.d$; formally, every object o with:

$$Overlap([o.t_{st}, o.t_{end}], [q.t_{st}, q.t_{end}]) = TRUE, \textbf{ and } o.d \supseteq q.d$$

Note that if O contains e.g., document versions in an archive, our goal is to find the *versions*, *not* the distinct documents, that contain the query elements (terms); a similar assumption was made in [4].

Example 2.2. Figure 1 introduces our running example of 8 objects $\{o_1, \ldots, o_8\}$. The descriptive elements for the objects are drawn from the $\mathcal{D}$ = {a,b,c} dictionary. Consider the time-travel IR query q with its time interval corresponding to the red shaded area and $q.d$ = {a,c}. The answer to q consists of objects o_2, o_4 and o_7 whose time intervals overlap with the shared area and their descriptions contain both user-given elements.

2.2 Temporal IR indexing

As explained in the introduction, state-of-the-art temporal IR indexing builds on top of the inverted index. Therefore, we start our discussion with the base *temporal inverted index* which we denote by *t*IF.[4] The index associates every element e in the global dictionary $\mathcal{D}$ with a time-aware postings list $\mathcal{I}[e]$ of the objects that contain e. Every $\langle o.id, [o.t_{st}, o.t_{end}] \rangle$ entry of such list includes the identifier and the $[o.t_{st}, o.t_{end}]$ time interval of the corresponding object o.[5] For instance, the $\mathcal{I}$[a] list in our running example contains the $\langle o_1, [\ldots] \rangle$, $\langle o_2, [\ldots] \rangle$, $\langle o_4, [\ldots] \rangle$ and $\langle o_7, [\ldots] \rangle$ entries.

[4]We use the IF abbreviation originating from the term inverted file index.
[5]Compression techniques used to reduce the space requirements in IR systems, e.g., those proposed in [56]. are orthogonal to the above scheme.

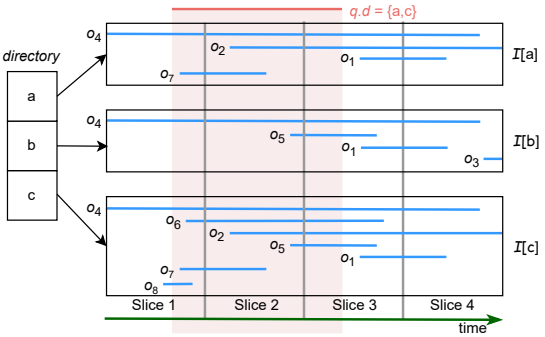

Figure 2: *t*IF+Slicing for the running example

Given a *t*IF index for a collection of objects, to answer a time-travel IR query $q = \langle [t_{st}, t_{end}], d \rangle$ it suffices to intersect the postings lists of all elements $e \in q.d$, omitting the entries for the objects whose time interval does not overlap the $[q.t_{st}, q.t_{end}]$ query interval. For typical IR containment search, a common practice is to maintain the entries inside a postings list ordered by their object id. Such sorting allows us to efficiently compute list intersections in a merge-sort fashion. Moreover, the elements in $q.d$ are considered by their frequency in the global dictionary, in increasing order. This way we can reduce the size of the intermediate lists which contain candidate results, and accelerate the subsequent intersections. To further enhance the evaluation process, before computing the first list intersection, we can exclude from the postings list of the least frequent element in $q.d$, the objects that do not qualify the temporal overlap predicate. Under this, we avoid checking objects that are guaranteed not to be results, during the next list intersections. Algorithm 1 provides a high-level pseudocode for computing time-travel IR queries. For our running example, we first access $\mathcal{I}$[a] as element a is less frequent than c. We then filter out object o_1 and intersect candidates $C = \{o_2, o_4, o_7\}$ with $\mathcal{I}$[c].

Slicing. To accelerate the temporal overlap predicate of the query, Berberich et al. [7] adopted a breakdown of the time domain into a sequence of non-overlapping *slices*. Under this slicing, every postings list $\mathcal{I}[e]$ of the inverted index is divided into smaller sub-lists, each representing a slice of the time domain. The relevant domain slices for an $\mathcal{I}[e]$ are essentially those overlapping the $[\mathcal{I}[e].t_{st}, \mathcal{I}[e].t_{end}]$ time interval which represents the temporal span of the entire list, with $\mathcal{I}[e].t_{st} = \min_{o \in \mathcal{I}[e]}\{o.t_{st}\}$ and $\mathcal{I}[e].t_{end} = \max_{o \in \mathcal{I}[e]}\{o.t_{end}\}$. An entry $\langle o.id, [o.t_{st}, o.t_{end}] \rangle$ in the original $\mathcal{I}[e]$ list, is replicated to all slices (and their sub-lists) it overlaps. Figure 2 shows the *t*IF index with Slicing for our running example where each postings list is divided into 4 sub-lists, using the depicted slices. Object o_4 whose interval overlaps all 4 slices, is replicated to every sub-list of $\mathcal{I}$[a], $\mathcal{I}$[b] and $\mathcal{I}$[c].

This replication is necessary to ensure the correctness of query processing but may also lead to duplicate results.[6] Duplicates can be discarded by hashing or more efficiently, using the reference value method from [25], which was originally proposed for rectangles but can be applied to intervals as well. Given a time-travel IR query q, the evaluation process in the base *t*IF is adapted so that for each

[6]Note that Berberich at al. [7] only consider stabbing queries in time, which define a single timestamp $q.t$, instead of an interval $[q.t_{st}, q.t_{end}]$. Yet, the work can be extended in our setting as long as we deal with duplicate results.

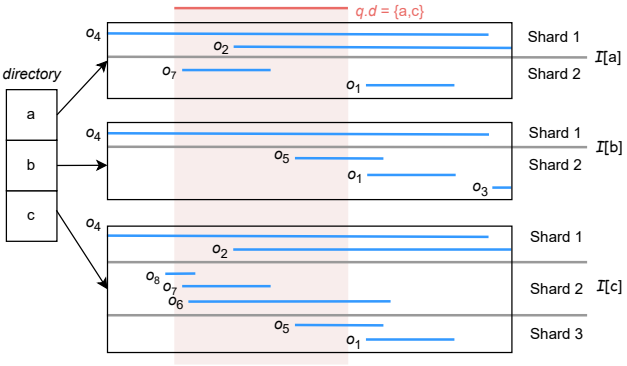

Figure 3: *t*IF+Sharding for the running example

element $e \in q.d$, only the sub-lists of $\mathcal{I}[e]$ whose slices overlap $[q.t_{st}, q.t_{end}]$ are considered. For instance, in our running example, we access only the sub-lists for the first three slices in $\mathcal{I}[a]$, $\mathcal{I}[c]$.

The authors in [7] also investigated how to set the number of slices in the time domain. They modeled this tuning task as an optimization problem. Given a user-defined upper bound on the tolerable index-size increase (due to replicating object entries) compared to the original temporal inverted index, the goal is to determine the best slicing that satisfies the above constraint while minimizing the expected query processing cost. This query cost is defined as the expected number of index entries read.

Sharding. Instead of dividing the time domain to partition the postings lists, Anand et al. [4] proposed to group the contained $\langle o.id, [o.t_{st}, o.t_{end}]\rangle$ entries according to their $o.t_{st}$, completely avoiding the need for replication and thus, for result de-duplication. These groups are called *shards*. The contents of a shard are ordered by t_{st} and ideally, satisfy the *staircase* property; for every two entries $\langle o_1.id, [o_1.t_{st}, o_1.t_{end}]\rangle$, $\langle o_2.id, [o_2.t_{st}, o_2.t_{end}]\rangle$ with $o_1.t_{st} \leq o_2.t_{st}$ in the shard, $o_1.t_{end} \leq o_2.t_{end}$ also holds. Figure 3 depicts the *t*IF index with Sharding for our running example. Every postings list is horizontally divided into shards that satisfy the staircase property. For instance, the $\mathcal{I}[a]$ list is split into two shards, the first containing the entries for o_4, o_2 and the second, for o_7, o_1.

When evaluating a temporal IR query, a shard is scanned until the first entry that starts after the $[q.t_{st}, q.t_{end}]$ intervals is found. The remaining entries in the shard are guaranteed not to overlap the query. To accelerate the shard scanning, the authors designed an auxiliary structure called the *impact* list which maintains pairs of t_{st} points and offsets to the shard contents. Upon querying, the impact list is first probed to determine the offset position from which the shard should be scanned according to the query interval.

In practice, the number of ideal shards per postings list can be overwhelmingly large, which may affect the query performance. To deal with this issue, the authors in [4] proposed a cost-aware merging approach of ideal shards, which intuitively relaxes the staircase property requirement.

2.3 Indexing intervals with HINT

HINT [19, 20] hierarchically and uniformly divides the domain into 2^ℓ partitions for $\ell = 0$ to m, defining $m+1$ levels, as shown in Figure 4. Partitions at level ℓ are denoted by $P_{\ell,0}$ to $P_{\ell,2^\ell-1}$. Each

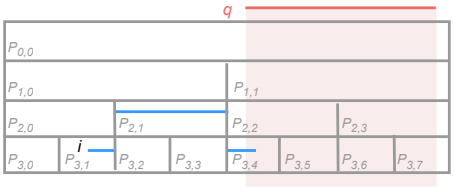

Figure 4: HINT example

interval i is normalized, discretized in the $[0, 2^{m-1}]$ domain, and assigned to the smallest set of partitions from all levels that cover i (at most 2 partitions per level). For example, in Figure 4, the interval i is assigned to partitions $P_{3,1}$, $P_{2,1}$, and $P_{3,4}$. The intervals in each partition P are split into two divisions (sub-partitions): those that start *inside* P (called *originals*), denoted by P^O, and those that start *before* P (called *replicas*), denoted by P^R. Hence, interval i is stored inside $P^O_{3,1}$ division which stores originals in $P_{3,1}$, while in $P_{2,1}$ and $P_{3,4}$, inside the $P^R_{2,1}$ and $P^R_{3,4}$ replica divisions, respectively.

Given a (selection) range query $q = [q.st, q.end]$, at each index level ℓ only the sequence of partitions $P_{\ell,j}$ that intersect q are accessed; we call these, *relevant* partitions. For query q in Figure 4, only partitions $P_{3,4} - P_{3,7}$, $P_{2,2}$, $P_{2,3}$, $P_{1,1}$ and $P_{0,0}$ will be accessed. To avoid producing duplicated results, originals are accessed for all relevant partitions at each level ℓ while replicas, only for the first relevant partition. Finally, the endpoints of an interval i are compared to query q only for the first and the last relevant partition at a level; for every (original) interval i inside the rest (intermediate) partitions $q.st < i.st < q.end$ holds by construction of the index.

Bottom-up traversal. The number of partitions where comparisons are required can be further reduced by traversing HINT in a *bottom-up* fashion, instead of a conventional *top-down*. Under this, comparisons are necessary only in 4 partitions. Consider again Figure 4. For query q, no comparisons are needed in partition $P_{2,3}$, because all intervals assigned to $P_{2,3}$ should overlap with $P_{3,6}$ and the extent of $P_{3,6}$ is covered by q. Hence, the start of all intervals in $P_{2,3}$ is guaranteed to be before $q.end$ (which is inside $P_{3,7}$).

Algorithm 2 illustrates range queries on HINT. The algorithm uses the *compfirst*, *complast* auxiliary flags to mark if comparisons are necessary 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 5–6, based on the ℓ-prefixes of $q.st$ and $q.end$, denoted by f and l, respectively. Every relevant partition $P_{\ell,j}$ is then processed in Lines 7–22. For the first, $P_{\ell,f}$ 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 $q.st \leq i.end$ comparisons, while if only *compfist* is set, regardless whether $f = l$, we only perform $q.st \leq i.end$ 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 19) Algorithm 2 considers $P^O_{\ell,l}$ and applies only the $i.st \leq q.end$ test for each interval there. Finally, for every partition in-between the first and the last relevant ones, all original intervals are simply reported.

ALGORITHM 2: Range query on HINT

Input : HINT index $\mathcal{H}$, query $q = [st, end]$
Output : set of all intervals that overlap q

$\mathcal{R} \leftarrow \emptyset$; ▷ initialize result set
$compfirst \leftarrow TRUE$;
$complast \leftarrow TRUE$;
foreach *level* $\ell = m$ **to** 0 **do** ▷ bottom-up fashion
  $f \leftarrow prefix(\ell, q.st)$; ▷ first relevant partition
  $l \leftarrow prefix(\ell, q.end)$; ▷ last relevant partition
  foreach *partition* $j = f$ *to* l **do**
    if $j = f$ **then**
      if $p = l$ **and** $compfirst$ **and** $complast$ **then**
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^O_{\ell,j}, q.st \leq i.end \wedge i.st \leq q.end\}$;
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^R_{\ell,j}, q.st \leq i.end\}$;
      else if $i = l$ **and** $complast$ **then**
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^O_{\ell,j}, i.st \leq q.end\}$;
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^R_{\ell,j}\}$;
      else if $compfirst$ **then**
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^O_{\ell,j} \bigcup P^R_{\ell,j}, q.st \leq i.end\}$;
      else
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^O_{\ell,j} \bigcup P^R_{\ell,j}\}$;
    else if $j = l$ **and** $complast$ **then** ▷ $l > f$
      $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^O_{\ell,j}, i.st \leq q.end\}$;
    else ▷ in-between or last ($l > f$), no comparisons
      $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{i.id | i \in P^O_{\ell,j}\}$;
  if $f \bmod 2 = 0$ **then** ▷ last bit of f is 0
    $compfirst \leftarrow FALSE$;
  if $l \bmod 2 = 1$ **then** ▷ last bit of l is 1
    $complast \leftarrow FALSE$;
return $\mathcal{R}$;

Optimizations. First, the number of performed comparisons are reduced by further dividing P^O and P^R 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 index size. So far, each interval i is stored as a $\langle i.id, i.st, i.end \rangle$ triple. But, only the $P^{O_{in}}$ subdivisions require both endpoints. For $P^{O_{aft}}$ and $P^{R_{in}}$, $i.st$ and $i.end$ are only needed, respectively, while for $P^{R_{aft}}$, none of the endpoints matter, 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 Novel IR-first Indexing

We next study how to further enhance the performance of *t*IF by capitalizing on the state-of-the-art index for intervals HINT.

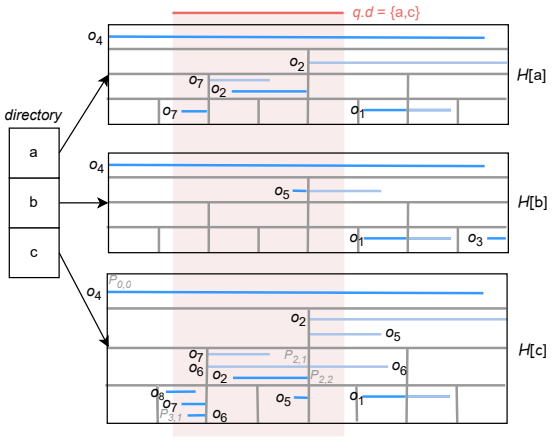

Figure 5: *t*IF+HINT for the running example

3.1 The *t*IF+HINT index

As discussed in the introduction, the slicing extension to *t*IF discussed in Section 2.2, organizes every postings list using an 1D grid. In view of the findings in [19, 20], we present a novel extension to the temporal inverted index, which organizes every postings list using a HINT instead. We denote this novel composite index by *t*IF+HINT. For simplicity, we will describe *t*IF+HINT in what follows, utilizing the unoptimized version of HINT but the optimizations discussed in Section 2.3 are orthogonal to our discussion and hence can be applied; an exception arises for the beneficial/temporal sorting, which we will clarify in the next paragraphs.

To build a *t*IF+HINT index, we combine the construction process of base *t*IF with the construction of HINT. Consider the $\langle o.id, [o.t_{st}, o.t_{end}] \rangle$ entry in postings list $\mathcal{I}[e]$. The $[o.t_{st}, o.t_{end}]$ interval is first rescaled to the $[0, 2^{m-1}]$ domain and then stored in the corresponding HINT for element e, denoted by $\mathcal{H}[e]$, inside at most 2 relevant (overlapping) partitions per level, as explained in Section 2.3.[7] Updates can be also implemented combining the updating process of *t*IF with the updating process of HINT. To accommodate updates that grow the time domain, we can take advantage of the time-expanding extension to HINT presented in [21]. Figure 5 depicts the *t*IF+HINT index for our running example. Observe how every postings list $\mathcal{I}[e]$ is replaced by the $\mathcal{H}[e]$ HINT; for simplicity, assume that all postings HINTs contain 4 levels, i.e., $m = 3$. To distinguish between the originals O and the replicas R divisions inside each HINT partition, we color replicas by a lighter shade of blue. Take for instance object o_6 in $\mathcal{H}[c]$; the object is stored as an original in $P^O_{3,1}$ and as a replica in $P^R_{2,1}$ and $P^R_{2,2}$.

We now discuss how to evaluate a time-travel IR query. A straightforward approach is to directly employ HINT's querying process to quickly determine candidate results which qualify the temporal condition of the query. Specifically, we first execute the $[q.t_{st}, q.t_{end}]$ range query on the HINT for the least frequent element e^*, which determines the initial set of candidates C, i.e., the objects that contain element e^* and whose time interval overlaps with $[q.t_{st}, q.t_{end}]$. Then, in order to produce the final answer to the time-travel IR query, we need to remove all candidates from C whose description

[7] One option to set m parameter is to use the cost model proposed in [19, 20] independently for each postings list. We elaborate on the alternative in Section 5.2.

ALGORITHM 3: Time-travel IR query on *t*IF+HINT (using binary search for intersections)

```
   Input  : tIF+HINT index, query q = ⟨[t_st, t_end], d⟩
   Output : set of object ids
1  sort q.d by element frequency;                                  ▷ in increasing order
2  e* ← first element in q.d;                                      ▷ least frequent element
3  C ← RangeQuery(H[e*], [q.t_st, q.t_end]);                       ▷ determine candidates using Algorithm 2
4  foreach element e ∈ q.d \ {e*} do
5      sort C;                                                     ▷ by object id
6      R ← ∅;                                                      ▷ initialize temporary result set
7      foreach level ℓ = m to 0 in the H[e] HINT do                ▷ bottom-up
8          f ← prefix(ℓ, q.t_st);                                  ▷ first relevant partition
9          l ← prefix(ℓ, q.t_end);                                 ▷ last relevant partition
10         foreach partition i = f to l do
11             if i = f then
12                 if p = l and compfirst and complast then
13                     R ← R ⋃ {o.id | o ∈ P^O_{ℓ,j}, q.t_st ≤ o.t_end ∧ o.t_st ≤
                           q.t_end ∧ o.id ∈ C};
14                     R ← R ⋃ {o.id | o ∈ P^R_{ℓ,j}, q.t_st ≤ o.t_end ∧ o.id ∈ C};
15                 else if i = l and complast then
16                     R ← R ⋃ {o.id | o ∈ P^O_{ℓ,j}, o.t_st ≤ q.t_end ∧ o.id ∈ C};
17                     R ← R ⋃ {o.id | o ∈ P^R_{ℓ,j} ∧ o.id ∈ C};
18                 else if compfirst then
19                     R ← R ⋃ {o.id | o ∈ P^O_{ℓ,j} ⋃ P^R_{ℓ,j}, q.t_st ≤
                           o.t_end ∧ o.id ∈ C};
20                 else
21                     R ← R ⋃ {o.id | o ∈ P^O_{ℓ,j} ⋃ P^R_{ℓ,j} ∧ o.id ∈ C};
22             else if i = l and complast then                     ▷ l > f
23                 R ← R ⋃ {o.id | o ∈ P^O_{ℓ,j}, o.t_st ≤ q.t_end ∧ o.id ∈ C};
24             else                ▷ in-between or last (l > f), no comparisons
25                 R ← R ⋃ {o.id | o ∈ P^O_{ℓ,j} ∧ o.id ∈ C};
26         if f mod 2 = 0 then                                     ▷ last bit of f is 0
27             compfirst ← FALSE;
28         if l mod 2 = 1 then                                     ▷ last bit of l is 1
29             complast ← FALSE;
30     C ← R;                                                      ▷ update candidates
31 return C;
```

does not contain the remaining elements in $q.d$. To this end, we traverse the $\mathcal{H}[e]$ HINT hierarchy for each element $e \in q.d \setminus \{e^*\}$, and intersect set C with the contents of the P^O or P^R divisions from all relevant partitions to determine the final results.

The unoptimized HINT does not store the contents of divisions ordered by any attribute, but even when the sorting optimization (see Section 2.3) is activated, each division has its own beneficial sorting. Under this, the intersections with candidates C cannot be computed in a typical, efficient merge-sort fashion. Nevertheless, to fastly compute these intersections, we maintain set C sorted by object *id*, which allows us to compute every intersection as a series of binary searches on C. Algorithm 3 illustrates the pseudocode of this query evaluation method for *t*IF+HINT. After computing the initial set of candidates C in Lines 1–3, the algorithm employs the bottom-up strategy in Lines 4–30 to traverse the $\mathcal{H}[e]$ HINT for each element $e \in q.d \setminus \{e^*\}$ and intersect set C with the divisions of the relevant partitions. For this purpose, the candidates set C is first sorted in Line 5. The Lines 7–29 where each $\mathcal{H}[e]$ HINT is traversed, are almost identical to the Lines 4–26 in Algorithm 2; note that the notation is adapted to the temporal IR search setting (see Table 1). The only meaningful difference is that for every object o in a P^O or P^R division whose time interval overlaps $[q.t_{st}, q.t_{end}]$, we also check if its $o.id$ is contained in the candidates set C using binary search. Finally, in Line 30, set C is updated to include only the *id*s of the objects that contain all examined so far elements, before continuing with the HINT for the next element in $q.d$. Consider again our running example. We first evaluate the typical range (time-travel) query in $\mathcal{H}[\mathrm{a}]$, which defines the initial candidates set $C = \{o_7, o_2, o_4\}$. We then sort C by object *id* and juxtapose it against every relevant division in $\mathcal{H}[\mathrm{c}]$, using binary search. Take for instance $P^O_{3,1} = \{\langle o_6, [\ldots]\rangle, \langle o_7, [\ldots]\rangle, \langle o_8, [\ldots]\rangle\}$. The time interval of all contained objects overlaps with the query interval but we output only o_7 which is included in C. The algorithm continues in a similar fashion, outputting o_2 from $P^O_{2,1}$ and o_4 from $P^O_{0,0}$.

ALGORITHM 4: Time-travel IR query on *t*IF+HINT (using merge-sort for intersections)

```
   Input  : tIF+HINT index, query q = ⟨[t_st, t_end], d⟩
   Output : set of object ids
1  sort q.d by element frequency;                                  ▷ in increasing order
2  e* ← least frequent element in q.d;
3  C ← RangeQuery(H[e*], [q.t_st, q.t_end]);                       ▷ determine candidates using Algorithm 2
4  foreach element e ∈ q.d \ {e*} do
5      sort C;                                                     ▷ by object id
6      foreach level ℓ = m to 0 in the H[e] HINT do
7          f ← prefix(ℓ, q.t_st);                                  ▷ first relevant partition
8          l ← prefix(ℓ, q.t_end);                                 ▷ last relevant partition
9          C ← C ⋂ P^R_{ℓ,f};                                      ▷ update candidates
10         foreach partition i = f to l do
11             C ← C ⋂ P^O_{ℓ,j};                                  ▷ update candidates
12 return C;
```

Although the initial set of candidates C already contains only the objects whose time interval overlaps the query one, we observe that Algorithm 3 still conducts the temporal comparisons in Lines 13, 14, 16, 19 and 23 when traversing the $\mathcal{H}[e]$ HINT for an element e. The reason for this practice is to avoid performing the potentially costly binary search of the $o.id \in C$ condition for all objects contained in a division. In view of this tradeoff, we next revisit the merge-sort as a viable option for the intersections between C and the divisions.

Specifically, we consider a modified version of HINT which orders the contents of each division by the object *id*.[8] Algorithm 4 illustrates the pseudocode of the new query evaluation method for *t*IF+HINT which now utilizes merge-sort to intersect candidates C with HINT divisions. Similar to Algorithm 3, we initialize again the set of candidates C using the $\mathcal{H}[e^*]$ from the least frequent element e^*. Then, for the remaining elements in $q.d$, it suffices to traverse their HINT and directly intersect current set C with the P^O or P^R divisions of the relevant partitions in Lines 6–11. Observe that the P^R divisions are considered only for the first overlapping partition f at each level, while P^O, for all relevant partitions, as discussed in Section 2.3. In addition, also notice that the *compfirst* and *complast* flags from Algorithm 2 are no longer used, simplifying the traversing of the hierarchy; intuitively, there is no difference in traversing HINT in a bottom-up or a top-down fashion. This

[8] Naturally, this sorting is incompatible with the beneficial/temporal sorting optimization (by t_{st} or t_{end}), discussed in Section 2.3. Intuitively, we expect this modification to slow down the traditional range queries on HINT but drastically accelerate the intersections. On the other hand, we can still utilize the subdivisions, the storage optimization and the optimizations for handling skewness & sparsity and cache misses.

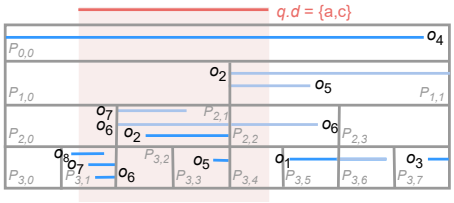

Figure 6: irHINT for the running example: partitioning

is because the algorithm does not perform any temporal comparisons; the correctness of the result is guaranteed as the initial set C contains temporally qualifying objects.

3.2 A Hybrid Index

Despite employing merge-sort to efficiently intersect the candidates set C, Algorithm 4 exhibits a critical shortcoming. Intuitively, the algorithm is able to take full advantage of the HINT infrastructure only for the first element in $q.d$ (i.e., the least frequent element e^*) when determining the initial candidates set C. In the subsequent intersections, all optimizations proposed for HINT (e.g., the bottom-up traversal) are never used with the exception of how to determine the temporally relevant partitions per level. Hence, the querying process of *t*IF+HINT in Algorithm 4 overall intersects exactly the same candidates sets as for *t*IF+Slicing, but with a larger number of partitions (or divisions). This fragmentation of the intersecting process will slow down the queries as the number of elements contained in $q.d$ increases beyond one.

In view of the above, we devise a hybrid index denoted by *t*IF+HINT+Slicing, which pairs the advantages of *t*IF+HINT with those of *t*IF+Slicing. This hybrid IR-first index adopts a dual-structure design. Specifically, the postings list for each term e is stored twice: the *first* copy inside a HINT $\mathcal{H}[e]$ whose divisions are sorted by object id, similar to the *t*IF+HINT variant in Algorithm 4; the *second* copy, divided into sub-lists following a breakdown of its time domain into slices, similar to *t*IF+Slicing. Under this design, given a $q = \langle [t_{st}, t_{end}], d \rangle$ time-travel IR query, we rely on the range querying on $\mathcal{H}[e^*]$ to quickly determine the initial candidates set from the least frequent element e^*, and then, execute the subsequent intersections using the relevant partitions/slices in $\mathcal{I}[e]$, which are typically fewer than the relevant divisions in $\mathcal{H}[e]$, avoiding this way the fragmentation of the intersection process.

For query processing, we expect *t*IF+HINT+Slicing to outperform or at least to be competitive to the best method among *t*IF+Slicing and the other two *t*IF+HINT variants; Section 5 tests this claim. However, we also expect to occupy more space than these methods due to its dual-structure and copies design. The key to reduce the space requirements of *t*IF+HINT+Slicing is the observation that after determining the initial candidates set, the subsequent intersections no longer need to check the temporal condition; the same idea is employed in Algorithm 4 as well. A straightforward way to build upon this observation is to completely omit the time interval $[o.t_{st}, o.t_{end}]$ inside the slice sub-lists for each element e, storing only object's $o.id$. But, in this case, we need to apply hashing or sorting to deal with duplicates. Instead, we store a $\langle o.id, o.t_{st} \rangle$ pair which allows us to employ the more efficient reference value [25] technique, as discussed in Section 2.2 for *t*IF+Slicing.

4 The irHINT Index

We next embark on a different indexing direction to boost temporal IR search. We propose a novel composite index termed irHINT which directly builds upon the state-of-the-art HINT index for intervals. The key idea of irHINT is to use a single HINT to hierarchically index the time domain but inject its structure with inverted indices. Under this premise, time-travel IR queries can benefit from HINT's efficiency to quickly determine the divisions which may contain temporally qualifying objects, and from the inverted indexing to efficiently determine the final results. In what follows, we present two variants of irHINT, elaborating on their strong and weak points; later in Section 5, we experimentally compare these variants for different query workloads and datasets.

4.1 Focus on performance

The first variant primarily focuses on high query performance. The irHINT index comprises a HINT hierarchy for which the contents of every P^O and P^R division are maintained inside a base temporal inverted index as described in Section 2. The construction and the maintenance process are driven by the HINT component; e.g., for inserting a new object o, we first determine which partitions (and their divisions) should store o and then invoke the temporal inverted index building process to include an entry for o in these divisions. A growing time-domain can again be handled as discussed in [21]. Figure 6 illustrates the partitioning of the domain applied by HINT with $m = 3$, for our running example; similar to Figure 5, we color replicas by a lighter shade of blue. For every non-empty division, we construct the $\mathcal{I}^O$, $\mathcal{I}^R$ temporal inverted indices, shown in Table 2.

Given a time-travel IR query, the evaluation process is also driven by the HINT component. Specifically, the hierarchy is traversed following the bottom-up approach (similar to Algorithm 2) and for each relevant P^O or P^R division, a time-travel IR query is issued to the corresponding *t*IF to collect results. The duplicate avoidance principle of HINT which mandates P^R divisions to be checked only for the first relevant partition per level, guarantees that there is no overlap among the outputs of the inverted index searches, and therefore, no need for a de-duplication step. On the other hand, thanks to the *compfirst* and *complast* flags, it is possible to further accelerate the inverted index search in each division. Contrary to Line 5 in Algorithm 1, we no longer have to check both the $q.t_{st} \leq o.t_{end}$ and $o.t_{st} \leq q.t_{end}$ conditions for all divisions, unless both *compfirst* and *complast* flags are set. Algorithm 5 illustrates the query evaluation process of the first irHINT variant, highlighting the necessary changes in the bottom-up approach employed by Algorithm 2; for simplicity, we only include the modified lines. We denote by $\mathcal{I}^O_{\ell,j}$ and $\mathcal{I}^R_{\ell,j}$ the *t*IF indices of the $P^O_{\ell,j}$ and $P^O_{\ell,j}$ divisions for the j-th relevant partition $P_{\ell,j}$ in level ℓ, respectively. In each case (determined by the values of the *compfirst* and *complast* flags, and the nature of current partition j), the algorithm extends the current result set $\mathcal{R}$ with the output of QueryTemporalIF which computes the time-travel IR query on a division's temporal inverted index. For this purpose, QueryTemporalIF employs a slightly modified Algorithm 1 which alters Line 5 according to which temporal conditions need to be checked; the comment over each line in Algorithm 5 specifies the temporal conditions checked in each case.

Table 2: irHINT for the running example: division *t*IFs

element	$\mathcal{I}^O_{0,0}$	$\mathcal{I}^R_{1,1}$	$\mathcal{I}^O_{2,1}$	$\mathcal{I}^R_{2,1}$	$\mathcal{I}^R_{2,2}$	$\mathcal{I}^O_{3,1}$	$\mathcal{I}^O_{3,3}$	$\mathcal{I}^O_{3,5}$	$\mathcal{I}^R_{3,6}$	$\mathcal{I}^O_{3,7}$
a	$\langle o_4, [\ldots]\rangle$	$\langle o_2, [\ldots]\rangle$	$\langle o_2, [\ldots]\rangle$	$\langle o_7, [\ldots]\rangle$	-	$\langle o_7, [\ldots]\rangle$	-	$\langle o_1, [\ldots]\rangle$	$\langle o_1, [\ldots]\rangle$	$\langle o_7, [\ldots]\rangle$
b	$\langle o_4, [\ldots]\rangle$	$\langle o_5, [\ldots]\rangle$	-	-	-	-	$\langle o_5, [\ldots]\rangle$	$\langle o_1, [\ldots]\rangle$	$\langle o_1, [\ldots]\rangle$	-
c	$\langle o_4, [\ldots]\rangle$	$\langle o_2, [\ldots]\rangle$,$\langle o_5, [\ldots]\rangle$	$\langle o_2, [\ldots]\rangle$	$\langle o_6, [\ldots]\rangle$,$\langle o_7, [\ldots]\rangle$	$\langle o_6, [\ldots]\rangle$	$\langle o_6, [\ldots]\rangle$,$\langle o_7, [\ldots]\rangle$,$\langle o_8, [\ldots]\rangle$	$\langle o_5, [\ldots]\rangle$	$\langle o_1, [\ldots]\rangle$	$\langle o_1, [\ldots]\rangle$	$\langle o_7, [\ldots]\rangle$

ALGORITHM 5: Time-travel IR query on irHINT (focus on performance variant)

Input : irHINT index $\mathcal{H}$, query $q = \langle [t_{st}, t_{end}], d\rangle$
Output : set of object ids $\mathcal{R}$

▷ Modified lines from Algorithm 2

▷ for each object $o \in P^O_{\ell,j}$, check $q.st \le o.end \wedge o.st \le q.end$
10 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^O_{\ell,j}, q$);

▷ for each object $o \in P^R_{\ell,j}$, check $q.st \le o.end$
11 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^R_{\ell,j}, q$);

▷ for each object $o \in P^O_{\ell,j}$, check $o.t_{st} \le q.t_{end}$
13 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^O_{\ell,j}, q$);

▷ no temporal checks needed
14 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^R_{\ell,j}, q$);

▷ for each object $o \in P^O_{\ell,j} \bigcup P^R_{\ell,j}$, check $q.st \le o.end$
16 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^O_{\ell,j}, q$) $\bigcup$ QueryTemporalIF($\mathcal{I}^R_{\ell,j}, q$);

▷ no temporal checks needed
18 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^O_{\ell,j}, q$) $\bigcup$ QueryTemporalIF($\mathcal{I}^R_{\ell,j}, q$);

▷ for each object $o \in P^O_{\ell,j}$, check $o.t_{st} \le q.t_{end}$
20 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^O_{\ell,j}, q$);

▷ no temporal checks needed
22 $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryTemporalIF($\mathcal{I}^O_{\ell,j}, q$);

Regarding the HINT optimizations from Section 2.3, the first irHINT variant can directly adopt the subdivisions (with a *t*IF per subdivision) and storage ones. The optimizations that deal with skewness & sparsity and cache misses are also still valid but less meaningful while their integration is more complicated. In contrast, the sorting optimization is incompatible since the postings lists of *t*IF are sorted by object *id* to accelerate the IR search.

4.2 Focus on index size

Apart from a high query throughput, we also expect the first irHINT variant to have a large size. To better understand this issue, assume that the $o.d$ description of every object o contains n elements on average. Also, consider the P^O originals division of a partition P in the hierarchy. The temporal inverted index $\mathcal{I}^O$ for P^O will contain n entries on average, for every object assigned to the division; each of these entries includes both the *id* of the object and its associated time interval. In contrast, the original HINT index would store a single copy of this information per every object assigned to P^O. To make matters worse, the time intervals are in fact used only once, when processing a division; i.e., in Lines 4–6 of Algorithm 1 when scanning the postings list of the least frequent element in $q.d$. The followup list intersections in Lines 7–8 consider only the object *id*s.

In view of the above, we devise a second variant of irHINT which trades query performance for index size. To this end, irHINT now employs two data structures for each division in the hierarchy, essentially decoupling and seperately indexing the two attributes of the objects (besides their *id*). The first structure indexes *solely* the time intervals identically to the original HINT; i.e., for an unoptimized version of the index, we store $\langle o.id, [o.t_{st}, o.t_{end}]\rangle$ entries. The second structure is a *traditional* inverted index that indexes *only* the description attribute $o.d$; i.e., a postings list associates every element e in the global dictionary $\mathcal{D}$ to the ids of the objects inside the division that include e in their description. Back to our running example, the size irHINT variant utilizes the same partitioning of the domain as in Figure 6. The $\mathcal{I}^O$ and $\mathcal{I}^R$ inverted indices constructed for the non-empty divisions are indentical to Table 2, storing however only the object *id*s.

The benefits of this dual-structure approach are multifold. First, the temporal interval of each division object is stored only once which reduces the size of the index. As a consequence (second benefit), this will also reduce the number of cache misses when traversing the irHINT to answer time-travel IR queries. Third, we can take full advantage of all HINT optimizations discussed in Section 2.3. In particular, we can now apply the beneficial sorting which accelerates the scanning of a division (or subdivision) because the sorting by object *id* occurs only in the inverted index; essentially, the two structures for each (sub)division employ different sortings.

Given a time-travel IR query $q = \langle [t_{st}, t_{end}], d\rangle$, the answering process follows once again the bottom-up approach of HINT but the processing of each relevant division consists of two steps. The first step determines the objects whose time interval overlaps $[q.t_{st}, q.t_{end}]$, i.e., we execute Lines 8–22 in Algorithm 2 using the structure that maintains the temporal information of the objects. The determined objects define an initial set of candidates for the results that will come out from the current relevant division. Then, we progressively intersect this candidates set with the postings list of each element in $q.d$ in the division inverted index. To efficiently compute the necessary intersections in a merge-sort fashion, the initial set of the candidates (determined by the range query) is sorted by object *id* before the first intersection. Algorithm 6 provides a pseudocode of time-travel IR search on the second irHINT variant. The algorithm replaces Lines 10, 11, 13, 16 and 20 in Algorithm 2 where overlapping intervals are identified, with a sequence of three statements. The first statement (Lines 10, 13, 17, 20, 24, 28, 32, 36) executes a typical range query on intervals to determine a set of candidates C, the second (Lines 11, 14, 18, 21, 25, 29, 33, 37) sorts C and the third statement (Lines 12, 15, 19, 22, 26, 30, 34, 38) intersects C with the inverted lists in the division to update the results $\mathcal{R}$. The latter invokes QueryIF to execute a typical containment IR search.

5 Experimental Analysis

We implemented all indices in C++, compiled using gcc (v4.8.5) with flags `-O3`, `-mavx` and `-march=native`. [9] Our tests ran on an Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20GHz with 512GBs of RAM, running AlmaLinux 8.5. All data, i.e., inputs and indices, reside in main memory. We did not utilize any inverted file compression to

[9] Source code available in https://github.com/chrauch/irhint.

ALGORITHM 6: Time-travel IR query on irHINT (focus on index size variant)

Input : irHINT index $\mathcal{H}$, query $q = \langle [t_{st}, t_{end}], d \rangle$
Output : set of object ids $\mathcal{R}$

$\mathcal{R} \leftarrow \emptyset$; ▷ initialize result set
$compfirst \leftarrow TRUE$;
$complast \leftarrow TRUE$;
foreach *level* $\ell = m$ **to** 0 **do** ▷ bottom-up fashion
  $f \leftarrow prefix(\ell, q.t_{st})$; ▷ first relevant partition
  $l \leftarrow prefix(\ell, q.t_{end})$; ▷ last relevant partition
  foreach *partition* $j = f$ *to* l **do**
    if $j = f$ **then**
      if $p = l$ **and** $compfirst$ **and** $complast$ **then**
        $C \leftarrow \{o.id | o \in P^O_{\ell,j}, q.t_{st} \leq o.t_{end} \wedge o.t_{st} \leq q.t_{end}\}$;
        sort C; ▷ by object id
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^O_{\ell,j}, C, q.d)$;
        $C \leftarrow \{o.id | o \in P^R_{\ell,j}, q.t_{st} \leq o.t_{end}\}$;
        sort C; ▷ by object id
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^R_{\ell,j}, C, q.d)$;
      else if $i = l$ **and** $complast$ **then**
        $C \leftarrow \{o.id | o \in P^O_{\ell,j}, o.t_{st} \leq q.t_{end}\}$;
        sort C; ▷ by object id
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^O_{\ell,j}, C, q.d)$;
        $C \leftarrow \{o.id | o \in P^R_{\ell,j}\}$;
        sort C; ▷ by object id
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^R_{\ell,j}, C, q.d)$;
      else if $compfirst$ **then**
        $C \leftarrow \{o.id | o \in P^O_{\ell,j} \bigcup P^R_{\ell,j}, q.t_{st} \leq o.t_{end}\}$;
        sort C; ▷ by object id
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^O_{\ell,j}, C, q.d) \bigcup$ QueryIF$(\mathcal{I}^R_{\ell,j}, C, q.d)$;
      else
        $C \leftarrow \{o.id | o \in P^O_{\ell,j} \bigcup P^R_{\ell,j}\}$;
        sort C; ▷ by object id
        $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^O_{\ell,j}, C, q.d) \bigcup$ QueryIF$(\mathcal{I}^R_{\ell,j}, C, q.d)$;
    else if $j = l$ **and** $complast$ **then** ▷ $l > f$
      $C \leftarrow \{o.id | o \in P^O_{\ell,j}, o.t_{st} \leq q.t_{end}\}$;
      sort C; ▷ by object id
      $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^O_{\ell,j}, C, q.d)$;
    else ▷ in-between or last ($l > f$), no comparisons
      $C \leftarrow \{o.id | o \in P^O_{\ell,j}\}$;
      sort C; ▷ by object id
      $\mathcal{R} \leftarrow \mathcal{R} \bigcup$ QueryIF$(\mathcal{I}^O_{\ell,j}, C, q.d)$;
  if f *mod* 2 = 0 **then** ▷ last bit of f is 0
    $compfirst \leftarrow FALSE$;
  if l *mod* 2 = 1 **then** ▷ last bit of l is 1
    $complast \leftarrow FALSE$;
return $\mathcal{R}$;

reduce the storage requirements; such techniques are orthogonal to our focus on query efficiency and so, left out as future work.

5.1 Setup

For the indices built on HINT [19, 20], we used the source code provided by the authors; specifically the `subs+sort` version, which employs the subdivisions and sorting (whenever applicable) optimizations (see Section 2.3). We activated the cache misses optimization, but not the skewness & sparsity for a more update-friendly indexing; see [19]. Also, we dropped the storage optimization in line with [20]. As no code was available, we re-implemented both *t*IF-Slicing and *t*IF+Sharding according to [7] and [4], respectively. Note that HINT's cache misses optimization cannot be paired with the

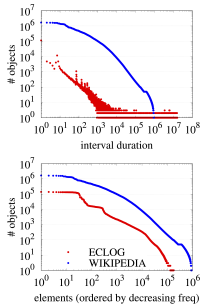

Figure 7: Stats of real datasets

Table 3: Characteristics of real datasets

	ECLOG	WIKIPEDIA
Cardinality	300311	1672662
Size [MBs]	171	4715
Time domain [secs]	15807599	126230391
Min. interval duration [secs]	1	1
Max. interval duration [secs]	15802098	126169456
Avg. interval duration [secs]	1325118	6587819
Avg. interval duration [%]	8.4	5.2
Dictionary size [# elements]	178478	927283
Min. description size [# elems]	1	1
Max. description size [# elems]	14399	6982
Avg. description size [# elems]	72	367
Min. element frequency	1	1
Max. element frequency	140423	1671696
Avg. element frequency	122	675
Avg. element frequency [%]	0.04	0.05

Table 4: Parameters of synthetic datasets

parameter	**values** (defaults in **bold**)
cardinality	100K, 500K, **1M**, 5M, 10M
time domain size	32M, 64M, **128M**, 256M, 512M
α (interval duration)	1.01, 1.1, **1.2**, 1.4, 1.8
σ (interval position)	10K, 100K, **1M**, 5M, 10M
dictionary size	10K, 50K, **100K**, 500K, 1M
$\|d\|$ (description size)	5, 10, **50**, 100, 500
ζ (element frequency)	1.0 1.25, **1.5**, 1.75, 2.0

*t*IF+Slicing and *t*IF+Sharding competitors. Maintaining object ids separately makes sense only when results can be directly produced with no comparisons. However, in *t*IF+Slicing, we always need $o.t_{st}$ along with $o.id$ to apply the reference point de-duplication test, while in *t*IF+Sharding, we scan the shards and compare the object interval endpoints to the query for outputting results.

We experimented with 2 real-world datasets; Table 3 summarizes their characteristics and Figure 7 shows the distribution of interval duration and the frequency distribution for their elements. ECLOG [18] was derived from e-commerce data on HTTP requests, from December 1, 2019, to May 31, 2020. The requests were grouped by session; every session defined an object o with $o.t_{st}$ and $o.t_{end}$ determined from the timestamps of the first and last requests within each session, while $o.d$ was constructed based on the requested URIs. For WIKIPEDIA, we randomly selected 100K articles using the official API[10] and downloaded all their versions (or revisions) from 2020 until 2024. Each version provided an object o with $o.t_{st}$ equal to its creation timestamp and $o.t_{end}$, to the creation timestamp of the next version. The terms in each version were used for $o.d$.

We also generated synthetic datasets by extending the approach from [19] to include object descriptions. Table 4 summarizes the construction parameters and their default values. The datasets cardinality ranges from 100K to 1M, while their time domain from 32M to 512M units and their dictionary size from 100K to 1M elements. The duration of each object interval follows a zipfian distribution, controlled by parameter α. A small value of α makes most intervals relatively long, while with a large value, the majority of intervals have length 1. The element frequency in the global dictionary also follows a zipfian distribution, controlled by the parameter ζ. The middle point of every interval is positioned according to a normal distribution centered at the middle point of the time 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. Last, the size of every object description $|d|$ ranges from 5 to 500.

[10] https://www.mediawiki.org/wiki/API:Main_page

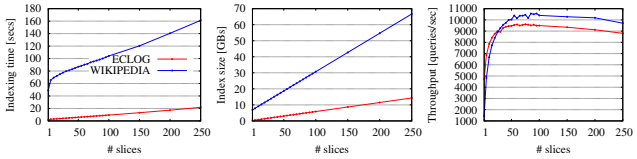

Figure 8: Tuning *t*IF+Slicing

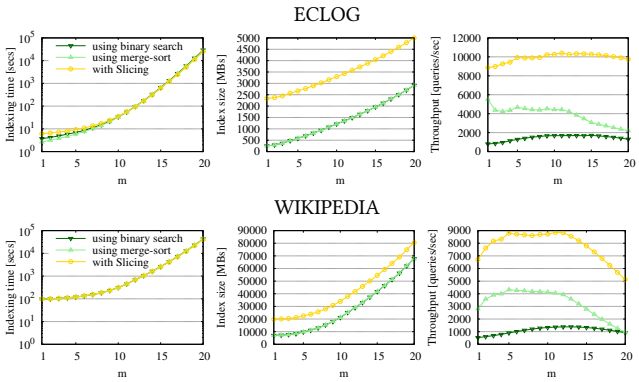

Figure 9: Tuning *t*IF+HINT variants

To assess the efficiency of the indexing methods, we measure their throughput (as the number of evaluated queries per second)[11], while varying the following four experimental parameters: (1) the extent of the query interval as a percentage of the entire domain inside $\{0.01\%, 0.05\%, 0.1\%, 0.5\%, 1\%, 5\%, 10\%, 50\%, 100\%\}$, which allows to monitor the performance from small to large extents, including the 100% extreme case when the query is no longer a time-travel IR one but a typical IR containment query, (2) the number of elements in the query description $|q.d|$ inside the $\{1, 2, 3, 4, 5\}$ value set, (3) the frequency of the query elements drawn from the $[* - 0.1\%]$, $(0.1\%–1\%]$, $(1\%–10\%]$, $(10\%–*]$ bins; for instance, in the $(0.1\%–1\%]$ bin, we generate queries using elements which appear in 0.1% to 1% of the dataset objects, and (4) the selectivity of the query as a percentage of the input cardinality, drawn from the percentage bins 0%, $(0\% - 0.001\%]$, $(0.001\% - 0.01\%]$, $(0.01\% - 0.1\%]$, $(0.1\% - 1\%]$, and $(1\% - 10\%]$.[12] In each test, we ran 10K random time-travel IR queries with a *non-empty* result set; an exception is the 0% bin for (4), where the result is empty. When testing (1) and (2), we vary one of the parameters while fixing the other to its default value, i.e., 0.1% for the query interval extent and 3, for the query description. For (3), we set the query interval extent and the query description to the above default values. Lastly for (4), we never fix the parameter values. Mixed queries allow us to cover different cases.; e.g., zero results occur either because of a very short query interval extent or of many query elements, or due to very infrequent elements.

5.2 Tuning

We first investigate the best parameter setting for *t*IF+Slicing and the *t*IF+HINT variants. In contrast, *t*IF+Sharding is tuned using the cost-aware merging approach of ideal shards which relaxes

[11] Our focus is on applications that manage large volumes of data, offering a search interface to multiple users simultaneously (e.g., public archives). Under this, we expect systems to receive large numbers of short and fast time-travel IR queries and so, reporting query throughput instead of the average time per query is more appropriate.
[12] Note that the query interval extent, the number of query elements and the element frequency parameters indirectly control the query selectivity.

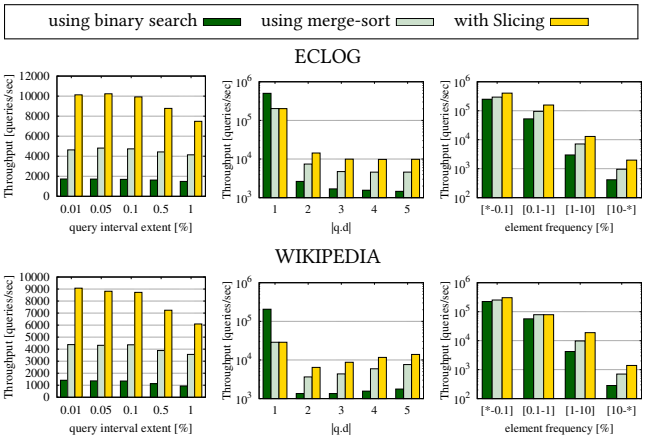

Figure 10: Comparing the throughput for *t*IF+HINT variants

the staircase property requirement in [4]. Figure 8 reports on determining the best number of domain slices for *t*IF+Slicing. As expected the index initially benefits from increasing the number of slices/partitions because the query processing algorithm can more efficiently determine the temporally qualifying objects but the creation of too many sub-lists will eventually impact the execution time due to the higher fragmentation of the intersecting process. Naturally, as the number of slices increases, the index occupies more space and its construction time rises. Hence, for the rest of our analysis, we set the number of slices to 50, the smallest value in the range where the highest query throughput is observed.

We now turn our focus to the *t*IF+HINT variants. For the *t*IF+HINT+Slicing hybrid, we set the number of slices according to Figure 8. To set the number of bits m for the postings HINTs, one option is to directly use the cost model from [19]. We do not expect this approach to be effective since this model was designed for interval data and range queries, without an additional attribute (i.e., description d in our case) involved in the query. As an alternative, we monitor the indexing costs and the query performance while increasing m, and then select the lowest m value which guarantees the highest performance or in its vicinity. Figure 9 reports on these tests. The indexing costs naturally rise as the postings HINTs contain more levels (larger m value). However, in a typical trade-off style, the query efficiency initially improves but then drops. Especially, for the variants that use merge-sort, the performance deteriorates because the subdivisions in the postings HINTs contain increasingly fewer entries which renders computing list intersections via merge-sort inefficient. Under this, the best setting for the *t*IF+HINT variant that uses merge-sort and *t*IF+HINT+Slicing is with $m = 5$, while for the variant that uses binary search, with $m = 10$. In contrast, if we use the cost model from [19], the index costs are significantly higher, similar to $m = 16$ case for both datasets. In regards to querying, we observe no performance improvement for the *t*IF+HINT that uses binary search, while for *t*IF+HINT that uses merge-sort and the *t*IF+HINT with Slicing hybrid, we saw a 45% and 15% slowdown, respectively. Hence, in the rest of our analysis, we set the value of m for the *t*IF+HINT variants according to Figure 9.

Table 5: Indexing costs (no compression used)

index		time [secs]		size [MBs]	
		ECLOG	WIKIPEDIA	ECLOG	WIKIPEDIA
*t*IF+Slicing		5.83	86.8	3124	19150
*t*IF+Sharding		8.57	288	295	7180
*t*IF+HINT	using binary search	74.7	529	1215	21221
	using merge-sort	7.18	103	583	9740
	with Slicing	12.2	143	2667	22507
irHINT	for performance	26.9	580	1218	18022
	for size	22.9	594	415	7124

5.3 Comparing *t*IF+HINT variants

We next compare the three variants of *t*IF+HINT in terms of their indexing costs and query performance. Table 5 and Figure 10 report our findings. The results completely align with our discussions and intuitions in Section 3. First, regarding the query performance, we observe that employing merge-sort to compute the necessary intersections of the candidates sets with HINT's subdivisions, leads to a higher throughput. The variant that uses binary search is efficient only when queries contain a single element (see the second column of Figure 10), where no intersections are performed and the initial set of candidates is output. This variant fully benefits from all HINT optimizations to fast answer the range query used to define the candidate set from the only element e in $q.d$. In contrast, recall that the variant that uses merge-sort does not traverse the $\mathcal{H}[e]$ HINT in a bottom-up fashion and in addition does not utilize any temporal sorting in the subdivisions (see Section 2.3). Last, the *t*IF+HINT+Slicing hybrid exhibits the best performance, excluding again the single-element queries, as it successfully combines the advantages of HINT with merge-sort and of Slicing for organizing the postings lists (more on Slicing in the next paragraphs).

In regards to the indexing costs, there exist two key takeaways. The merge-sort variant has the lowest construction time since no sorting occurs; the subdivisions are implicitly sorted by adding the objects in the order of their *id*. Naturally, the *t*IF+HINT+Slicing hybrid exhibits the highest indexing time as two structures are built for each postings list. In a similar fashion, *t*IF+HINT+Slicing occupies the most space compared to the other two variants, second by the variant that uses binary search. The reason why the non-hybrid variants have different space requirements (contrary to Figure 9 plots) although they use the same structures, is because we set m differently to achieve the highest possible performance; the binary search variant typically requires a larger m. In view of the above, we consider only the *t*IF+HINT+Slicing variant for the rest of our analysis since our focus is primarily on query efficiency.

5.4 *t*IF+HINT and irHINT against competition

The next set of experiments compares our *t*IF+HINT+Slicing and our two irHINT variants against competition, i.e., *t*IF+Slicing [7] and *t*IF+Sharding [4]. For irHINT, out tests showed that the cost model in [19] effectively determines the best m value because of the HINT-first design, in contrast to the *t*IF+HINT variants which are IR-first. Figure 11 reports the throughput of each indexing method on time-travel IR queries and Table 5 report theirs indexing costs.

We start off with our best IR-first method. We observe that *t*IF+HINT+Slicing exhibits a similar overall performance to the best competitor *t*IF+Slicing. Specifically, *t*IF+HINT+Slicing is faster for single-element queries due to benefiting from HINT's fast range (time-travel) querying, but for queries with 2 or more elements, the fragmentation of the intersecting process impacts its relative performance. Nevertheless, *t*IF+HINT+Slicing outperforms *t*IF+Slicing for ECLOG in most tests, because the time intervals of the contained objects are longer compared to WIKIPEDIA (see Table 3), making temporal indexing with HINT more effective.

On the other hand, the performance irHINT variant is overall the fastest indexing method for time-travel IR queries. It significantly outperforms the best IR-first methods, i.e., *t*IF+Slicing and our *t*IF+HINT+Slicing hybrid, on both datasets. Essentially, an IR-first method can outperform the performance irHINT variant only on single-element queries for ECLOG, or on queries that include very rare elements (typically the first two bins, in the third column of Figure 11). The size irHINT variant outperforms the IR-first competition but compared to the performance variant, it is typically slower, as expected. Due to its dual-structure design, probing both structures to answer each query impacts the performance.

In relation to how the experimental parameters affect the performance, the throughput of all indexing methods naturally goes down when the queries become less selective. Specifically, as we increase the extent of the query interval, more objects are temporally relevant and as we increase the frequency of the query elements, the postings lists become longer, rendering the intersection with the candidates list more expensive. The same effect is observed also when we directly reduce the selectivity of the queries in the fourth column of Figure 11. At the same time, we also observe that in all three tests, the advantage of the irHINT variants over the competition rises, as the queries become less selective; the IR-first indices are competitive only when the queries are highly selective, notice for instance the case of zero results. On the other hand, when the number of elements in a query increases (second column), the throughput improves; this is due to the following trade-off. As the value of $|q.d|$ increases, the methods do need to consider more postings lists but the queries become more selective which renders intermediate candidate lists shorter and their intersection with the index partitions, faster. This behavior is better observed for WIKIPEDIA where average length of a postings list is higher compared to ECLOG (see the "Avg. element frequency" in Table 3).

Regarding the index size, both irHINT variants occupy less space than the query-efficient IR-first methods *t*IF+Slicing and *t*IF+HINT+Slicing, due to their HINT-first design. This advantage is of course more pronounced in case of the size irHINT variant, by design. Strictly speaking, the most space efficient indexing method is *t*IF+Sharding because no replication takes place at the expense however to a significantly lower throughput than the irHINT variants, especially the performance one. Finally, in regards to the indexing time, the construction of both irHINT indices takes more time than the others with the size irHINT variant being the most time-consuming due to constructing two structures per subdivision.

Figure 12 reports on the synthetic datasets; we tuned indices similar to Section 5.2. The plots unveil an identical trend to Figure 11. The performance irHINT variant is the most efficient method, followed by the size variant. The dataset cardinality, the domain size, the object description size and the skewness of the element frequency all impact the index efficiency. For instance, increasing the

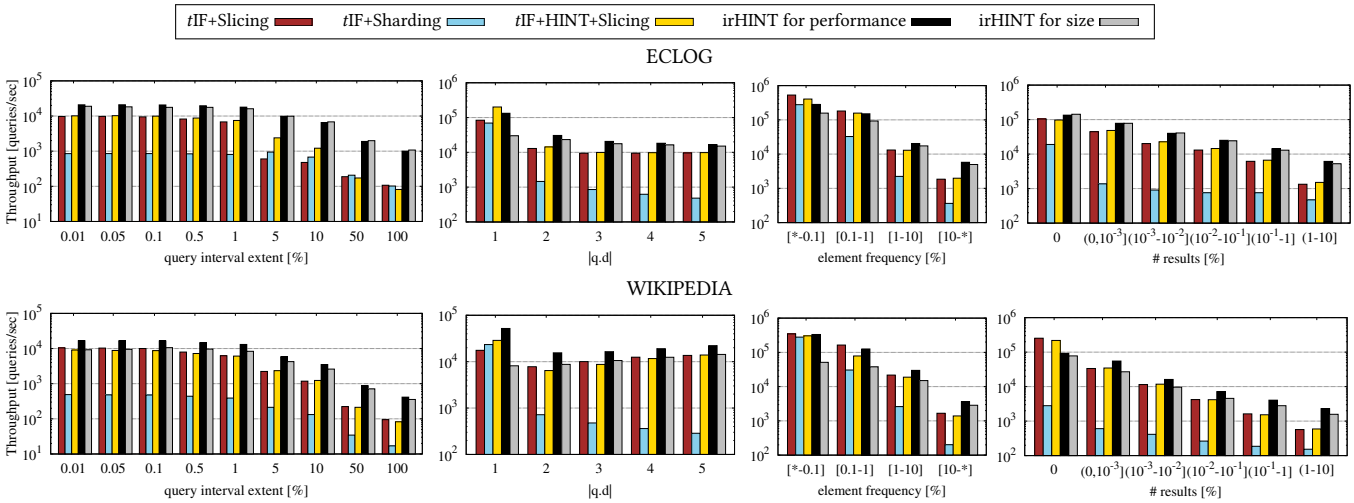

Figure 11: Comparing the throughput for *t*IF+HINT+Slicing and irHINT variants against competition on real datasets

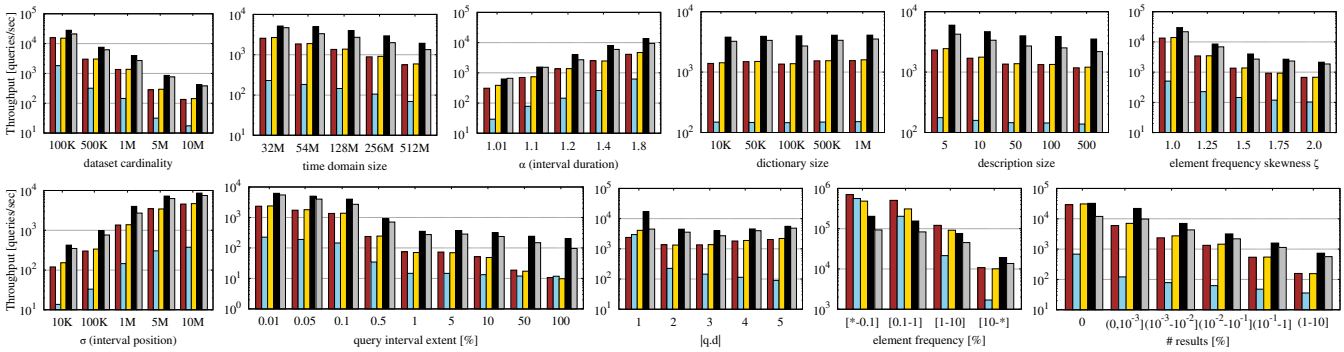

Figure 12: Comparing the throughput for *t*IF+HINT+Slicing and irHINT variants against competition on synthetic datasets

Table 6: Update time [secs] for insertions

index		ECLOG			WIKIPEDIA		
		1%	5%	10%	1%	5%	10%
*t*IF+Slicing		0.17	0.95	1.98	1.16	6.10	11.8
*t*IF+Sharding		0.13	0.61	1.18	2.68	16.0	32.0
*t*IF+HINT	using binary search	1.23	1.81	3.69	7.23	38.5	76.2
*t*IF+HINT	using merge-sort	0.12	0.62	1.60	1.85	9.97	19.7
*t*IF+HINT	with Slicing	0.31	3.28	3.51	3.23	16.8	33.4
irHINT	for performance	0.16	0.87	1.79	3.41	17.8	34.1
irHINT	for size	0.22	1.13	2.31	5.31	28.0	54.0

Table 7: Update time [secs] for deletions

index		ECLOG			WIKIPEDIA		
		1%	5%	10%	1%	5%	10%
*t*IF+Slicing		0.44	2.02	3.95	6.43	32.5	65.9
*t*IF+Sharding		4.74	21.4	42.3	338	1707	3364
*t*IF+HINT	using binary search	0.16	0.78	1.58	6.62	33.8	67.7
*t*IF+HINT	using merge-sort	0.15	0.75	1.48	4.77	24.2	48.3
*t*IF+HINT	with Slicing	0.58	2.83	5.59	11.6	58.3	118
irHINT	for performance	0.30	1.44	2.91	9.22	46.4	96.3
irHINT	for size	0.46	2.22	4.38	15.9	78.4	156

domain size under a fixed query interval extent, affects the performance similar to increasing the query extent, i.e., the queries become longer and less temporally selective. In contrast, when α grows, object intervals become shorter, so the performance of all indices improves. Also, when increasing σ the intervals are more widespread, making the temporal predicate is more selective.

5.5 Updates

Finally, we study the efficiency of all implemented indices in updates. We start off with insertions of new objects. For this purpose, we first index offline 90% of the objects for each dataset and then measure the cost of updating the indexing structure with a batch of 1%, 5% or 10% of the remaining objects. Table 6 reports the insertion times for every batch size and dataset. In most of the cases, the simpler tested IR-first methods, i.e., *t*IF+Slicing and *t*IF+Sharding, exhibit the lowest update times, but the performance irHINT variant is always competitive. The size irHINT is as expected the slowest irHINT variant due to its dual-structure design and the need to maintain the temporal sorting in HINT subdivisions. Similar issues impact the hybrid *t*IF+HINT+Slicing variant (the dual-structure design) and the variant that uses binary search (maintaining temporal sorting). Observe how the merge-sort *t*IF+HINT variant which utilizes only HINT and no temporal sorting outperforms both the other two *t*IF+HINT variants. Note that the sorting by object *id* feature employed by *t*IF+Slicing, the merge-sort *t*IF+HINT variant

and the performance irHINT variant is automatically maintained as the new objects carry larger *id*s than the already indexed ones.

For the deletion updates, we first index offline each real dataset and then measure the cost of removing 1%, 5%, 10% of the indexed objects. In line with previous works, e.g., [19, 30, 47, 54], these objects are not actually deleted; instead, we place tombstones for a logical deletion. Table 7 reports the deletion times. Handling deletion updates partially resembles to querying; i.e., we first need to locate the entries for each deleted object inside the index partitions. Under this, *t*IF+Sharding which has the lowest querying throughput, also exhibits the highest deletion cost, despite there is a single index entry for each deleted object, due to the lack of replication. We also see that the hybrid *t*IF+HINT with slicing and the irHINT variant for size incur high deletion costs due to their dual structure design. Overall, *t*IF+HINT with merge-sort has the lowest deletion time due to its lower replication factor compared to *t*IF+Slicing.

6 Related Work

We last discuss additional related work besides Section 2.

6.1 IR and temporal IR indexing

There exist two main indexing approaches for IR containment queries in the literature: inverted indices or files [67] and signature files [28, 29], although tries have been also considered [59, 61]. We already discussed the traditional inverted index in previous paragraphs. Nevertheless, extensions have been also proposed to enhance its performance, based on the idea to treat long and short postings lists, differently. The *Hybrid Trie Inverted* file (HTI) [61] breaks up the larger inverted lists to smaller sub-lists that contain known combinations of items. König et. al. [44] propose a index structure that similarly to HTI creates inverted lists for combinations of items. The *Ordered Inverted File* (OIF) [60] introduces an ordering for the database elements and records; under this, a B^+-tree organizes the access to all the parts of each postings list.

On the other hand, the idea behind signature files is to hash every element to a fixed size word. By superimposing the codes from all the elements in the object description, the object *signature* is defined. These signatures are used as cheap filters to quickly remove the objects that do not contain the query elements. Extensions which organize object signatures into hierarchical structures, include the signature trees [15, 23, 49, 51]. Similar to previous work on temporal IR indexing, our study focuses exclusively on the inverted index, because previous surveys showed that they outperform signature-based methods for containment queries both on low cardinality set-valued database attributes [35] and on text documents [66].

Besides the slicing and sharding techniques, other work on temporal IR indexing can be found in indexing versioned archives e.g., [52, 53]. The approaches are built upon the idea of maintaining, along with the current document, small delta updates for every version, each one of them with their own creation date timestamp. In case the temporal index includes positional information, e.g., [8, 64], to answer phrase queries for instance, previous work typically partitions every document version into a number of fragments, which are then indexed individually. Contrary to most methods where each version is a separate document, He et al. [34] takes a slightly different approach by modeling a versioned collection using a two-level index, a document level where each distinct document has a global identifier and a version level index. Finally, Huo and Tsotras [36] propose rank-based partitioning solutions which are however applicable only for top-k time-travel *relevance* IR search.

6.2 Indexing intervals

A simple and practical data structure for intervals is a *1D-grid*, which divides the domain into k pair-wise disjoint, partitions $P_1, P_2, \ldots, P_k$. Input intervals are assigned to all partitions they intersect and results to a range query q are obtained by accessing all partitions P_i that overlap with q. To avoid duplicates if the query interval spans multiple partitions, the reference value method [25] can be used.

The *interval tree* [26] offers optimal worst-case space and time guarantees. The tree divides the input 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. This process is repeated recursively for the left and right subtrees using the centers of the corresponding subdomains. The intervals assigned to each tree node are sorted in two lists based on their starting and ending values, respectively. To answer a range query, the interval tree is traversed top-down comparing the center on each node to the query range. A relational interval tree RI-tree for disk-resident data was proposed in [45]. Another binary search tree for intervals is the *segment tree* [22], which however was designed for stabbing (or point) queries where the goal is to determine the intervals that contain a specific value.

In other solutions for indexing intervals, the timeline index [43] is a general-purpose access method for temporal (versioned) data, implemented as SAP-HANA tables. A table called the event list stores a $\langle time, id, isStart \rangle$ triple for the endpoints of all intervals, where $time$ is either the start or end of an interval, specified accordingly by the boolean $isStart$ flag. In addition, at certain timestamps, called *checkpoints*, the entire set of active objects is materialized, i.e., those with an interval that contain the checkpoint. Range queries $q = [q.st, q.end]$ are evaluated by comparing the contents of the closest checkpoint before $q.st$ and the entries in the event list after the checkpoint, against the query range. The period [5] and the RD-index [13] are self-adaptive structures which split the domain into coarse partitions, and then further divide each partition hierarchically to organize the contained intervals based on their positions and durations. They are specialized to range and duration queries.

6.3 Keyword search in temporal databases

Keyword search has also been studied in temporal databases [55], although receiving significantly less attention than its counterpart in typical relational databases. Jia et al. [38] designed a target-oriented search on a augmented data graph (modeling the temporal data) to efficiently evaluate temporal keyword queries. In a different line of approach, Gao et al. [33] showed how temporal keyword queries could be rewritten to SQL queries (including temporal joins), which allows the RDBMS engine to directly evaluate them. Gao et al. [32] shows how temporal aggregation can be employed in temporal keyword search, allowing to query statistical information over time. Nevertheless, the works above do not propose a native indexing

scheme for efficient query processing and so, they are not comparable either to our solutions nor to *t*IF+Slicing and *t*IF+Sharding.

7 Conclusions and Future Work

We studied time-travel IR queries, where the goal is to identify objects whose lifespan temporally overlaps the query time interval and their description contains the query elements. We proposed novel IR-first solutions, which build on the inverted index but organize every postings list using the state-of-the-art interval index HINT. In addition, we devised a novel approach termed irHINT, which injects HINT with inverted indices to organize the contents of every partition. Our tests showed that the performance variant of irHINT outperforms all IR-first solutions (both existing and our proposed) while the size variant achieves the lowest storage requirements. In the future, we will extend our work towards several directions. First, we plan to consider bag semantics and language models for the object description and relevance-based query definitions, e.g., to find the most relevant objects overlapping the query time interval. Furthermore, we plan to investigate the role of compression techniques for both the IR-first and irHINT indices. Finally, we also intend to study other types of temporal IR queries, e.g., joins.

References

[1] Omar Alonso and Michael Gertz. 2006. Clustering of search results using temporal attributes. In *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006*. ACM, 597–598. doi:10.1145/1148170.1148273

[2] Omar Alonso, Michael Gertz, and Ricardo Baeza-Yates. 2009. Clustering and exploring search results using timeline constructions. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*. ACM, 97–106. doi:10.1145/1645953.1645968

[3] Avishek Anand, Srikanta J. Bedathur, Klaus Berberich, and Ralf Schenkel. 2010. Efficient temporal keyword search over versioned text. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*. ACM, 699–708. doi:10.1145/1871437.1871528

[4] Avishek Anand, Srikanta J. Bedathur, Klaus Berberich, and Ralf Schenkel. 2011. Temporal index sharding for space-time efficiency in archive search. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*. ACM, 545–554. doi:10.1145/2009916.2009991

[5] 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. doi:10.1145/3340964.3340965

[6] Klaus Berberich, Srikanta J. Bedathur, Omar Alonso, and Gerhard Weikum. 2010. A Language Modeling Approach for Temporal Information Needs. In *Advances in Information Retrieval, 32nd European Conference on IR Research, ECIR 2010, Milton Keynes, UK, March 28-31, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 5993)*. Springer, 13–25. doi:10.1007/978-3-642-12275-0_5

[7] Klaus Berberich, Srikanta J. Bedathur, Thomas Neumann, and Gerhard Weikum. 2007. A time machine for text search. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27, 2007*. ACM, 519–526. doi:10.1145/1277741.1277831

[8] Andrei Z. Broder, Nadav Eiron, Marcus Fontoura, Michael Herscovici, Ronny Lempel, John McPherson, Runping Qi, and Eugene J. Shekita. 2006. Indexing Shared Content in Information Retrieval Systems. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3896)*. Springer, 313–330. doi:10.1007/11687238_21

[9] Brian Brost, Rishabh Mehrotra, and Tristan Jehan. 2019. The Music Streaming Sessions Dataset. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2594–2600. doi:10.1145/3308558.3313641

[10] Ricardo Campos, Gaël Dias, and Alípio Jorge. 2011. An Exploratory Study on the Impact of Temporal Features on the Classification and Clustering of Future-Related Web Documents. In *Progress in Artificial Intelligence, 15th Portuguese Conference on Artificial Intelligence, EPIA 2011, Lisbon, Portugal, October 10-13, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7026)*. Springer, 581–596. doi:10.1007/978-3-642-24769-9_42

[11] Ricardo Campos, Gaël Dias, Alípio Mário Jorge, and Adam Jatowt. 2014. Survey of Temporal Information Retrieval and Related Applications. *ACM Comput. Surv.* 47, 2 (2014), 15:1–15:41. doi:10.1145/2619088

[12] Ricardo Campos, Gaël Dias, Alípio Mário Jorge, and Célia Nunes. 2014. GTE-Rank: Searching for Implicit Temporal Query Results. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*. ACM, 2081–2083. doi:10.1145/2661829.2661856

[13] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2023. Indexing Temporal Relations for Range-Duration Queries. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management, SSDBM 2023, Los Angeles, CA, USA, July 10-12, 2023*. ACM, 3:1–3:12. doi:10.1145/3603719.3603732

[14] Matteo Ceccarello, Anton Dignös, Johann Gamper, and Christina Khnaisser. 2025. Indexing temporal relations for range-duration queries. *Distributed Parallel Databases* 43, 1 (2025), 7. doi:10.1007/S10619-024-07452-6

[15] Yangjun Chen. 2005. On the Signature Trees and Balanced Signature Trees. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 742–753. doi:10.1109/ICDE.2005.99

[16] Zhida Chen, Lisi Chen, Gao Cong, and Christian S. Jensen. 2021. Location- and keyword-based querying of geo-textual data: a survey. *VLDB J.* 30, 4 (2021), 603–640. doi:10.1007/S00778-021-00661-W

[17] Shiwen Cheng, Anastasios Arvanitis, and Vagelis Hristidis. 2013. How fresh do you want your search results?. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*. ACM, 1271–1280. doi:10.1145/2505515.2505696

[18] Grzegorz Chodak, Grażyna Suchacka, and Yash Chawla. 2020. EClog: HTTP-level e-commerce data based on server access logs for an online store. doi:10.7910/DVN/Z834IK

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

[20] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. HINT: a hierarchical interval index for Allen relationships. *VLDB J.* 33, 1 (2024), 73–100. doi:10.1007/S00778-023-00798-W

[21] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proc. ACM Manag. Data* 2, 1 (2024), 20:1–20:27. doi:10.1145/3639275

[22] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer. https://www.worldcat.org/oclc/227584184

[23] Uwe Deppisch. 1986. S-Tree: A Dynamic Balanced Signature Index for Office Retrieval. In *SIGIR'86, Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Pisa, Italy, September 8-10, 1986*. ACM, 77–87. doi:10.1145/253168.253189

[24] Leon Derczynski, Jannik Strötgen, Ricardo Campos, and Omar Alonso. 2015. Time and information retrieval: Introduction to the special issue. *Inf. Process. Manag.* 51, 6 (2015), 786–790. doi:10.1016/J.IPM.2015.05.002

[25] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. IEEE Computer Society, 535–546. doi:10.1109/ICDE.2000.839452

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

[27] Jonathan L. Elsas and Susan T. Dumais. 2010. Leveraging temporal dynamics of document content in relevance ranking. In *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM 2010, New York, NY, USA, February 4-6, 2010*. ACM, 1–10. doi:10.1145/1718487.1718489

[28] Christos Faloutsos. 1992. Signature Files. In *Information Retrieval: Data Structures & Algorithms*, William B. Frakes and Ricardo A. Baeza-Yates (Eds.). Prentice-Hall, 44–65.

[29] Christos Faloutsos and Stavros Christodoulakis. 1984. Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation. *ACM Trans. Inf. Syst.* 2, 4 (1984), 267–288. doi:10.1145/2275.357411

[30] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175. doi:10.14778/3389133.3389135

[31] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (1998), 170–231. doi:10.1145/280277.280279

[32] Qiao Gao, Mong-Li Lee, and Tok Wang Ling. 2021. Temporal Keyword Search with Aggregates and Group-By. In *Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13011)*. Springer, 160–175. doi:10.1007/978-3-030-89022-3_14

[33] Qiao Gao, Mong-Li Lee, Tok Wang Ling, Gillian Dobbie, and Zhong Zeng. 2018. Analyzing Temporal Keyword Queries for Interactive Search over Temporal Databases. In *Database and Expert Systems Applications - 29th International Conference, DEXA 2018, Regensburg, Germany, September 3-6, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11029)*. Springer, 355–371. doi:10.1007/978-3-319-98809-2_22

[34] Jinru He, Hao Yan, and Torsten Suel. 2009. Compact full-text indexing of versioned document collections. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*. ACM, 415–424. doi:10.1145/1645953.1646008

[35] Sven Helmer and Guido Moerkotte. 2003. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.* 12, 3 (2003), 244–261. doi:10.1007/S00778-003-0106-0

[36] Wenyu Huo and Vassilis J. Tsotras. 2012. A Comparison of Top-k Temporal Keyword Querying over Versioned Text Collections. In *Database and Expert Systems Applications - 23rd International Conference, DEXA 2012, Vienna, Austria, September 3-6, 2012. Proceedings, Part II (Lecture Notes in Computer Science, Vol. 7447)*. Springer, 360–374. doi:10.1007/978-3-642-32597-7_31

[37] Adam Jatowt, Mari Sato, Simon Draxl, Yijun Duan, Ricardo Campos, and Masatoshi Yoshikawa. 2024. Is this news article still relevant? Ranking by contemporary relevance in archival search. *Int. J. Digit. Libr.* 25, 2 (2024), 197–216. doi:10.1007/S00799-023-00377-Y

[38] Xianyan Jia, Wynne Hsu, and Mong-Li Lee. 2016. Target-Oriented Keyword Search over Temporal Databases. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9827)*. Springer, 3–19. doi:10.1007/978-3-319-44403-1_1

[39] Hideo Joho, Adam Jatowt, and Roi Blanco. 2015. Temporal information searching behaviour and strategies. *Inf. Process. Manag.* 51, 6 (2015), 834–850. doi:10.1016/J.IPM.2015.03.006

[40] Rosie Jones and Fernando Diaz. 2007. Temporal profiles of queries. *ACM Trans. Inf. Syst.* 25, 3 (2007), 14. doi:10.1145/1247715.1247720

[41] Nattiya Kanhabua, Roi Blanco, and Kjetil Nørvåg. 2015. Temporal Information Retrieval. *Found. Trends Inf. Retr.* 9, 2 (2015), 91–208. doi:10.1561/1500000043

[42] Nattiya Kanhabua and Kjetil Nørvåg. 2012. Learning to rank search results for time-sensitive queries. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*. ACM, 2463–2466. doi:10.1145/2396761.2398667

[43] 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. doi:10.1145/2463676.2465293

[44] Arnd Christian König, Kenneth Ward Church, and Martin Markov. 2009. A Data Structure for Sponsored Search. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. IEEE Computer Society, 90–101. doi:10.1109/ICDE.2009.37

[45] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. 2000. Managing Intervals Efficiently in Object-Relational Databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Morgan Kaufmann, 407–418. http://www.vldb.org/conf/2000/P407.pdf

[46] Anagha Kulkarni, Jaime Teevan, Krysta M. Svore, and Susan T. Dumais. 2011. Understanding temporal query dynamics. In *Proceedings of the Forth International Conference on Web Search and Web Data Mining, WSDM 2011, Hong Kong, China, February 9-12, 2011*. ACM, 167–176. doi:10.1145/1935826.1935862

[47] David B. Lomet. 1975. Scheme for Invalidating References to Freed Storage. *IBM J. Res. Dev.* 19, 1 (1975), 26–35. doi:10.1147/RD.191.0026

[48] Wei Lu, Zhanhao Zhao, Xiaoyu Wang, Haixiang Li, Zhenmiao Zhang, Zhiyu Shui, Sheng Ye, Anqun Pan, and Xiaoyong Du. 2019. A Lightweight and Efficient Temporal Database Management System in TDSQL. *Proc. VLDB Endow.* 12, 12 (2019), 2035–2046. doi:10.14778/3352063.3352122

[49] Nikos Mamoulis, David W. Cheung, and Wang Lian. 2003. Similarity Search in Sets and Categorical Data Using the Signature Tree. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 75–86. doi:10.1109/ICDE.2003.1260783

[50] Donald Metzler, Rosie Jones, Fuchun Peng, and Ruiqiang Zhang. 2009. Improving search relevance for implicitly temporal queries. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009, Boston, MA, USA, July 19-23, 2009*. ACM, 700–701. doi:10.1145/1571941.1572085

[51] Alexandros Nanopoulos and Yannis Manolopoulos. 2002. Efficient similarity search for market basket data. *VLDB J.* 11, 2 (2002), 138–152. doi:10.1007/S00778-002-0068-7

[52] Kjetil Nørvåg and Albert Overskeid Nybø. 2005. Improving Space-Efficiency in Temporal Text-Indexing. In *Database Systems for Advanced Applications, 10th International Conference, DASFAA 2005, Beijing, China, April 17-20, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3453)*. Springer, 791–802. doi:10.1007/11408079_72

[53] Kjetil Nørvåg and Albert Overskeid Nybø. 2006. DyST: Dynamic and Scalable Temporal Text Indexing. In *13th International Symposium on Temporal Representation and Reasoning (TIME 2006), 15-17 June 2006, Budapest, Hungary*. IEEE Computer Society, 204–211. doi:10.1109/TIME.2006.12

[54] Mark H. Overmars. 1983. *The Design of Dynamic Data Structures*. Lecture Notes in Computer Science, Vol. 156. Springer. doi:10.1007/BFB0014927

[55] Gultekin Özsoyoglu and Richard T. Snodgrass. 1995. Temporal and Real-Time Databases: A Survey. *IEEE Trans. Knowl. Data Eng.* 7, 4 (1995), 513–532. doi:10.1109/69.404027

[56] Giulio Ermanno Pibiri and Rossano Venturini. 2021. Techniques for Inverted Index Compression. *ACM Comput. Surv.* 53, 6 (2021), 125:1–125:36. doi:10.1145/3415148

[57] Kira Radinsky, Krysta M. Svore, Susan T. Dumais, Jaime Teevan, Alex Bocharov, and Eric Horvitz. 2012. Modeling and predicting behavioral dynamics on the web. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*. ACM, 599–608. doi:10.1145/2187836.2187918

[58] Stefano Giovanni Rizzo, Matteo Brucato, and Danilo Montesi. 2023. Ranking Models for the Temporal Dimension of Text. *ACM Trans. Inf. Syst.* 41, 2 (2023), 49:1–49:34. doi:10.1145/3565481

[59] Iztok Savnik, Mikita Akulich, Matjaž Krnc, and Riste Škrekovski. 2021. Data structure set-trie for storing and querying sets: Theoretical and empirical analysis. *PLoS ONE* 16, 2 (2021), 1–38. doi:10.1371/journal.pone.0245122

[60] Manolis Terrovitis, Panagiotis Bouros, Panos Vassiliadis, Timos K. Sellis, and Nikos Mamoulis. 2011. Efficient answering of set containment queries for skewed item distributions. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*. ACM, 225–236. doi:10.1145/1951365.1951394

[61] Manolis Terrovitis, Spyros Passas, Panos Vassiliadis, and Timos K. Sellis. 2006. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006*. ACM, 728–737. doi:10.1145/1183614.1183718

[62] Leong Hou U, Nikos Mamoulis, Klaus Berberich, and Srikanta J. Bedathur. 2010. Durable top-k search in document archives. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 555–566. doi:10.1145/1807167.1807228

[63] Pengming Wang, Qing Chen, and Bin Wang. 2019. Temporal Smoothing: Discriminatively Incorporating Various Temporal Profiles of Queries. In *Information Retrieval - 25th China Conference, CCIR 2019, Fuzhou, China, September 20-22, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11772)*. Springer, 26–38. doi:10.1007/978-3-030-31624-2_3

[64] Jiangong Zhang and Torsten Suel. 2007. Efficient search in large textual collections with redundancy. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. ACM, 411–420. doi:10.1145/1242572.1242628

[65] Yating Zhang, Adam Jatowt, Sourav S. Bhowmick, and Katsumi Tanaka. 2016. The Past is Not a Foreign Country: Detecting Semantically Similar Terms across Time. *IEEE Trans. Knowl. Data Eng.* 28, 10 (2016), 2793–2807. doi:10.1109/TKDE.2016.2591008

[66] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2 (2006), 6. doi:10.1145/1132956.1132959

[67] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. 1992. An Efficient Indexing Technique for Full Text Databases. In *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*. Morgan Kaufmann, 352–362. http://www.vldb.org/conf/1992/P353.PDF