FILM: a Fully Learned Index for Larger-than-Memory Databases

Chaohong Ma
Renmin University of China
chaohma@ruc.edu.cn

Xiaohui Yu
York University
xhyu@yorku.ca

Yifan Li
York University
yifanli@eecs.yorku.ca

Xiaofeng Meng*
Renmin University of China
xfmeng@ruc.edu.cn

Aishan Maoliniyazi
Renmin University of China
aishan@ruc.edu.cn

ABSTRACT

As modern applications generate data at an unprecedented speed and often require the querying/analysis of data spanning a large duration, it is crucial to develop indexing techniques that cater to larger-than-memory databases, where data reside on heterogeneous storage devices (such as memory and disk), and support fast data insertion and query processing. In this paper, we propose FILM, a **F**ully learned **I**ndex for **L**arger-than-**M**emory databases. FILM is a learned tree structure that uses simple approximation models to index data spanning different storage devices. Compared with existing techniques for larger-than-memory databases, such as anti-caching, FILM allows for more efficient query processing at significantly lower main-memory overhead. FILM is also designed to effectively address one of the bottlenecks in existing methods for indexing larger-than-memory databases that is caused by data swapping between memory and disk. More specifically, updating the LRU (for Least Recently Used) structure employed by existing methods for cold data identification (determining the data to be evicted to disk when the available memory runs out) often incurs significant delay to query processing. FILM takes a drastically different approach by proposing an adaptive LRU structure and piggybacking its update onto query processing with minimal overhead. We thoroughly study the performance of FILM and its components on a variety of datasets and workloads, and the experimental results demonstrate its superiority in improving query processing performance and reducing index storage overhead (by orders of magnitudes) compared with applicable baselines.

PVLDB Reference Format:
Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. FILM: a Fully Learned Index for Larger-than-Memory Databases. PVLDB, 16(3): 561 - 573, 2022.
doi:10.14778/3570690.3570704

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/chaohcc/film.

* Corresponding author: Xiaofeng Meng.

Proceedings of the VLDB Endowment, Vol. 16, No. 3 ISSN 2150-8097.
doi:10.14778/3570690.3570704

1 INTRODUCTION

With the increasing deployment of big science observation infrastructures, IoT devices, and Web services, there has emerged a new category of applications with the following data and workload characteristics. (1) *large volume and high velocity:* data keep pouring in from high-frequency data collection equipment such as IoT sensors and large area space telescopes; (2) *append-only:* data is produced and stored in an append-only fashion with no modification allowed, to ensure the authenticity of the records; and (3) *diverse workloads:* query workloads display a high degree of diversity and complexity, varying from simple point queries to complex analytical queries involving both newly arrived and historical data.

Data and workloads with the above-mentioned properties can be observed in a variety of scenarios and are of great research importance. For example, in a typical astronomy application, 1-2TB of data are generated in every observation night at the rate of 85MB/s [40, 43]. Likewise, in an industrial setting [15, 24], vast amounts of data are collected from assets and machines equipped with sensors. In these scenarios, the data collected need to be quickly ingested and analyzed to support the downstream tasks. For example, it would be highly desirable for astronomers to detect sky events from astronomical data with low latency. The detection process may involve querying events (such as transient astronomical events and periodic variable stars) with time spans ranging from seconds to years [9, 46] and retrieving not only newly generated data but also data from a distant past regarding a particular event and/or observation object, which jointly make the detection task challenging [30]. As another example, when detecting anomalies from IoT sensor readings, non-periodic failure alerts need to be carefully validated against historical data, which involves running queries on recent data as well as relevant data from the past with similar patterns. Such queries must be processed efficiently to allow for fast and reliable anomaly detection, which will form the basis of high-stake decisions on what actions to take in response (e.g., shutting down the assembly line). Similar scenarios also exist elsewhere, such as the nearline operational analysis task at LinkedIn [11].

Such data and workloads present unique challenges to data management. To support downstream tasks leveraging such data, it is essential to build index structures on the data that can not only handle newly arriving data with high performance, but also support fast query processing for efficient data retrieval and analysis. Given the data and workload requirement in our target setting, such index structures have to be main-memory based. However, since the data volume usually far exceeds the memory capacity, we

have to consider more advanced techniques proposed in the literature that support larger-than-memory databases. Among them, anti-caching [7] is one of the most scalable and robust techniques. Compared with alternative methods, it provides more control over the identification and movement of cold data from memory to disk as the database grows in size. Moreover, anti-caching facilitates fine-grained tracking of data evicted to disk, in that cold data are spilled to disk at the tuple level rather than the page level unlike in techniques based on the use of virtual memory [38].

Nonetheless, existing methods for larger-than-memory databases, exemplified by anti-caching, are insufficient to address challenges arising from our target settings. First, the indexes occupy a large portion of memory, as such indexes have to reside in memory to provide low update and query latency [7, 28, 45]. As indicated in [45], existing indexes may consume up to 55% of the memory in a main-memory database, and this percentage is even higher in our target settings considering the data indexed in a larger-than-memory database is (perhaps orders of magnitude) larger in size. Second, when the available memory runs out, the system needs to free up space for hot (newly arrived or recently accessed) data. Cold data is identified and evicted to disk, typically using the Least Recently Used (LRU) strategy, while tracking data access causes additional CPU overhead at query processing time [7, 28]. Our experiments show that tracking data access accounts for about 35% of the total query time, even with aggressive optimizations applied to reduce its cost (see Section 6) .

In this paper, we propose FILM, a **F**ully learned **I**ndex for **L**arger-than-**M**emory DBMSs, which addresses the challenges in larger-than-memory databases. We take a learned approach, as recent research [8, 14, 22, 27, 42] on learned indexes yields more lightweight index structures than conventional methods, which significantly reduce the storage overhead by using machine learning models to predict the positions of keys. However, different from existing learned indexes which are designed for data storage and retrieval in homogeneous storage settings (i.e., only a single type of storage is involved, either memory [8, 14, 22, 42] or disk [27]), FILM aims to facilitate both low-cost cold data identification and high-performance index lookup for larger-than-memory databases across heterogeneous storage (i.e., involving different storage devices with varying hardware characteristics). In particular, we study the case of data spanning both memory and disk, but the methodology is general enough to be applied across a broader range of heterogeneous storage settings involving byte-addressable and block-addressable devices [28].

FILM utilizes approximate learned models with controllable error guarantees to serve a dual purpose: (1) to fully index data across heterogeneous storage and support fine-grained data access, and (2) to efficiently maintain an integrated LRU data structure and reduce the cost of cold data identification. At a high level, the design of FILM has three key elements. **First of all**, FILM uses simple machine learning models to capture the cumulative distribution function of the underlying data, which are lightweight and reduce the memory consumption by orders of magnitude compared with conventional indexes. **Secondly**, FILM features a unified tree structure that insulates its upper levels from changes in data location caused by data swapping between memory and disk, and thus enjoys a very low maintenance cost in the presence of frequent data swapping. **Finally**, FILM integrates an adaptive LRU module that embeds the access information of data into the learned model to achieve low-cost and fine-grained cold data identification.

In summary, we make the following contributions:

- We propose **FILM**, a fully learned index for larger-than-memory databases. To the best of our knowledge, FILM is the first learned approach specifically proposed for data indexing and cold data identification on heterogeneous storage.
- We develop index update procedures that is able to dynamically and efficiently handle data swapping between memory and disk with reduced I/O overhead.
- We design an adaptive LRU structure based on the learned data patterns that can reflect the access patterns of query workloads as claimed in [17], which provide fine-grained cold data identification while incurring lower CPU overhead..
- We propose algorithms for point and range queries based on FILM to process queries involving one or more types of storage.
- We conduct extensive experiments on several real and synthetic datasets. The results show that FILM can reduce the index size by at least 50× and accelerate query processing by 10× to 100×.

The rest of this paper is outlined as follows. Related work is reviewed in Section 2. We then present an overview of FILM in Section 3. We introduce the details of FILM in Section 4 and the query algorithms in Section 5. Section 6 presents results from the empirical studies, and Section 7 concludes the paper.

2 RELATED WORK

Cold data identification. Cold data identification is a common task in larger-than-memory databases and cache replacement. Generally, it has two methods [28]: 1) online and 2) offline. The former maintains fine-grained access information for all tuples in real-time, typically uses LRU, and the latter logs and analyzes the accessed tuples offline.

Larger-than-memory databases. A widely-used method supporting larger-than-memory datasets is to apply main memory distributed cache [13, 18] atop the disk-based DBMS. However, this method leads to duplicated data between the distributed cache and the buffer pool of DBMS [7]. Main-memory DBMSs usually provide faster query answering than disk-resident databases [28], but their performance is limited by the memory capacity. More advanced approaches are proposed to support main-memory DBMSs with secondary storage such as disks, and use virtual memory (VM) swapping in the operating system (OS) [38] to manage larger-than-memory datasets. However, VM is a "black box" for a DBMS since it does not know whether a page is in memory or disk and has no way to predict when it will encounter a page fault [28].

Anti-caching [7] is a new architecture for main-memory DBMSs. It supplements the main-memory database with disks and separates data into different storage devices. Its key advantage is its ability to make fine-grained eviction such that the data is evicted at tuple-level instead of page-level of VM.When the DBMS moves cold data onto disk, it inserts "tombstone" in memory to keep track of each tuple on disk to handle queries accessing data that do not reside in memory [7, 45], and all indexes are updated according to tombstones. However, the internal structures of anti-caching

consume a larger portion of the total available memory and the LRU-based method for cold-data identification can be very costly due to the maintenance cost of the LRU-chain. Siberia [10] avoids the high overhead of LRU via an offline process. However, it requires more storage to log the accessed tuples, and providing access frequency information may require the analysis of all the log records [28]. In contrast, FILM uses an online LRU-based identification method with low maintenance cost, providing real-time identification and avoiding the offline procedure.

HTAP systems. In Hybrid Transactional and Analytical Processing (HTAP) systems, data owners simultaneously run transactional and analytical workloads over the same data and hardware [6, 19, 20, 23, 25, 34, 37, 44]. Similar to the task studied herein, HTAP systems need to manage hot and cold data efficiently within a single database instance [23]. However, the application scenarios we target have different technical demands which lead to different optimization objectives. While HTAP considers a mixture of transactional and analytical workloads, FILM targets specifically the append-only settings, which allows us to design a dedicated approach that drastically improves system performance and reduces hardware footprint over general-purpose HTAP systems.

Methods for reducing index storage overhead. While many compression methods [3, 14] have emerged to reduce index size, they all involve decompression that causes query latency and the index size still constantly grows with the number of keys to be indexed. Eventually, the index itself is too large to fit in memory, and the index has to be dropped or a partial index built (e.g., only for data in memory). A state-of-the-art method is a hybrid index with dual-stage transformation which builds dynamic and static stages [45]. In experiments, we apply the compaction and structural reduction guidelines of the hybrid index to build baselines.

Learned indexes. Learned indexes [22] provide opportunities to reduce the storage consumption of indexes. However, existing works are designed for homogeneous storage, mostly for in-memory data [8, 14, 22, 39, 42]. [12] can build learned models for data either in memory or on disk (but not both). They cannot effectively handle the larger-than-memory settings where data is stored across heterogeneous storage with frequent data swapping. Besides, they do not support efficient cold data identification. [22] uses Recursive Model Index (RMI) to fit the cumulative distribution function (CDF) of data. There are works following the idea of RMI [8, 12, 14, 27, 39], among which the most related work is PGM-index [12] which incorporates recursive index structures and $pwlfs$ (Piece-Wise Linear Functions). However, it assumes data is stored in a contiguous array, whereas FILM does not have such restrictions.

3 FILM OVERVIEW

We first discuss the challenges of data indexing in larger-than-memory settings, motivating the design of FILM, and then present the high-level design of FILM.

3.1 Basic Idea

FILM utilizes approximation models to capture the distribution of keys and predict their approximate positions (with a specified error bound ε). Since the data in our target settings usually have a large number of keys, it is generally infeasible to train a single model to capture their distribution unless the model is extremely complex as indicated by [12, 14, 22]. We therefore take a divide-and-conquer approach following existing work [22] and partition a range of (sorted) keys into sub-ranges. We then learn a simple approximation model for each sub-range with minimal computational cost. We refer to a sub-range of data together with the approximation model fitted on the sub-range as a ***piece***. The first key and last key in a sub-range are referred to as the start key and end key of the *piece*, respectively.

What distinguishes our proposal from existing approaches that also use learned approximation models for indexing is that (1) we explicitly address issues arising from indexing data across heterogeneous storage, and (2) the proposed index has a built-in structure that performs efficient cold data identification for data swapping between memory and disk with minimal overhead.

To see why the heterogeneous storage presents unique challenges for indexing, let us take a closer look at existing learned indexes, most of which are designed for main memory data and provide only a single type of prediction granularity (byte address in memory). That is, they predict for a given key a range of byte locations (e.g., array elements) where the given key may appear and then examine these bytes to find the key. However, such a design is not directly applicable to our target setting of larger-than-memory databases as the data is spanned over both main memory and disk. Since it is generally infeasible to build a model that is capable of making predictions with different granularities (e.g., byte for main memory and page for disk), the only seemingly sensible alternative is to build two models, one for the portion of data in memory and the other for the portion residing on disk. While the index on the in-memory portion may work well (as in existing work on indexing main-memory data), the index for disk-resident data would not, as predicting a range of disk pages (instead of bytes) to further examination may result in a large number of I/Os for even a moderate ε [27] (e.g., with $\varepsilon = 16$, 32 pages have to be retrieved from disk). Even worse, with the frequent data swapping across heterogeneous storage, both learned models would have to be frequently retrained to guarantee the accuracy of prediction.

Another major challenge lies in cold data identification, which plays a crucial role in larger-than-memory databases. Anti-caching maintains a chain of items using a doubly-linked list sorted in LRU order [7]. However, such an approach is not directly applicable to our setting due to its high maintenance overhead. The complexity of maintaining the LRU chain on data access is $O(n)$ in the worst case, where n denotes the number of items in the LRU chain. While sampling can be applied to skip some access operations to reduce the maintenance overhead of LRU chain, as empirically shown in Section 6.3.5, the cold data identification thus optimized still incurs significant CPU overhead and may lead to sub-optimal data eviction as the LRU chain no longer accurately reflects the data access order, both degrading the overall query performance. Moreover, the LRU chain in anti-caching is a standalone structure that is independent from the index, limiting the opportunity to share operations between index lookup and LRU maintenance for further cost saving.

To overcome the above challenges, FILM provides a unified learned model for data stored across heterogeneous storage and does not require retraining after data swapping. The first key idea in the design of FILM is that we would like it to be "location-oblivious", meaning that the learned model does not have to be aware of

whether a record resides in memory or on disk. This is achieved through the introduction of an in-memory "directory" that maps each key to either a location on disk or a record in memory. For a given key, the learned model only needs to predict its range of locations in the directory in a similar fashion to predicting key locations in existing learned indexes, which can be used to locate the records, no matter whether they are in memory or on disk.

The second key idea in the design of FILM is to reduce the overhead of maintaining the LRU chain by piggybacking the maintenance of the LRU order on the index lookup. This is achieved through judicious maintenance of the LRU information within the learned model. By building the LRU into the learned model itself, LRU update can be performed at the same time as index lookup for query processing and thus incurs minimal overhead.

3.2 FILM Design

Assume that D is a set of N records produced by an incremental process with an attribute k as the key, where the value of k is drawn from a discrete domain and monotonically increasing (e.g., the timestamp attribute of the record). Following the practice in [7, 11], we keep the newly generated or recently accessed data (hot data) in memory, with the underlying assumption that such data are more likely to be accessed in the immediate future. When the available memory runs out, cold data is swapped to disk to make room for hot data.

Fig. 1 depicts the architecture of FILM, which is tree-structured as follows. (1) Each internal (i.e., non-leaf) node contains a *piece* (except for the root). For a given key, the approximation model in *piece* is used to predict the set of *pieces* to be accessed in the next level. (2) The leaf nodes, each containing a *piece*, partition the entire domain of k according to its distribution; the leaf level as a whole also serves as a directory to insulate the non-leaf nodes from changes in the locations of data records due to swapping. To support cold data identification, we design a structure that we call the *adaptive LRU* that consists of a linked list called the *global chain*, which references all leaf nodes to maintain a global access order of leaf *pieces*, as well as a linked list called the *local chain* in each leaf *piece*, which stores the data within the *piece* in the LRU order.

For newly arriving records, we either update existing leaf nodes according to the mapping from the keys to their positions, or create new leaf nodes. Each time a new leaf is created, its parent nodes and ancestor nodes are recursively updated.

To see how FILM supports query processing, let us take the point query (finding the locations of records with a specific k) as an example, but the same idea applies to the range query (finding the records with k values in a specific range) as well. We may process a point query by starting from the root and successively predicting the *piece* in the lower level until the leaf *piece* containing k is reached. We then access the identified leaf *piece* and use the directory at the leaf level to obtain the location of the requested record (be it in memory or on disk). Finally, we retrieve the record and update the adaptive LRU accordingly.

If the amount of available memory runs out, FILM will make evictions to free up the memory space. It first leverages the global chain to locate the least recently used leaf *piece* and then uses the local chain to identify the least recently accessed data record. FILM

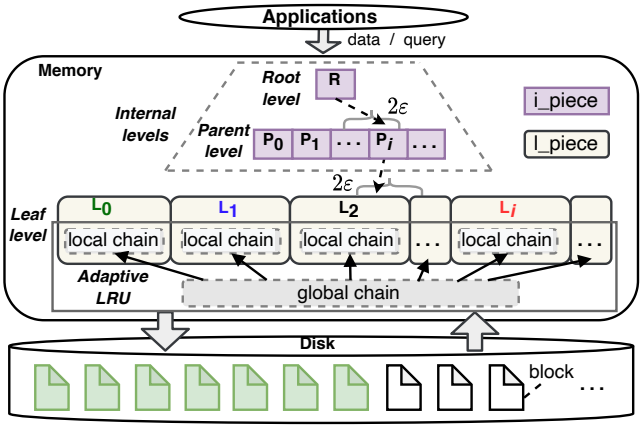


Fig. 1: The architecture of FILM

buffers in memory the evicted records and uses a fixed-size block as the unit of data swapping from memory to disk. A fixed-size block consists of a set of contiguous pages of disk which can be read from or written to disk with a single sweep of the disk arm [32, 35]. As the last step, FILM updates the leaf nodes with the current information on the evicted keys, which keeps track of each record that has been swapped out to disk.

To summarize, FILM is an integrative structure that provides a unified learned model for data stored across heterogeneous storage that avoids retraining after data swapping and maintains a built-in adaptive LRU to reduce the cost of cold data identification.

4 THE FILM INDEX

This section details the major aspects of FILM, including the design of *piece*, the learned model, and the adaptive LRU.

4.1 The Design of *Piece*

As mentioned in Section 3.1, a *piece* contains a sub-range of data and an approximation model fitted on the sub-range. The model predicts for a given key its position in the corresponding sub-range, and the true position $true\_pos$ of the key is guaranteed to be within a specified distance bound (ε) to the predicted position $pred\_pos$, i.e.,

$$|pred\_pos - true\_pos| \leq \varepsilon,$$

which facilitates the efficient locating of a key by focusing the search on a particular portion of the sub-range. A newly inserted key that breaks the constraint will be inserted into a new *piece*, and the new key is called $break\_k$. Intuitively, the value of ϵ is inversely related to the number of sub-ranges for a fixed range of keys. By choosing a proper number of sub-ranges and using a separate approximation model for each sub-range, we can expect the value of ε to be small and thus only a limited number of positions need to be accessed to locate the key. As indicated by previous studies [12, 14, 22], the linear model can effectively approximate the positions of keys while incurring low training and inference cost. Therefore we also build FILM utilizing linear models.

A design choice we have to make when creating sub-ranges from the keys where the range of the keys is constantly expanding due to newly arrived data, is whether consecutive *pieces* should be connected or not. Here, "connected" means that the end key of a

piece doubles as the start key of the next *piece* (i.e., overlapping). On one hand, having connected *pieces* brings extra work in deciding which *piece* to access. For example, assume that 5 is the end key of a *piece* and the start key of the succeeding *piece*. To search for key 5 we may have to access both *pieces*. On the other hand, having disconnected *pieces* results in the "cold start" problem since at least two keys are required to create a new *piece* (to calculate the slope of the linear model). This can be handled by temporarily saving a single key in a buffer until the next key arrives. Since the requested data may be saved in the buffer, the buffer needs to be checked for each search key, causing extra space and search time overhead. In FILM we use connected *pieces* for the internal levels and disconnected *pieces* for the leaf level, denoted by *i_piece* and *l_piece* respectively. The *i_piece* in internal level is to alleviate the complexity of repetitively checking buffer, and the *l_piece* in leaf level can achieve non-overlapping key partitioning at runtime.

The *i_piece* in internal level can be denoted by a triple $\{startK, sl, ipt\}$, where *startK* is the start key that represents the beginning of the sub-range (which is also the end key of the preceding *piece*), and *sl* and *ipt* are the slope and intercept of the corresponding linear model fitted on this sub-range respectively. The *l_piece* in leaf level can be represented by a quadruple $\{startK, endK, sl, ipt\}$, where *startK* is a *break_k* rather than the end key of the previous *piece*, and *endK* denotes the last point in the sub-range. We further discuss the detailed design of *l_piece* in Section 4.2.2

Methods from computational geometry [33] can help determine the optimal number of *pieces* used to partition the key range, which are used in PGM-index [12] for index construction. In this work we adopt the same idea and use a list of *pwlfs* (piece-wise linear functions) to partition the range of keys at each level.

Considering the unique requirements of data indexing in larger-than-memory databases, FILM has the following differences from those works using *pwlfs* [12, 14, 31]. First, different from the *pwlfs* designed for homogeneous storage that are only used to map keys to (ε-approximate) positions, the *pieces* partitioned by *pwlfs* in FILM can also actively assist the cold data identification which is an important and expensive task for larger-than-memory databases. Secondly, previous works [12, 14] adopt a single type of *pwlfs* designed for identical data layouts. FILM, on the other hand, contains two types of *pwlfs*, *l_pwlfs* and *i_pwlfs*, catering to the requirements of *l_pieces* (disconnected sub-ranges) at leaf level and *i_pieces* (connected sub-ranges) at internal levels respectively.

4.2 Dynamic Learned Model

At a high level, the *i_pieces* and *l_pieces* together capture the distribution of the whole data and constitute the learned model of FILM. Below we detail the construction process of the learned model and discuss how FILM handles out-of-order insertions.

4.2.1 The Incremental Construction of FILM. When a new key arrives, FILM attempts to index it using the last *piece* if the ε error constraint can be satisfied, otherwise a new *l_piece* will be created for the newly arriving key (i.e., *break_k*). A direct problem incurred by the above process is that at least 2 points are required to create a *l_piece* while only one key exists during the initialization of the new *l_piece*. Thus, we design queuebuf for the leaf level to decouple the update of index from data insertion. The queuebuf is used to temporarily store the *break_k* until the next key arrives when the two keys can be utilized to create a new *piece*.

Algorithm 1 Construction of the learned model

Input: $[keys, Poss]$, ε, initial = $True$
Output: learned model
1: **while** initial **do** queuebuf.put(k,pos)
2: **while** queuebuf.full() **do**
3: k_1, p_1 = queuebuf.get(), k_2, p_2 = queue-buf.get()
4: piece = $\{k_1, (k_2), sl, ipt\}$ = INITIALPIECE($k_1, p_1, k_2, p_2, \varepsilon$)
5: initial = $False$
6: update parent level
7: **if** UPDATEPIECE(k, pos) = false **then**
8: initial = $True$, queuebuf.put (k, p)
9: UPDATEPIECE of parent level

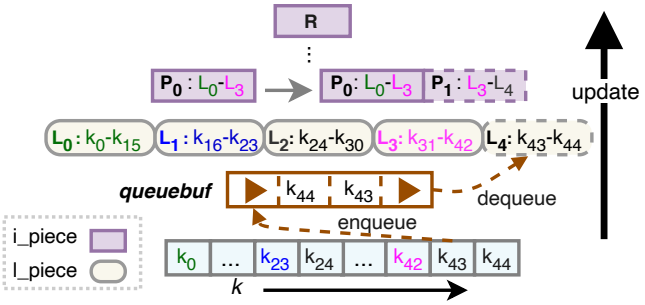


Fig. 2: The learned model with two types of *pieces*

As depicted in Fig. 2, the *l_pieces* of the leaf level are fit on the keys and their positions, while a *i_piece* at an internal level is fit on the *startKs* of its child *pieces* at the lower level. Thus higher levels have less *pieces* than lower levels, and the root R contains only one *piece*.

Algorithm 1 shows how the learned model is constructed recursively. We first put newly arrived keys into queuebuf with $initial = true$ (Line 1). Once the queuebuf is full, two keys are removed from queuebuf to initialize a leaf *piece*, and we set $initial = false$ (Lines 2-4). Then we recursively check whether the parent level needs to be built or updated (Line 5). If the number of *pieces* at the root level becomes 2, FILM will create a new *piece* based on the current level's two *startKs* and use the new *piece* as the new root. The procedure of initializing a *piece* by two points (line 4) is a well-studied computational geometry algorithm [33]. The function of updating a piece (line 7) takes an inserted key, its position and ε as inputs to validate whether the inserted key satisfy the ε constraint, and sets initial = true if the constraint is violated and input this new key into queuebuf until the next key arrives to create a new *piece*.

4.2.2 Leaf Piece of FILM. We detail the design of leaf *piece* and present how it efficiently maps keys to their positions in memory or on disk and helps to achieve low-cost updating of LRU order.

As illustrated in Fig.3, a leaf *piece* in FILM contains five components. 1) A linear function predicting the position of a key (in the corresponding *piece*) (*pred_pos*). 2) The key array recording the keys belonging to this leaf node, which is utilized to get all keys within 2ε of *pred_pos*, from which we identify the key's actual position. While the key array of each leaf *piece* needs to be stored in a contiguous chunk of memory, the key arrays of different leaf nodes do not have to be contiguous in memory. 3) A directory that

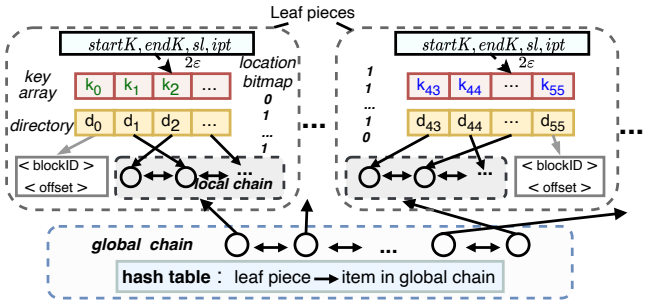


Fig. 3: Leaf *piece* and adaptive LRU of FILM

dynamically tracks the actual storage positions of all keys, which directs a key to either a memory address or a location on disk. 4) The local chain that maintains the LRU order of keys and stores the payloads of keys in memory. 5) A lightweight location bitmap with the same length as the key array that tracks whether the corresponding data record is in memory or not. A bit "1" indicates that the entry in the directory is pointing to an item in the local chain; a bit "0" indicates that the entry is pointing to an element in an array that stores the addresses of all evicted records, where each element takes the form of a pair ⟨blockID, offset⟩.

These five interrelated components are adaptive to data swapping, making the learned model location-oblivious. Also, the local chain within leaf *piece* enables the high-performance learned model to actively participate in the cold data identification process (denoted by adaptive LRU), greatly reducing its overhead. We introduce the details of this adaptive LRU process in the Section 4.3.

4.2.3 Handling Out-of-Order Insertions. While FILM is designed for the settings when data arrives in an append-only fashion, it also naturally supports out-of-order insertions (i.e., the key of the newly inserted record is less than at least one key currently indexed), as discussed below. The update of a record (in attributes other than the key) for a given key can be easily conducted by first locating the record and then modifying its payload.

A seemingly straightforward solution to out-of-order insertions is to create a sort_list for each leaf *pieces* to absorb all the out-of-order keys inserted to the *piece*, and periodically merge the sort_lists into the learned index, as in [14, 39]. However, as will be introduced in Section 4.3, keys in the sort_lists cannot participate in the adaptive cold data identification process with FILM and thus will never be swapped to disk.

Based on the requirement of larger-than-memory databases and FILM, we design sort_piece, which can be viewed as a special type of *l_piece* that directly locates keys using binary search instead of relying on the model prediction. When out-of-order insertion occurs, FILM inserts the keys into sort_piece and ensures that elements in sort_piece remain sorted. In addition, to actively fit in the larger-than-memory settings, the payloads of these keys are also stored in the local chain to track their LRU orders, and the sort_piece is an item in global chain and can participate in cold data identification.

The overall runtime of inserting a new key into sort_piece is the complexity of binary search $O(\log n_s)$, where n_s is the number of keys in sort_piece. In practice, FILM allows the users to set a threshold on the size of sort_piece to prevent it from growing too large (for performance consideration), and once the size of a sort_piece reaches the threshold, we rebuild a new learned model for all the keys in sort_piece as in [22, 36].

4.3 Adaptive LRU

Instead of treating the update of LRU as a separate task from index lookup, FILM consists of local chains and the global chain which supports adaptive LRU, i.e., updating the access order of the requested data when performing index lookups, incurring no extra cost. Next we present the design of local chain and global chain, and discuss how the low-cost cold data identification is conducted.

4.3.1 Local chain. As depicted in Section 4.2.2, a local chain is one of the components in a leaf *piece*, which stores the payloads of the keys belonging to this *piece* in the LRU order. The local chain is implemented as a doubly-linked list supporting fast item removal and insertion. We integrate the local chain into the index by making each entry in the directory reference an item in the chain.

When making evictions using the local chain, the cold records are identified from tail to head[1]. More specifically, the tail contains the payload corresponding to the key that will be spilled to disk. When evicting a key to disk, FILM will locate the position of the key within the corresponding *piece* to update the location bitmap and the tracking information in the directory for fine-grained tracking of the evicted record. With FILM we store a 4-byte offset within the item that points to the corresponding key's position in the location bitmap and the position in the directory.

The key benefit of local chain is that locating the corresponding item in LRU is done simultaneously as the index lookup with FILM. More specifically, when a query arrives, FILM uses the learned model built by Algorithm 1 to locate the key. If the key appears in FILM, the learned model will locate its exact position in the key array of the leaf *piece* containing the key. The located key points to a local chain item that stores the payloads of the key. The payload will be returned as the query result; meanwhile, the corresponding item will be moved to the head of the local chain, and the global chain will also be updated as introduced in the next section.

4.3.2 Global chain. While local chain is efficient to identify the cold records and maintain the LRU order within each leaf *piece* in real-time, it only tracks the local access order of keys in a particular *piece*. Thus, we further design global chain to track the global data access order across *pieces*.

Each item in the global chain points to a leaf piece, and the tail of the global chain denotes the least recently accessed leaf *piece*. Every time an eviction is to be performed, we locate the tail of the global chain, and use the corresponding local chain to identify the least recently accessed key from the *piece*. When a query comes, the global chain is updated by first locating the corresponding item of the access leaf within the chain and then moving the located item to the head of the global chain.

To further improve the LRU efficiency, FILM builds a hash table for the fast location of nodes in the global chain, achieving complexity $O(1)$ on average. Note that the extra space overhead resulting from the hash table is negligible, as the hash table size

[1]The tail in chain is the least recently used item; the head is the most recently used.

(N_l, i.e. the number of leaf *pieces* in leaf level) is expected to be much smaller than the number of records in memory (N_{in}). We empirically evaluate the performance and the storage overhead of adaptive LRU in Section 6.3.5.

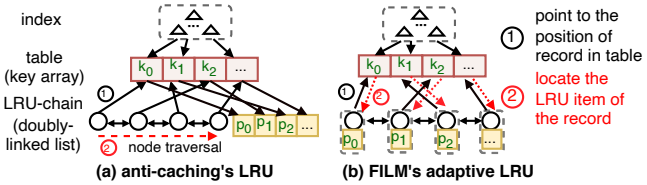
4.3.3 Cold data identification. Algorithm 2 provides the detailed procedure of cold data identification. For a given query, if the requested data can be located in memory (Lines 2-5), FILM updates the local chain and then the global chain, which are completed by moving the corresponding item of local chain or global chain to the head of the chain. If the requested data is not in memory and the available memory runs out, the global chain will identify the leaf *piece* to evict data from, then evict the data associated with the tail of the local chain (Lines 7-11). The process of updating global chain with hash table is presented in Lines 12-18. To reduce the cost of frequent random writes to disk, FILM evicts batches of pages in a single sequential write when the database size (including the memory footprint used for storing data, learned model, LRU, and the tracking information for evicted records) reaches a user-specified threshold, as introduced in Section 3.2.

Ideally, the above-introduced cold data identification could evict the coldest (the least frequently accessed) record of all the leaf *piece* by traversing the global chain from tail to head and moving the coldest record of each local chain to disk until the database size is below the threshold. However, traversing the global chain may break the cache locality. In addition, this strategy makes the records of a certain leaf be scattered across multiple evicted blocks, which results in prohibitively expensive I/O costs when executing a range query containing a series of continuous keys and a large number of blocks need to be retrieved from disk. To overcome these problems, FILM conducts evictions by moving a number of cold records (rather than a single cold record) from the coldest leaf to a block until the block is full, and then a new block will be created to continue the process of eviction. During the above process, we continue traversing the global chain to evict records from the coldest leaf until the database size reaches below the user-defined threshold. Thus the data from a certain leaf will be placed in the same block or adjacent blocks, which greatly reduces the number of blocks retrieved during range queries and ideally lead to sequential reads. We expect such a strategy to be effective in practice, especially when the access to *pieces* is skewed, and the exploration of other strategies would be interesting future work.

There are two major differences between FILM's adaptive LRU and anti-caching's LRU, as illustrated in Fig. 4. First, anti-caching uses node traversal on the doubly-linked list from head to tail to locate an LRU item. However, FILM avoids the time-consuming node traversal by piggybacking the locating of the LRU item onto the high-performance index lookup. Secondly, an LRU item in FILM's adaptive LRU stores the payload of the corresponding record (p_i represents the payload of the record) instead of just indicating the LRU order as done in anti-caching.

5 QUERY PROCESSING AND COST ANALYSIS

In this section, we present algorithms for processing point and range queries based on FILM, and analyze their cost.



Note: In anti-caching, the pointers of LRU chain are embeds in the records' headers. To show the difference when locating LRU items more intuitive, we separate them in this toy example.

Fig. 4: LRU in anti-caching vs. adaptive LRU in FILM

Algorithm 2 Cold data identification

```
Input: queried key k, Adaptive LRU
Output: the cold record, updated adaptive LRU
    access_leaf = lookup k with learned model
 2: if the k is stored in memory then
        access the LRU item of k in local chain
 4:     move the LRU item to the head
        updateInterChain (leaf)
 6: else
        evict_leaf = read the tail of global chain
 8:     evict_record = evict_leaf.tail
        make eviction
10:     if evict_leaf.tail = None then
            remove evict_leaf from global chain
12: function updateGlobalChain(access_leaf))
        if access_leaf is in hash table then
14:         target_item = hash table (access_leaf)
            move the target_item to the head
16:     else
            append access_leaf to the global chain
18:         add access_leaf to hash table
```

5.1 Point Query

We first consider the processing of point queries with FILM, where the query involves searching for the data record with a given key k. Note that for cases when k exists in queuebuf (i.e., it is not indexed by the learned model yet), the data record can be directly retrieved from it with negligible overhead considering that queuebuf only contains at most two keys and always resides in L1 cache. Thus our focus is locating the requested data record using the learned model. Specifically, the procedure consists of three steps.

Step 1: **Find the leaf *piece* that k belongs to.** As described in Section 4, FILM is a learned tree structure, and each level contains multiple *pieces* [33] fitted on the $startK$s of the next level (or keys for leaf *pieces*). Starting from the root of FILM, we recursively predict the candidate *pieces* in the next level containing k until the leaf *piece* containing k has been identified. The query result is empty if no such *piece* can be found.

Step 2: **Locate k in the *piece*.** Let L_i be the leaf *piece* that contains k. We use the linear function in L_i to predict the position of k: $pred\_pos = k \times L_i.sl + L_i.ipt$. The actual position of k is guaranteed to be within $pred\_pos \pm \varepsilon$. We then exhaustively search this range to locate k. Note that FILM is able to determine whether the payload associated with k resides in memory or on disk using the location bitmap introduced in Section 4.2.2. We return an empty result if k cannot be found in this *piece*.

Step 3: **Retrieve the data record and update the adaptive LRU.** If the requested data record resides in memory, then k points to an item in the *local chain*, and the record can be directly accessed using the item. At the same time, the accessed item is moved to the

head of the *local chain*. Otherwise, k must be pointing to a ⟨blockID, offset⟩ pair, which is used to retrieve the requested data record from disk at the given offset in the block indicated by blockID.

For each query, the accessed leaf will be moved to the head of the global chain. Once the available memory runs out, FILM makes eviction using the adaptive LRU.

There are two strategies for determining how much data from the retrieved block to be merged into memory (i.e. put the records back into their corresponding leaf nodes) [7], namely *block-merge*, which merges all records of the retrieved block back into memory, and *request-merge* which only saves the requested records in memory. Considering the larger merge costs of the block-merge strategy to merge all the records from a block, FILM adopts the request-merge strategy. The chosen strategy can also avoid thrashing where some key is fetched into memory and then re-evicted immediately. Once the desired records are merged, the fetched block will be discarded, which means a record may have an in-memory version and a stale version on disk causing "holes" in the block. The "holes" are handled with a lazy merge strategy as done in anti-caching [7]. It tracks the number of "holes" in each block, and checks the number of "holes" when the block is retrieved from disk. If the number of "holes" exceeds a user-defined threshold, we will execute a block-merge operation instead of request-merge.

5.2 Range Query

We next consider range queries of the form $Q_R = (lp, rp)$ requesting a set of data records, with lp and rp being the start and end points of the queried range.

The cases when all the requested data records reside in a single type of storage or across different types of storage can be handled similarly by FILM. To answer a range query Q_R, FILM first determines all the leaf *piece*s overlapping with Q_R. More specifically, it first finds the leaf containing lp (denoted by l_leaf) and the position of lp within l_leaf, then continuously checks the succeeding leaf *piece* until rp is located (since keys in FILM are sorted).

For each leaf *piece* overlapping with Q_R, FILM reads the requested data records contained therein. When retrieving records from disk, the algorithm involves a pre-processing phase during which we determine all the blocks on disk containing the records that Q_R requests, so that the blocks can then be retrieved from disk together to reduce the I/O cost. The adaptive LRU will be updated during the process as is done for point queries.

5.3 Cost Analysis

Assume that the number of leaf *piece*s is N_l, and the "height" (the number of levels) of FILM is H. The cost of finding a leaf *piece* for a given k is thus $O(\log H)$. Note that $H = \log N_l$ in the worst case, and its value is usually much smaller than $\log N_l$, depending on the choice of ε. In the current implementation of FILM, we use linear scan[2] to identify the exact position of the query key in candidates (defined by the predicted position and ε) which is of cost $O(2\varepsilon)$.

For a point query that accesses data in memory, the total time complexity is $O(\log H + 2\varepsilon)$. When the queried data resides on disk, FILM requests data records from disk, and at the same time identifies cold data and makes eviction. The adaptive LRU can locate the cold item in $O(1)$ time, and the total cost of accessing data residing on disk also includes the cost of disk I/O. As the performance breakdown in Section 6.3.1 shows, this total cost is dominated by the I/O of disk accesses. Besides, the I/O cost of eviction is amortized over sequential writes and the maximum number of records contained in an evicted block which consists of a set of contiguous disk pages.

We can perform a similar cost analysis on range queries accessing only data in memory. For range queries requiring disk access, the eviction strategy of FILM as shown in Section 4.3.3 can greatly reduce the number of blocks retrieved from disk and may often lead to sequential reads of multiple blocks. Likewise, the total cost of range query accessing data on disk is also dominated by disk I/O. Note that the actual cost of a range query accessing p blocks in practice may be close to the cost of accessing a single block due to sequential disk page access.

[2]While other search methods such as binary search work as well, we choose to use linear scan since in some cases it is more efficient than binary search [16].

6 EXPERIMENTS

In this section we empirically evaluate the performance of FILM as well as its components using real and synthetic datasets with various workloads. The results demonstrate that FILM is able to significantly accelerate query processing while incurring much lower storage overhead for indexing larger-than-memory databases.

6.1 Baselines

This section introduces the baseline methods for indexing and cold data identification that can be applied or adapted to our target setting. Detailed discussions can be located in Section 2.

Index methods. We compare FILM against B+trees (OriB+tree, CptB+tree, HybridB+tree) and learned indexes (ALEX, FITING-Tree). We adapt these baseline methods to the larger-than-memory setting using the architecture of anti-caching [7], which maintains tombstones in memory for fine-grained tracking of each record on disk; all the indexes are updated according to tombstones. Previous learned indexes designed for homogeneous storage [8, 12, 14] usually take the standard B+tree (**OriB+tree**) as the primary baseline. We further optimize **OriB+tree** by applying hybrid index guidelines [45] to build **HybridB+tree** as introduced in Section 2. The **CptB+tree** is a compact B+tree where the node utilization is set to 100%. We adopt the implementation of B+tree from the standard STX B+Tree [4] library, and set the parameters based on the optimal setting of STX's implementation. For learned indexes, we choose ALEX [8] and FITING-Tree [14]. We use the open-source implementation of ALEX [1], and implement FITING-Tree in C++ on our own since its artifacts are not available.

Cold data identification. To better study the LRU performance of FILM, we compare the adaptive LRU of FILM (**adaLRU**) against 1) traditional LRU chain that a doubly-linked list with sampling rate $a = 0.01$ is used to maintain the LRU chain (i.e., only one out of every one hundred queries updates the LRU chain) as per [7], denoted by **aLRU**, and 2) traditional LRU chain that performs real-time updates of the LRU order for every query (**LRU**) [7]. Comparison of LRU methods is provided in Section 6.3.5, and for other sections we adopt aLRU for OriB+tree, CptB+tree, HybridB+tree, ALEX and FITING-Tree since it incurs less overhead than traditional LRU.

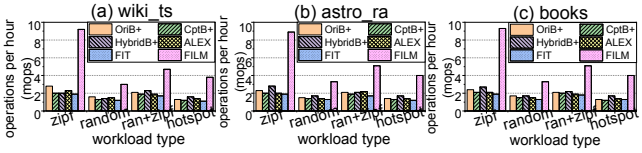


Fig. 5: FILM vs. Baselines: insertion and query performance

Table 1: Breakdown of performance

time (s)	oriB+	CptB+	HyB+	FIT	ALEX	FILM
LRU update	1070.85	1155.99	724.67	1357.42	1230.16	6.13
read disk	338.97	338.67	462.3	317.34	500.48	248.7
write disk	6.05	5.27	4.59	4.7	6.86	1.84
index lookup	3.21	3.17	4.41	2.31	1.04	0.02

6.2 Experiment Setup

FILM is implemented in C++. All experiments are conducted on an Ubuntu machine with 3.6GHz Intel CPU with 256KB L1 cache, 128GB memory (4 × 32GB) and 557GB disk. All experiments are run on a single thread and disk accesses are carried out with direct I/O to avoid OS-level caching.

Besides, we choose synchronous retrieval (SR) [7] of anti-caching as the cold data retrieval policy to avoid the overhead of aborting queries and restarting the long-running execution. SR stalls the execution of a query that requests evicted records until the data is brought into memory.

6.2.1 Datasets. We introduce the 3 real-world datasets and 2 synthetic datasets used in the experiments below:

- *wiki_ts*. The keys are the timestamps of a subset of English wikipedia [29, 41].
- *books*. The key is the sale popularity of each book on Amazon[21].
- *astro_ra*. The keys are the **r**ight **a**scensions of observation stars [40].
- *synthetic*. The keys are randomly generated long int numbers following the Zipfian distribution with a factor of 0.5.
- *YCSB*. The keys are generated based on YCSB [5] using the tool proposed in [45].

Since the datasets consist of keys only, we attach to each key a fixed-size payload to form a record, with the default record size being 128B; we study the influence of record size on FILM in Section 6.5.1. All keys are in 64-bit format. The sizes of the three real datasets are 4GB each, and the sizes of datasets *synthetic* and *YCSB* are up to 128GB. The CDF of each dataset is provided in Fig. 4 in [2].

6.2.2 Workloads. The workloads consist of query keys selected from the existing keys in each dataset according to four types of access patterns, including Zipfian [5, 7], random [12, 42], Zipfian+random [8, 10], and hotspot [26, 39]. Specifically, the Zipfian distribution is controlled by a factor z in the range [0.25,1.5], which simulates skewed workloads where older data are accessed much less frequently than newer data [7], and higher z leads to higher skewness.

To produce a range query, as in [8], we first generate a query key that serves as the left bound, and then choose a random uniform number (upper bound is 100) which indicates the number of keys contained in the range and thus also defines the right bound.

In the following, we first compare the performance and storage overhead of FILM against baselines. Then, we evaluate the sensitivity of FILM to changes in parameters including hardware-related parameters (available memory, block size), and data-and-model-related parameters (record size, different datasets, ε, update ratio). We run three trials per workload and report the average.

6.3 Comparison with Baselines

In the experiments, we configure the amount of memory available to indexing and query processing, Θ (a administrator-tunable parameter), according to the data size such that the ratio of data size to Θ is 2:1 unless otherwise noted. We study its effect on the performance in Section 6.4.1.

6.3.1 Insertion and query performance. We first evaluate the insertion and query performance of all methods. This experiment uses the three real datasets. We first initialize the index with the first $1.5 \times \Theta$ records from the dataset, and then during the measurement phase, the workload interleaves insertions and queries (generated according to different types of workloads as described in Section 6.2.2) with a 1:1 ratio. We run the measurement phase for 3600 seconds and report the total number of operations (a mixture of inserts, point and range queries) completed in this phase in Fig. 5.

As can be observed from Fig. 5, FILM can execute 2× to 5× number of operations compared with baselines, across various datasets and workloads. One of the main reasons is that FILM reduces the query latency by piggybacking the (expensive) LRU update to the high-performance index lookup. For baselines, even though using sampling to update LRU helps to reduce the overhead (for range queries, a sampling rate of 0.001 is used as having more keys within the queried range would result in prohibitive cost for LRU update), it still incurs a high latency to locate the accessed item in the LRU chain each time the chain is updated and may lead to inaccurate cold data identification. Another reason is that FILM improves memory efficiency using the lightweight learned model and the adaptive LRU chain, so that more data could reside in memory, incurring less data transfer between memory and disk.

Another observation from Fig. 5 is that, as the skewness increases (with random being the lowest and Zipf the highest), the superiority of FILM becomes more significant across all datasets. The reason is that skewed queries tend to access data in memory, which leads to fewer data retrieval from disk. Other baselines also benefit from the increase in skewness, but less significantly.

For more detailed analysis of the performance gains, we provide a breakdown of the insertion and query execution times of different methods under Zipf workload in Table. 1, with the same number of operations executed. The total execution time of a workload can be broken down into four parts: retrieving data from disk, evicting records to disk, LRU maintenance, and index lookup. Table. 1 shows that the main factors affecting the performance are disk accesses and LRU maintenance. Clearly, FILM reduces the cost of both disk access and LRU updates on various workload types. Disk accesses becomes the dominant overhead in FILM, since the expensive LRU maintenance is piggybacked to index lookup.

Fig. 7 presents the 95% confidence interval (CI) of latency for wiki_ts with Zipf workload. FILM has the lowest average latency in both insertion and query, with a much narrower CI than other

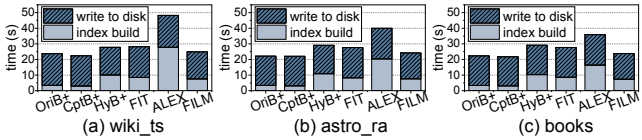


Fig. 6: The comparison of index construction

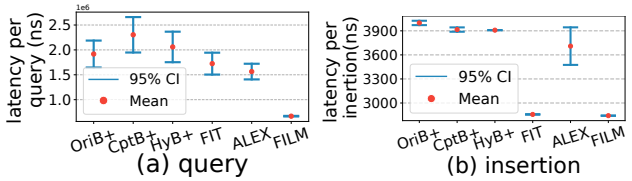


Fig. 7: 95% confidence interval of query and insertion latency

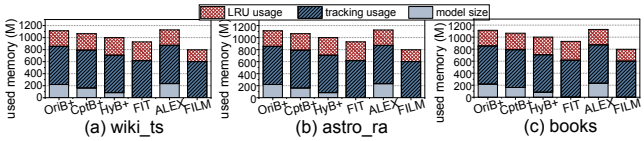


Fig. 8: FILM vs. Baselines: storage overhead

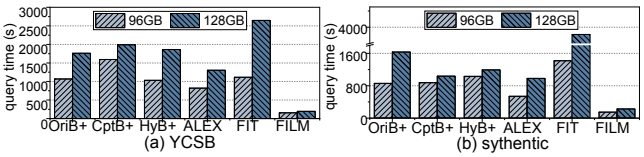


Fig. 9: FILM vs. Baselines: scalability w.r.t data size

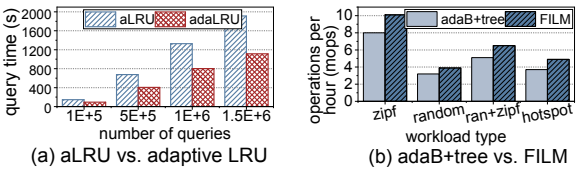


Fig. 10: Evaluating adaptive LRU

baselines, indicating that FILM is more stable and consistent in execution time. Further discussion can be found in the full report [2].

6.3.2 Index construction. The initialization of index described in Section 6.3.1 also serves as an evaluation of the index construction and eviction process. Fig 6 shows the elapsed time of index construction and eviction for three datasets.

The results show that index construction takes longer with learned indexes than with traditional indexes (with HybridB+tree being an exception due to the transformation between the dynamic and static stages). More specifically, the construction time for FILM is on average 2.2-2.5 times that of OriB+tree and CptB+tree, which is a fair trade-off for the great saving in memory usage and reduction in search time. Another observation is that FILM takes 9%-17% less time to evict records than other baselines due to memory savings.

6.3.3 Storage overhead. We next compare the storage overhead of each method based on fixed data size and available memory and the results are shown in Fig. 8. The model size refers to the size of the learned model (which includes the pre-allocated gaps in the leaves) or the the size of the tree for traditional tree indexes. LRU usage refers to the size of the LRU chain (and the hash table in FILM), and tracking usage denotes the amount of memory consumed by saving the information used to access the evicted records on disk including the directory, the location bitmap and the disk addresses.

Fig. 8 shows the storage overhead with data size equals to $4096MB$ under $\Theta = 2048MB$ on three real datasets; other datasets and settings show similar trends. FILM incurs 28%, 25% 20%, 29% and 14% less overhead than OriB+tree, CptB+tree, HybridB+tree, ALEX, and FITING-TREE, respectively, demonstrating that FILM is more storage-efficient and can keep more data in memory for larger-than-memory databases under the same storage capacity.

The reduction of storage overhead of FILM mainly comes from two aspects. First, the index size of FILM is 212×, 157×, 82×, 227×, and 1.9× smaller than that of oriB+tree, CptB+tree, HybridB+tree, ALEX, and FITING-Tree respectively (note: ALEX has a large index size because the leaf node in ALEX is set with a max node size and pre-allocates empty space with gapped arrays.) Second, FILM achieves up to 37% less LRU memory footprint as FILM uses 4-byte pointers in the local chain of adaptive LRU, and it has lower tracking usage since less data needs to be evicted to disk.

6.3.4 Scalability w.r.t data size. We set the available memory to 64GB and increase the data size to up to 128GB to evaluate the scalability of FILM to large datasets. Fig. 9 shows the query time when executing 10^5 random point queries on large datasets *YCSB* and *synthetic*. FILM outperforms baselines across all data sizes, and as the data size increases from 96GB to 128GB, FILM's query time increases at a much slower rate than those for the baselines. This is because, as the data size increases, the length of the LRU chain increases, leading to notable performance reduction in the baselines with aLRU due to the increasing number of item visits on the chain. However, with the adaptive LRU, its maintenance is piggybacked to query processing with minimal overhead.

As the data size increases from 3GB to 128GB, the height of traditional indexes increases by 2-3 on average, while the height of learned indexes does not increase significantly since they can better adapt to the data distribution (with FITING-Tree as an exception, since its internal levels are a B+tree).

6.3.5 LRU overhead. In this experiment, we compare the query processing time with different LRU methods used. We integrate aLRU and LRU into the learned index (the same as FILM) for a fair comparison. The processing time and LRU usage on *books* with a data size of $4096MB$ and Θ=$2048MB$ when executing 10^5 point queries are provided in Table 2. Range queries contain more keys in the queried span that require LRU update and are thus orders of magnitude slower than running range queries over FILM, so we do not include it in this experiment. Fig. 10(a) shows the performance of aLRU and adaLRU when more queries are executed.

As observed from Table 2 and Fig. 10, FILM with adaLRU achieves lower query latency and smaller storage overhead. LRU and aLRU are slower than adaLRU due to the high cost of locating an item in the LRU chain, which is an $O(N_{in})$ operation in the worst case (N_{in} is the number of records residing in memory).

Table 2: LRU comparison

methods	query time (s)			LRU usage (M)
	$z = 0.25$	$z = 0.75$	$z = 1.25$	
LRU	3979.35	4841.99	4306.14	228.218
aLRU	194.50	146.02	89.36	228.218
adaLRU	149.60	97.52	41.06	174.655

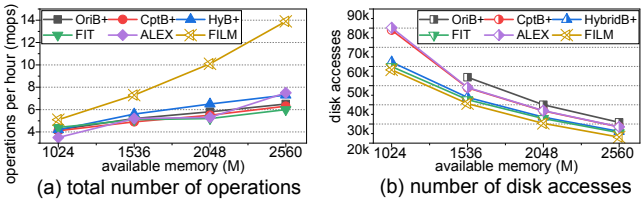


Fig. 11: Varying available memory

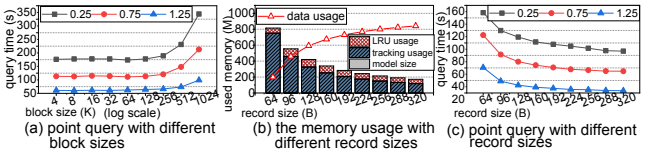


Fig. 12: Effect of block size and record size

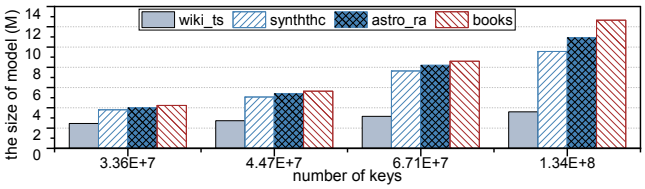


Fig. 13: The size of the learned model on different numbers of keys and different datasets

As shown in Fig. 10(a), the query time of all methods increases when more queries are executed, in this case, however, the superiority of adaLRU over aLRU becomes more pronounced. The reason is that, with more queries, aLRU needs more times to update LRU, and it is not a real-time update and thus may lead to suboptimal eviction. On the contrary, FILM binds the real-time LRU update on the high-performance index lookup, making it efficient in tracking the access information at runtime.

In summary, straightforward adaptation of learned indexes into larger-than-memory databases is problematic. Because simply improving memory efficiency would lead to more data residing in memory, it will result in an increased length of the LRU chain, driving up the traversal cost on the chain when updating the LRU.

To further evaluate the integration of the adaptive LRU and the learned index, Fig. 10(b) shows the comparison of FILM against the integration of adaptive LRU and B+tree (adaB+tree), in terms of the number of operations executed in an hour. As indicated by Fig. 10(b), FILM is still out-performed by adaB+tree across all workloads, indicating that the superiority of FILM comes from both the learned index and the adaptive LRU.

6.4 Study of Environment Parameters

6.4.1 Available memory. This experiment evaluates the performance of FILM and other baselines in terms of the total number of operations executed and the number of disk accesses, varying the amount of available memory Θ. The results of Zipfian point query on *wiki_ts* with data size = 4096MB are shown in Fig. 11. Other datasets and workloads show similar trends.

As depicted in Fig. 11(a), increasing Θ results in a higher number of operations executed in an hour for all methods (since more records could reside in memory), and the superiority of FILM over baselines becomes more pronounced. This can be attributed to the judicious integration of the learned model and adaptive LRU, which saves memory and reduces the overhead of LRU maintenance.

As observed from Fig. 11(b), when executing 10^5 queries, FILM reduces the number of disk accesses by 3% to 38% under various Θ. The results indicate that both memory saving and reduced LRU overhead contribute to the performance gains in the larger-than-memory setting. Note that OriB+tree with 1024MB of available memory cannot support the data of size 4096MB and the corresponding result is not shown.

6.4.2 Block size. We investigate the performance of FILM with different block sizes ranging from 4KB (the standard and minimal page size of disk) to 1024KB. A larger block size means more records in each evicted block. The point query performance of *books* is reported in Fig. 12(a) with data size 4096MB under Θ=2048MB. Similar trends are observed in range query and other datasets.

As observed from Fig. 12(a), when the block size is between 4KB and 128KB, the performance is not sensitive to the change in block size, since the sequential reads and writes of disk do not require the arm to be re-positioned, and the block can be read from or written to disk with a single sweep of disk arm [28]. However, the query time increases rapidly after the block size reaches approximately 128KB, mainly because of the added costs of reading larger blocks, which as indicated by [7], leads to higher overhead in retrieving data from disk. We thus recommend a block size in the range of 4KB-128KB when deploying FILM, and in other experiments we use a block size of 64KB.

6.5 Sensitivity to Data and Model Parameters

6.5.1 Record size. This experiment studies the influence of record size. For the same data size and key distribution, the record size directly affects the number of keys to be indexed, and thus results in different sizes of internal structures. We set data size=2048MB, Θ=1024MB, and vary record sizes from 64B to 320B. The details of FILM's memory usage and the query performance on *wiki_ts* are reported in Fig. 12(b) and (c).

As shown in Fig. 12(b), smaller records lead to higher memory consumption than larger records. The overhead mainly comes from the tracking of evicted records, since each evicted key leaves a "tombstone" in memory and a larger number of keys, thus increasing the cost of fine-grained control of eviction data. The model size overhead is observed to be less than 0.1% of the available memory at any record size, and the tracking information consumes a larger portion of available memory, especially with smaller record sizes.

Fig. 12(c) presents the point query performance, and range query has a similar trend. The query time is longer with smaller record

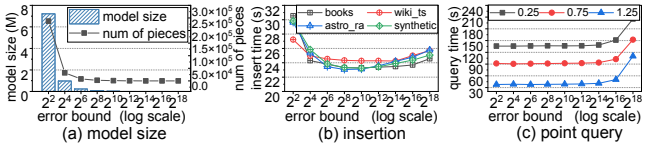


Fig. 14: Performance with ε in larger-than-memory settings

sizes since more evicted data needs to be retrieved from disk. However, for record sizes in a wide range (128B to 320B), the query time is stable, proving that FILM is robust with different record sizes.

6.5.2 Different datasets and number of keys. In this experiment, we report the size of the dynamic learned model on different datasets in Fig. 13. We vary the number of keys indexed by FILM and set Θ =2048MB, ε =8. Regardless of the datasets, the size of the learned model grows as the number of keys increases, but at a fairly slow rate. This is because, with a larger number of keys, FILM needs more *pieces* to partition the keys domain.

Another observation is that the growth in model size is at a different pace for various datasets when the number of keys changes. As depicted in Fig. 13, under the same number of keys, the model sizes of different datasets differ from each other, because the size of the learned model depends on the inherent data distribution (shown in Fig. 4 in [2]). We observe smaller model sizes on datasets with more skewed (i.e., non-uniform) patterns.

6.5.3 Error bound ε. Next, we present the empirical investigations on the influence of ε. The model size, insertion time and query performance with respect to ε are provided in Fig. 14, with ε ranging from 4 to 2^{18}. The data size is set to 4096MB with Θ = 2048MB.

The line chart in Fig. 14(a) displays that the number of *pieces* fitted by FILM has a sharp fall in the range of $4 \leq \varepsilon \leq 16$ and then becomes stable with a minor decrease. The bar chart coincides with the line chart since the model size mainly depends on the number of pieces created. Obviously, the model size of FILM consumes only 1% or even 0.1% of the available memory.

As is clear from Fig. 14(b), the insertion time of all datasets first drops rapidly as ε increases, and then becomes stable at approximately $\varepsilon = 16$. The decrease is because that smaller ε leads to more *pieces* to be created by FILM which increases the insertion time. Particularly, there is a diminishing return as ε further increases, since a large number of records are inserted to a single leaf *piece* due to the loose ε.

Fig. 14(c) shows the point query performance on *wiki_ts*. Similar trends are observed in other datasets and query types. As can be observed, the query time is stable when the error bound is within the range from 4 to 2^{12}. However, after ε becomes larger than 2^{12} the query time increases rapidly. The reason is that, for a small ε the query processing overhead is dominated by the I/O operations of retrieving data from disk. For an overly large ε, a huge candidate set needs to be checked, and thus the dominant cost comes from scanning the candidate set, which grows linearly with ε. The results attest that FILM is robust for ε in the range of 16 to 2^{12}. In our experiments, we set ε to 16 by default.

6.5.4 Handling Out-of-Order Insertions. We evaluate the performance of FILM in handling out-of-order insertions, in which the insertion keys are randomly selected from the original domain with different sampling ratios (i.e., out_of_order_frac). Fig. 15 presents the results on *wiki_ts*; we observe similar trends on other datasets.

Table 3: the overhead of searching the sort_piece

out_of_order_frac	0.1	0.2	0.3	0.4	0.5
index lookup (s)	18.68	16.24	15.08	13.89	14.60
sort_piece overhead (s)	0.19	0.18	0.15	0.14	0.13
ratio	1.0%	1.1%	0.99%	0.97%	0.89%

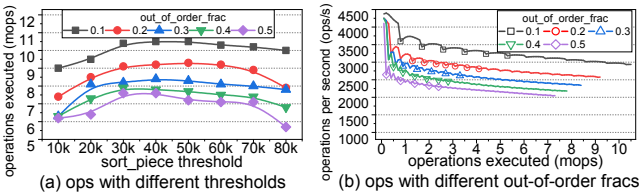


Fig. 15: The performance of out-of-order insertions

FILM allows users to set a threshold for the sort_piece to prevent it from growing too large. As indicated by Fig. 15(a), as the threshold increases, the total number of operations executed in an hour for different out_of_order_fracs first increases, and then becomes stable, followed by a decline. This is because smaller thresholds result in more frequent index reconstructions, whereas larger thresholds lead to large sizes of sort_piece, both of which degrade the performance. We empirically observe that sort_piece thresholds between 30,000 to 70,000 lead to better performance of FILM.

Fig. 15(b) further presents the performance with the threshold set to 40,000. The curves indicate that the performance decreases due to out-of-order insertions, and then bounces back when FILM rebuilds the learned model when reaching the threshold. The downward trend in curves comes from the increasing number of disk accesses as more data is inserted into the system when more operations are executed. Another observation is that the higher the value of out_of_order_frac, the earlier the curve ends, which means fewer operations are executed in an hour, i.e., lower throughput.

Table 3 shows the overhead of searching the sort_piece during the one-hour execution, which is only a tiny fraction (about 1%) of the total index lookup time.

7 CONCLUSIONS AND FUTURE WORK

We have proposed FILM, the first fully learned structure for data indexing and retrieving problems across heterogeneous storage. The experiments demonstrate the ability of FILM in improving the query performance while reducing the storage overhead and achieving fast cold data identification in larger-than-memory databases.

Many interesting problems are worth future investigation following the work studied herein. One possible direction is to extend FILM to multi-core systems and design more sophisticated concurrency control strategies to improve the performance when processing concurrent updates/queries.

ACKNOWLEDGEMENTS

This work was supported by NSFC [No. 62172423, 91846204] and NSERC [RGPIN-2022-04623]. We also thank Beibei Fan, Ning Bao and the anonymous reviewers for their help to improve the paper.

REFERENCES

[1] 2020. *alex*. Retrieved June 18, 2022 from https://github.com/microsoft/ALEX

[2] 2022. *FILM: a Fully Learned Index for Larger-than-Memory Databases*. http://123.57.224.239:8866/s/HGipgKybfgCrjss#pdfviewer

[3] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proc. VLDB Endow.* 7, 14 (2014), 1881–1892.

[4] Timo Bingmann. [n.d.]. . *STX B+ Tree*. Retrieved Augest 28, 2021 from https://panthema.net/2007/stx-btree/

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[6] Oracle database. 2019. *Oracle Database In-Memory*. Retrieved January 27, 2022 from https://www.oracle.com/a/tech/docs/twp-oracle-database-in-memory-19c.pdf

[7] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (2013), 1942–1953.

[8] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 969–984.

[9] Z. Duan, C. Yang, X Meng, Y Du, and C. Wu. 2019. SciDetector: Scientific Event Discovery by Tracking Variable Source Data Streaming. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*.

[10] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942.

[11] Raul Castro Fernandez, Peter R Pietzuch, Jay Kreps, Neha Narkhede, Jun Rao, Joel Koshy, Dong Lin, Chris Riccomini, and Guozhang Wang. 2015. Liquid: Unifying nearline and offline big data integration.. In *CIDR*. Citeseer.

[12] 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.

[13] B. Fitzpatrick. Aug. 2004. Distributed Caching with Memcached. In *Linux Journal*, Vol. 2004. 5–.

[14] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1189–1206.

[15] Franco Giustozzi, Julien Saunier, and Cecilia Zanni-Merk. 2019. Abnormal situations interpretation in industry 4.0 using stream reasoning. *Procedia Computer Science* 159 (2019), 620–629.

[16] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. *CoRR* abs/2006.16411 (2020). arXiv:2006.16411

[17] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. Deepdb: Learn from data, not from queries! *Proceedings of the VLDB Endowment* 13, 7 (2020), 992–1005.

[18] Xiaoli Hu, Chao Li, Huibing Zhang, Hongbo Zhang, and Ya Zhou. 2017. Distributed Caching Based Memory Optimizing Technology for Stream Data of IoV. In *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*. Springer, 15–24.

[19] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[20] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.

[21] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[22] 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*. 489–504.

[23] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*. 311–326.

[24] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. 2011. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference*. Springer, 370–388.

[25] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1598–1609.

[26] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[27] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2119–2133.

[28] Lin Ma, Joy Arulraj, Sam Zhao, Andrew Pavlo, Subramanya R. Dulloor, Michael J. Giardino, Jeff Parkhurst, Jason L. Gardner, Kshitij Doshi, and Stanley Zdonik. 2016. Larger-than-Memory Data Management on Modern Storage Hardware for in-Memory OLTP Database Systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Article 9, 7 pages.

[29] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[30] Daniel Muthukrishna, Gautham Narayan, Kaisey S Mandel, Rahul Biswas, and Renée Hložek. 2019. RAPID: early classification of explosive transients using deep learning. *Publications of the Astronomical Society of the Pacific* 131, 1005 (2019), 118002.

[31] 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 Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 985–1000.

[32] Patrick E O'Neil. 1992. The SB-tree an index-sequential structure for high-performance sequential access. *Acta Informatica* 29, 3 (1992), 241–265.

[33] Joseph O'Rourke. 1981. An on-line algorithm for fitting straight lines between data ranges. *Commun. ACM* 24, 9 (1981), 574–578.

[34] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1771–1775.

[35] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[36] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.

[37] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. 2021. Performance Characterization of HTAP Workloads. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1829–1834.

[38] Radu Stoica and Anastasia Ailamaki. 2013. Enabling efficient OS paging for main-memory OLTP databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. 1–7.

[39] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.

[40] Meng Wan, Chao Wu, Jing Wang, Yulei Qiu, Liping Xin, Sjoerd Mullender, Hannes Mühleisen, Bart Scheers, Ying Zhang, Niels Nes, et al. 2016. Column store for GWAC: a high-cadence, high-density, large-scale astronomical light curve pipeline and distributed shared-nothing database. *Publications of the Astronomical Society of the Pacific* 128, 969 (2016), 114501.

[41] wiki Team. 2021. *Wikipedia: database download*. Retrieved June 28, 2021" from https://en.wikipedia.org/wiki/Wikipedia:Database$_$download

[42] 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.

[43] Chen Yang, Xiaofeng Meng, and Zhihui Du. 2018. Cloud based Real-Time and Low Latency Scientific Event Analysis. In *2018 IEEE International Conference on Big Data (Big Data)*. 498–507.

[44] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

[45] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1567–1581.

[46] Keming Zhang and Joshua S Bloom. 2021. Classification of periodic variable stars with novel cyclic-permutation invariant neural networks. *Monthly Notices of the Royal Astronomical Society* 505, 1 (2021), 515–522.