Adaptive Dynamic Array: Keeping Data in Order from Exploration to Manipulation

ABSTRACT

Ordering data is important for interactively exploring and manipulating data. When accessing data on a tabular interface, users need to order tuples, to allow categorizing, comparing, and focusing on data. Such *order awareness* for tabular data is fundamentally missing in relational databases. This paper studies *ordered access* for relational data. To maintain *rank-to-row* mapping, we propose an index structure, Adaptive Dynamic Array (ADA), addressing three challenges. First, to support updates, which can cause cascading changes, ADA features *indirect indexing* to isolate context dependence. Second, to support collective manipulations like swap and reorder, ADA is *two-layered* to organize pointers by contiguity while keys by uniformity. Third, to support from exploration (query-only) to manipulation (update-mostly), ADA features *adaptive dynamic* so that its growth depends on the amount of changes– and not size of data. We derive the ADA index structure from the requirements, analyze its complexity, and prove its optimality w.r.t. both scale and dynamics of data. Our experiments validated that ADA can efficiently support ordered data access from read-only to write-mostly scenarios with significant performance margin from baselines.

ACM Reference Format:

. 2020. Adaptive Dynamic Array: Keeping Data in Order from Exploration to Manipulation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The abundance and wide spread of data in our digital era have mandated its democratization to end users for interactive processing. Data are no longer only processed by machines via programs; instead, users are participating as "human in the loop" to explore data. Further, data are no longer handled only in a batch mode; instead, users demand interactive processing to manipulate data. User-driven interactive data processing presents new requirements.

Ordering data is important for users to interact with data in a *direct manipulation* manner [1, 2]. When users access a table, they need to order rows for exploration and refer to their ordered positions. Consider a user reviewing graduate applications in a table Applicants (or a small business owner manages customers, a scientist analyzes experiment results, etc.) She *orders* students by country to categorize them. She *scrolls* down and *selects* those "US" applicants, and *reorder* them by major. She *swaps* those "ece" to next to "cs" ones and again *reorders* by gpa to compare. Another user may open Applicants for recruiting teaching assistants and order them by domestic or foreign with good language scores. He may *insert* to or *delete* from the TAs table.

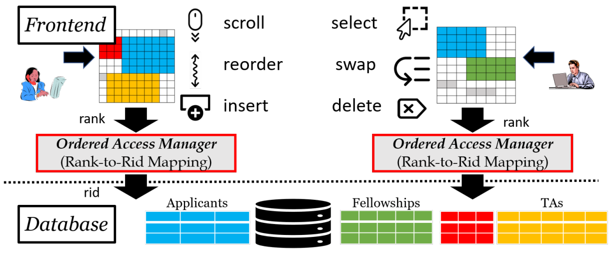


Figure 1: Problem: Supporting ordered access for relations.

Problem. We study *ordered access* for a relation R, as Fig. 1 shows, where R is represented in a tabular interface with tuples arranged in a *linear order* or *permutation* α. Ordered access is familiar to users in common tabular interfaces– e.g., spreadsheets like Excel and Google Sheets– which allows users to order rows. However, a relation is a *set* of tuples and thus inherently order oblivious (and SQL ORDER BY is only for query results). We aim to enable "order awareness" for RDBMS.

Requirements. As this common scenario motivates, for each user u over a relation R, we aim to support a *per-user* order α_u to allow her *ad-hoc* control of a part or the entire order in arbitrary ways, with a *mixed* set of *ordered operations*: Scroll, Select, Reorder, Swap, Insert, and Delete, in a *persistent* manner across sessions.

Solution. We first propose a framework, *Ordered-Access Manager* (OAM), to support order awareness at a *frontend* client, which provides access to, rather than building into, an underlying RDBMS, as Fig. 1 shows, for two reasons: First, since ordering is ad-hoc and per-user, it should be decoupled from the generic storage of tables. Second, for wide applicability, it should be independent of choices of RDBMS (e.g., MySQL). Thus, via OAM, in a tabular interface, users can access any RDBMS and keep data in order.

We further propose *rank-to-rid* mapping to maintain an order α_u for each user u, as the core of the OAM framework, to realize its order awareness. To support the six ordered operations (e.g., select rows 100-106th, insert row at 950917th), we need to map a *rank* in an order, as the lookup *key*, to the row identifier, or *rid*, of a tuple. Ordered access by each user is thus translated to rows in an underlying table, allowing her to see an ordered view. This rank-to-rid indexing is a novel problem.

On the one hand, we can consider our problem as indexing tuples (by ranks). While the database community has extensively studied such indexing, our problem is different. In database indexing, traditional structures (e.g., B-tree) index *attribute* (e.g., gpa, major), an intrinsic value of a tuple, which rank is not, since it depends, contextually, on preceding tuples. If we directly index rank as key, after inserting a tuple, all succeeding tuples must change their ranks. As

Challenge 1, how to avoid such *cascading changes*, an expensive $O(n)$ operation for a table of n tuples?

On the other hand, we can consider our problem as supporting a list. In classic data structure, the *list representation* problem maintains an (ordered) *list* of items (rids) with select, insert, and delete. The lower bound has been proved to be $\Omega(\log n/\log\log n)$ in [3] and an actual structure has realized this bound in [4].

However, ordered access is different from the classic list representation. Ordered access needs to manipulate orders, to reorder a *range* or swap with another, which are *collective manipulations* over a set of tuple rids. List representation only supports point and not range-based queries or updates. Such operations, done tuple by tuple, would involve an $O(n)$ cost. As **Challenge 2**, how to handle costly *collective manipulations*?

Further, for interactive data processing, ordered access must support any ad-hoc mix of *queries* (select, scroll) and *updates* (insert, delete). In a *session*, a user may be *exploring* to analyze data, which is query-only. In another, she may be *manipulating* to change data, which is update-mostly. Supporting rank-to-rid mapping for varying user cases is challenging: E.g., an *array* A, where $A[r]$ returns rank r in $O(1)$ with a cost of $O(n)$ for updates (with cascading changes); in cotrast, a linked list has the opposite trafe-off. Such a *fixed* $O(n)/O(1)$ tradeoff does not adapt: While optimal for query-only, it is unacceptable for write-mostly sessions. Thus, traditional list representation, as a "programmatic" data structure, does not consider varying human-driven workloads. As **Challenge 3**, how to adapt *query-update tradeoff* to different workloads to achieve the optimal cost?

We propose *Adaptive Dynamic Array* or ADA, which addresses the challenges with some novel insights:
• *Insight 1: Indirect Keys.* To decouple cascading changes, we index not lookup keys (ranks) but their "proxies" that are intrinsic to tuples indexed. For rank, the *count* of tuples is such a proxy, since we can use counts to recover ranks.
• *Insight 2: Non-uniform Structure.* To facilitate collective manipulation over a "collection" of rids of varying sizes, we shall allow non-uniform structures. Traditional indexing often maps keys to rids in a regular structure, e.g., B-tree's internal (for keys) and leave nodes (for rids) satisfy the same size and capacity requirements. We will exploit non-uniformity– a two-layer structure– to manipulate rids collectively in a *contiguous* rid layer while looking up keys with log-efficiency in a *balanced* key layer.
• *Insight 3: Change-driven Indexing.* To adapt query-update tradeoff, we develop adaptive indexing that captures "changes." We wish querying to be as efficient as $O(1)$ (like an array) and only slow down "when necessary". With the insight that adaptation means to react to changes, we propose a new principle: *To adapt, index changes but not data.* Upon this principle, our index achieves an adaptive performance with $O(\log c)$ cost (for both queries and updates), where c is the number of changes (updates or swaps)– which is indeed "change-proportional", when c is small compared to n. When c increases to be comparable to or larger than n, the complexity will not grow with c but rather becomes limited to $O(\log n)$. We prove the adaptive complexity (Theorem 1) of ADA and show that it is indeed optimal (Theorem 2). To the best of our knowledge, our work is the first to quantify complexity and prove optimality in terms of changes c (in addition to data size n).

We summarize the contributions as follows.
• We are the first to identify the meaning and requirements of **order awareness** for RDBMS.
• We propose a **general framework** OAM and design a novel **indexing mechanism** ADA for supporting ordered access.
• We **prove the adaptive complexity and optimality** of ADA with respect to both the number of changes and the size of data.
• We perform **experiments** and **ablation study** to validate the performance and design of the techniques.

2 RELATED WORK

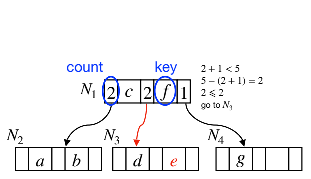
We find that our ordered access problem is related to *list representation* problem but different that we have additional operations and requirements. Unlike list representation problem which focuses only on querying and updating items in a list, we aim to support human operations in human-data interaction for RDBMS, so we need ad-hoc, per-user and persistent ordering with a mixed set of ordered operations.

We propose a framework OAM to provide a frontend client with ordered access to an RDBMS. There are many SQL frontend clients in the industry, such as MySQL Workbench and Oracle Database Workbench, but none of them have order awareness. Users are not able to select or swap tuples based on order.

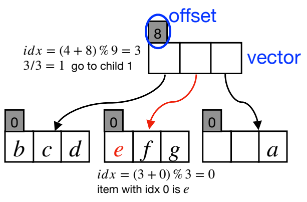
We design an index structure ADA to support rank-to-rid mapping under ordered operations. List representation problem studies a subset of operations we need to support. It has been actively studied in 1980s and 1990s, with an optimal lower bound $O(\log n/\log\log n)$ [3, 4]. The list representation problem is to maintain an ordered list and supports querying the i^{th} item, inserting into the i^{th} position and deleting the i^{th} item. It is closely related to the *partial-sums problem* that shares the same tight lower bound [3], which maintains an array $A[1\cdots n]$ that support update(k,δ): $A[k] \leftarrow A[k]+\delta$ and sum(k): return $\Sigma_{i=1}^{k} A[i]$.

In prior works, a trade-off lower bound, as well as the time complexity lower bound of the partial-sums problem is proved based on cell-probe model where the memory is divided to cells of fixed size, and the running time of an operation is represented by the number of cells accessed in either reads or writes. The cell-probe model was first used to prove the lower bound of partial-sums problem by Fredman and Saks in [3], where they proved a lower bound $O(\log n/(\log\log n + \log b))$ for partial-sums problem in $O(\mathbb{Z}/2\mathbb{Z})$ by chronogram technique. Fredman and Saks's lower bound is achieved by a data structure by Dietz [4], whose time complexity is $O(\log n/\log\log n)$ provided that $\delta = O(\log\log n)$. In [5], a lower bound for all cases is given: $O(1+\log n/\log(b/\delta))$.

To achieve the above optimality, Dietz [4] proposes *WBB-tree* (weight balanced B-tree) to solve list representation problem in amortized optimal logarithmic time. Each internal node of the tree stores two arrays to represent the partial sums of the number of items in every sub-tree, one storing an "old" version of partial sums and the other storing recent updates. To know which sub-tree to go to in a query, we compute partial sums in constant time, so the query time is logarithmic, proportional to the height of the tree. However, the balance condition depends on the total number of items, so it is changing dynamically. When a node violates the balance condition,



(a) counted B-tree.



(b) Tiered Vector.

Figure 2: Query 5th item in sequence *abcdefg* in counted B-tree and tiered vector.

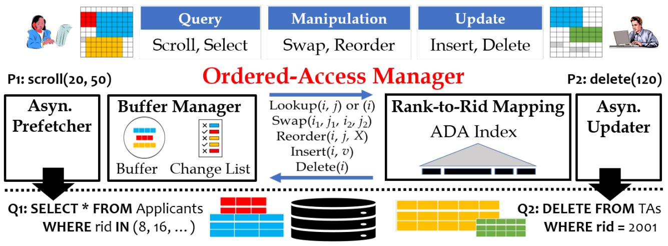


Figure 3: Framework for supporting ordered access.

the whole sub-tree needs linear time to be reconstructed, which is $O(1)$ amortized work per update.

On the one hand, WBB-tree is an amortized optimal data structure, and its time complexity can be linear without amortization. Counted B-tree (also called Order Statistic Tree) [6] improves it to time complexity $\Omega(\log n)$ without amortization, by allowing dynamic local node reconstruction (e.g., splitting and merging). As shown in Fig. 2(a), counted B-tree is a B-tree [7] with additional *counts* showing the numbers of keys in corresponding sub-trees. Lookup by rank is done by comparing the rank with each count, and either returning the key or recursively going to a sub-tree. For example, in Fig. 2(a), to find the 5th item, we compare 5 with counts in node N_1, go to node N_3 and return the 2nd item e.

On the other hand, while the trade-off of WBB-tree is optimal, there could be other ways of trade-off that achieve the same optimality. Tiered vector [8, 9] demonstrates a different possibility of trade-off: keep query time constant, and under this constraint, make update as fast as possible. It achieves $O(1)$ query time and $O(n^{\epsilon})$ update time for $\epsilon > 0$. As shown in Fig. 2(b), a tiered vector maintains several levels of equal-sized vectors and each one is ordered but starts from an offset not necessarily 0. The equal size leads to constant query time. Storing and modifying the start position of each vector makes update faster. However, because of the existence of offset, range query cannot be executed by traversing, and swap or move cannot be executed by grafting substructures, so linear cost is needed for these operations.

The *standard array* (SA) achieves constant query time but needs linear time for changes. To achieve a balance, the *indexed linked list* (LL) [10] uses a linked list to store ever m items and an auxiliary array to index these lists so queries can be faster. However, its update is inefficient, which needs to rebuild the auxiliary array every time. In addition, LL does not support order manipulation.

All the lower bounds achieved in the prior works, either by theoretical proof or actual data structures, only parameterize data set size n, without considering the amount of changes c. We are motivated to handle varying workloads for human-data interaction from query-only ($c \ll n$) to write-mostly scenarios ($c \approx n$). Thus, our solutions are adaptive to the amount of updates. In particular, our modeling of changes to quantify the complexity and prove the optimality of a data structure is unique.

3 OVERALL FRAMEWORK AND DESIGN

To support ordered access to a table, we need a framework to maintain an order α_u for user u and mediate her rank-based access at the frontend to rid-based at the database. This section presents the overall OAM framework and the conceptual design of the ADA index at its heart for maintaining the order.

Objective. Our purpose is to support *ordered access*: to access the tuples in a relation R via their *permutation* α, a bijective *rank-to-rid mapping* from a *rank* to the corresponding rid, i.e., α: $\{1, 2, \ldots, |R|\} \rightarrow R$, so that $\alpha(i)$ returns the rid at rank i.

3.1 OAM: Ordered-Access Framework

We present the overall framework in Fig. 3, which is our solution for the *ordered-access manager* (OAM) in Fig. 1.

How to support ordered access responsively? This is challenging as OAM needs to "mediate" between a user and database. Observing the nature of user-data interaction, we can hide much latency by asynchronous database access, with three key insights:

• **Sequantial Order:** Ordered accesses are predictable with their sequentiality. Viewing a screen of rows, a user triggers an access when she selects from this screenful or scrolls to the next or last ones. We can thus *prefetch* tuples into a *buffer* to hide retrieval time.

• **Buffered Access:** As tuples in use are buffered, as soon as any updates are made in the buffer, we can *return control early* to users without waiting to propagate updates to the database.

• **Viewport Focus:** Each user interacts with data via a viewport (screen) of visible tuples. We can *defer concurrent updates* from other users by pending them in a *change list* and only "merged" with tuples in the buffer when they become visible.

OAM indexes order and buffers prefetched tuples to provide ordered access. As Fig. 3 shows, upon receiving an operation (e.g., P1 or P2), the *Buffer Manager* will handle it by looking up *Rank-to-Rid mapping* in the ADA index, return those tuples from the buffer, and request the *Asynchronous Prefetcher* to prefetch via an SQL query (e.g., Q1) from or the *Asynchronous Updater* to push updates (e.g., Q2) to the database.

Specifically, Buffer Manager stores prefetched tuples and returns them when requested. To handle an operation with some rank i or range $[i, j]$, it will Lookup in ADA to get the rids. For select/scroll, it simply returns these rows from the buffer– since *hits* are guaranteed due to prefetching– merged with any pending updates in the change list. Then, it triggers a further prefetch of the next/last screen of rows. For insert/delete, it updates the buffer, calls ADA to update, requests asynchronous updates to the database, and immediately returns the control to users. For swap/reorder, it only calls ADA to perform the changes to update the indexed order.

Critical to latency hiding, we handle both reads and writes asynchhrounsly. Asynchronous Prefetcher ensures that tuples are always "hit" in the buffer when accessed. The prefetcher is *aynchronous* [11]: When a lookup *hits* (instead of misses) on tuples in the buffer, it will prefetch the next/last screenful of tuples. On

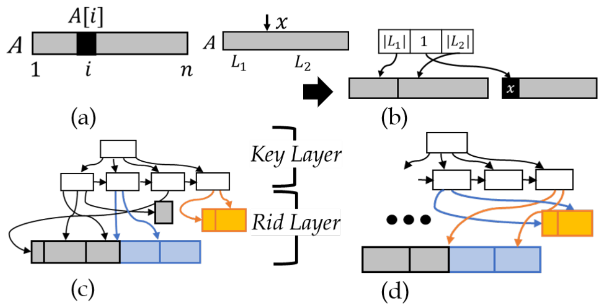


Figure 4: Conceptual design of ADA.

the other hand, Asynchronous Updater propagates updates to the database, so that the changes are visible to other users. On the other hand, it also regularly visits the database to receive updates from other users into the change list.

At the core of OAM, Rank-to-Rid Mapping indexes the user-defined order via the new ADA index. It supports five methods: Lookup to find rids by (point and range) ranks, Insert and Delete to update tuples indexed, and Swap and Reorder to change the order of tuples. We will discuss its conceptual design next.

3.2 ADA: Adaptive Dynamic Array

To support access by an order α, we need an efficient rank-to-rid mapping– as an *index*– which we now design. To highlight the guidelines for concrete implementation, we will identify a set of *design points*, based on three insights as Sec. 1 proposed.

First, following Insight 3 *Change-driven Indexing*, we should index updates, and not tuples, unlike traditional indexing. The index should grow only with changes– Before any update, it should be as efficient as an array, for queries, which is $O(1)$. Thus, as a user starts a session to open a table, the index is ideally an array A of the recorded order, i.e., $A[i] = \alpha(i)$, as Fig. 4a shows. For closure, ending the session, the index should also be stored as an array (on disk). Thus our first design point is:

DP1: *Start with and end as an array.*

How do we change the array to adapt to updates? When she inserts a new rid x at rank i, a standard array would have to first expand the array length, add x to $A[i]$, and shift $A[i+1{:}n]$ by 1– a prohibitive $O(n)$ cost plus memory allocation. However, observing that neither the left side of x, i.e., $L_1 = A[1{:}i\text{-}1]$, nor the right $L_2 = A[i{:}n]$ need to change– the orders within both remains the same. So, we will split the array at i to reuse the sub-arrays and only create a new one for x, as Fig. 4b shows, thus reducing the $O(n)$ cost.

DP2: *Reuse arrays by splitting to decouple subarrays.*

Now that we have multiple arrays, how do we find our way around? We need some additional structure to "index" these subarrays so we can navigate to them. However, such structure will also add overhead. Per Insight 3, to avoid unnecessary overhead, we will reuse the original array as much as possible and only index its changes, i.e., when new subarrays are generated, due to insertion, deletion, reorder, or swap. Thus, we add a node containing keys, or "key nodes", to point to the new subarrays (Fig. 4, part b) which will grow to more nodes to index more changes (part c).

DP3: *Index and only index changes.*

To organize key nodes and rid subarrays, following Insight 2 *Non-uniform Structure*, to support them with different operations,

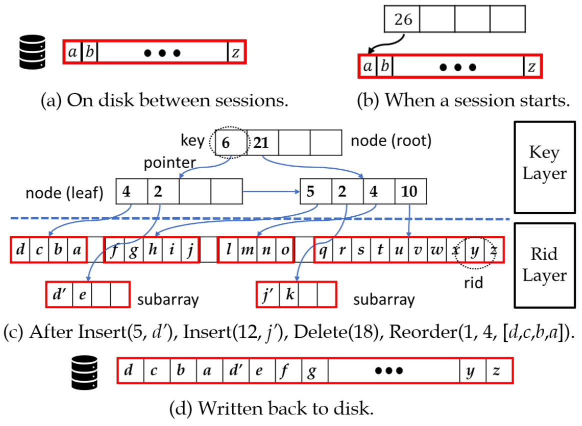

a
b
•••
z
(a) On disk between sessions.
26
a
b
•••
z
(b) When a session starts.
key
6
21
node (root)
pointer
node (leaf)
4
2
5
2
4
10
Key Layer
d c b a f g h i j l m n o q r s t u v w x y z
rid
Rid Layer
d′ e
subarray
j′ k
subarray
(c) After Insert(5, d′), Insert(12, j′), Delete(18), Reorder(1, 4, [d,c,b,a]).
d c b a d′ e f g ••• y z
(d) Written back to disk.

Figure 5: The structures of ADA.

we structure a "key layer" for key nodes on top of the "rid layer" for rid subarrays, as Fig. 4c shows.

DP4: *Organize a two-layered structure.*

We will structure the two layers differently. On the one hand, we organize the index layer like a B+-tree. As we aim to support ordered access– point and range lookups– over sequential ranks, B+-tree is a good choice. We will thus balance the key layer by fixing the size (some M keys) and capacity (at least $\frac{1}{2}M$) for each node, which we call a *key node*. On the other hand, to reuse subarrays as much as possible, we should allow them to be "irregular"– for preserving their contiguity.

DP5: *Balance key nodes but allow variably-sized array nodes.*

However, we cannot simply use B+-tree as is for the key layer: While we are indexing ranks, per Insight 1 *Indirect Keys*, we will use counts, i.e., lengths of subarrays as keys (e.g., $|L_1|$, 1, and $|L_2|$, Fig. 4b), to avoid cascading changes.

DP6: *Use lengths of subarrays as keys.*

Also unlike a B+-tree, we need to support collective swap and reorder of ranks. Per Insight 2 (Two-Layered Structures), since each whole subarray is linked from a key node, as Fig. 4c to d shows, we can swap rank ranges (orange and blue subarrays) simply by "grafting" their key nodes (same colored) between branches.

DP7: *Support grafting of whole subarrays.*

4 CONCRETE REALIZATION

We now develop the concrete realization of ADA based on the conceptual design identified in Sec. 3. We will first present the index structure and then define the basic internal operations that underlie all functions to use the index.

4.1 Index Structure

ADA stores the desired order α of a table of n tuples for a user, as Fig. 5 shows for rids $\{a, \cdots, z\}$. Between sessions, ADA is stored on disk compactly as an array A (sequential file) of rids, $|A| = n$, as Fig. 5a shows, taking the minimum space (**DP1**). At the start of a session, ADA is brought into memory, initially just like an array (**DP1**), with one additional node in Fig. 5b. (We will use the initial size $n_0 = |A|$ for our complexity analysis; Sec. 6.1.) After some changes, ADA grows to index them (**DP3**), as in Fig. 5c.

We next introduce the general structure of ADA (Fig. 5c). The index is essentially an array of rids "indexed" by a B^+-tree of keys–

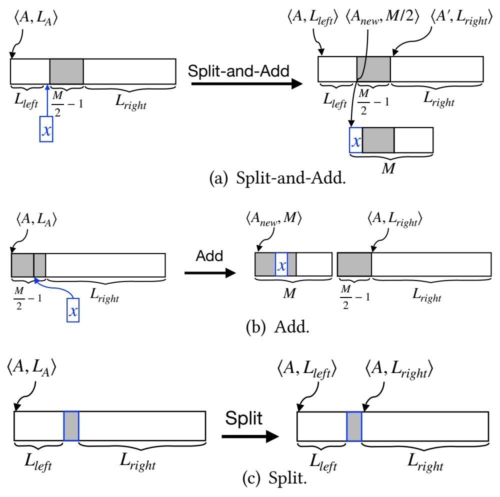


Figure 6: Basic operations.

thus, in two layers: an rid layer for storing rids in subarrays and a key layer for storing keys and pointers (**DP4**).

The *rid layer* is a set of subarrays which form a logical array of rids. ADA starts with one array A, which may evolve to more subarrays when necessary due to changes (**DP3**). A *subarray* S is part of the overall array A which stores the desired order α. In some cases, S can be logically split from A (**DP2**) with a size no less than M (a parameter) to avoid over fragmentation and allow maximum reusing of A. Else, S can be a physically allocated with a new space of size M. Thus, each subarray S can have a different allocated size, or *capacity* (**DP5**): $M \leq |S| \leq n$. A subarray S can contain one rid (when S is newly formed) or more, i.e., its *length* $L_S \in [1, |S|]$.

The *key layer* is self-balanced similar to a B$^+$-tree (**DP5**): each node has a *capacity* M (e.g., $M = 4$) and must be at least half-full (e.g., 2 or more children). The leaf nodes are chained with pointers from left to right. However, unlike B$^+$-tree, each node in ADA has an equal number of keys and pointers, i.e., one key for each pointer. A *key* records the total lengths of the subarrays in the subtree that the *pointer* leads to (**DP6**). E.g., key 6 points to two subarrays of lengths 4 and 2 respectively.

4.2 Basic Operations

We next present the basic internal operations of ADA, which will be used to implement the external functions. Since the key layer is a B$^+$-tree, it has the same operations such as merging and splitting nodes. We will focus on the new operations, for the rid layer to manipulate subarrays.

Session Operations. As just introduced, ADA is stored on disk (Fig. 5a) before a session. To start a session, the open() operation reads the n rids in order into an array A. This first subarray of the rid layer is indexed by a key $|A| = n$ and pointer to A (Fig. 5b). ADA is thus initialized with the order α of the last session. When a session ends, the close() operation writes all the rids in order back to disk. It merges all the fragmented subarrays (Fig. 5c) into one simple array (Fig. 5d) for next session to start with simply.

Lookup Operations As an index, ADA provides the rank-to-rid mapping for querying α. In general, we want to find the rids from rank i to j, i.e., $\alpha(i, j)$, which is a range query if $j > i$ or a point lookup if $j = i$. We thus need two basic operations.

First, we need locate(i) to locate the subarray in the rid-layer that contains the i-th rid in α. The process is similar to lookup in a B$^+$-tree; however, since each key is a count of its subtree, we sum up these counts from left to right, to locate the subtree whose range contains i. E.g., in Fig. 5c, the key 21 covers [6+1, 6+21] or [7, 27]; the key 10 covers [6+5+2+4+1, 6+5+2+4+10] or [18, 27]. We do this counting recursively until reaching the leaf of the key layer, which points to the target subarray in the rid layer. E.g., to locate(24), we follow the pointer from key 21 (whose range [7, 27] contains 24) and 10 ($24 \in [18, 27]$) to reach the subarray with content $(q, r, \ldots, z)$ where the target rank 24 is indexed 6th (i.e., 24-18, 0-based index), which is w. As shown in the following pseudo-code, we travel from root to a subarray (lines 2-7), keep subtracting the keys at the left side of the target, to find the subarray at node n and offset index i'; the target will be at $n[i']$.

```
0. Operation locate($i$): Input $i$, target rank.
1. $n \leftarrow Root$
2. While ($n$ is node in the key layer)
3.     For ($k = 0$; $k < M$; $k$++)
4.         $key = n.Keys[k]$)
5.         If ($i \leqslant key$)
6.             $n = n.Pointers[k]$; Break
7.         $i = i - key$
8. return $n$, $i - 1$ // Target is $n[i - 1]$.
```

Second, traverse(n, i, s) traverse subarray at node n, from $n[i]$ for s items, with the help of pointers between adjacent leaf nodes.

Modification Operations. We need to modify the structure of ADA. First, we can add new subarrays via creation or splitting: Operation create() will create a new subarray n so that its capacity and length satisfy $n.cap = M, n.len = 0$, and returns n. Operation extract(n, i, j) will extract $n[i, j]$ from an existing subarray n as a new subarray n' by pointing n' into n, i.e., $n'.ptr = n.ptr + i - 1$, $n'.len = j - i + 1$, $n'.cap = \max(M, j - i + 1)$, and returns n'. Second, we can combine two subarrays: Operation merge(n', n) is used to merge n' into neighbor n. It moves all rids of n' into n so that $n.len$ += $n'.len$, and then deletes n'. Operation redistribute(n, i_1, i_2, n', j) copies $n[i_1, i_2]$ to overwrite n' from $n'[j]$, and calculate the new length of n' depending on specific situations.

5 USER FUNCTIONS

We present the functions for querying, updating, and manipulating ADA (Sec. 6 will analyze complexity). These functions exploit the index structure (Sec. 4.1) using the basic operations (Sec. 4.2).

5.1 Lookup

ADA supports both point and range lookup, to support various user operations that needs to query the order of rows.

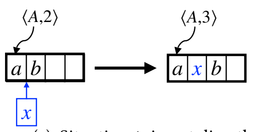
Lookup(i). The process of looking up the i^{th} rid is the same as basic operation $A, i' \leftarrow$ locate(i), but returns the target value $A[i']$.

Lookup(i, j). Looking up a range of rids is similar to that in a B$^+$-tree but different due to the two-layer structure of ADA. There are two steps as follows.

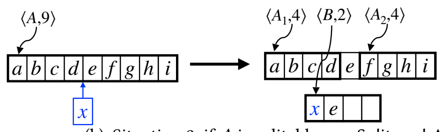
• **Step 1: Locate the i^{th} rid.** We first use $A, i' \leftarrow$ locate(i) to locate the first rid in the range.

• **Step 2: Traverse.** Since rids are stored in subarrays in order, we use traverse($A, i', j - i + 1$) to find all rids in the range.

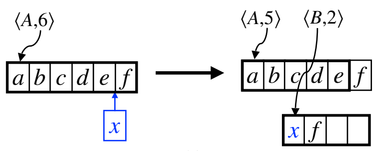
5.2 Update



(a) Situation 1: insert directly by shifting rids.



(b) Situation 2: if A is splitable, use Split-and-Add.



(c) Situation 3: use Add.

Figure 7: Three situations in insertion.

Insert(k, x)**.** In order to create and copy as little as possible in an insertion, we need to consider empty space, length of subarray A to which x should be inserted, and the insert position to choose an insert strategy. Our main idea is first trying to insert x into the current subarray. If the subarray is too full, we try to redistribute some rids to its neighbor, if the neighbor has empty space. Otherwise, we create a new subarray to redistribute.

Traditionally, redistribution should move rids at the start or end of A to another subarray. But unlike B-tree which balanced node size, a subarray has a variable length (**DP5**), so the time to do traditional redistribution is also a variable. To make it constant, when i is in the middle of A, we redistribute $\frac{M}{2}$ rids around index i to a new subarray (**DP3**), and extract the rids on the left and right side as two subarrays (**DP2**). We have the following three steps.

• **Step 1: Locate the k^{th} position.** We first use locate(k) to locate subarray A so that we should insert x to $A[i]$.

• **Step 2: Check $A.len$, i and A's neighbor to choose an insert strategy.** For different status of A, we have the following four situations with different insert strategies, as shown in Fig. 7.

Situation 1: If there are empty slots, insert directly. If $A.len < M$, this means the capacity of A is M and it has empty space, as shown in Fig. 7(a). We directly insert x into $A[i]$ by shifting rids, as shown in lines $3 - 5$ in the pseudo-code.

Situation 2: Otherwise, if A is long and i is in the middle, redistribute the middle part. In Fig. 7(b) and 7(c), there are no empty slots in A. If i is a middle position of A, with detailed condition in line 8 below, we redistribute $\frac{M}{2}$ contiguous rids starting from $A[i]$ to a new subarray, and then extract two subarrays before and after these $\frac{M}{2}$ rids, as shown in lines $8 - 13$. A in Fig. 7(b) satisfies these conditions, so we redistribute x, e, and extract a, b, c, d and f, g, h, i.

Situation 3: Otherwise, redistribute one rid to neighbor if possible If A's neighbor has empty space, we redistribute either the first or the last rid into the neighbor, depending on i, as shown in lines $13 - 15, 21 - 23$.

Situation 4: Otherwise, redistribute the start or end rids to a new subarray. This situation happens when A is not long enough, or the insert position i is too close to the start or end of A, so redistributing middle part will get illegal subarrays. This situation also means time to redistribute rids at the start or end of A is constant ($O(M)$), so we do as lines 17-19, 25-27 below. For example, in Fig. 7(c), A only has 6 slots, not enough to extract two independent subarrays like Fig. 7(b), so we redistribute e, f and extract other rids.

```
1. $A, i \leftarrow$ locate($k$)
2. $A \leftarrow$ list of rids after inserting $x$ into $A$ conceptually
3. If ($A.len < M$)
4.     Directly insert $x$ to $A[i]$, and return
6. $L_1 \leftarrow i, L_2 \leftarrow A.len - L_1 - \frac{M}{2}$
7. $N_l \leftarrow A$'s left neighbor, $N_r \leftarrow A$'s right neighbor
8. Case $L_1 \geqslant M$ && $L_2 \geqslant M$
9.     $B \leftarrow$ create()
10.     redistribute($A, i, i + \frac{M}{2} - 1, B, 1$)
11.     $A_1 \leftarrow$ extract($A, 1, i - 1$)
12.     $A_2 \leftarrow$ extract($A, i + \frac{M}{2}, A.len$)
13.     In the key layer, change $A$ to $A_1$, insert $A_2, B$ after $A_1$
14. Case $i \leqslant \frac{A.len}{2}$ && $N_l$ has empty space
15.     redistribute($A, 1, 1, N_l, N_l.len + 1$)
16.     $A \leftarrow$ extract($A, 2, A.len$)
17. Case $i \leqslant \frac{A.len}{2}$ && $N_l$ has no empty space
18.     $B \leftarrow$ create()
19.     redistribute($A, 1, \frac{M}{2}, B, 1$)
20.     $A \leftarrow$ extract($A$, min($\frac{M}{2}, A.len - M + 1$), $A.len$)
21.     In the key layer, insert $B$ before $A$
22. Case $i > \frac{A.len}{2}$ && $N_r$ has empty space
23.     ... //similar to lines $15 - 16$
24. Case $i > \frac{A.len}{2}$ && $N_r$ has no empty space
25.     ... //similar to lines $18 - 21$
```

Delete(k)**.** The deletion process is similar insertion above, so we only show the pseudo-code here.

```
1. $A, i \leftarrow$ locate($k$)
2. If ($A.len = \frac{M}{2}$)
3.     redistribute($A, i, A.len, A, i - 1$)
4.     ... //Merge or redistribute with a neighbor like b-tree
4.     $N_l \leftarrow A$'s left neighbor, $N_r \leftarrow A$'s right neighbor
4.     Case $N_l.len \leqslant \frac{M}{2} + 1$
4.         merge($A, N_l$)
4.     Case $N_r.len \leqslant \frac{M}{2} + 1$
4.         merge($A, N_r$)
4.     Default case
4.         redistribute($A, 1, A.len, A, 2$)
4.         redistribute($N_l, N_l.len, N_l.len, A, 1$)
4.         $N_l \leftarrow$extract($N_l, 1, N_l - 1$)
5. $L_1 \leftarrow i, L_2 \leftarrow A.len - i$
7. If ($L_1 \geqslant M$ && $L_2 \geqslant M$)
8.     $A_1 \leftarrow$ extract($A, 1, i - 1$)
9.     $A_2 \leftarrow$ extract($A, i + 1, A.len$)
10.     In the key layer, change $A$ to $A_1$, insert $A_2$ after $A_1$
11. Else if ($i \leqslant \frac{A.len}{2}$)
12.     redistribute($A, 1, i - 1, A, 2$)
```

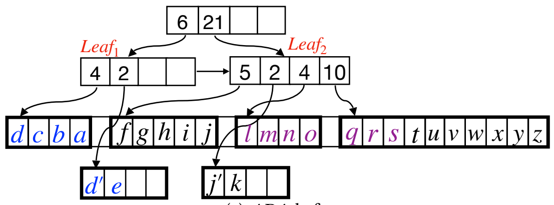
```
13.     $A \leftarrow$ extract($A$, 2, $A.len$)
14. Else
15.     redistribute($A$, $i+1$, $A.len$, $A$, $i$)
16.     $A \leftarrow$ extract($A$, 1, $A.len - 1$)
```

5.3 Order Manipulation

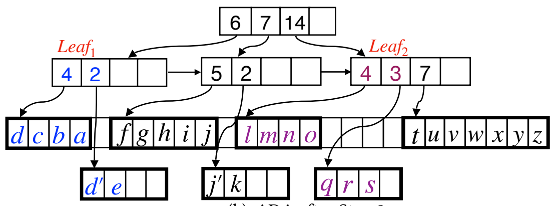
Reorder(i, j, X)**.** Reorder is a rid redistribution within a range, so it does not change the structure of ADA but it needs to modify rids. The two steps are as follows.

• **Step 1: Locate the i^{th} rid.** We use locate(i) to locate the i^{th} rid.
• **Step 2: Traverse and modify.** We observe that before and after reorder, only rids within the range change their ranks, and the new ranks are still within this range. This means we only need to redistribute rids in this range in corresponding subarrays, without touching other part of the data structure. So after locating the first rid in the range, we traverse along the subarrays and modify each rid to the new corresponding value. Following is the pseudo-code of reorder operation.

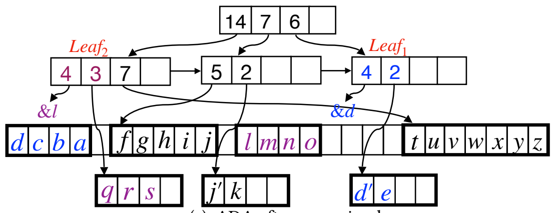
```
1. $A, i' \leftarrow$ locate($i$)
2. Use traverse($A, i', j-i+1$) to traverse along the $i^{th} \sim j^{th}$ rids and change them on the way
```



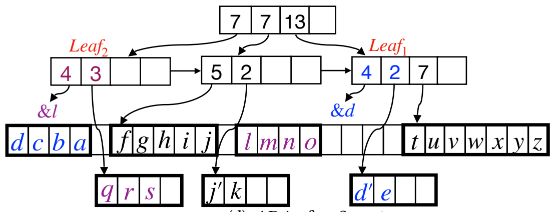
(a) ADA before swap.



(b) ADA after Step 2.



(c) ADA after swapping leaves.



(d) ADA after Step 4.

Figure 8: Swap$(1, 6, 14, 20)$ **in ADA**

Swap(i_1, j_1, i_2, j_2) To realize **DP6**, we graft whole leaves instead of deleting and reinserting rids so as to reduce time complexity. We describe process of Swap(1, 6, 14, 20) in Fig. 8.
• **Step 1: Do range lookups.** First of all, we need to locate the two ranges in ADA by doing range lookups. The two ranges are shown in Fig. 8(a) highlighted with blue and purple colors, respectively.
• **Step 2: Align subarrays with leaves.** We need to align subarrays with leaves to avoid moving rids which should not be moved. In Fig. 8(a), all seven purple rids are indexed by $Leaf_2$, but there are other rids $t, u, ..., z$ also indexed by $Leaf_2$. When we graft $Leaf_2$, $t, u, ..., z$ will also be moved, though they should stay in their original positions. This is because the purple range does not ends at the end of a subarray, and we need to align it. Similarly, if a range does not begin at the beginning of a subarray, we also need to align it. We split between s and t and get Fig. 8(b).
• **Step 3: Swap leaves.** We swap whole leaves: $Leaf_1$ and $Leaf_2$ in Fig. 8(b) by moving the corresponding ⟨key, pointer⟩ pairs and get Fig. 8(c). We do not change subarrays but they are moved logically since they are indexed by leaves. However, we find the third pair $\langle 7, \&t \rangle$ in $Leaf_2$ does not belong to a range but it is moved together with $Leaf_2$. So next, we need to recover such pairs.
• **Step 4: Recover separate subarrays.** If a leaf overlaps only partially with a range, like $Leaf_2$ and the purple range, then some of its ⟨key, pointer⟩ pairs are mistakenly moved in Step 3 and in this step we recover them to original positions. We move $\langle 7, \&t \rangle$ after the blue range and get Fig. 8(d).
• **Step 5: Check legality.** Recall that in Step 2, we align subarrays with leaves, which can make a subarray less than half-full. So our last step is: for either side of each range, we check the length of subarrays. If we find a subarray not half-full, we either merge it or redistribute it with its neighbour. In Fig. 8(d), all these subarrays are half-full, so we do nothing in this step.

We summarize Swap(i_1, j_1, i_2, j_2) in the following pseudo-code.

```
1. Locate two ranges by Lookup($i_1, j_1$) and Lookup($i_2, j_2$)
2. Align 4 subarrays at the edges of two ranges
3. Swap leaves of the two ranges
4. Move separate subarrays back to original positions
5. Check the legality of the 4 subarrays at the edges of two ranges
```

6 MATHEMATICAL ANALYSIS

We first study the time complexity (Sec. 6.1) of our framework and compare it to state of the art. Further, we analyze the optimal (lower-bound) complexity of the ordered-access problem (Sec. 6.2) and show that our framework is indeed optimal. We focus on time complexity, as the primary goal of ordered access is to support interactive, responsive data exploration. As all methods store n rids in some way, the space complexity is similarly linear and practically insignificant (e.g., 4 MB for 1 million rows with 4 bytes/rid).

We model the cost with respect to the sizes of not only data and but also its changes of a workload. The *data size* n, number of rids in the table, is the amount of data to manage. Moreover, as we aim to adapt to changes, we capture the *change size* c, number of change operations, i.e., Insert, Delete, Swap, and Reorder.

For simplicity, we leave out several less significant parameters. We leave out machine characteristics: the number b of bits in a *word* (e.g., 32 bits) as the unit of memory access and the number δ of bits

	ADA		CBT	SA	LL
	$c \ll n_0$	$c = \Omega(n_0)$			
Lookup	$O(\log c)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(m)$
Insertion	$O(\log c)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(m + \frac{n}{m})$
Deletion	$O(\log c)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(m + \frac{n}{m})$
Swap	$O(\log c)$	$O(\log n)$	$O(L \log n)$	$O(n)$	$O(n)$
Reorder	$O(\log c)$	$O(\log n)$	$O(L \log n)$	$O(1)$	$O(m)$

Figure 9: Time complexity of ADA and state of the art.

in a *rid* (to represent an update). These characteristics are constants in a specific computing environment. As we will see, our analysis captures only the *number* of information transfers, i.e., memory probes. We also leave out the retrieval size L, length of a range in an operation, e.g., a range query Lookup(i, j), where $L = j - i + 1$, when it is the inherent work (e.g., sequential access to read every in a range) that cannot be optimized. With this omission, we do not distinguish point and range lookups.

6.1 Time Complexity

We now analyze the complexity of ADA in contrast to representative baselines (as Sec. 2 discussed), which Fig. 9 summarizes.

• **Standard Array (SA)** supports $O(1)$ lookups but with $O(n)$ insertions and deletions. To swap two ranges of lengths L_1 and L_2, we need to shift $|L_1 - L2|$ items, which is also $O(n)$. To reorder a range, we locate the range in $O(1)$ and rewrite the items (which is the same cost for all algorithms and thus ignored).

• **Indexed Linked List (LL)** [10] stores every m items in a separate linked list and uses an array of length $\frac{n}{m}$ to index these lists, so it has $O(m)$ lookup time. Although m is a constant, it helps to make LL faster at updates by sacrificing some lookup efficiency, compared to SA. For insertion and deletion, it needs to update the array so the time complexity is $O(m + \frac{n}{m})$. Similar to SA, the swap time of LL is $O(n)$, and the reorder time is the lookup time $O(m)$.

• **Counted B-tree (CBT)** has $O(\log n)$ time for lookup, insertion, and deletion as a B-tree. However, as each item appears at a different level (unlike B$^+$-tree), swap and reorder of size L need $O(L \log n)$.

We now analyze the complexity for ADA to observe its adaptive cost which our change-driven indexing aims to achieve. As a desirable contrast to the state of the art, ADA has an interesting cost behavior which adapts to the amount of changes c. While having a balanced tree structure (for the key layer) like a B-tree, it only reaches $O(\log n)$ when c is close to the data size, i.e., $c = \Omega(n_0)$, where n_0 is the number of items when a session starts (i.e., $n_0 = |A|$; see Sec. 4). When c is small, with $c \ll n_0$ (e.g., 10 insertions among 10K rows), the cost depends *only* on c, i.e., $O(\log c)$. We note that, in practice, changes c often remains smaller than n in a human-driven workload, making the adaptive cost appealing.

When $c \ll n_0$: At the begging of a user session (Sec. 4.1), *ADA* starts with only one subarray in the rid layer and thus one node (root) in the key layer. A new change will introduce a constant number (e.g., 2 for an insertion) of subarrays. The number of new subarrays is thus proportional to c, which needs to be indexed by the key layer. Proportional to the height of the key tree, the complexity of all operations, which need to locate a rank through the key layer, is thus $O(\log c)$.

When $c = \Omega(n_0)$: As c increases, since each change split the original array A add/or create new "regular" subarrays, both of which have a minimum capacity $\frac{1}{2}M$, the number of subarrays, rather than growing with c, will be limited to $\frac{2n}{M}$, for current data size n– with the cost is $O(\log c)$. Since each change split the original array A into some constant k subarrays (and creating new regular subarrays), the threshold happen when A (with n_0 items) is completely broken into regular sizes, i.e., $kc \geq n_0/M$, i.e., $c = \Omega(n_0)$.

Theorem 1 states the complexity results with a precise range of the turning point. The exact threshold depends on the particular mix of change operations. Intuitively, a workload with all swaps will reach the turning point quicker, since each swap can cut A into four subarrays, thus giving the lower value of the threshold range. In contrast, an insertion cuts only one subarray out of the original array, and thus gives the higher threshold. Due to space limitation, we give the full proof in an extended report.

Theorem 1. *With data size n, initial data size n_0, change size c, and a threshold t s.t. $\lceil (\lceil \frac{n_0}{M} \rceil - 1) / 4 \rceil \leqslant t \leqslant \lceil \frac{n_0}{M} \rceil - 1$, for all operations, including lookup, insert, delete, swap and reorder, the time complexity of ADA is $O(\log c)$ when $c \leqslant t$ and $O(\log c)$ when $c > t$.*

6.2 Optimality

Now we present the optimality– that the cost of ADA is the lowest possible. We will first develop the lower bound for the cost of the ordered access problem, and then show that the cost of ADA meets the lower bound. Our result is novel, and our proof adopts the technique developed for proving lower bounds in the cell-probe model for dynamic data structures (such as the partial-sums problem).

For ordered access, as a main-memory data structure problem, we will use the cell-probe model [12] to analyze the cost of computation. In the *cell probe model*, the memory consists of fixed-sized *cells* (or words), and we count the cost of computation as the number of cells it reads or writes. The model is general to capture the main cost of memory-based computation, and has been used to prove lower bounds on data structures (e.g., [3]). Our proof is inspired by the techniques for proving logarithmic bounds [13], but our problem is distinct and our modeling of changes c is novel.

Changes impact queries. A change c(i) will *hit* a rank i and change every position after. Consider an order A with length n. Insert(i), now denoted i(i) for brevity, and Delete(i) or d(i) will shift every rank from i by +1 and -1 respectively. Swap(i, j, i', j') also changes every rank from i when $j = i'$ and $j' = n$ and similarly Reorder(i, j, X) when $j = n$, so we simplify them as s(i) and r(i). Thus, every change c(i), hitting rank i, will change the return value of Query(j) or q(j), if c precedes q in time, and leads q in ranks, i.e., $i \leq j$. We say q *depends* on c.

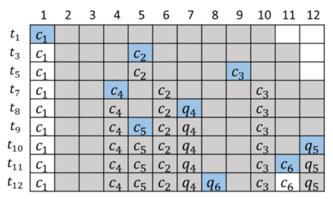
We develop the lower bound by identifying the amount of information transfer– and thus the number of cell accesses– that are *required.* We will create a possible hard instance where the bound can be identified. Consider ordered access over n items with c changes and q queries in a session. Fig. 10a shows an an example sequence of operations over 10 items ($n = 10$), from time t_1 to t_{12}.

To model the cost, we will consider the information transfer required, due to dependencies, between *any* pair of adjacent equal-length intervals of operations, which we call the *left* and *right* intervals. In Fig. 10a, consider the pair at time 1-6 and 7-12 as the left and right. How many cell accesses *must* happen in the right to read the contents written in the left? To answer, we will find the

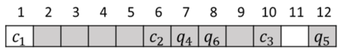
Number	c_1	q_1	c_2	q_2	c_3	q_3	c_4	q_4	c_5	q_5	c_6	q_6
Operation	d(1)	q(3)	i(4)	q(2)	s(8)	q(9)	i(3)	q(6)	r(4)	q(11)	d(10)	q(7)
Time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}

Left Interval *Right Interval*

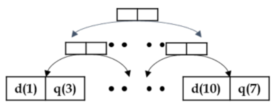
(a) Operations.



(b) Evolution of operations.



(c) Left changes and right queries.



(d) Intervals in a balanced tree.

Figure 10: Example operations and their dependencies.

number of *required* information transfers– each indicating one cell probe– from the left to right. Summing up such transfers between all adjacent intervals, we will obtain the lowest cost possible.

To study dependencies, we inspect how their change and query operations interleave. Consider the first and second halves as the left and right intervals. At the beginning, there are n (e.g., $n = 10$) ranks. Each d, s, or r operation will hit an existing rank and an i operation will create and hit a new rank. To visualize, Fig. 10b shows the evolution of the ranks, where shaded cells are occupied ranks and blue cells are ranks hit at each time. The first operation $c_1 = \text{d}(1)$ hits the 1^{st} place, which we leave in place to indicate the presence of the event c_1. Next, $c_2 = \text{i}(4)$ adds a new rank to become the new 4^{th} place. At t_{12}, there are 12 ranks (due to two insertions).

The simulation of operations over ranks reveals the dependencies between intervals, since *all* operations are "recorded" in the *common* coordinates of ranks at the end, despite their dynamic evolution. At time t_{12}, as Fig. 10c shows, we observe how the changes (c_1, c_2, c_3) of the left interleave with the queries (q_4, q_5, q_6) from the right interval. The left changes hit indexes 1, 6, 10 (as numbered at t_{12}), before the right queries read 7, 8, 12. To see dependencies, we line them up along the indexes hit: $c_1, c_2, q_4, q_6, c_3, q_5$.

Dependencies exist when a change directly transitions to a query in their interleaving. Since a change c(i) will affect any query q(j) on a larger rank ($i \le j$), q_4 depends on c_1 and c_2. Are both *required*? For two changes c(i_1) and c(i_2), w.r.t. query q(j), if $i_1 < i_2 \le j$, the effect of c(i_1) may be "absorbed" by c(i_2). The converse (for c(i_2) to be absorbed by c(i_1)) is not true since another query between i_1 and i_2 should not wrongly see the effect of c(i_2). So, only c_2 *must*, although c_1 *may*, be transferred. Next, succeeding q_4 with no interleaving changes, q_5 may not require new information. Thus, an information transfer must take place when a change c(i) *immediately transitions* to query q(j) without operations in between. So, e.g., two transfers are mandatory from the left to right half interval: c_2 and c_3.

Lemma 1 quantifies the required information transfer between two intervals (by estimating immediate transitions). Consider intervals with x changes and x queries. Intuitively, with few operations, i.e., x is small relative to the number of ranks n (specifically, when $x = O(n^{1/3})$), they will hit *different* ranks. Thus they will uniformly interleave into c-to-q transitions among possible combinations of adjacent operations– i.e., *every* change among x (thus $\Omega(x)$) has a chance to interleave. With more operations, however, changes (and queries) will start to collide at the same ranks, and n instead of x becomes the limiting factor for transitions to occur. The phenomenon is the standard *birthday paradox* of how x people have distinct birthdays over n days. We leave the proof to the extended report.

Update Performance			
insert-asyn	insert-syn	delete-asyn	delete-syn
1.3 µs	41 ms	2.5 µs	41 ms
Query Performance			
prefetching		no prefetching	
7.5 µs		49 ms	

Figure 11: Framework validation.

Lemma 1 (Information Transfer). *Consider two adjacent intervals each containing x changes and x queries and $x = O(n^{1/3})$. The required information transfer from the left to right is $\Omega(x)$.*

Finally, we present the tight lower bound of the ordered access problem in Theorem 2, modeling both n and c. As Lemma 1 gives the minimum cost *between* a pair of adjacent intervals, we need to sum up *all* such pairs. To do so, we organize all c changes with c queries (to "consume" changes) into a binary tree, as Fig. 10d shows. Each node connects two intervals at a different time point, e.g., the root is the half-time t_7, connecting our example intervals. To apply Lemma 1, which nodes satisfy the $O(n^{1/3})$ condition? If c is small, the nodes at every level will do, and thus the amortized cost boils down to the tree height $\log c$. Otherwise, only those nodes with small intervals can– they are at the lower levels. Thus the cost is limited by the height of such lower levels or $\log n$. The turning point of c is $\sqrt[3]{n}$. We prove the theorem in the extended report. The bound is tight: Our ADA achieves the lower bound, and Theorem 2 predicts its performance including the turning point (Fig. 9).

Theorem 2 (Lower Bound). *For the ordered access problem with c change operations over n data items, any cell-probe data structure has an amortized per-operation cost lower bound $\Omega(\log c)$, when $c \le n^{1/3}$, and $\Omega(\log n)$ otherwise.*

7 PERFORMANCE EVALUATION

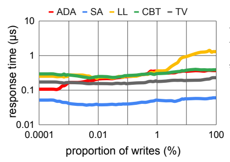
We perform experiments to validate the effectiveness of our overall OAM framework for supporting interactive ordered access, measure the efficiency/scalability of ADA, and assess its design points via an ablation study.

7.1 Experimental Setup

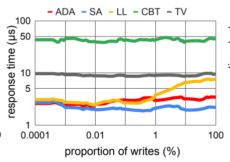
Computing Platform. We conducted all the experiments on a machine with 12 CPUs (12 threads, Intel (R) Core i7-6850K CPU @3.60 GHz) and 62 GB memory. Each processor has three levels of cache: 32KB L1 data cache and 32KB L1 instruction cache, 256KB L2 cache, and 15MB L3 cache.

Studies. We present four set of studies.

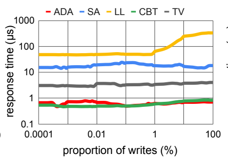
- **Framework.** We validate the efficiency of our OAM framework by comparing the response time with and without simulated asynchronous prefetcher and asynchronous updater. We run the same workload and execute an insertion, deletion or lookup at different points where different amount of changes that have been done. For example, when 1% of operations in the workload have been done, we insert an rid into the index structure and record the response



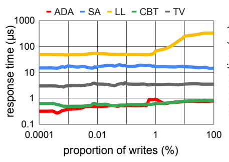
(a) Point Lookup.



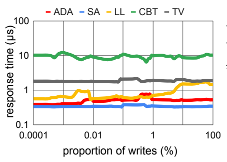
(b) Range Lookup.



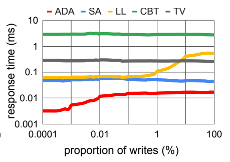
(c) Insert.



(d) Delete.

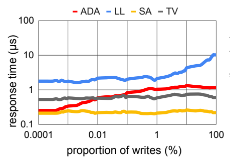


(e) Reorder.

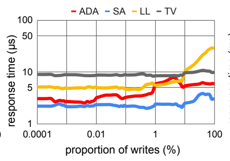


(f) Swap.

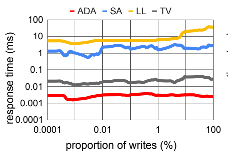
Figure 12: Index structure comparison with 10K dataset.



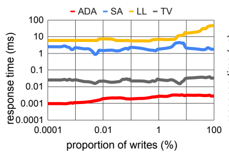
(a) Point Lookup.



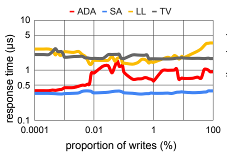
(b) Range Lookup.



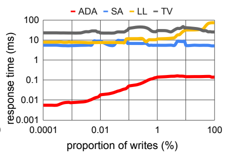
(c) Insert.



(d) Delete.

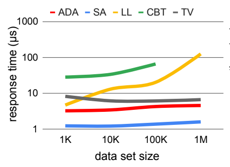


(e) Reorder.

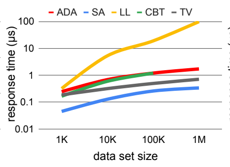


(f) Swap.

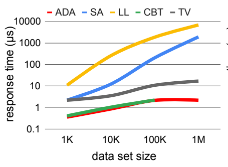
Figure 13: Index structure comparison with 1M dataset.



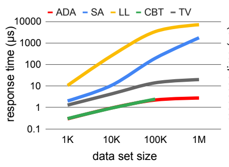
(a) Range Lookup.



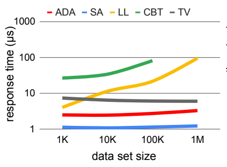
(b) Point Lookup.



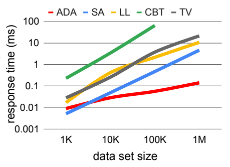
(c) Insert.



(d) Delete.

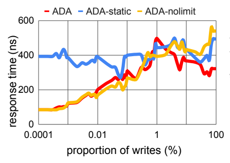


(e) Reorder.

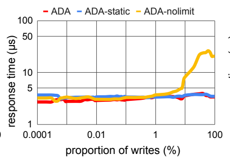


(f) Swap.

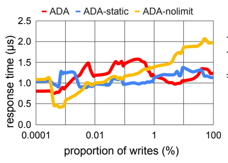
Figure 14: Scalability study with 1K, 10K, 100K and 1M datasets.



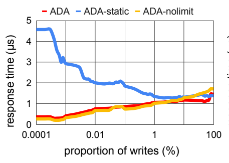
(a) Point Lookup.



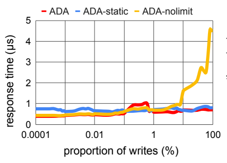
(b) Range Lookup.



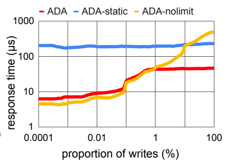
(c) Insert.



(d) Delete.



(e) Reorder.



(f) Swap.

Figure 15: Ablation study with 1M dataset.

time of this insertion, and we do the same for 10% ,50%, etc. We repeat the same experiment for 10 times and take average results.
• **Index Structure.** To evaluate the efficiency and trade-off of ADA, we compare it with the start-of-the-art. We choose CBT which achieves optimal time complexity w.r.t. the size of dataset [13], LL which makes a trade-off between changes and queries by adding an auxiliary array, and tiered vector (TV) which achieves the same logarithmic lower bound in [13] but has a different choice of the trade-off. We run experiment in the same way as we evaluate the framework.
• **Scalability.** To evaluate the scalability of ADA, we compare it with the other four index structures same as above, but we compare the average response time of a whole workload rather than showing response time at different points of a workload.
• **Ablation Study.** To validate the design of ADA, we perform ablation study on design points mentioned in Sec. 3.2.

Dataset and Workload. We want to validate performance of a wide range of user operation scenarios, from small to large tables and from read-only to write-mostly. For this objective, the content of data, represented by the primary keys in a table, does not matter; only the size of data matters. Thus, we use synthetic datasets with specific sizes to represent small and large tables. Further, to systematically assess varying and mixed operations, we use synthetic workloads with different mixes of operations. We test different points during the sequence of a workload, to simulate different user behaviors.

We generate four datasets: *1K Dataset*, *10K Dataset*, *100K Dataset* and *1M Dataset*. They have 1K, 10K, 100K and 1M rids, respectively, and the rids are positive integers starting from 1, simulating user interacting with small tables, e.g., profiles of students in a class, to large tables, e.g., an informative business table. We use a workload consisting of 1M changes in random order, and each of the four changes accounts for 25%, and do tests at different time points during the workload.
Metrics. We compare *response time* in each study, which is the time spent on indexing of one specific operation during a workload.
Baselines. We choose CBT which achieves optimal time complexity w.r.t. the size of dataset [13], LL which makes a trade-off between changes and queries by adding an auxiliary array, and tiered vector (TV) which achieves the same logarithmic lower bound in [13] but has a different choice of the trade-off. All data structures and experiments are implemented in c++ and open sourced. CBT, SA and LL are implemented by ourselves. Tiered vector [9] offers an open source code [14] in c++, but it has a restriction that the capacity of the tiered vector cannot exceed a predefined threshold, which disagrees from ordered access requirement. Based on their source code, we make some modifications to enable it to be extensible. In

addition, none of these four index structures support swap, so we implement swap manually by deleting and reinserting rids one by one.

7.2 Framework Validation

As mentioned in Sec. 3.1, our framework features asynchronous updater and asynchronous prefetcher, so that the response time of the system only depends on the index structure, to hide latency by asynchronous database access. To validate the efficiency of OAM, we use multi-thread experiments to simulate scenarios where a user is scrolling, inserting and deleting, with an interval of at least 500 ms between every two user operations to simulate real user scenarios. We use ADA as the index structure to support the client to access MySQL database, with 100K dataset and a workload consisting 100K operations. From [15], 100 ms is the time limit for a system to response so that users feel that the system is reacting instantaneously. We favor a system with response time far less than 100 ms. Figure 11 shows the average response time with or without asynchronous updater and prefetcher.

Asynchronous Updater. We study the average response time of asynchronous/synchronous insert/delete operations. Synchronous response time consists of both the time of ADA and the time to access MySQL, while asynchronous one is the ADA time. It is obvious that asynchronous update is much more efficient than synchronous version, because it does not include the time to access the database. And compared to the baseline 100 ms, asynchronous update is much below it, while synchronous version is clost to the limit. So asynchronous updater helps to make the system interactive.

Asynchronous Prefetcher. We focus on the average response time of query with/without prefetching. In the experiment, we set the size of a *user window*, which is the maximum number of rows a user can look at on the screen, to be 30, and buffer 5 windows every time: current user window, plus 2 windows before and 2 windows after the current user window. Users scroll continuously up or down so we buffer data in the neighbourhood. If the tuples are not ready at the time of a scrolling, we fetch the new user window immediately and synchronously. We find that the average response time with prefetching is much smaller than 100 ms baseline, while that without prefetching is close to 100 ms baseline, which means we need asynchronous prefetcher to achieve responsiveness.

7.3 Index Structure Comparison

In index structure comparison, we compare our ADA with existing works including CBT, SA, LL and TV on 10K dataset and 1M dataset.

Responsiveness. Fig. 12 and 13 show response time in index structure comparison with 10K and 1M dataset, respectively. In the two figures, the x-axis is the proportion of changes that has been executed, and the y-axis is the average response time of 10 operations. For example, Fig. 12(a) shows the response time of point lookup when certain proportion of changes have been executed, e.g., 1%, 10%. We test CBT only with 10K dataset but not with 1M dataset because in its response time is too long with 1M dataset. For example, a swap or reorder operation needs about 0.8s, which means a workload containing 500K swap and reorder will need more than 100 hours. We exclude CBT in advance.

The trends are similar between Fig. 12 and 13 so we analyze them together. CBT is the slowest one for range-based operations like range lookup, reorder, and swap, because it queries or reorders a range by looking up or updating one rid in logarithmic time repeatedly. And CBT is fast on updates, due to logarithmic time complexity. SA is good at lookup and reorder due to optimal lookup performance $O(1)$, but bad at others due to linear time complexity for update and order manipulation. LL is the slowest in point lookup because it needs to traverse a small range, slowest in update because the auxiliary array always needs updating which needs $O(m + \frac{n}{m})$ time. But it is faster than CBT and TV in range-based operations since the linked list structure makes it easy to traverse. TV has a performance between SA and ADA for point lookup and update, because it has a different choice of trade-off between update and query from ADA. ADA is slower than only SA in lookup and reorder, for which SA has constant time complexity, and the best in other operations. For all operations, the response time of ADA first grows, which corresponds to time $O(\log c)$ before turning point due to change-driven indexing; and then ADA enters a plateau, which means most subarrays have regular lengths, so that the depth of ADA depends on the data size which almost maintains unchanged.

Scalability. Fig. 14 compares the scalibility of ADA with baselines. The x-axis is the size of datasets, from 1K to 1M, and the y-axis is the average response time of the corresponding operations executed during the workload. ADA generally has the most stable lines across different operations, due to its logarithmic time complexity on every operation. The lines of SA increase slowly for most operations, but quickly for insert, delete and swap, because it needs linear time to update items when executing these operations. LL always has a fast growth, since its time complexity is linear to the dataset size, even if its auxiliary array helps to avoid much traversal. Counted B-tree grows fast for range-based operations since it handles each rid at a time, and grows slowly for other operations, due to its logarithmic time complexity. TV grows relatively slowly, since for different datasets, we set the TV to have the same number of levels, but different vector sizes.

7.4 Ablation Study

In ablation study, we aim to show the effect of design points in Sec. 3.2. Among all the 6 design points, **DP1**, **DP4** and **DP6** are evaluated by comparing ADA with CBT; **DP2** and **DP7** is evaluated by comparing ADA with SA. So we focus on **DP3** and **DP5** in ablation study. We show results on 1M dataset in Fig. 15. Results on other datasets are similar and thus omitted. We compare ADA with the following two "variants".

• **ADA-static** is an ADA without change-driven indexing: it indexes not based on changes but a uniform node size. When we initially load data, the capacity of all subarrays is M, so the tree is already fully constructed. Its height grows because of full or empty nodes rather than a changes that split subarrays.

• **ADA-nolimit** is an ADA without subarray length restriction. In other words, subarrays do not have minimum length restriction ($M/2$) and can be as short as 1. Their capacities are always equal to lengths so there are no empty slots to facilitate insertions.

As shown in Fig. 15, ADA-static is stable for most operations, but sometimes slower than ADA, mainly because of the loss of **DP3**. ADA-static has a balanced tree structure with regular-sized nodes all the time. The growth of the tree is only based on the increasing number of items. So when the number of changes is small, which is often the case in practice, ADA has $O(\log c)$ time complexity which is smaller than $O(\log n)$ time of ADA-static. In addition, we have equal number of insertions and deletions in workloads which makes the number of items maintains roughly unchanged. Only Fig. 15(d) shows an exception: the blue line decreases. This is because when we initialize ADA-static, every time the new item is inserted into the last leaf, because data are inserted sequentially. Every time when last leaf is full, it is split into two half-full leaves, and we continue to insert into the new last one, leaving the original one still half-full. After initialization, all leaves are half-full. So deletions at the beginning of the workload often make a leaf too empty and need merging or redistribution, which is time-consuming. When there are more changes, there are less half-full leaves, and the response time of ADA-static becomes stable.

ADA-nolimit always has a growing line, overlapping with ADA at first but keeps growing after ADA enters a plateau. Both ADA-nolimit and ADA index changes, i.e., their tree heights grow by changes, so their tree height is $O(\log c)$ at the beginning, having similar growing lines at first. Both of them will enters a plateau when all subarrays have regular sizes, but due to length restriction M, ADA enters early. ADA-nolimit almost does not enter within the workload, since changes always cut subarrays into smaller pieces until reaching length1, causing a large growing number of subarrays. Thus, **DP5** is effective to control the scale of the structure from growing too large.

Compared to the two variant, the line of ADA first grows from a low point, which corresponds to $O(\log c)$ time complexity before the turning point, and then enters a plateau, which corresponds to $O(\log n)$ time complexity. Before the turning point, its response time is much shorter than ADA-static and similar to ADA-nolimit, due to **DP3**. After the turhing point, its response time is shorter than ADA-nolimit and similar to or also shorter than ADA-static, due to **DP5**.

8 CONCLUSION

In this paper, in order to enable order awareness in relational databases, we propose a general framework OAM and a novel index structure ADA which maintains a rank-to-row mapping. ADA features indirect keys to solve cascading changes, two-layered structure to decouple keys and rids, and change-driven indexing so that the complexity only depends on changes rather than data size when the number of changes is small. Modeling both changes and data, we prove the optimality of ADA by proving the lower bound of the ordered access problem. We perform experiments to validate the performance and design of our framework OAM and ADA. We find that OAM helps to improve responsiveness of the system effectively so that the response time is far less than the interactivity limit. We also observe that ADA have the best overall performance, i.e., responsiveness and scalability compared to other index structures. Ablation study shows that the design of ADA effectively enhances its responsiveness. Our new framework and index structure improves usability and interactivity for acccessing database systems in an intuitive tabular interface, which inherently requires persistent and personalized ordering of rows.

REFERENCES

[1] Waqas Javed, Niklas Elmqvist, Ji Soo Yi, et al. Direct manipulation through surrogate objects. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 627–636. ACM, 2011.
[2] Ben Shneiderman. Direct manipulation: A step beyond programming languages. In *ACM SIGSOC Bulletin*, volume 13, page 143. ACM, 1981.
[3] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1989.
[4] Paul F Dietz. Optimal algorithms for list indexing and subset rank. In *Workshop on Algorithms and Data Structures*, pages 39–46. Springer, 1989.
[5] Mihai Pǎatraşcu and Erik D Demaine. Tight bounds for the partial-sums problem. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 20–29. Society for Industrial and Applied Mathematics, 2004.
[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
[7] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.
[8] Philip Bille, Anders Roy Christiansen, Mikko Berggren Ettienne, and Inge Li Gørtz. Fast dynamic arrays. *arXiv preprint arXiv:1711.00275*, 2017.
[9] Michael T Goodrich and John G Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In *Workshop on Algorithms and Data Structures*, pages 205–216. Springer, 1999.
[10] Richard Karl Kirkman. Method and apparatus for maintaining a linked list, June 17 2003. US Patent 6,581,063.
[11] Binny S. Gill and Dharmendra S. Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 33, USA, 2005. USENIX Association.
[12] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM (JACM)*, 28(3):615–628, 1981.
[13] Mihai Patrascu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
[14] Source code related of tiered vector. https://github.com/mettienne/tiered-vector.
[15] Response times: The 3 important limits. https://www.nngroup.com/articles/response-times-3-important-limits/.

A PROOF OF THEOREM 1

Proof. The time complexity of ADA depends on the height of the tree structure, i.e., $O(\log s)$, where s is the number of subarrays and is affected only by changes. We study how the number of subarrays is affected by different kinds of changes. Since reorder does not affect the structure of ADA, we only consider insertion, deletion and swap.

When $c \ll n_0$, the time complexity is $O(\log c)$, since every change increases the number of subarrays by a constant number. When all subarrays have regular lengths $\frac{M}{2} \sim M$, which means the number of subarrays is $\frac{2n}{M} \sim \frac{n}{M}$. Since M is a constant, in this case the complexity is $O(\log n)$. There is a turning point from $O(\log c)$ to $O(\log n)$ and we aim to find this turning point.

We study the turning point t when the workload only consists of one kind of change, and then show that the turning point is constrained in a range. Since we want to get the worst case complexity, we should find the earliest possible turning point, i.e., we make the number of arrays grow as fast as possible until the turning point.

We consider how insertion, deletion and swap affects the number of subarrays. An insertion cuts the subarrays at one position at most. A delete also cuts once but it also removes an item and makes it faster to reach the turning point, i.e., make it easier to let all subarrays have regular lengths. A swap cuts a subarray at four different positions at most. To get the worst case turning point, we consider the situation when all subarrays have length M, so the number of subarrays is $\lceil \frac{n_0}{M} \rceil$, which means $\lceil \frac{n_0}{M} \rceil - 1$ cuts are needed. On the one hand, a change cuts the subarrays at no more than four position, so the number of changes before the turning point is no less than $\lceil \left(\lceil \frac{n_0}{M} \rceil - 1\right)/4 \rceil$. On the other hand, a change cuts the subarrays at no less than one position, so the number of changes before the turning point is no more than $\left(\lceil \frac{n_0}{M} \rceil - 1\right)$.

Based on above, for mixed workloads, the turning point falls in the range $\lceil \left(\lceil \frac{n_0}{M} \rceil - 1\right)/4 \rceil \sim \left(\lceil \frac{n_0}{M} \rceil - 1\right)$. Theorem holds. □

B PROOF OF LEMMA 1

Proof. With n original ranks and x changes, an operation can hit a rank among at least n choices. The set of ranks is only changed by insertions: Any insertion will increase the rank by 1.

1) Distinct hits. We show that all operations hit different ranks with a probability $\Omega(1)$

From the result of the birthday paradox, for p people over b possible birthdays, the expected "collision" X , number of people sharing a birthday with others, is $E[X] = b\left(1 - (1 - \frac{1}{b})^{p-1}\right)$. With the binomial expansion of the inner term, $E[X] = \frac{p(p-1)}{b} - \frac{p(p-1)(p-2)}{2!b^2} + \frac{p(p-1)(p-2)(p-3)}{3!b^3} + \cdots = O(\frac{p^2}{b})$ - $O(\frac{p^3}{b^2}) + O(\frac{p^4}{b^3}) + \cdots = O(\frac{p^2}{b})$, if $\frac{p}{b} < 1$. The probability that some people share birthdays with others, by Markov's inequity, is thus upper bounded, i.e., $P(X \geq 1) \leq E[X]$. So, the probability of all people having a distinct birthday is lower bounded: $P(X = 0) \geq 1 - E[X] = 1$ - $O(\frac{p^2}{b})$.

We have totally $p = O(n^{1/3})$ operations over $b \geq n$ ranks. The probability that the operations are over distinct ranks is at least 1 - $O(\frac{n^{2/3}}{n}) = 1$ - $O(n^{-1/3})$ and thus $\Omega(1)$.

2) Transitions. We show that the expected number of transitions $E[T] = \Omega(x)$.

Since all operations hit different ranks, we order the operations by the ranks each operation hits. Consider each pair of adjacent operations a and b. To form a transition, a needs to be from the left (1/2 chance) and a change (1/2), and b needs to be from the right (1/2) and a query (1/2); thus the probability is 1/16. Among totally $2(x + x) - 1$ or $4x - 1$ such pairs, $E[T] = 1/16 \times (4x - 1) = \Omega(x)$.

3) Transfers. Here we complete the proof. Since each transition represents a required information transfer, the expected transfer is $\Omega(x)$. □

C PROOF OF THEOREM 2

Proof. Since there are c changes, suppose there are also c queries and all the $2c$ operations hits different ranks uniformly at random. To help sum up the required information transfers, we organize the $2c$ operations in a binary tree whose leaves are the sequence of operations; Fig. 10(d) show the tree for our example. As a binary tree, each internal node u represents the concatenation of two sub-intervals at a particular time point. We quantify the required information transfer at each node, which thus represents the transfer needed at a particular time; e.g., the root node is the 1/2 time point, its children are the 1/4 and 3/4 time points, respectively. Since every node represents a distinct time, we can sum up all the nodes to get the total transfer throughout the entire sequence of operations.

The binary tree has $\log 2c$ levels, which we number from the leaves as level 0 to the root as level $\log 2c$. At the k^{th} level, each node u contains 2^k operations in the subtree rooted at u.

1) When $c \leq n^{1/3}$: In this case, Lemma 1 holds for each node u at every level, since its subtree contains at most $2c$ or $O(n^{1/3})$ operations (when u is the root). Summing up all the nodes at each level, which together cover all c changes and c queries, we get the information transfer as $\Omega(c)$. Since there are $\log c$ levels, the total transfer is $\Omega(c \log c)$ and thus, over $2c$ operations, the amortized per-operation cost is $\Omega(\log c)$.

2) Otherwise: Now, Lemma 1 holds only for nodes u at the lowest $k \leq \frac{1}{3} \log n$ levels, so that 2^k is $O(n^{1/3})$. Summing up these levels, we obtain the total transfer $\Omega(c \cdot \frac{1}{3} \log n)$ and the amortized cost $\Omega(\log n)$ per operation. □