Adaptive Ensemble Indexing for Temporal Information Retrieval via Learned Cost Models

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 substantial interest in Information Retrieval (IR) and related fields, including database search. Our focus is on efficient, adaptive indexing for the fundamental time-travel IR query which returns the objects whose descriptions contain all query elements and their temporal lifespan overlaps with the query time range. However, we deviate from the traditional paradigm where a single structure is used for the entire dataset; instead, we explore the prospect of an index ensemble which employs multiple structures (one for every partition of the input) to better cope with the data and the query characteristics. The blueprint of such an index is called a configuration. We formulate the optimization problem of finding the configuration that minimizes the expected cost, given a representative workload of queries. Due to the vast design space of all possible configurations, we opt for a heuristic approach which synthesizes an approximate solution. Our framework utilizes partitioning and existing temporal IR indexing structures as its building blocks. To support the synthesis, we also propose a novel learned cost model which rapidly evaluates candidate configurations without the need to build and test the actual index. We experimentally evaluate our synthesis process on real-world datasets and compare the performance of the recommended index configuration against single-structure temporal IR indexing.

CCS CONCEPTS

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

KEYWORDS

Temporal data, information retrieval, adaptive indexing, cost models

ACM Reference Format:
Christian Rauch and Panagiotis Bouros. 2026. Adaptive Ensemble Indexing for Temporal Information Retrieval via Learned Cost Models. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD '27).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

SIGMOD '27, June 13–19, 2027, Huntington Beach, CA, USA

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

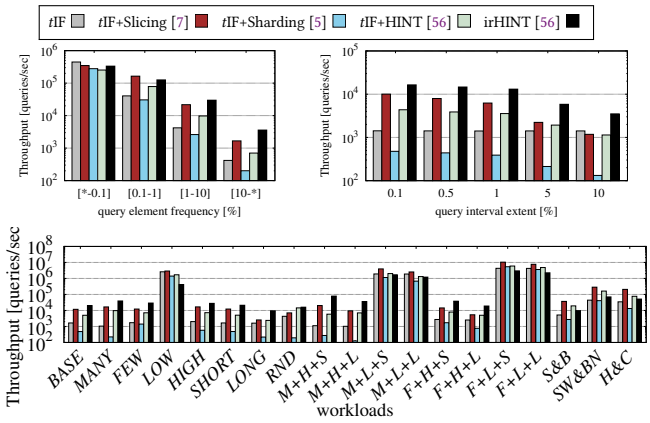

Figure 1: Performance analysis on a WIKIPEDIA versioning dataset; top plots adapted from [56] to include base temporal IF (*t*IF), bottom plots from our experiments in Section 6

1 INTRODUCTION

Temporal Information Retrieval [8, 15, 35] extends the field of traditional Information Retrieval (IR) by incorporating temporal dynamics. Given a collection of objects, the goal is to retrieve the most relevant objects with respect to both their content and their temporal lifespan. Temporal IR search plays a key role in document and web archives, versioning systems, temporal and time-evolving databases, multimedia and scientific databases. Consider for instance the task of retrieving all early articles relevant to COVID-19 pandemic in January 2020 from media and news outlets. We can model such a task as a temporal IR query which returns all articles containing the keywords "COVID", "coronavirus" and "Wuhan", published from 2020-01-01 to 2020-01-31. As another example, consider a network system which monitors HTTP sessions described by requested URIs and the timespan from the first to last packet. To determine possible intrusions, the system inspects sessions that hit sensitive endpoints. We can model such a task as a temporal IR query which returns all sessions requesting e.g., "login.html" and "admin.html" in [2026-08-01 02:00, 2026-08-01 04:00]. Such detection queries are continuously evaluated, repeated by thousands per second over sliding time windows.

Indexing and query processing for temporal IR has received significant attention in the past [4, 5, 7, 24, 26, 50, 51, 56, 60]. Our focus is on the fundamental *time-travel IR* query [5, 56] which retrieves the objects whose description contains all query elements and their lifespan (expressed as a time interval) overlaps with the query time

range.[1] For processing this query, a straightforward approach is to extend classic IR indexing (e.g., the inverted file which is the most dominant IR index [69]) to include object time intervals. Despite its simplicity, such an approach fails to temporally prune the search space which highlights the need for efficient composite or hybrid indexing. Berberich et al. [7] proposed a vertical (time) partitioning of every posting list by first dividing the time domain into disjoint slices. Later, Anand et al. [5] presented a horizontal partitioning, where the contents of each posting list are grouped into temporal shards. Recently, Rauch and Bouros [56] reorganized every posting list as a HINT [11, 12] which is the state-of-the-art in-memory interval index. Alternative to these IR-first approaches that build upon IR indexing, Rauch and Bouros [56] also proposed irHINT which injects the partitions of HINT's hierarchy with inverted indexing.

Motivation. Primarily, the above solutions aim at high query throughput, the most critical factor in modern data systems and cloud services that receive hundreds or thousands of queries per second.[2] The analysis in [56] showed that irHINT is on average the fastest temporal IR index, outperforming its IR-first competitors especially as we increase the extent of the query time range. However, the same study also showed that irHINT is outperformed by either the time-aware inverted file *t*IF or its extension with Slicing [7], when the query elements are very selective due to irHINT's time-first design. This is depicted in Figure's 1 top two plots which we adapted from [56]. Our experiments in Section 6 conducted on a wider range of workloads draws a similar picture at the bottom plot, revealing even more cases where irHINT is outperformed. Consequently, no single method consistently dominates across *all* workloads, which unveils the need for *adaptive* indexing, able to better cope with both the temporal and the IR aspects of the queries.

Adaptive indexing has been extensively studied in the literature under different perspectives: index selection and design, e.g., [6, 9, 19, 27, 31, 33, 38, 42, 44], database cracking, e.g., [22, 23, 28–30, 40, 41, 54, 58, 65, 66], and learned indexing, e.g., [13, 17, 18, 21, 39, 49, 62, 63]. The vast majority of these works adopt the traditional monolithic, "one size fits all" paradigm where a single structure indexes the entire input collection; although we can organize parts of the index with different layouts, e.g., the hot vs cold nodes in [6], or employ independent adjustments per node, e.g., the size ratio and Bloom filter settings in [42]. The most relevant effort to our work is GENE [16] which utilizes a genetic algorithm to breed a synthetic index that combines multiple data structures. Nevertheless, similar to the rest, GENE cannot target composite indexing for temporal IR data. Under this, the prospect of a *workload-aware ensemble* index which employs multiple data structures, one for every different partition of the input, remains still unexplored for temporal IR. To our knowledge, even in other contexts, ensemble indices [36] have received little attention. For instance, to index spatio-textual data, Rocha-Junior et al. [57] first divide keywords into frequent and rare to employ different structures for each. Yet, this design is empirical and the authors do not propose a systematic, generic solution.

[1] In the future, we plan to also consider relevance-based temporal IR search.
[2] E.g., https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/

Contributions. We present a novel generic framework for adaptively indexing temporal IR data, given a set of representative time-travel IR queries. The goal is not to propose a new indexing structure but to re-use and combine the existing as building blocks, hence we focus on formulating the problem and modeling the solution framework. Our framework is extendable, i.e., incorporating new structures is straightforward. Our key contributions are as follows:

- We formally define workload-aware adaptive indexing for temporal IR data as an optimization problem. To deal with the vast index design space and the complexity of the task, we devise a heuristic approach which approximately solves the problem.
- We propose a compositional modeling framework for designing index configurations. A configuration is a concrete plan on how to index the input objects and it may specify a single indexing structure or an ensemble via partitioning.
- We present a synthesis process that efficiently explores the design space of all possible configurations to synthesize an approximation of the best configuration for a given input dataset and a given representative query workload.
- We devise a novel Learned Cost Model which supports the evaluation of candidate index configurations without the need to actually build and test every index. A similar idea was also employed by the Data Calculator in [33] but for evaluating index components (primitives) instead of an entire index.
- We experimentally study our solution by evaluating the cost model, showcasing the synthesis process and comparing synthesized indices against the existing temporal IR indices. Our experiments on three real-world temporal IR datasets showed the benefits of ensemble adaptive indexing. The synthesized indices always match or even exceed the performance of traditional single-structure indices across a variety of diverse workloads.

Outline. Section 2 revisits querying and indexing temporal IR data, and then formulates the problem of workload-aware indexing. Sections 3 and 4 present our solution for the problem at hand; the design framework for index configurations and the synthesis process to determine the best configuration. Section 5 details our novel learned cost model which supports the synthesis. Section 6 reports our experimental analysis. Last, Section 7 discusses related work and Section 8 concludes our study.

2 PRELIMINARIES

We first introduce the necessary notation and recap on the existing solutions for temporal IR search. Then, we formulate the problem of *workload-aware* temporal IR indexing.

2.1 Notation and terminology

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

$$i_1 \cap i_2 = [\max\{i_1.t_{start}, i_2.t_{start}\}, \min\{i_1.t_{end}, i_2.t_{end}\}]$$

We model a data object o as an $\langle id, [t_{start}, t_{end}], \psi \rangle$ triple, where $o.id$ is the object's identifier (used to access other types of information potentially attached to the object), $[o.t_{start}, o.t_{end}]$ is the time interval associated with o representing its lifespan, and $o.\psi$ is the description of the object, containing elements drawn from a global dictionary $\mathcal{D}$. The nature of the attribute $o.\psi$ depends on the application; for example, if o is a document (or a version of a document) then a set $o.\psi$ includes the terms contained in o.[3]

Given an object collection O, we next define temporal IR search, which blends its classic boolean (containment) search counterpart with time-travel search (the latter as a range query on intervals).

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

$$[o.t_{start}, o.t_{end}] \cap [q.t_{start}, q.t_{end}] \neq \emptyset \textbf{ and } o.\psi \supseteq q.\psi$$

If collection O contains e.g., archived document versions, a time-travel IR query finds the versions (not the distinct documents) that contain the query elements (terms); a similar assumption to [5, 56].

2.2 Temporal IR indexing

Temporal IR indices fall under two categories based on which component of the data and which predicate of the query they prioritize. Section 2.2.1 discusses those prioritizing the IR component, building on the inverted index [69], while the indices in Section 2.2.2 prioritize the temporal component, building on time (interval) indexing.

2.2.1 IR-first. A straightforward approach is to augment the posting lists of the classic inverted index with temporal intervals; we denote this solution as *t*IF. Specifically, *t*IF associates every element e in the global dictionary $\mathcal{D}$ with a *time-aware* posting list $L[e]$ of the objects that contain e. Each $\langle o.id, [o.t_{start}, o.t_{end}] \rangle$ entry of the list includes the identifier $o.id$ and the $[o.t_{start}, o.t_{end}]$ time interval of an object o. To answer a time-travel IR query $q = \langle [q.t_{start}, q.t_{end}], q.\psi \rangle$, it suffices to intersect the posting lists of all elements $e \in q.\psi$, ignoring the entries for the objects whose time interval does not overlap the $[q.t_{start}, q.t_{end}]$ query interval. For performance, the entries inside a posting list are ordered by their object *id* to efficiently compute list intersections in a merge-sort fashion. Further, the elements in $q.\psi$ are examined by their frequency in the global dictionary in increasing order, to reduce the size of the intermediate lists and hence accelerate list intersections.

Despite its simplicity, *t*IF fails to efficiently filter out objects based on their time interval as no temporal indexing is in place. The solutions below directly target this shortcoming.

***t*IF+Slicing**. To enrich *t*IF with temporal indexing, Berberich et al. [7] first define a breakdown of the time domain into a sequence of non-overlapping *slices*. Under this *slicing*, every posting list $L[e]$ is then divided into smaller sub-lists, each representing a slice of the time domain. An entry $\langle o.id, [o.t_{start}, o.t_{end}] \rangle$ in the original $L[e]$ *t*IF list is now assigned to all slices it temporally overlaps and replicated in their sub-lists. This replication is necessary to ensure the correctness of query processing. To eliminate potential duplicate results, we can use hashing or the reference value technique from [20]. Determining the number of domain slices is modeled as an optimization problem. To mitigate over-fragmentation from excessive slicing, adjacent time slices can be merged to balance query efficiency and storage overhead [7]. Given a time-travel IR query q, only the sub-lists of $L[e]$ whose slices overlap $[q.t_{start}, q.t_{end}]$ are considered for each element $e \in q.\psi$, avoiding this way to scan temporally irrelevant entries.

***t*IF+Sharding**. Instead of slicing the time domain to partition *t*IF posting lists, Anand et al. [5] reorganize their entries on t_{start}, into groups called *shards*. An ideal shard satisfies the *staircase* property; i.e., for any two $\langle o_1.id, [o_1.t_{start}, o_1.t_{end}] \rangle$ and $\langle o_2.id, [o_2.t_{start}, o_2.t_{end}] \rangle$ entries in a shard, if $o_1.t_{start} < o_2.t_{start}$, then $o_1.t_{end} \leq o_2.t_{end}$. Compared to *t*IF+Slicing, *t*IF+Sharding requires no entry replication and so, result de-duplication is no longer necessary. Upon querying, a shard is scanned until its first entry that starts after the $[q.t_{start}, q.t_{end}]$ interval is found; the remaining entries are guaranteed not to overlap the query. In practice, the number of ideal shards per posting list can be overwhelmingly large. To address this issue, the authors introduced relaxed sharding which permits some violations of the staircase property. The relaxation factor is a tunable parameter balancing pruning efficiency and fragmentation.

***t*IF+HINT**. Recently, Rauch and Bouros [56] stored *t*IF posting lists, using the state-of-the-art interval index HINT [11, 12]. For an element e, HINT $H[e]$ hierarchically and uniformly divides the time domain covered by the original $L[e]$ list, into 2^ℓ partitions for $\ell = 0$ to m, defining m+1 levels. The interval of every $\langle o.id, [o.t_{start}, o.t_{end}] \rangle$ entry is normalized, discretized in the $[0, 2^{m-1}]$ domain, and the entry is assigned to the smallest set of partitions from all levels that cover $[o.t_{start}, o.t_{end}]$ (at most 2 partitions per level). For a time-travel IR query q, it suffices to first evaluate a range interval query on the $H[e^*]$ HINT of the least frequent element e^* in $q.\psi$, to quickly obtain the temporally qualifying candidates. Then, these candidates are refined through intersections with $H[e]$ partitions for each $e \in q.\psi \setminus \{e^*\}$. Three intersection strategies are supported: (1) verification via the object descriptions, (2) binary-search on sorted candidate sets, and (3) merge-sort when HINT partitions are sorted by object *id*. HINT's duplicate avoidance principle ensures that no de-duplication is necessary. The number of HINT levels and the intersection strategy are tunable parameters.

2.2.2 Time-first. Rauch and Bouros [56] also devised a time-first, composite indexing approach called irHINT which directly builds on the temporal (interval) indexing of HINT. Specifically, irHINT injects every partition of HINT's hierarchy with time-aware inverted indexing, i.e., instances of base *t*IF. Given a time-travel IR query q, computation extends the interval range query process of HINT. It suffices to traverse HINT's hierarchy in bottom-up fashion and independently evaluate query q on each partition overlapping $[q.t_{start}, q.t_{end}]$ using the corresponding *t*IF index. This time-first organization enables efficient pruning at multiple time granularities before IR search matching, while again HINT's duplicate avoidance principle ensures that no de-duplication is necessary. The number of levels in the hierarchy is a tunable parameter that balances temporal pruning granularity and index size.

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

The authors additionally proposed a second irHINT variant which trades query performance for a smaller index size and therefore, lower space requirements. This variant adopts a dual structure design per partition, decoupling temporal from IR indexing. Under this, each temporal interval is stored only once rather than being replicated across multiple posting lists. In particular, a separate data structure is maintained for the temporal component of the input objects using the $\langle o.id, [o.t_{start}, o.t_{end}]\rangle$ entries (similar to vanilla HINT) while traditional (non-temporal) inverted indexing is used for the object's description $o.\psi$. Upon querying, it suffices to compute a typical interval range query to determine a list of temporally qualifying objects and then intersect these candidates with posting lists, similar to a traditional IR boolean search.

2.3 Problem definition

Let O be a collection of data objects (as defined in Section 2.1). An *index configuration* $\mathcal{I}$ defines a concrete plan on how to index the objects in O for computing time-travel IR queries. A configuration specifies either a *single* indexing structure (i.e., one of the structures discussed in Section 2.2), which covers the entire global dictionary $\mathcal{D}$ and the entire time domain, or an index *ensemble* which utilizes multiple structures, each indexing a different part of O. We denote by $\mathcal{S}(O)$ the *design space* of all possible index configurations for an object collection O.

Now, to formally define the problem at hand, assume that a workload of representative time-travel IR queries Q is available for collection O.[4] Under this assumption, we define the optimization problem of determining the index configuration $\mathcal{I}$ in $\mathcal{S}(O)$ that minimizes its expected cost captured by the cost function $C(\mathcal{I}, Q)$.

Definition 2.2 (Workload-aware temporal IR indexing). Given an object collection O and a workload of time-travel IR queries Q, the goal is to find the index configuration $\mathcal{I}^*$ in $\mathcal{S}(O)$ such that

$$\mathcal{I}^* = \arg \min_{\mathcal{I} \in \mathcal{S}(O)} C(\mathcal{I}, Q)$$

The nature of the cost function $C(\cdot, Q)$ depends on the application. Without loss of generality, for the rest of our analysis, we consider a weighted linear combination of the query throughput and the index size, prioritizing the former.[5]

The key challenge in solving the problem of Definition 2.2 stems from the prohibitively large design space $\mathcal{S}(O)$. Under this, Sections 3–5 describe our heuristic approach to synthesize an index configuration that approximates optimal $\mathcal{I}^*$.

3 INDEX CONFIGURATION FRAMEWORK

Our framework for designing temporal IR index configurations comprises three components: data partitioning, hierarchical composition and adaptive parameter tuning. In what follows, we describe these components and then explain how to handle construction, queries and maintenance for the index described in a configuration.

[4] By "representative", we refer to historical queries with similar features to current or future ones. This assumption is typical in adaptive indexing [6, 9, 59]. In Section 4, we explain how to deal with a cold start (where no workload is available) or with data distribution shifts which significantly alter the workload characteristics.

[5] Such a setting represents modern data systems which service large numbers of queries per second. Under this, index size is still critical while the one-time building cost is less important. Nevertheless, our work can be extended to incorporate or prioritize other factors such as build and update time.

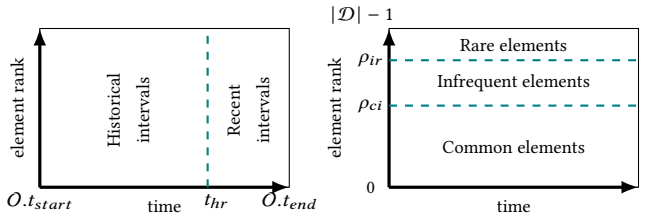

Figure 2: Examples of splitting strategies for data partitioning; time-based (left) and dictionary-based (right); t_{hr}, ρ_{ci}, ρ_{ir} boundaries and historical/recent, common/infrequent/rare labels are symbolic placeholders for illustration purposes

3.1 Data partitioning

As discussed in Section 2.3, an index configuration $\mathcal{I}$ may specify an ensemble of data structures, contrary to the classic paradigm which employs a single structure. For this purpose, our framework considers a break down of the input collection O into a set of potentially *overlapping* partitions $O_1, \ldots, O_n$. Each partition is indexed independently from the rest, using one of the data structures in Section 2.2. The collection of these partition indices form the ensemble. The framework considers two partitioning schemes.[6]

3.1.1 Time-based partitioning. The first option is to temporally define the partitions such that each one covers a specific time range. We divide the $[O.t_{start}, O.t_{end}]$ domain into a number of ranges. For instance, an intuitive practice would be to separate the recent from the historical objects according to a specific time point, e.g., t_{hr} exemplified in Figure 2 (left). Nevertheless, our framework supports *any* number of splitting time points to accommodate different access patterns. Every data partition O_i is defined for either a single time range or a contiguous sequence of ranges, and O_i includes all objects o whose $[o.t_{start}, o.t_{end}]$ interval overlaps with the partition's time range. Our time-based partitioning resembles an irregular 1-dimensional grid, allowing time slices (ranges) to overlap if necessary. Each object in collection O will be replicated to all partitions its interval overlaps. In Figure 2 (left), we define two partitions: O_1 for the historical objects, O_2, for the recent; objects whose interval spans both ranges, i.e., cut by t_{hr}, will be assigned to both partitions.

3.1.2 Dictionary-based partitioning. The second option is to define the partitions by dividing the global dictionary $\mathcal{D}$. To this end, we introduce the *rank* $\rho(e)$ of every element e in $\mathcal{D}$, based on how many times it appears in the input collection. The most frequent element gets rank 0, while the least frequent, $|\mathcal{D}|-1$. We then divide $[0, |\mathcal{D}| - 1]$ into a number of potentially overlapping rank ranges, typically separating high-frequency elements from low-frequency ones. For instance, Figure 2 (right) exemplifies how three disjoint rank ranges are defined using boundaries ρ_{ci}, ρ_{ir}: the $[0, \rho_{ci}]$ range for the most frequent (common) elements, the $[\rho_{ci}, \rho_{ir}]$ for less frequent elements (called infrequent in the figure) and $[\rho_{ir}, |\mathcal{D}|-1]$ for the least frequent (rare). The three dictionary partitions derive

[6] For this study, the two schemes are only exclusively applied; in the future, we will investigate hybrid approaches that combine them.

from prior findings in [56] and our testing, but similar to time-based partitioning, we can support any number of splitting rank boundaries (Section 6 tests two partitions too as in [57]). To ensure that rank ranges capture meaningful distinctions across the element distribution, the framework uses logarithmic scaling to specify rank boundaries as $\lceil |\mathcal{D}|^{\gamma} \rceil$, where $\gamma \in [0, 1]$ denotes a normalized log-scaled position: a target rank ρ maps to $\gamma = \frac{\log \rho}{\log |\mathcal{D}|}$. Finally, an O_i partition is defined for each rank range, including all objects whose description contains elements of a rank in range. Under this assignment, an object may be replicated to multiple partitions.

3.2 Hierarchical composition

We model an index configuration $\mathcal{I}$ as a two-level hierarchy which details the definition and the construction of the index ensemble. We consider the following two types of nodes as the building blocks:

fanout node (root) partitioning $O_1, \ldots, O_n$ { Time-based | Dictionary-based }

terminal node (leaf) indexing structure for partition O_i { base *t*IF | *t*IF+Slicing | *t*IF+Sharding | *t*IF+HINT | irHINT }

The fanout (root) node (colored in green) represents an $O_1, \ldots, O_n$ partitioning of the input collection O; the node is labeled after the partitioning scheme, i.e., time-based or dictionary-based. Every child is a terminal (leaf) node (colored in orange), which represents a partition O_i.[7] Terminal nodes are labeled after the structure utilized to index their partition. Our model currently supports the data structures discussed in Section 2.2 but it can be extended to include additional structures. The range (time or rank) used to define every partition is specified as a label on the directed edge that connects the fanout node to the corresponding terminal.

Figure 3 exemplifies the index configurations for our running example. The left configuration employs a time-based partitioning using the left splitting in Figure 2. Without loss of generality, the ensemble index comprises an irHINT for the historical objects and a *t*IF+HINT for the recent ones.[8] The right configuration defines a dictionary-based partitioning using the right splitting in Figure 2. The index ensemble consists of an irHINT for the common elements, a *t*IF+HINT, for the infrequent and a base *t*IF for the rare.

3.3 Adaptive parameter tuning

Next, we elaborate on the tunable parameters of the data structures at the terminal nodes. All *t*IF variants in Section 2.2.1 (besides base *t*IF) include such parameters to control the posting list fragmentation; *t*IF+Slicing has the number of domain slices p; *t*IF+Sharding, the staircase relaxation factor r which controls the number of shards; *t*IF+HINT, the number of levels m per HINT hierarchy.

Setting p, r, and m uniformly across all posting lists may over-fragment small lists and under-fragment large ones, both leading to inefficient query processing. Over-fragmentation additionally incurs a space overhead. Existing adaptive schemes [5, 7, 11] address this to varying degrees but have limitations. For setting p, the cost model in [7] merges slices of the time domain to balance index size and query performance, but does not directly consider the extent of the query time interval or the selectivity of the query elements.

[7] If the configuration specifies a single indexing structure then the hierarchy is reduced to a single terminal node representing the entire collection O.
[8] Section 4 elaborates on how to select the best data structures on each terminal node.

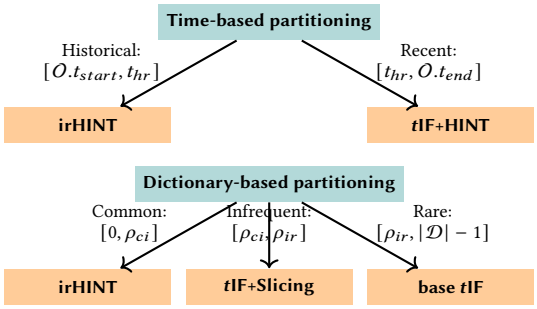

Figure 3: Examples of configurations for Figure 2's splittings

Moreover, the merging process is time-consuming and good merging thresholds may vary with posting list size. The model in [5] for setting r only accounts for the ratio of random to sequential access cost and determines a single factor for all posting lists, therefore failing to adapt independently to each posting list. For setting m, the cost model in [11] considers the extent of the query time interval but ignores IR-specific characteristics. The experiments in [56] confirmed these limitations, forcing the authors to ultimately opt out for manual tuning. Overall, existing approaches only coarsely adapt to the input data and the query workload.

In view of the above, we propose an alternative, adaptive tuning strategy which sets p, r, m as a function of the posting list size $|L[\cdot]|$ and workload-tuned parameters α, β. Intuitively, α determines the base scaling and β, the sub-linear adaptation to posting list size. This adaptive approach allocates fine-grained partitioning to frequent elements with long posting lists for efficient pruning, while rare elements maintain compact structures. Specifically, for *t*IF+Slicing, the time domain covered by every posting list $L[\cdot]$ is divided into

$$p = \lceil 10^{\alpha \cdot (\log_{10} |L[\cdot]| - \beta)} \rceil \tag{1}$$

slices, using exponential scaling. For *t*IF+Sharding, we set the relaxation factor according to

$$r = e^{-\alpha \cdot \max(0, \log_{10} |L[\cdot]| - \beta)} \tag{2}$$

with exponential decay which yields maximum relaxation for posting lists with fewer than 10^{β} records. Last, for *t*IF+HINT, we set the number of levels m in the HINT hierarchy for each list $L[\cdot]$ as

$$m = \lceil \alpha \cdot (\log_{10} |L[\cdot]| - \beta) \rceil \tag{3}$$

using logarithmic scaling. For all formulas, α, β are tuned as hyper-parameters during the index synthesis process (see Section 4). Last, for irHINT's number of levels m we use a single hyper-parameter; unlike *t*IF+HINT, irHINT employs a single HINT hierarchy.

3.4 Index operations

All operations on the index defined by a configuration are driven by the hierarchical composition model. We use the JSON format to materialize index configurations, including now additional implementation details such as the index variant and the values for the tunable parameters; Figure 4 exemplifies the configurations from Figure 3. Building the ensemble index for an input collection O is straightforward. We first compute the partitions according to the partitioning scheme and the ranges specified in the configuration. Then, for

```
{"partitioning":"time-based", "mode":"split", "terminals": [
   {"range":"O.t_start, t_hr", "structure":"irhint;performance;m"},
   {"range":"t_hr, O.t_end", "structure":"tif;hint;merge;α;β"},
```

```
{"partitioning":"dictionary-based", "mode":"refine", "terminals": [
   {"range":"0, ρ_ci", "structure":"irhint;perfomance;m"},
   {"range":"ρ_ci, ρ_ir", "structure":"tif;slicing;α;β"},
   {"range":"ρ_ir, |D| − 1", "structure":"tif;base"} ] }
```

Figure 4: Materializing Figure 3's configurations

each partition O_i, we construct the indexing structure invoking its native build method. Figure 4 lists the data structures (specific variant as well) to build and the values of their tunable parameters. For example, "structure":"irhint;perfomance;m" specifies a performance irHINT variant of m levels, while "structure":"tif;hint;merge;α;β" specifies a *t*IF+HINT variant which employs merge-sort for list intersections; values α, β are used to determine the number of levels per HINT hierarchy as detailed in Section 3.3.[9]

Queries are handled in a top-down fashion. A time-travel IR query q is submitted to the fanout (root) node of the configuration hierarchy. The role of this node is twofold. First, it determines which indices from the terminal (leaf) nodes should be probed. If a time-based partitioning is defined then query q is forwarded to the terminal nodes whose time range overlaps with the $[q.t_{start}, q.t_{end}]$ interval. Otherwise, if a dictionary-based partitioning is defined then q is forwarded to the terminal nodes whose rank range includes the rank of an element in the $q.\psi$ query description. Second, the fanout node orchestrates how the query results are produced; the key task is how to combine the results from the relevant terminal indices. The framework offers two modes of operation. Under the *split* mode, the relevant terminal indices process the submitted query independently of each other. The final result is produced by unifying the local results from the indices, however, if the relevant to query q partitions are not disjoint (typically the case), extra care is taken for removing duplicate results. For this, we can simply use hashing or sorting; for the time-based partitioning, we can alternatively use the more efficient reference value technique from [20]. Our time-based partitioning utilizes the split mode; see the top JSON listing in Figure 4. In contrast, the dictionary-based partitioning employs the *refine* mode where the terminal indices are used in a serial fashion. The index responsible for the first (typically the least frequent) element in $q.\psi$ generates a list of candidate results, which are then refined against the index for the next element and so on, until all relevant indices are probed. In Figure 4, the bottom listing specifies a refine mode for the dictionary-based partitioning.

Last, we discuss index maintenance. Similar to previous work, we focus on batch updates. The key is to use the index configuration to determine which terminal nodes are affected; after that we rely on the native update operations of each data structure. Consider insertions first. For the time-based partitioning, the case when the $[O.t_{start}, O.t_{end}]$ domain remains fixed, is straightforward, i.e., the time ranges of the terminal nodes will route every new object to its relevant partitions. Otherwise, if the domain is extended then we accordingly update the partition ranges that contain $O.t_{end}$ before indexing the new objects.[10] For instance, in our example, we modify the range for the second partition (recent intervals) to $[t_{hr}, O.t'_{end}]$, where $O.t'_{end}$ is the updated domain end. For the dictionary-based partitioning, if the new objects contain only elements already in the global dictionary $\mathcal{D}$, we simply route the objects to their relevant partitions. Typically, we do not expect the updates to alter the element ranks; if they do however, we preserve the existing ranked-based partitions to limit the maintenance cost. On the other hand, to deal with new elements, we incorporate gaps when computing the ranks to accommodate additions. Regarding deletions, we adopt a lazy update approach for both partitioning schemes, i.e., we never modify the splitting boundaries and partition ranges.

4 INDEX SYNTHESIS

The framework in Section 3 provides the tools to design an (ensemble) index for input collection O. We now discuss how to determine the configuration $\mathcal{I}^*$ with the lowest expected cost for workload Q, i.e., how to solve the problem in Definition 2.2. As already explained, due to the vast design space $\mathcal{S}(O)$, we opt for a heuristic approach to synthesize a configuration that approximates the optimal $\mathcal{I}^*$.

For this purpose, we introduce the concept of a *configuration template* or simply *template*, denoted by $\mathcal{T}$, as a family of configurations that share exactly the same partitioning scheme (time-based or dictionary-based and number of partitions) and the same choices of indexing structures for the terminal nodes. Intuitively, a template $\mathcal{T}$ introduces *variables* which replace concrete values for the splitting time points or ranks in partitioning and for the tunable parameters of the indexing structures.[11] Consider for example the template listed in Figure 5 (left). Similar to an index configuration, a template is modeled by a configuration hierarchy and then materialized in JSON format. This particular template represents variations of the dictionary-based configuration from Figures 3 and 4, where the number of levels m in the performance irHINT (common elements partition, first terminal node) and the α, β parameters in the merge-sort *t*IF+HINT (infrequent elements, second terminal node) are defined as variables and marked by a "?". For simplicity, we considered the rank ranges fixed in this example. When defining the m, α, β variables, we can also specify a range for their possible values, e.g., m should take a value from 5 to 16.

Algorithm 1 illustrates a high-level pseudocode of the synthesis process. In principle, the algorithm iterates through configuration templates selected from the $\mathcal{S}(O)$ design space (for-loop in Lines 2–18).[12] For each template $\mathcal{T}$, the algorithm performs a multi-stage grid search, organized in two phases. The purpose of the *selection* phase (Lines 3–8) is to pick k candidate configurations from $\mathcal{T}$'s design space with low $C(\cdot, Q)$ cost. The goal of the *refinement* phase (Lines 9-16) is to improve this initial candidates set by synthesizing configurations very similar to each candidate. As the algorithm goes through every template, it maintains the best configuration $\mathcal{I}^*$ found so far (Lines 17–18), which is reported at the end.

[9] In both configurations shown, m, α, β represent specific fixed values.

[10] Case when the domain is extended by moving $O.t_{start}$ back in time, is symmetric.

[11] In the future, we plan to investigate more generic templates where for instance the number of partitions are also a variable.

[12] Currently, the selected templates are defined manually based on experimentation and intuitive heuristics, e.g., base *t*IF is typically the best structure for rare elements [56]. In the future however, we will investigate how to generate these templates automatically.

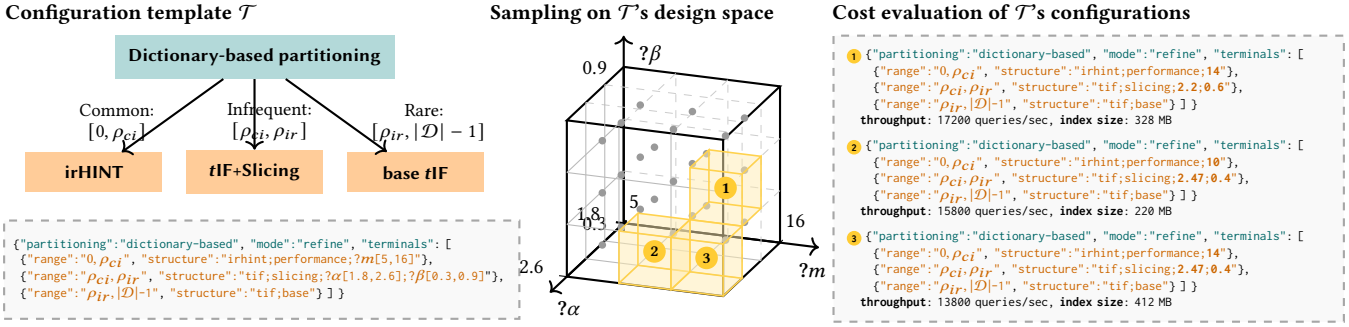

Figure 5: Example of the synthesis process for a dictionary-based partitioning template; t_{hr}, ρ_{ci}, ρ_{ir} boundaries are symbolic placeholders representing fixed values for illustration purposes

ALGORITHM 1: Index synthesis

Input : object collection O, query workload Q, parameter k
Output : index configuration $\mathcal{I}^*$
Variables : top-k configuration list K organized by evaluation cost $C(\cdot, Q)$

1 $\mathcal{I}^* \leftarrow \emptyset$; ▷ $C(\mathcal{I}^*, Q) = \infty$
2 **foreach** *template* $\mathcal{T} \in \mathcal{S}(O)$ **do**
▷ selection phase
3 $K \leftarrow \emptyset$; ▷ initialize top-k list
4 **synthesize** candidate configurations from $\mathcal{T}$; ▷ sample $\mathcal{T}$'s design space
5 **foreach** *candidate configuration* $\mathcal{I}$ **do**
6 **calculate** $C(\mathcal{I}, Q)$;
7 **if** $C(\mathcal{I}, Q) < C(K.\text{top}(), Q)$ **then**
8 **update** top-k configurations with $\mathcal{I}$; ▷ update K
▷ refinement phase
9 **repeat**
10 **synthesize** candidate configurations from K; ▷ local search
11 $K \leftarrow \emptyset$; ▷ reset top-k configurations
12 **foreach** *candidate configuration* $\mathcal{I}$ **do**
13 **calculate** $C(\mathcal{I}, Q)$;
14 **if** $C(\mathcal{I}, Q) < C(K.\text{top}(), Q)$ **then**
15 **update** top-k configuration with $\mathcal{I}$; ▷ update K
16 **until** *no cost improvement observed*;
17 **if** $C(\mathcal{I}^*, Q) > C(K.\text{top}(), Q)$ **then**
18 $\mathcal{I}^* \leftarrow K.\text{top}()$; ▷ update best configuration so far
19 **return** $\mathcal{I}^*$;

We elaborate on how the two phases navigate the design space of a template $\mathcal{T}$. For this, we construct a multi-dimensional space defined by the variables in the template's configuration as dimensions; the extent of this space is set according to the value ranges provided for the variables in the configuration. Each concrete index configuration that materializes $\mathcal{T}$ is a point in this space whose coordinates correspond to its specific variable values. The selection phase in Algorithm 1 conducts a grid search in Line 4 over the template's multi-dimensional space to systematically sample configurations at regular intervals across each parameter dimension; the grid granularity (resolution) is a system, tunable parameter. This sampling process provides good coverage and is effective for the low-dimensional parameter spaces typically found in our index configurations. Figure 5 (middle) shows the 3-dimensional design space for our example template (left), based on the $?m$, $?\alpha$ and $?\beta$ variables; the figure also plots the configurations sampled by the grid search, which correspond to the center of every cube cell. The best k of the sampled configurations (with the lowest expected cost) serve as anchors for local search during the refinement phase.

The refinement phase recursively employs grid-based sampling to search subspaces of $\mathcal{T}$'s design space. At each round, Algorithm 1 iterates over every candidate configuration selected in the previous round, and defines new configurations in close vicinity. Essentially, the algorithm applies the same grid-based sampling employed by the selection phase but now in the subspace defined around every configuration. Returning to Figure 5, assume that the selection process picked the 3 configurations that center the yellow shaded cells. The first round of the refinement phase applies the grid-based sampling to the subspace defined by each yellow cell, synthesizing k' (from every cell) candidate configurations, with $k' \leq k$. Finally, the algorithm selects the k index configurations from this round with the lower expected cost $C(\cdot, Q)$, which will act as the new anchor points for the next round to sample their subspace. The refinement phase terminates when we observe no cost improvement in the selected synthesized configurations between two rounds, although in practice, we typically set a maximum number of rounds.[13]

Finally, the right part of Figure 5 exemplifies the evaluation of the index configurations. In our current implementation, the $C(\mathcal{I}, Q)$ cost is a function of the query throughput and the space requirements; the figure shows the numbers for the top-3 candidate index configurations picked by the selection phase.

Complexity analysis. Each template $\mathcal{T}$ undergoes the selection phase and ρ rounds of refinement, all performing a grid search at a granularity g over the d-dimensional variable space. So, the per template cost is $O((g^d + \rho k g^d) \cdot c_f)$, where k is the number of selected candidate configurations and c_f is the cost of computing the $C(\cdot, Q)$ of a candidate. Overall, with τ templates, Algorithm 1 runs in $O(\tau(g^d + \rho k g^d) \cdot c_f)$. With this cost scaling linearly to τ, k and ρ, considering additional templates, candidates, or refinement rounds incurs only proportional overheads. Also, although the grid search incurs the exponential g^d cost, in practice we expect a low d value; in our tests $d \leq 5$ (see Figure 10). Yet, for high-dimensional spaces, grid search can be replaced by (quasi-)random sampling

[13] For our experiments we set this number to 3.

under a fixed budget, decoupling the candidates count from d; we list gradient-guided exploration in Section 8 as future work.

Remarks. We follow up on our discussion in Section 3.4. After a large number of updates, the quality of the index may deteriorate, affecting the query performance. A common practice in this case is to drop and re-synthesize the index, accommodating the updated characteristics of the data objects. We handle workload shifts analogously. The key is to monitor the workload features over time and apply a statistical divergence test (e.g., a two-sample Kolmogorov–Smirnov test) against the reference distribution during synthesis. Upon a significant deviation, the index is re-synthesized. Last, when no workload Q is yet available (*cold start*), we capitalize on [56]'s findings building the on-average best structure irHINT, until a representative workload is formed from accumulated query statistics.

5 LEARNED COST MODEL

Calculating the $C(\mathcal{I}, Q)$ cost for an index configuration $\mathcal{I}$ on the available query workload Q is critical to our synthesis process (see Lines 6 and 13 in Algorithm 1). Although we need to evaluate a sample out of all possible configurations, constructing these indices and executing the queries to calculate their $C(\mathcal{I}, Q)$ is prohibitively expensive. Instead, we devise a novel *Learned Cost Model* (LCM) able to approximate $C(\mathcal{I}, Q)$ without physically testing the index, allowing c_f and Algorithm 1's overall cost, to be independent of the input size $|O|$.

5.1 Model

LCM solves a supervised regression problem, predicting continuous-valued evaluation metrics $\mathcal{P}$ from statistics describing the object collection O and the query workload Q. Specifically, it defines a f_{lcm} mapping that predicts $\mathcal{P}$ for individual time-travel IR queries $q \in Q$ processed by an index configuration $\mathcal{I}$:

$$f_{\text{lcm}} : \langle O_{\text{stats}}, q_{\text{stats}}, \mathcal{I} \rangle \rightarrow \mathcal{P}$$

where O_{stats} represents object collection metrics and q_{stats} captures q's individual characteristics (Section 5.2). In our implementation, $\mathcal{P}$ includes query throughput and index size, but the model can be extended to support other evaluation metrics (e.g., index construction time, maintenance costs).

We realize f_{lcm} as a neural network parameterized by weights and biases θ. On training, we optimize θ to minimize the discrepancy between predicted and observed evaluation metrics.[14] The training dataset consists of samples collected from actual query executions:

$$\{(\mathcal{X}^{(i)}, \mathcal{P}^{(i)}) \mid i = 1, \ldots, n\}$$

where $\mathcal{X}^{(i)} = \langle O^{(i)}_{\text{stats}}, q^{(i)}_{\text{stats}}, \mathcal{I}^{(i)} \rangle$ represents the input features for the i-th training example. The training objective is to find optimal parameters $\theta^* = \arg\min_\theta \mathcal{L}_{\text{total}}(\theta)$, where the total loss is the average of per-sample losses:[15]

$$\mathcal{L}_{\text{total}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{\text{sample}}(f_{\text{lcm},\theta}(\mathcal{X}^{(i)}), \mathcal{P}^{(i)})$$

The per-sample loss $\mathcal{L}_{\text{sample}}(\mathbf{p}, \hat{\mathbf{p}})$ measures the prediction error for individual training examples. We employ a weighted mean squared error over the m evaluation metrics:

$$\mathcal{L}_{\text{sample}}(\mathbf{p}, \hat{\mathbf{p}}) = \frac{1}{m} \sum_{j=1}^{m} w_j \left(p_j - \hat{p}_j\right)^2$$

where $\mathbf{p}$, $\hat{\mathbf{p}}$ are the true, predicted values, and vector $\mathbf{w}$ has metric-specific weights for certain evaluation measures during training.[16]

5.2 Statistics

We utilize a variety of metrics to statistically characterize an object collection O and a query workload Q which reflect the key cost primitives identified in [56]. Collection metrics capture the input scale. Specifically, we consider its cardinality $O_{\text{card}} = |O|$ as the number of contained objects, the extent of its time domain $O_{\text{dom}} = O.t_{end} - O.t_{start} = \max_{o \in O} o.t_{end} - \min_{o \in O} o.t_{start}$, the size of the global dictionary as $O_{\text{dict}} = |\mathcal{D}|$, and the collective length of object descriptions as $O_{\text{post}} = \sum_{o \in O} |o.\psi|$.

The element frequency distribution in IR objects typically follows a Zipfian law, where few elements appear frequently while most are rare, however, our model is able to accommodate any distribution. For this purpose, we encode the empirical rank-frequency profile at logarithmically spaced ranks: for each sample position s, we record the element's frequency O^s_{freq}. Together with quantile-based summaries, these features capture the observed distribution shapes efficiently. Last, O's temporal distribution captures the relationship between time extent and object complexity. We sort objects by their time span, i.e., $o.t_{end} - o.t_{start}$ and partition them into quantile-based bins of approximately equal size. For each bin j, we record its range $[O^j_{\text{span-min}}, O^j_{\text{span-max}}]$, average element count $O^j_{\text{elem-avg}}$, and standard deviation $O^j_{\text{elem-dev}}$ (calculated using log-normal modeling to account for multiplicative variability). This captures how object complexity varies with temporal scope.[17]

For a query q in the workload Q, q_{stats} includes statistical metrics that capture q's selectivity, predicate structure, and access pattern. Specifically, We record the normalized $q_{\text{start-rel}} = q.t_{start}/O_{\text{dom}}$ and $q_{\text{end-rel}} = q.t_{end}/O_{\text{dom}}$ positions relative to the collection's time domain, as well as the logarithmic extent $q_{\text{extent-rel-log}} = \log((q.t_{end} - q.t_{start})/O_{\text{dom}})$; the element count $q_{\text{elem-cnt}} = |q.\psi|$. Finally, to characterize the distribution of element ranks within each query, we compute the logarithmic rank of every element (relative to the dictionary size) and record the top-k ranks as $q^1_{\text{rank-log}}, \ldots, q^k_{\text{rank-log}}$. This compactly encodes the diversity and selectivity of query predicates, allowing LCM to distinguish between queries targeting rare versus common elements, and between narrow versus broad temporal ranges. Intuitively, among the selected workload metrics,

[14] The definition of f_{lcm} is orthogonal to the model implementation. In practice, we use two neural networks, one for the throughput and the other for index size. The latter does not take q as input since the index size is query-independent. Single-metric prediction (throughput or index size) may be preferable to multi-metric prediction, as it simplifies training and improves accuracy.

[15] In the probabilistic formulation, this corresponds to minimizing the negative log-likelihood $\mathcal{L}_{\text{total}}(\theta) = -\sum_{i=1}^{n} \log \Pr(\mathcal{P}^{(i)} \mid \mathcal{X}^{(i)}; \theta)$, quantifying how well the model explains observed evaluation metrics.

[16] E.g., we penalize the misprediction of $\mathcal{P}_{\text{throughput}}$ higher than $\mathcal{P}_{\text{size}}$.

[17] We assume that element frequencies in $o.\psi$ do not vary per extent bin. We also do not model temporal skewness of the indexed objects.

rank-frequency and query-rank features expose list-scanning and selectivity; query extent and positions capture temporal pruning; and query element count captures list-intersection cost.

To encode an index configuration $\mathcal{I}$, we employ one-hot encoding for categorical features. Specifically, we one-hot encode the partitioning type (none, dictionary-based, time-based) and represent partition boundaries as normalized continuous values. We further one-hot encode the indexing structure and algorithm choices (e.g., irHINT, *t*IF+HINT, base *t*IF), and represent their tunable parameters as continuous features. This encoding scheme allows the neural network to learn interactions between design choices and their impact on performance across varying workload characteristics.

5.3 Implementation

We implement LCM as a neural network with fully connected layers and ReLU activations, decreasing from input dimension $dim(\mathcal{X})$ to output dimension $dim(\mathcal{P})$; features with exponential-scale variation are log-transformed; detailed hyper-parameters in Section 6.

For robust generalization, we generate diverse synthetic training data spanning varying cardinalities, domain sizes, selectivities, and distribution patterns. Our tests in Section 6 show that training on synthetic data enables effective prediction for real collections, even without explicit training on similar real data. Also, with this design choice, neither updates in the input features nor shifts in query patterns affect our cost model and so, no re-training is necessary.

For parameter updates, we employ AdamW [37, 43], whose direct regularization helps mitigate overfitting by encouraging smaller weights, which aids generalization.[18] A reduce-learning-rate-on-plateau scheduler monitors validation loss and reduces the learning rate η when improvement stalls for a set patience interval. Early stopping triggers after a specified number of non-improving epochs. Since query execution exhibits natural variability due to system load and cache states, the stopping threshold is set to account for this measurement noise, ensuring that the model captures genuine patterns rather than transient fluctuations.

The model scope is adaptable: input or output dimensions can be omitted when constant or irrelevant (e.g., O_{stats} for similar collections, or q_{stats} for size prediction). Mapping f_{lcm} can incorporate hardware-specific features into $\mathcal{X}$; transfer learning then enables adaptation to new platforms with limited target-specific data.

6 EXPERIMENTAL ANALYSIS

We conducted our analysis on a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz with 512 GBs of RAM running AlmaLinux 8.5 (Kernel 4.18.0). We implemented our indexing framework with all temporal IR structures from Section 2.2 in C++[19], compiled using gcc (v13.3.0) with flags -O3 -mavx -march=native.[20] For our Learned Cost Model (training and design space navigation), we used Python 3.12.3, PyTorch 2.8.0, NumPy 2.3.2, pandas 2.3.2 and scikit-learn 1.7.1. All data, i.e., inputs and indices, reside in main memory.

Table 1: Dataset characteristics

	ECLOG [10, 56]	WIKIPEDIA [56]	DIGINETICA [48]
Cardinality	300311	1672662	573935
Size [MBs]	171	4715	768
Time domain [secs]	15807599	126230391	24527463
Min / max interval duration [secs]	1 / 15802098	1 / 126169456	1 / 12780663
Avg interval duration [%]	8.4	5.2	0.25
Dictionary size [# elements]	178478	927283	130987
Min / max description size [# elements]	1 / 14399	1 / 6982	1 / 12097
Avg description size [# elements]	72	367	173
Min / max element frequency	1 / 140423	1 / 1671696	1 / 31896
Avg element frequency [%]	0.04	0.05	0.18

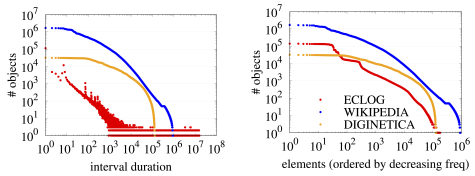

Figure 6: Dataset statistics

6.1 Setup

Datasets. We experimented with 3 real-world datasets (2 provided by [56]); Table 1 summarizes their characteristics and Figure 6 shows the distribution of their interval duration and the frequency distribution of their elements. ECLOG [10, 56] contains e-commerce data on HTTP requests, from Dec 1, 2019 to May 31, 2020. Every session is an object o where $o.t_{start}$, $o.t_{end}$ are the timestamps of the first, last requests in the session; $o.\psi$ stores the requested URIs. WIKIPEDIA [56] contains all versions of 100K random articles from 2020 to 2024. Each object o is a version with its creation timestamp $o.t_{start}$ and the creation timestamp $o.t_{end}$ of the next version; $o.\psi$ stores the terms in each version. DIGINETICA was created from the Personalized E-commerce Search CIKM 2016 Challenge [48]. Every session is an object o with $o.t_{start}$, the timestamp of the first user query issued in the session and $o.t_{end}$, of the last; $o.\psi$ stores all product ids returned as results in the queries.

Workloads. To assess our synthesis process and showcase adaptive temporal IR indexing, we defined 19 types of time-travel IR query workloads that represent different querying patterns and contexts. For this, we control the extent of the query time intervals and their distribution in the time domain (temporal skewness). Further, we control the length of the query description $|q.\psi|$ and the selectivity of the query elements, i.e., how frequently they appear in a dataset.

Table 2 summarizes the features of the tested workload types. The first 8 workloads stem from the default query setting *BASE* in [56], i.e., time-travel IR queries with $|q.\psi| = 3$ elements, where each element is sampled independently from the element frequency distribution of O (i.e., $\Pr[e] \propto \text{freq}(e, O)$), and with a time interval covering 0.1% of the time domain, uniformly distributed inside it. Each new workload is defined by deviating from *BASE* on one query feature. We have *LOW*, *HIGH* when varying element frequency, *SHORT*, *LONG* when varying the temporal query extent

[18] Let $g^{(t)} = \nabla_\theta \mathcal{L}^{(t)}$ denote the gradient of the loss at iteration t. AdamW computes exponentially weighted moment estimates of the gradients: the first moment $m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)}$ and the second moment $v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)(g^{(t)})^2$, where the square is taken elementwise. Bias-corrected estimates are computed as $\hat{m}^{(t)} = m^{(t)}/(1-\beta_1^t)$ and $\hat{v}^{(t)} = v^{(t)}/(1-\beta_2^t)$. AdamW decouples weight decay from gradient-based optimization by applying it directly to the parameters: $\theta^{(t+1)} = (1-\eta\lambda)\theta^{(t)} - \eta\,\hat{m}^{(t)}/(\sqrt{\hat{v}^{(t)}} + \epsilon)$, where η denotes the learning rate and λ the weight decay coefficient.

[19] Code for tIF variants and irHINT from https://github.com/chrauch/irhint.

[20] Our source code is available in https://github.com/chrauch/synthdex.

Table 2: Tested workloads; parameters deviating from *BASE* marked in bold; ratios for mixed workloads in parentheses

workload		$\|q.\psi\|$	elem. freq. [%]	temp. ext. [%dom.]	skew [%dom.]
BASE		3	0–100	0.1	0–100
MANY		**4–9**	0–100	0.1	0–100
FEW		**1–2**	0–100	0.1	0–100
LOW		3	**0–0.1**	0.1	0–100
HIGH		3	**0.1–100**	0.1	0–100
SHORT		3	0–100	**0–0.1**	0–100
LONG		3	0–100	**0.1–10**	0–100
RND		**1–9**	**varied**	**varied**	0–100
MANY + HIGH + SHORT	(M+H+S)	**4–9**	**0.1–10**	**0–0.1**	0–100
MANY + HIGH + LONG	(M+H+L)	**4–9**	**0.1–10**	**0.1–10**	0–100
MANY + LOW + SHORT	(M+L+S)	**4–9**	**0–0.1**	**0–0.1**	0–100
MANY + LOW + LONG	(M+L+L)	**4–9**	**0–0.1**	**0.1–10**	0–100
FEW + HIGH + SHORT	(F+H+S)	**1–2**	**0.1–10**	**0–0.1**	0–100
FEW + HIGH + LONG	(F+H+L)	**1–2**	**0.1–10**	**0.1–10**	0–100
FEW + LOW + SHORT	(F+L+S)	**1–2**	**0–0.1**	**0–0.1**	0–100
FEW + LOW + LONG	(F+L+L)	**1–2**	**0–0.1**	**0.1–10**	0–100
SELECTIVE (2:1)		**1**	**0–0.1**	**0.1–100**	0–100
&BROAD	(S&B)	**1–3**	0–100	**0–0.1**	0–100
SELECTIVE-WIDE (20:1)		3	**0–0.01**	**1–10**	0–100
&BROAD-NARROW	(SW&BN)	**1–2**	**1–100**	**0–0.01**	0–100
HOT (20:1)		3	**0–0.01**	**5–10**	**70–100**
&COLD	(H&C)	3	**1–100**	**0–0.01**	**0–80**

and *FEW*, *MANY* when varying $|q.\psi|$. The *RND* workload samples all parameters uniformly across logarithmically spaced ranges, ensuring balanced coverage across multiple orders of magnitude. The next 8 workloads systematically combine meaningful selections of the *LOW*/*HIGH*, *SHORT*/*LONG*, and *FEW*/*MANY* traits to provide better coverage of the workload space. Every trait contributes its key feature to the combined workload; e.g., *FEW + LOW + SHORT* inherits $|\psi| \in [1, 2]$ from *FEW*, 0–0.1% element frequency from *LOW* and 0–0.1% extent of from query interval *SHORT*.

We also defined 3 deliberately contrastive *mixed* workloads that combine opposing selectivity profiles, creating conditions in which a partitioned index ensemble is particularly beneficial. *SELECTIVE&BROAD* (*S&B*) pairs element-selective queries (rare elements, wide temporal extent) with element-broad queries (any frequency, narrow extent); *SELECTIVE-WIDE&BROAD-NARROW* (*SW&BN*) sharpens this contrast by combining very rare elements with wide extents against common elements with narrow extents. *HOT&COLD* (*H&C*) introduces temporal skew: *HOT* queries concentrate on the recent portion of the domain (skew 70–100%) with rare elements and wide extents, while *COLD* queries address historical data (skew 0–80%) with common elements and narrow extents. Last, in Section 6.4 we test even more mixed workloads.

Queries. We generated 10K queries per workload; 10% used as the representative workload for the synthesis process to construct the adaptive index, and 90%, to assess the performance of the synthesized indices against the competitors in Section 2.2. For the latter, we run every query 10 times and took its median execution time.

6.2 LCM design, hyper-parameters and training

We first elaborate on LCM's implementation, justifying at the same time our design decisions and our feature selection in Section 5.2.

Model architecture. To determine the underlying multi-layer perceptron design, we evaluated 5 neural networks with capacities from ~400K to ~6K trainable parameters and the following hidden layers: XL = 512→512→128→64→32→16, L = 256→256→128→64→32→16, M = 256→128→64→32→16, S = 128→128→32→16, and XS = 64→32→16. Figure 7 shows that ranking quality remains stable around Kendall's $\tau = 0.91$ as capacity decreases and drops only for the XS. We therefore selected S as the smallest architecture without a noticeable loss in ranking quality.[21] Network S trains at least 4 times faster than XL and less than 50% slower than XS.

Table 3: LCM hyper-parameters

hyper-parameter	value
MLP Layers and dimensions	57 (in) → 128 → 64 → 32 → 16 → 2 (out)
Activation function	ReLU
Learning rate	2×10^{-3}
Weight decay	10^{-4}
Loss function	weighted MSE
Gradient clip norm	1.0
Batch size and epochs	64 and 30
Early stop	5
Reduce LR (fac., pat., min.)	0.5, 3, 10^{-6}
Data split (train/val.)	9:1 (dataset-level)

Table 4: Sampling ranges for generating LCM train datasets

	parameter	sampling range
Global	cardinality O_{card}	$[10^{4.0},\ 10^{6.3}]$
	temporal domain O_{dom}	$[10^{6.0},\ 10^{8.2}]$
Per extent bin (J=8 bins in total)	max extent $O^j_{\text{ext-max}}$	$[10^{0},\ 10^{8.2}]$
	avg. elements per bin $O^j_{\text{elem-avg}}$	$[1,\ 10^{2.7}]$
	element std. dev. $O^j_{\text{elem-dev}}$	$[10^{-1},\ 10^{3.1}]$
Per frequency bin (K=12 bins in total)	Rank upper bound O^k_{hi}	$[10^{0.3},\ 10^{6.0}]$

On the grounds of the ranking quality again in Figure 7, we decided to describe input collection O using 8 extent bins and 12 element-frequency bins. This representation yields a total of 57 features for the $(O_{\text{stats}}, q_{\text{stats}}, \mathcal{I})$ input space and two evaluation metrics, throughput and space, as output in $\mathcal{P}$. Overall, Tables 3 and 4 (1st column) list the hyper-parameters for our learned cost model and the parameters for the training datasets, respectively. Apart from the architecture and feature representation, we set the remaining neural-network, optimization, and train/validation-split hyper-parameters in Table 3, to commonly used defaults [55].

Model training. To train our generic LCM, we generated synthetic random datasets. Each dataset O was constructed by sampling its O_{stats} uniformly from ranges spanning real-world datasets with at least a ±10% margin (Tables 1 and 4). We then materialized the objects in O from O_{stats}[22], and generated a query workload Q of 1000 queries and varied characteristics, using the *RND* workload. For each (O, Q) pair, we constructed and measured multiple index configurations $\mathcal{I}$ from our design space (Figure 10), sampling each template according to its dimensionality: 1 configuration per parameter-free template (e.g., base *t*IF), 2 per single-parameter template (e.g., irHINT), 4 per two-parameter template (e.g., *t*IF variants), and 10 per five-parameter template (partitioned ensemble indices).[23] Resulting $(O_{\text{stats}}, q_{\text{stats}}, \mathcal{I}, \mathcal{P})$ formed the training set.

To tune the training corpus size and gauge the achievable accuracy, we incrementally add batches of 50 synthetic collections (with 10 random configurations), re-train, and evaluate on a held-out set

[21] Rigorous k-fold cross-validation could identify the smallest adequate architecture.
[22] To accelerate the generation, we (1) skipped datasets of raw size over 8 GB, (2) imposed a 400 MB limit for 90% of generated datasets, (3) manually excluded configurations deemed too costly or of little value, and (4) subsampled statistics.
[23] Reducing these numbers lowers quality below $\tau = 0.91$ of network S in Figure 7.

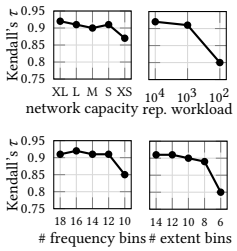

Figure 7: Ranking quality under model variations

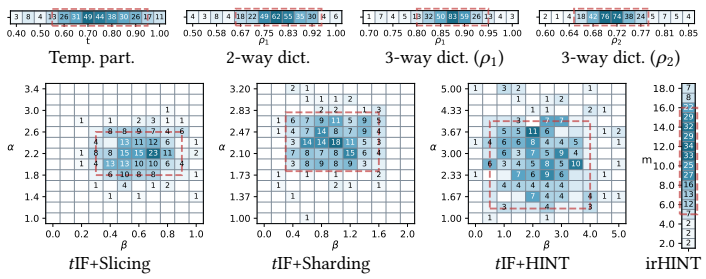

Figure 8: Calibrating template parameters; every heatmap frames the area where the highest concentration (over 90%) of fast configurations (ranked among top-3) was observed

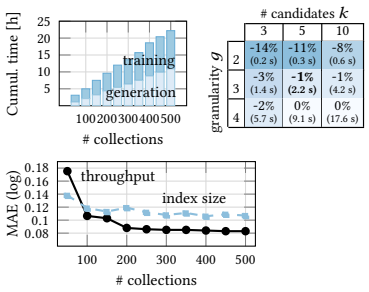

Figure 9: Cumulative training cost per template; Synthesis tuning; LCM generalization and accuracy

of 50 collections (10 configurations, 1000 queries each).[24] Training stops when validation error reaches the measurement noise floor.[25] Figure 9 (top) shows the cumulative cost for generating the training data and actual training per template, while Figure 9 (bottom) reports the mean absolute error MAE $= \frac{1}{n}\sum_{i=1}^{n} |\log \hat{y}_i - \log y_i|$ on log-transformed predictions. As expected, the total cost rises when increasing the corpus size. However, with respect to how accurately the model estimates the index size and query throughput, we observe that both curves drop sharply up to 100 collections and then flatten; from 300 onward the throughput MAE approaches natural fluctuations, yielding diminishing returns. Under this, our corpus has 300 collections producing an accurate, synthesis-ready LCM from scratch in less than 15 hours per template.

Model accuracy and inference efficiency. We further elaborate on LCM's accuracy and efficiency, both critical for our synthesis process. To measure the former, we query 100 synthetic collections under variations of the *RND* workload with $|Q| = 1000$. For each collection, we selected up to $n = 5$ candidates within 50% of the predicted best throughput and at least 8% apart, exceeding the ~6% noise floor. We then tested the candidates and compared their predicted rankings using Kendall's τ. We obtained a mean $\tau = 0.91$ and a top-1 accuracy = 95%, confirming how reliable and accurate LCM ranking is and providing empirical evidence that the selected statistics capture the necessary information to determine promising configurations.[26] Regarding how fast LCM estimates $C(\cdot, Q)$ costs, we tested 3 probe sizes $|Q| \in \{100, 1000, 10000\}$ for which LCM evaluates over 5000, 1800, 200 configurations per sec, respectively.

6.3 Synthesis process and index construction

We next elaborate on the synthesis process, justifying our design choices and studying the index cost of synthesized configurations.

Template definition and calibration. We created a catalog of 15 templates after extensively experimenting with the temporal IR indices in Section 2.2; Figure 10 lists our catalog. The first 5 templates model single-structure indices, allowing the synthesis process to potentially recommend a traditional index. Every structure in Section 2.2 has its own template; variables are included for

```
{"structure":"tif;base"}
{"structure":"tif;slicing;?α[1.8,2.6];?β[0.3,0.9]"}
{"structure":"tif;sharding;?α[1.8,2.8];?β[0.3,1.6]"}
{"structure":"tif;hint;merge-sort;?α[1.3,4.0];?β[0.5,4.0]"}
{"structure":"irhint;performance;?m[5,16]"}

{"partitioning":"time-based", "mode":"split", "terminals": [
  {"range":"?t[0.55,0.95], 1", "structure":"tif;base"},
  {"range":"0, ?t", "structure":"irhint;performance;?m[5,16]"} ] }
{"partitioning":"time-based", "mode":"split", "terminals": [
  {"range":"?t[0.55,0.95], 1", "structure":"tif;slicing;?α[1.8,2.6];?β[0.3,0.9]"},
  {"range":"0, ?t", "structure":"irhint;performance;?m[5,16]"} ] }
{"partitioning":"time-based", "mode":"split", "terminals": [
  {"range":"?t[0.55,0.95], 1", "structure":"tif;hint;merge;?α[1.3,4.0];?β[0.5,4.0]"},
  {"range":"0, ?t", "structure":"irhint;performance;?m[5,16]"} ] }
{"partitioning":"time-based", "mode":"split", "terminals": [
  {"range":"?t[0.55,0.95], 1", "structure":"tif;irhint;performance;?m[5,16]"},
  {"range":"0, ?t", "structure":"tif;base"} ] }
{"partitioning":"time-based", "mode":"split", "terminals": [
  {"range":"?t[0.55,0.95], 1", "structure":"tif;irhint;performance;?m[5,16]"},
  {"range":"0, ?t", "structure":"tif;slicing;?α[1.8-2.6];?β[0.3-0.9]"} ] }
{"partitioning":"time-based", "mode":"split", "terminals": [
  {"range":"?t[0.55,0.95], 1", "structure":"tif;irhint;performance;?m[5,16]"},
  {"range":"0, ?t", "structure":"tif;hint;?α[1.3-4.0];?β[0.5-4.0]"} ] }

{"partitioning":"dictionary-based", "mode":"refine", "terminals": [
  {"range":"?ρ1[0.80,0.95], 1", "structure":"tif;base"},
  {"range":"?ρ2[0.65,0.78], ?ρ1", "structure":"tif;slicing;?α[1.8,2.6];?β[0.3,0.9]"},
  {"range":"0, ?ρ2", "structure":"irhint;performance;?m[5,16]"} ] }
{"partitioning":"dictionary-based", "mode":"refine", "terminals": [
  {"range":"?ρ1[0.80,0.95], 1", "structure":"tif;base"},
  {"range":"?ρ2[0.65,0.78], ?ρ1", "structure":"tif;hint;merge;?α[1.3,4.0];?β[0.5,4.0]"},
  {"range":"0, ?ρ2", "structure":"irhint;performance;?m[5,16]"} ] }
{"partitioning":"dictionary-based", "mode":"refine", "terminals": [
  {"range":"?ρ1[0.65,0.95], 1", "structure":"tif;slicing;?α[1.8,2.6];?β[0.3,0.9]"},
  {"range":"0, ?ρ2", "structure":"irhint;performance;?m[5,16]"} ] }
{"partitioning":"dictionary-based", "mode":"refine", "terminals": [
  {"range":"?ρ1[0.65,0.95], 1", "structure":"tif;hint;merge;?α[1.3,4.0];?β[0.5,4.0]"},
  {"range":"0, ?ρ1", "structure":"irhint;performance;?m[5,16]"} ] }
```

Figure 10: Configuration templates catalog

adaptive parameter tuning (Section 3.3) with the exception of *t*IF where we have a concrete configuration. The next 6 templates are for the time-based partitioning scheme. Our empirical analysis and testing showed that utilizing a single time point t to distinguish between recent and historical objects is a good splitting strategy. To determine a value range for $?t$, we conducted the calibration study in Figure 8 (top row) over 100 random synthetic collections, each with 10 random configurations and *RND* + *H&C* workload variations. Under this, we set $?t$ range as $[0.55, 0.95]$, assuming a normalized $[0, 1]$ time domain for the input collection. The last 4

[24] Each round reuses earlier data, up to 400K training and 100K validation ($O_{\text{stats}}, q_{\text{stats}}, \mathcal{I}, \mathcal{P}$) tuples; these thresholds roughly balance data-generation and training time.
[25] We measured the noise floor (natural throughput fluctuation) at ~6% using 50 synthetic collections, 10 configurations per collection, 5 runs with shuffled queries.
[26] Accuracy tests on our real datasets confirmed these findings.

templates are for dictionary-based partitioning, modeling 2 alternative strategies: (1) the *common-infrequent-rare* split we discussed in Section 3.1, which results into 3 partitions, and (2) a *common-rare* split, which results into 2 partitions. The 3-way split derives naturally from our building blocks which are either time-first or IR-first (see prior findings in [56]). With common elements, time-first processing (i.e., irHINT) narrows down the candidates faster. Infrequent elements benefit from IR-first processing (i.e., *t*IF variants with base *t*IF being the best choice for rare elements where no further temporal sub-partitioning pays off). The 2-way split is an alternative, coarser access pattern inspired by [57]. Rank boundaries are modeled as variables whose value ranges are again determined by the calibration study in Figure 8 (top row), which evaluates *RND* variations over 100 random synthetic collections with 10 random configurations per collection.

In both partitioning schemes, the templates consider different choices for the structures in the terminal nodes. For their tuning (apart from base *t*IF), we used the calibration study in Figure 8 (bottom row) to determine the best value ranges of $?\alpha$, $?\beta$ for each *t*IF variant and $?m$ for irHINT.

Synthesis calibration. We tested combinations of grid-search sampling granularity $g \in \{2, 3, 4\}$ and the number of selected candidates $k \in \{3, 5, 10\}$ using a 5-dimensional template.[27] For 100 random synthetic collections with 10 *RND* variations each, Figure 9 reports the worst throughput degradation relative to the top reference configuration, and the per-template search time. We observe that synthesis quality is sensitive to the grid granularity g, dropping significantly at $g = 2$ but is relatively insensitive to k once $g \geq 3$. Under this, we set $g = 3$ and $k = 10$ which achieves 99% of the reference performance in ~2 secs per 5-dimensional template (at $|Q| = 1000$), providing an effective operating configuration.

Indexing costs. We now report the costs of synthesized indices. Figure 11 compares the build time and the index size of the single-structure solutions in Section 2.2 against the synthesized index in each workload; we call the latter SynthDex, denoted by a "Sy" prefix. As expected, SynthDex numbers vary between workloads proving that a different index is indeed built in each case, with different structures and partitioning; although in some cases, the synthesis process may also determine a single-index configuration as the best. Overall, we observed that building an adaptive ensemble index does not incur a significant overhead. SynthDex's build times are comparable to irHINT's which was shown in [56] typically the fastest index for time-travel IR queries. The synthesis process itself incurs a fixed overhead of less than 40 secs for the entire template catalog which is significantly fasterthan manually tuning a *t*IF variant or irHINT. The index size also varies according to the workload; SynthDex is on average 12% smaller than irHINT on ECLOG, 11% smaller on WIKIPEDIA, and 10% larger on DIGINETICA. These space savings can be achieved alongside performance gains because SynthDex often selects a simpler *t*IF variant when the *LOW* trait is dominant, avoiding temporal over-partitioning or unnecessary ensemble components. We discuss the query-processing gains next.

[27] We set the number of refinement rounds $\rho = 3$; we saw no improvement with larger values. The 5-dimensional template is the most complex and challenging in our catalog.

Table 5: Query performance for mixed workloads

	M+H+S	*M+H+L*	*M+L+S*	*M+L+L*	*F+H+S*	*F+H+L*	*F+L+S*	*F+L+L*
M+H+S		68%	17%	11%	25%	69%	46%	50%
M+H+L	65%		41%	52%	58%	109%	84%	83%
M+L+S	0%	31%		0%	22%	42%	4%	4%
M+L+L	0%	22%	0%		8%	50%	0%	1%
F+H+S	25%	57%	0%	0%		66%	43%	38%
F+H+L	73%	116%	38%	33%	63%		58%	49%
F+L+S	38%	74%	6%	3%	37%	61%		0%
F+L+L	42%	71%	3%	1%	42%	52%	0%	

6.4 Comparison on query processing

Figure 12 reports the query throughput of SynthDex against the Section 2.2 single-structure competitors, across all workloads and datasets. Indices *t*IF+Slicing, *t*IF+Sharding, *t*IF+HINT and irHINT were tuned to *BASE* following [56]. As discussed for Figure 1, irHINT outperforms competition in several workloads, but there is still no single-structure solution dominating across *all* workloads. irHINT leads on ECLOG, WIKIPEDIA for medium-to-high element frequency workloads but the IR-first solutions prevail for very selective elements (*LOW*). On the temporally skewed DIGINETICA, *t*IF+Slicing often competes with or outperforms irHINT. In contrast, we observe that our synthesized adaptive indices consistently match or often outperform the single-index competitors, always coping with the special characteristics of every workload. Our synthesis frequently selects ensemble indices, especially when the workload combines diverse selectivity profiles that no single-structure index can simultaneously address. Even when a single structure is the fastest, our adaptive parameter tuning ensures that the recommended index configuration is well-suited to the workload.

For non-mixed workloads on ECLOG and WIKIPEDIA, SynthDex matches or slightly exceeds the best single-structure solution, correctly identifying the most suitable structure. On DIGINETICA, SynthDex already delivers substantial gains (e.g., +42% on *BASE*, +75% on *HIGH*, +50% on *FEW*), as DIGINETICA's skewed temporal distribution (over 98% of objects in the first 60% of the domain) makes ensemble indexing with time-based partitioning worthwhile even under uniform query characteristics. We observe similar trends also for the 8 workloads defined as combinations of the *LOW*/*HIGH*, *SHORT*/*LONG*, and *FEW*/*MANY* ones. On *RND*, the synthesized configuration reaches up to 3.7× the best baseline on WIKIPEDIA, confirming robustness even under unpredictable query mixes.

The most pronounced gains for SynthDex though arise on mixed workloads. On *S&B*, dictionary-based partitioning routes rare elements to an IR-first structure and frequent elements to irHINT, yielding +20% on ECLOG, 2.2× on WIKIPEDIA and +40% on DIGINETICA over the best single-structure index. The advantage grows on *SW&BN*: 2.3× on ECLOG, 2.1× on WIKIPEDIA, 2.5× on DIGINETICA. On *H&C*, time-based partitioning enables independent optimization of recent and historical data, improving throughput by 55% on ECLOG, 76% on WIKIPEDIA and 76% on DIGINETICA.

To further assess mixed workloads, we tested pairwise mixtures of the eight workloads defined from the *LOW*/*HIGH*, *SHORT*/*LONG*, and *FEW*/*MANY* traits.[28] Table 5 reports the average performance advantage of SynthDex over the best single-structure competitor, across all datasets. The lower triangle contains regular mixes, and

[28] Mix ratios were chosen to roughly balance the processing time of the constituents.

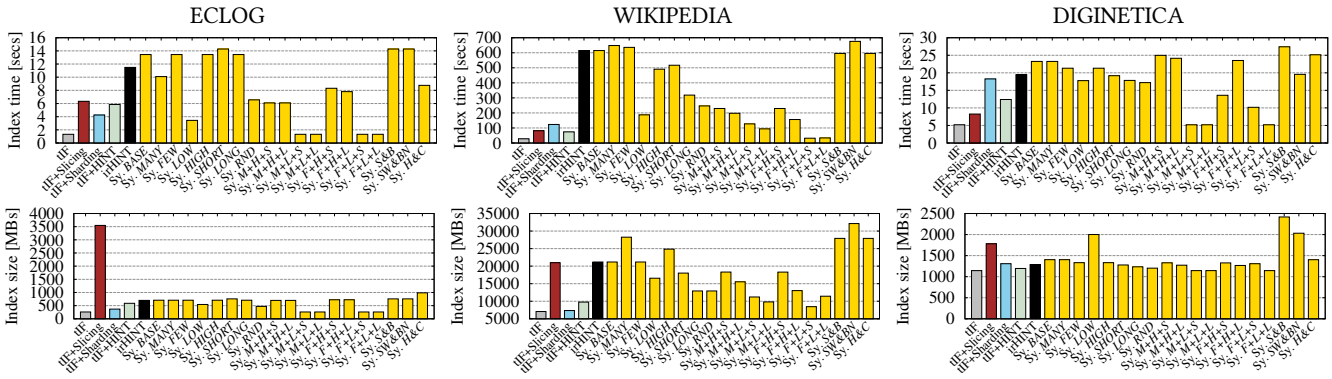

Figure 11: Comparison on indexing costs; workload-agnostic single-structure competitors represented by a single bar; "Sy" prefix denotes the synthesized index per workload

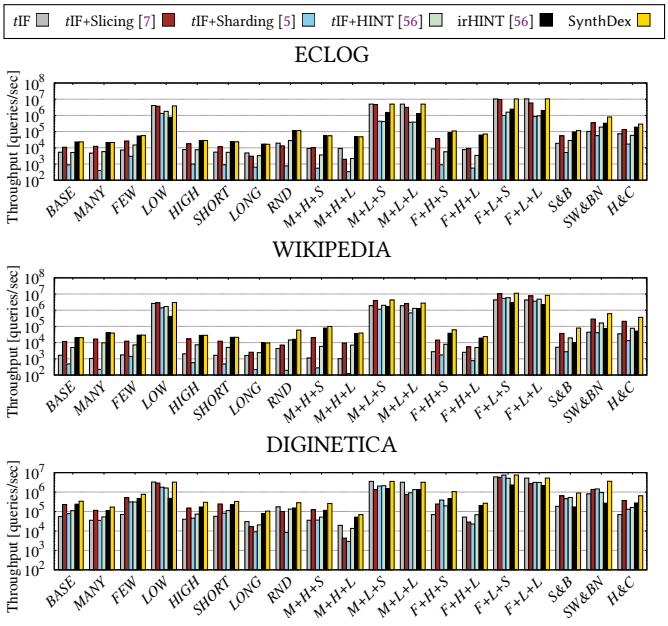

Figure 12: Comparison on query performance

the upper triangle, temporally skewed mixes. The largest gains occur for mixes with common terms, many elements, and long extents, reaching more than 100%. In contrast, mixes with rare terms generally benefit less and often favor the single-structure competitor; a 0% gain means that a competitor (typically, base *t*IF) outperforms any ensemble. Nevertheless, our synthesis correctly selects that single-structure index from the template catalog. Partitioning for temporally skewed mixes remain beneficial overall, but their gains vary and commonly come with negative time and size changes. Overall, the contrastive mixes, which combine substantially different workload traits, benefit the most from partitioning.

6.5 Index maintenance

Table 6 reports the total times for inserting or deleting a 1%, 3% or 5% batch for each dataset.[29] Typically, the maintenance cost scales linearly or sub-linearly to size of the update batch. Again, the times for SynthDex vary among different workloads since a different ensemble index is maintained. As expected the cheapest index to update is *t*IF due to its simplicity. Nevertheless, the maintenance costs for the synthesized indices are typically on par or slightly higher than irHINT, which was shown in [56] to be the most efficient single-structure solution for querying.

7 RELATED WORK

We last discuss related work in the context of adaptive indexing; previous work on temporal IR indexing was discussed in Section 2.2.

Ensemble indexing. Keogh et al. [36] pioneered an ensemble framework for similarity search in high-dimensional data that dynamically selects the most suitable representation for different data segments, accelerating queries through multiple small, heterogeneous indexes. Aggarwal [1] extended this idea with hierarchical subspace sampling, which adaptively reduces dimensionality based on local point behavior, improving compression and facilitating approximate nearest-neighbor search and selectivity estimation.

Cracking. Database cracking [28] is a form of adaptive indexing, where indices are incrementally built as a side-effect of querying. Subsequent work extended cracking with stochastic refinement [23], merged adaptive indices combining cracking and sorting [30], self-selecting and self-tuning indexes [22], and physical reorganization strategies [29, 54]. A comprehensive survey is provided by Schuhknecht et al. [58]. Recent work extends cracking to multi-dimensional and spatial data [40, 53, 65, 66], with Lampropoulos et al. [41] benchmarking adaptive multidimensional indices.

In contrast to these approaches, our framework synthesizes an ensemble of heterogeneous structures upfront for a given workload, also addressing the composite nature of temporal IR data.

Index selection and design. Automated index selection dates back to early optimization formulations for secondary key selection

[29] At terminal node indices, deletions were handled using tombstones.

Table 6: Maintenance costs; insertion/deletion times

	ECLOG			WIKIPEDIA			DIGINETICA		
	1%	3%	5%	1%	3%	5%	1%	3%	5%
*t*IF	0.01 / 0.05	0.04 / 0.06	0.05 / 0.09	0.37 / 0.75	0.44 / 1.01	0.45 / 1.23	0.07 / 0.15	0.11 / 0.23	0.12 / 0.25
*t*IF+Slicing	1.64 / 0.77	1.85 / 0.90	2.10 / 1.05	8.00 / 3.90	10.50 / 4.80	11.00 / 5.80	0.73 / 1.00	0.90 / 1.20	1.10 / 1.40
*t*IF+Sharding	4.90 / 2.30	6.11 / 4.32	8.92 / 5.88	19.8 / 20.9	21.0 / 25.0	22.5 / 31.5	5.71 / 4.29	8.38 / 5.31	12.69 / 6.13
*t*IF+HINT	0.37 / 0.33	0.65 / 0.55	0.95 / 0.80	3.37 / 2.50	4.50 / 3.62	5.80 / 4.80	0.64 / 0.82	0.80 / 1.00	0.95 / 1.20
irHINT	0.12 / 0.52	0.25 / 0.69	0.26 / 0.71	4.85 / 6.87	5.76 / 8.61	7.81 / 9.59	0.16 / 1.55	0.17 / 1.56	0.24 / 1.57
Sy. BASE	0.20 / 0.73	0.28 / 0.85	0.32 / 1.02	5.70 / 8.47	6.29 / 9.54	7.67 / 10.53	0.24 / 1.98	0.27 / 2.18	0.30 / 2.40
Sy. LOW	0.09 / 0.13	0.12 / 0.15	0.14 / 0.18	2.12 / 2.73	2.32 / 3.08	2.52 / 3.39	0.39 / 0.45	0.44 / 0.50	0.49 / 0.55
Sy. HIGH	0.22 / 0.73	0.30 / 0.85	0.35 / 1.02	4.94 / 7.49	6.40 / 8.44	7.90 / 9.31	0.51 / 2.23	0.57 / 2.45	0.64 / 2.71
Sy. SHORT	0.21 / 0.71	0.29 / 0.83	0.33 / 0.99	5.00 / 8.99	7.17 / 10.13	8.41 / 11.18	0.24 / 1.93	0.27 / 2.12	0.30 / 2.34
Sy. LONG	0.12 / 0.44	0.17 / 0.51	0.19 / 0.62	3.20 / 4.58	3.50 / 5.16	3.80 / 5.70	0.71 / 1.46	0.80 / 1.61	0.89 / 1.77
Sy. FEW	0.19 / 0.74	0.26 / 0.86	0.30 / 1.04	5.10 / 10.6	6.58 / 11.94	8.20 / 13.18	0.54 / 2.12	0.61 / 2.33	0.68 / 2.57
Sy. MANY	0.18 / 0.57	0.25 / 0.66	0.29 / 0.80	4.70 / 7.95	5.14 / 8.96	5.58 / 9.89	0.38 / 1.75	0.43 / 1.93	0.48 / 2.13
Sy. RND	1.24 / 0.63	1.71 / 0.73	1.96 / 0.88	6.30 / 5.19	7.39 / 5.85	7.48 / 6.45	0.31 / 0.84	0.35 / 0.92	0.39 / 1.02
Sy. S&B	0.21 / 0.69	0.29 / 0.80	0.33 / 0.97	5.20 / 8.21	5.69 / 9.25	6.17 / 10.21	0.61 / 2.16	0.69 / 2.38	0.76 / 2.62
Sy. SW&BN	0.20 / 0.70	0.28 / 0.81	0.32 / 0.98	4.90 / 9.45	5.36 / 10.65	5.82 / 11.75	0.36 / 1.95	0.40 / 2.15	0.45 / 2.37
Sy. H&C	0.22 / 0.70	0.30 / 0.81	0.35 / 0.98	5.50 / 8.33	6.42 / 9.38	6.53 / 10.36	0.49 / 2.09	0.55 / 2.30	0.61 / 2.54

[44] and has since evolved through cost-based approaches [9], with recent work leveraging machine learning for predictions. Kossmann et al. evaluate commercial techniques [38], while LINDAS [46, 47] selects optimal index algorithms per column using learned models.

Extensible indexing frameworks such as GiST [25] and XXL [14] provide generic templates for defining custom index structures. Automated index design from primitives has emerged. Idreos et al. introduced the Periodic Table of Data Structures [31] to systematically map the design space and continuum for self-designing key-value stores [27], and data structure alchemy for learning to synthesize structures [32]. The Data Calculator [33, 34] enables interactive synthesis using learned cost models for rapid design space exploration without implementation. GENE [19] uses genetic algorithms to assemble and evolve index structures from building blocks, matching or surpassing conventional designs. MOOSE [42] is a new LSM-structure [52] which enables independent adjustments under a single indexing paradigm, e.g., the number of runs per-level, size ratio, and Bloom filter settings at each LSM-tree level. Last, Anneser et al. [6] recently presented adaptive hybrid indices that dynamically select and migrate encodings at runtime based on query access patterns, enabling simultaneous optimization of space and performance by leveraging fine-grained workload statistics.

Our framework shares vision with some of the above but focuses on synthesizing workload-aware ensemble indices for the hybrid combinatorial-continuous design space in temporal IR, by determining partitioning schemes, combing diverse indexing structures, and their tunable parameters for a given workload.

Learned indexing. Learned indices employ data-driven models to optimize index structure design by learning from data distributions and access patterns. The seminal Recursive Model Index (RMI) [39] showed that learned models can replace traditional index components by mapping keys to records. Related work also explored learning data layouts for analytical workloads, such as Qd-tree, which learns hierarchical data partitionings to improve query performance in big data analytics [64]. Followup works considered both pure learned indexes with integrated error-correction mechanisms (e.g., LIPP [63], PGM-index [21]) and hybrid approaches that incorporate learned models into conventional structures (e.g., ALEX [17]). Recent work proposed hybrid construction methods that better balance query performance and memory consumption [45, 68], cost-based construction that optimizes cache performance [67], and automatic tuning approaches like LITune [61] that use reinforcement learning to adjust hyper-parameters for specific workloads. Learned indexing has been also extended to multi-dimensional data with Flood [49] learning grid layouts over multiple dimensions, Tsunami [18] handling correlated data and skewed workloads, and ML-Index [13] supporting point, range, and nearest-neighbor queries. Comprehensive surveys are available in [2, 3, 62].

While these approaches share our goal of using learned models for index optimization, they use them differently in the index-design process. Our framework synthesizes heterogeneous ensemble indices guided by workload statistics and learned cost models.

8 CONCLUSIONS

We studied adaptive, workload-aware temporal IR indexing. Contrary to traditional indexing, we explored the prospect of index ensembles with multiple (potentially diverse) structures, each for a different part of the input. We formulated the selection of the best ensemble (or index configuration) for an input dataset and a workload of representative queries, as an optimization problem. To deal with the vast design space of all possible configurations, we devised a heuristic approach which synthesizes an approximate solution. A novel Learned Cost Model supports the synthesis process, using data and query statistics to evaluate candidate configurations without building and querying the indices. Our tests on real datasets and diverse query workloads showed that synthesized indices perfectly cope with each workload, either matching or exceeding the performance of the fastest existing temporal IR index.

We plan to extend our work towards several directions. First, we will study relevance-based temporal IR search besides the boolean-based in this paper. Second, to improve our synthesis process, we will additionally consider the building and update costs. Furthermore, we will extend the definition of the configuration templates to include additional variables for the splitting ranges to define partitions, and we will study how these templates can be automatically extracted. We also plan to investigate alternative design-space

exploration strategies, including (quasi-)random sampling, gradient-guided search, Bayesian optimization, and genetic algorithms, to accelerate synthesis. Last, we will investigate the application of our indexing framework to other fields besides temporal IR.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the computing time granted on the supercomputer Mogon at Johannes Gutenberg University Mainz (https://hpc.uni-mainz.de/).

REFERENCES

[1] Charu C. Aggarwal. 2004. An Efficient Subspace Sampling Framework for High-Dimensional Data Reduction, Selectivity Estimation, and Nearest-Neighbor Search. *IEEE Trans. Knowl. Data Eng.* 16, 10 (2004), 1247–1262. https://doi.org/10.1109/TKDE.2004.49

[2] Abdullah Al-Mamun, Jianguo Wang, and Walid G. Aref. 2025. Learned Indexes From the One-dimensional to the Multi-dimensional Spaces: Challenges, Techniques, and Opportunities. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, 788–796. https://doi.org/10.1145/3722212.3725639

[3] Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G. Aref. 2026. A Survey of Learned Indexes for the Multi-dimensional Space. *ACM Comput. Surv.* 58, 4, 96:1–96:37. https://doi.org/10.1145/3768575

[4] 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. https://doi.org/10.1145/1871437.1871528

[5] 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. https://doi.org/10.1145/2009916.2009991

[6] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1626–1639. https://doi.org/10.1145/3514221.3526121

[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. https://doi.org/10.1145/1277741.1277831

[8] 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. https://doi.org/10.1145/2619088

[9] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 146–155. http://www.vldb.org/conf/1997/P146.PDF

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

[11] 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. https://doi.org/10.1145/3514221.3517873

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

[13] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. OpenProceedings.org, 407–410. https://doi.org/10.5441/002/EDBT.2020.44

[14] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. 2001. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 39–48. http://www.vldb.org/conf/2001/P039.pdf

[15] 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. https://doi.org/10.1016/J.IPM.2015.05.002

[16] Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Peter Moser. 2022. Leveraging range joins for the computation of overlap joins. *VLDB J.* 31, 1 (2022), 75–99. https://doi.org/10.1007/S00778-021-00692-3

[17] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 969–984. https://doi.org/10.1145/3318464.3389711

[18] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86. https://doi.org/10.14778/3425879.3425880

[19] Jens Dittrich, Joris Nix, and Christian Schön. 2021. The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures. *Proc. VLDB Endow.* 15, 3 (2021), 527–540. https://doi.org/10.14778/3494124.3494136

[20] 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. https://doi.org/10.1109/ICDE.2000.839452

[21] 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. https://doi.org/10.14778/3389133.3389135

[22] Goetz Graefe and Harumi A. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings (ACM International Conference Proceeding Series, Vol. 426)*. ACM, 371–381. https://doi.org/10.1145/1739041.1739087

[23] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proc. VLDB Endow.* 5, 6 (2012), 502–513. https://doi.org/10.14778/2168651.2168652

[24] 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. https://doi.org/10.1145/1645953.1646008

[25] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*. Morgan Kaufmann, 562–573. http://www.vldb.org/conf/1995/P562.PDF

[26] 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. https://doi.org/10.1007/978-3-642-32597-7_31

[27] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. https://vldb.org/cidrdb/2019/design-continuums-and-the-path-toward-self-designing-key-value-stores-that-know-and-learn.html

[28] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[29] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. ACM, 297–308. https://doi.org/10.1145/1559845.1559878

[30] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proc. VLDB Endow.* 4, 9 (2011), 585–597. https://doi.org/10.14778/2002938.2002944

[31] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75. http://sites.computer.org/debull/A18sept/p64.pdf

[32] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike S. Kester, Niv Dayan, Demi Guo, Minseo Kang, and Yiyou Sun. 2019. Learning Data Structure Alchemy. *IEEE Data Eng. Bull.* 42, 2 (2019), 47–58. http://sites.computer.org/debull/A19june/p47.pdf

[33] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 535–550. https://doi.org/10.1145/3183713.3199671

[34] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Internals of the Data Calculator. *CoRR* abs/1808.02066 (2018). arXiv:1808.02066 http://arxiv.org/abs/1808.02066

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

[36] Eamonn J. Keogh, Selina Chu, and Michael J. Pazzani. 2001. Ensemble-index: a new approach to indexing large databases. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26-29, 2001*. ACM, 117–125. https://doi.org/10.1145/502512.502531

[37] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980

[38] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395. http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf

[39] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[40] Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. 2023. Adaptive Indexing in High-Dimensional Metric Spaces. *Proc. VLDB Endow.* 16, 10 (2023), 2525–2537. https://doi.org/10.14778/3603581.3603592

[41] Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. 2025. Benchmarking Adaptive Multidimensional Indices. *Proc. VLDB Endow.* 18, 11 (2025), 4505–4517. https://doi.org/10.14778/3749646.3749709

[42] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in a Colossal Configuration Space. *Proc. ACM Manag. Data* 2, 3 (2024), 175. https://doi.org/10.1145/3654978

[43] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[44] Vincent Y. Lum and Huei Ling. 1971. An optimization problem on the selection of secondary keys. In *Proceedings of the 26th ACM annual conference, ACM 1971, USA, 1971*. ACM, 349–356. https://doi.org/10.1145/800184.810505

[45] Yongping Luo, Peiquan Jin, Zhaole Chu, Xiaoliang Wang, Yigui Yuan, Zhou Zhang, Yun Luo, Xufei Wu, and Peng Zou. 2024. Morphtree: a polymorphic main-memory learned index for dynamic workloads. *VLDB J.* 33, 4 (2024), 1065–1084. https://doi.org/10.1007/S00778-023-00823-Y

[46] Chaohong Ma, Xiaohui Yu, Yifan Li, Aishan Maoliniyazi, and Xiaofeng Meng. 2024. A Learned Approach to Index Algorithm Selection. In *IEEE International Conference on Data Mining, ICDM 2024, Abu Dhabi, United Arab Emirates, December 9-12, 2024*. IEEE, 311–320. https://doi.org/10.1109/ICDM59182.2024.00038

[47] Chaohong Ma, Xiaohui Yu, Yifan Li, Aishan Maoliniyazi, and Xiaofeng Meng. 2025. LINDAS: a learned approach to index algorithm selection. *Knowl. Inf. Syst.* 67, 12 (2025), 11381–11423. https://doi.org/10.1007/S10115-025-02552-W

[48] Snehasis Mukhopadhyay, ChengXiang Zhai, Elisa Bertino, Fabio Crestani, Javed Mostafa, Jie Tang, Luo Si, Xiaofang Zhou, Yi Chang, Yunyao Li, and Parikshit Sondhi (Eds.). 2016. *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*. ACM. https://doi.org/10.1145/2983323

[49] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD 2020, Portland, OR, USA, June 14-19, 2020*. ACM, 985–1000. https://doi.org/10.1145/3318464.3380579

[50] 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. https://doi.org/10.1007/11408079_72

[51] 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. https://doi.org/10.1109/TIME.2006.12

[52] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/S002360050048

[53] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-Aware Spatial Incremental Index. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. OpenProceedings.org, 325–336. https://doi.org/10.5441/002/EDBT.2018.29

[54] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. 2014. Database cracking: fancy scan, not poor man's sort!. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*. ACM, 4:1–4:8. https://doi.org/10.1145/2619228.2619232

[55] Simon J.D. Prince. 2026. *Understanding Deep Learning*. The MIT Press. http://udlbook.com

[56] Christian Rauch and Panagiotis Bouros. 2025. Fast Indexing for Temporal Information Retrieval. *Proc. ACM Manag. Data* 3, 4 (2025), 206:1–206:26. https://doi.org/10.1145/3725343

[57] João B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg. 2011. Efficient Processing of Top-k Spatial Keyword Queries. In *Advances in Spatial and Temporal Databases - 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings (Lecture Notes in Computer Science, Vol. 6849)*. Springer, 205–222. https://doi.org/10.1007/978-3-642-22922-0_13

[58] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *Proc. VLDB Endow.* 7, 2 (2013), 97–108. https://doi.org/10.14778/2732228.2732229

[59] Tarique Siddiqui and Wentao Wu. 2023. ML-Powered Index Tuning: An Overview of Recent Progress and Open Challenges. *SIGMOD Rec.* 52, 4 (2023), 19–30. https://doi.org/10.1145/3641832.3641836

[60] 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. https://doi.org/10.1145/1807167.1807228

[61] Taiyi Wang, Liang Liang, Guang Yang, Thomas Heinis, and Eiko Yoneki. 2025. A New Paradigm in Tuning Learned Indexes: A Reinforcement Learning Enhanced Approach. *Proc. ACM Manag. Data* 3, 3 (2025), 120:1–120:26. https://doi.org/10.1145/3725257

[62] Chaichon Wongkham. 2023. *A Benchmark Study on Updatable Learned Indexes*. Ph. D. Dissertation. https://www.proquest.com/dissertations-theses/benchmark-study-on-updatable-learned-indexes/docview/3171079392/se-2 Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2025-04-28.

[63] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288. https://doi.org/10.14778/3457390.3457393

[64] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 193–208. https://doi.org/10.1145/3318464.3389770

[65] Fatemeh Zardbani, Konstantinos Lampropoulos, Nikos Mamoulis, and Panagiotis Karras. 2025. Updating an Adaptive Spatial Index. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 1194–1206. https://doi.org/10.1109/ICDE65448.2025.00094

[66] Fatemeh Zardbani, Nikos Mamoulis, Stratos Idreos, and Panagiotis Karras. 2023. Adaptive Indexing of Objects with Spatial Extent. *Proc. VLDB Endow.* 16, 9 (2023), 2248–2260. https://doi.org/10.14778/3598581.3598596

[67] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (2022), 2679–2691. https://doi.org/10.14778/3551793.3551823

[68] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. 2024. Hyper: A High-Performance and Memory-Efficient Learned Index via Hybrid Construction. *Proc. ACM Manag. Data* 2, 3 (2024), 145. https://doi.org/10.1145/3654948

[69] 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

Received 17 April 2026; revised 19 August 2026; accepted 12 September 2026