FIFO Indexes for Decomposable Problems

Cheng Sheng
CUHK
Hong Kong
csheng@cse.cuhk.edu.hk

Yufei Tao
CUHK
Hong Kong
taoyf@cse.cuhk.edu.hk

ABSTRACT

This paper studies *first-in-first-out* (FIFO) *indexes*, each of which manages a dataset where objects are deleted in the same order as their insertions. We give a technique that converts a static data structure to a FIFO index for all decomposable problems, provided that the static structure can be constructed efficiently. We present FIFO access methods to solve several problems including *half-plane search*, *nearest neighbor search*, and *extreme-point search*. All of our structures consume linear space, and have optimal or near-optimal query cost.

Categories and Subject Descriptors

F2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems; H3.1 [**Information storage and retrieval**]: Content analysis and indexing—*indexing methods*

General Terms

Algorithms, theory

Keywords

Index, half-plane search, nearest neighbor search, extreme-point search

1. INTRODUCTION

Given a set D of objects, a (searching) *problem* Π is to construct a data structure for D so that the result $\Pi(q, D)$ of any legal query q can be computed efficiently. If D never needs to be updated, the structure on D is *static*. In this paper, we are interested in that D is dynamically changing according to some *update scheme*. For instance, if only new objects can be added to D while existing ones are never removed, the structure on D is required to support only insertions, and is hence *insertion only*. If both insertions and deletions are possible, the structure is *fully dynamic*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS'11, June 13–15, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0660-7/11/06 ...$10.00.

We consider the *online* setting, where a query must be answered (i.e., having its result reported in entirety) before handling the next operation, no matter whether that operation is an insertion, a deletion, or a query. We will denote by N the problem size, and by K the output size, i.e., $N = |D|$ and $K = |\Pi(q, D)|$.

1.1 FIFO update scheme and its applications

We introduce the notion of *first-in-first-out* (FIFO) *index*. As with fully-dynamic indexes, a FIFO structure must handle both insertions and deletions. However, deletions are restricted — only the *oldest* object, namely the one inserted the longest time ago (among the objects still in the index), can be removed. This scheme is useful in several areas, as explained below, that have been actively studied in the last decade.

Spatio-temporal databases. With the proliferation of intelligent location-aware devices (e.g., mobile phones, GPS devices, etc.), it is nowadays feasible to support spatial queries (such as range searching, nearest neighbor retrieval, and so on) on a large set of continuously moving objects [25, 31]. A simple, effective, way to keep track of object locations[1] is to have every object report its position periodically at an appropriate interval [31] (e.g., 1 minute). In other words, each location stored at the server automatically expires in a minute, at which time it is replaced by a fresh update. Therefore, locations expire in the same order that they are inserted, making FIFO indexes the key to query processing in these environments.

Sliding windows over data streams. By definition, sliding windows, which have been extensively studied since the pioneering paper of Babcock et al. [11], are well-known examples of the FIFO update scheme. There are two common types of sliding windows. The first one, *tuple-based sliding window*, simply covers the N most recently-received records for a fixed parameter N. The second one, *time-based sliding window*, includes all the records that arrived at the system within t timestamps from now, for a fixed parameter t. The number of tuples in the window varies, as it depends on the data's arrival rate during the past t timestamps.

[1]We consider objects whose future locations cannot be accurately predicted by simple motion functions, which makes the update schemes of [3, 26] inapplicable. Examples of such objects are vehicles in urban areas, pedestrians, etc. See [25] and the references therein for more discussion.

Evolving databases. Massive transaction records are generated on a daily basis in various businesses (such as supermarkets, banks, stock exchanges, etc.) to support analytical report generation and decision making. For these purposes, user queries are biased towards recent records, such that data older than a certain past timestamp are no longer interesting and hence, can be discarded from the system. Coining the concept of *evolving databases* for such systems, Shivakumar and Garcia-Molina [28] presented a detailed coverage on the importance of the so-called *wave indexes* to answering queries efficiently. Such an index is in fact a FIFO structure by our definition. It is worth mentioning that an evolving database is similar to a *transaction time database* [27] but with the oldest data gradually purged (in their insertion order).

1.2 Technical motivations

A problem Π under the FIFO scheme can obviously be settled by a fully dynamic structure for Π. However, since FIFO is no harder than the fully-dynamic update scheme, we may be able to find a FIFO structure that is better than the fully-dynamic state of the art, in terms of the space, query, and/or update performance. This benefits the applications mentioned in Section 1.1 because it allows the practitioners there to improve their implementation without having to wait for a new fully-dynamic structure to come out.

A prominent example is *half-plane search* (its formal definition will appear in Section 1.3), whose static version can be solved with linear space and logarithmic query time [19] in the RAM model. Dynamization, however, appears to be rather difficult. The best known result [6] uses $O(N^{1+\epsilon})$ space, processes a query in $O(\log N + K)$ time, and requires $O(N^{\epsilon})$ time for both an insertion and a deletion. We will show that, under the FIFO scheme, the problem can be solved with a structure that consumes linear space, answers a query in $O(\log N + K)$ time, and supports an update with $O(\log^2 N)$ cost.

At a higher level, a more ambitious goal is to develop a generic framework of designing FIFO structures. In particular, assuming that we already know how to solve the *static* version of a problem, is it possible to utilize *just* that solution for FIFO indexing? Proving a positive answer to this question, which we manage to do in this work, gives us a good starting point to approach some FIFO problems, as will be demonstrated later.

1.3 Problems, computation models, and basic notations

In the sequel, we define several problems to be discussed in detail. The purposes of studying these problems are twofold. First, they will be utilized to demonstrate how to apply the proposed generic framework to derive FIFO indexes. Second, all of them are well-known problems in the database literature. At the end of the subsection, we will clarify the computation models for our complexity analysis.

- *Half-plane search:* the dataset D is a set of 2-d points, a query q is a half-plane, and its result $\Pi(q, D) = q \cap D$, i.e., all the points of D falling in q. This problem is fundamental to, for example, non-isothetic range searching [29] and searching with linear constraints [4].
- *Nearest neighbor* (NN) *search:* D is again a set of points in the plane, q is also a point, and $\Pi(q, D)$ is the data point $p \in D$ minimizing the Euclidean distance $\|q, p\|$. This problem has been extensively investigated on moving objects; see [25] and its bibliography.
- *Extreme-point search:* D is still a planar point set, q is a linear function $q((x, y)) = ax + by$, and $\Pi(q, D)$ is the point $p \in D$ maximizing $q(p)$. This is equivalent to *top-1 search*, which is crucial to multi-criteria optimization [15, 23].

See Figure 1 for some illustrations.

The above problems share the common property of being *decomposable*. Specifically, a problem Π is decomposable if, for any dataset D and query q, $\Pi(q, D)$ can be computed efficiently from $\Pi(q, D_1)$ and $\Pi(q, D_2)$, where D_1 and D_2 form a *partition* of D, namely, $D_1 \cup D_2 = D$ and $D_1 \cap D_2 = \emptyset$. The proposed indexing framework can be applied to all decomposable problems.

We are interested in both main-memory and I/O-efficient data structures. For the former, our analysis is under the standard *unit-cost RAM model*. For the latter, we adopt the *external memory* (EM) *model* [7], which has been successful in capturing the characteristics of database algorithms [30]. In this model, a computer has a memory of M words, and a disk of unbounded size that has been formatted into *blocks* of size B. An I/O operation transfers a block of information between the memory and the disk. Space complexity is measured in the number of blocks occupied, while time complexity is gauged in the number of I/Os performed. We make the *tall cache* assumption[2] [5, 10] that $M \geq B^2$. Note that, a *linear* complexity should be understood as $O(N/B)$ in the EM model, instead of $O(N)$ as in RAM. Furthermore, *poly-logarithmic* should be interpreted as $O(\log_B^c N)$ for some constant c in EM, as opposed to $O(\log^c N)$ in RAM. All complexities are worst-case unless specifically stated otherwise.

In RAM, we characterize the performance of a *static* structure by three functions (assuming that the dataset has size N):

- $S(N)$ which bounds from above its space consumption;
- $Q(N)$ which bounds from above the time of answering a query;
- $U(N)$ such that $N \cdot U(N)$ bounds from above the time of constructing the structure. In other words, $U(N)$ can be understood as the amount of preprocessing cost amortized over each object.

Similarly, in EM, a static structure is also characterized by $S(N)$, $Q(N)$ and $U(N)$, except that the construction time is bounded above by $(N/B) \cdot U(N)$. Note that, strictly speaking, the query cost should include the overhead of outputting the query result. However, since the output overhead is *always* linear (i.e., $O(K)$ and $O(K/B)$ in RAM and EM, respectively) for all the algorithms discussed in this paper, we omit the term from $Q(N)$.

[2]A typical value of B in practice is 1024 words, in which case $M \geq B^2$ means that the memory should have size at least 1 mega words. Most computers today (even those in smart phones) have memory larger than this.

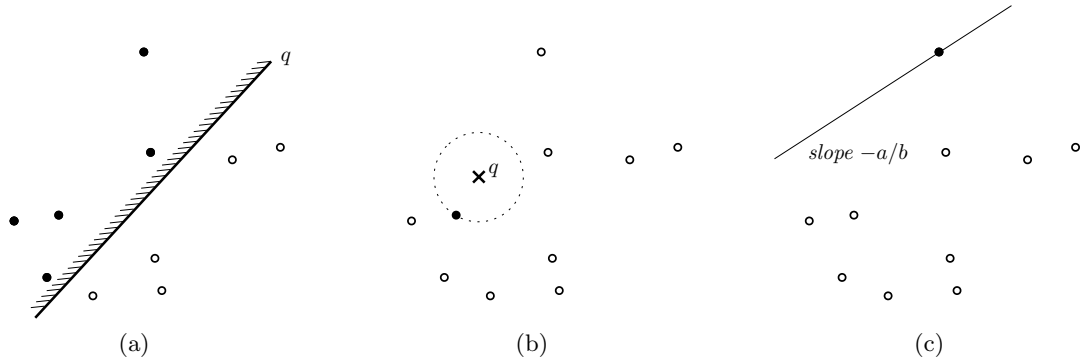

Figure 1: Illustrations of the problems studied: (a) half-plane search, (b) nearest neighbor search, and (c) extreme-point search. In each case, the query result includes the black dot(s). For (c), note that the answer to an extreme-point query with parameters (a, b) is the first data point hit as we move a line with slope $-a/b$ from infinity gradually towards the dataset.

For example, a binary tree uses $O(N)$ space, solves a 1-d range query in $O(\log N + K)$ time, and can be built in $O(N \log N)$ time. We have $S(N) = O(N), Q(N) = O(\log N)$, and $U(N) = O(\log N)$. Similarly, regarding the B-tree and 1-d range searching, $S(N) = O(N/B)$, $Q(N) = O(\log_B N)$, and $U(N) = O(\log_B N)$. In particular, note that the $U(N)$ indicates that the B-tree can be constructed in $O((N/B) \log_B N)$ I/Os.

We need to impose some constraints on $S(N)$, $Q(N)$, and $U(N)$ for our technique to work:

- All of them should be non-decreasing with N.
- $S(N)$ should grow at least linearly. Following [12], in RAM this means that $S(N)/N$ is non-decreasing. Accordingly, in EM, we require that $S(N)/(N/B)$ be non-decreasing with N.
- $U(N)$ needs to satisfy:

$$U(O(N)) = O(U(N)) \tag{1}$$

  which is fulfilled by a wide range of $U(N)$, including all the standard functions (e.g., constant, poly-logarithmic, polynomial, etc.).
- Both $Q(N)$ and $U(N)$ need to be $\Omega(1)$.

1.4 Previous results

Next, we survey the existing results related to our work. Our discussion consists of three parts. The first one describes the known general techniques for designing FIFO (or insertion-only) structures. The second clarifies the differences between FIFO and the *semi-online* update scheme of Dobkin and Suri [20]. Finally, the third part reviews the efficiency of the previous solutions to the problems listed in Section 1.3.

General techniques. Bentley and Saxe [12] initialized the research of developing general techniques to dynamize static structures for decomposable problems. They proposed several methods to convert static indexes to insertion-only structures (which can be regarded as special FIFO indexes where deletions never happen). The basic idea of their methods is to partition a dataset D into several subsets, each managed by a dedicated static structure. An insertion is handled by reorganizing some subsets and reconstructing their structures respectively. By the definition of decomposability, a query q can be correctly answered by issuing q on every subset, and then combining all results.

Bentley and Saxe presented three methods achieving various tradeoffs. Given a static structure with performance characterized by $S(N)$, $Q(N)$ and $U(N)$, their *binary transformation*, perhaps better known as the *logarithmic method* [10], yields an insertion-only index with $O(S(N))$ space, $O(Q(N) \log N)$ query cost, and $O(U(N) \log N)$ amortized insertion time. The other two transformations are less interesting for our purposes, as they yield either $O(Q(N)N^\epsilon)$ query time or $O(U(N)N^\epsilon)$ update time, where the multiplicative term N^ϵ is too expensive when $Q(N)$ and $U(N)$ are poly-logarithmic.

Arge and Vahrenhold [10] extended the logarithmic method to the EM model, achieving similar performance bounds. Specifically, the resulting insertion-only index consumes $O(S(N))$ space, answers a query in $O(Q(N) \log_B N)$ I/Os, and supports an insertion in $O(U(N) \log_B N)$ amortized cost.

Arasu and Manku [8] described a method we refer to as the *dyadic approach* for query processing on sliding windows. Although not explicitly mentioned in [8], this method can be modified to convert a static structure to a FIFO index in internal memory. The rationale is to index the *sequence number* of every object in D with a binary tree, and associate each node of the tree with a secondary, static, structure on all the objects in the node's subtree. Rebuilding of the secondary structures can be carried out in a manner that amortizes only a small amount of cost on every update. In general, the FIFO structure produced by the dyadic approach occupies $O(S(N) \log N)$ space, and supports a query in $O(Q(N) \log N)$ time, and an update in $O(U(N) \log N)$ amortized time. Note that there is an $O(\log N)$ blow-up in the space complexity. It is worth noting that this approach is not designed for the EM model. Naive adaptation will lead to the gigantic query cost of $O(Q(N) \cdot B \log_B N)$, with

	Model	Space	Query	Insertion	Deletion
logarithmic method [12]	RAM	$S(N)$	$Q(N)\log N$	$U(N)\log N$	no support
external logarithmic method [10]	EM		$Q(N)\log_B N$	$U(N)\log_B N$	
dyadic approach [8]	RAM	$S(N)\log N$	$Q(N)\log N$	$U(N)\log N$	$U(N)\log N$
external dyadic approach (adapted from [8])	EM	$S(N)\log_B N$	$B\cdot Q(N)\log_B N$	$U(N)\log_B N$	$U(N)\log_B N$
ours	**RAM**	$\boldsymbol{S(N)}$	$\boldsymbol{Q(N)\log N}$	$\boldsymbol{U(N)\log N}$	$\boldsymbol{U(N)\log N}$
	EM		$\boldsymbol{Q(N)\log_B N}$	$\boldsymbol{U(N)\log_B N}$	$\boldsymbol{U(N)\log_B N}$

Table 1: Comparison of the general techniques for deriving FIFO indexes. All complexities are in big-O. The update results are amortized while the others are worst case.

Problem	Model	Fully-dynamic				FIFO (by Theorem 1.1)			**FIFO (our best)**		
		Space	Query	Update	Ref	Space	Query	Update	**Space**	**Query**	**Update**
half-plane	RAM	$N^{1+\epsilon}$	$\log N+K$	N^{ϵ}	[6]	N	$\log^2 N+K$	$\log^2 N$	$\boldsymbol{N}$	$\boldsymbol{\log N+K}$	$\boldsymbol{\log^2 N}$
nearest	RAM	N	$\log^2 N$	$\log^6 N$	[14]	N	$\log^2 N$	$\log^2 N$	$\boldsymbol{N}$	$\boldsymbol{\log^2 N}$	$\boldsymbol{\log N}$
neighbor	EM		?			N/B	$\log_B^3 N$	$\log_B^2 N$	$\boldsymbol{N/B}$	$\boldsymbol{\log_B^3 N}$	$\boldsymbol{\log_B^2 N}$
extreme-	RAM	N	$\log N$	$\log N$	[13]	N	$\log^2 N$	$\log^2 N$	$\boldsymbol{N}$	$\boldsymbol{\log N}$	$\boldsymbol{\log N}$
point	EM		?			N/B	$\log_B^2 N$	$\log_B^2 N$	$\boldsymbol{N/B}$	$\boldsymbol{\log_B N}$	$\boldsymbol{\log_B N}$

? = we are not aware of any published result.

Table 2: Comparison of fully-dynamic and FIFO indexes. All complexities are in big-O. The update results are amortized while the others are worst case.

the space and update complexities being $O(S(N)\log_B N)$ and $O(U(N)\log_B N)$, respectively.

Semi-online. Dobkin and Suri [20] proposed a framework to convert a static structure to a dynamic one under a *semi-online* update scheme. Specifically, in that scheme, at the moment an object is inserted, we are told when the object will be deleted in the future. In particular, the future deletion time is given as the number of updates, *as opposed to* an absolute timestamp (e.g., by specifying 10, it means that the object will be deleted after 10 subsequent updates). This scheme is more general than FIFO in an *offline* setting, where all the update and query operations are given in advance, so that an algorithm can first scan the dataset once to attach each deletion time to its corresponding insertion.

In the online setting, however, the semi-online scheme does *not* subsume FIFO. Recall that, in this setting, every query must be answered before the algorithm can process the next operation. In other words, when answering a query q, the algorithm is not allowed to peek into the updates after q, and thus in general, cannot determine when an object o currently in the dataset will be removed – noticing that there can be an arbitrary number of insertions between q and the deletion of o. Unfortunately, knowledge of objects' deletion time is vital to the semi-online technique of [20], which thus becomes inapplicable. A FIFO algorithm, on the other hand, must work without knowing when objects will be deleted.

Concrete problems. For each problem in Section 1.3, we summarize below the performance of the best known static and fully-dynamic structures (if they exist, to our knowledge), under the computation model(s) where we will solve the problem in the FIFO scheme.

- Half-plane search: As mentioned in Section 1.2, in RAM, Chazelle et al. [19] gave a static solution that uses linear space and solves a query in $O(\log N + K)$ time. Their structure can be constructed in $O(N\log N)$ time. The dynamic state of the art is due to Agarwal and Matousek [6], having space, query, and update cost $O(N^{1+\epsilon})$, $O(\log N+K)$, and amortized $O(N^{\epsilon})$, respectively.

- NN search: In RAM, the static version can be settled in linear space and logarithmic query time, by combining a Voronoi diagram with a point-location structure. Its dynamization had been a long-standing open problem until recently Chan [14] proposed an index that occupies linear space (combining with ideas due to Afshni and Chan [1]), achieves $O(\log^2 N)$ query cost, and demands $O(\log^3 N)$ and $O(\log^6 N)$ amortized time to handle an insertion and deletion respectively.

  In EM, Goodrich et al. [21] showed that the Voronoi diagram of N points can be constructed in $O((N/B)\log_B N)$ I/Os. Agarwal et al. [2] presented a point-location structure that requires linear space, answers a query in $O(\log_B^2 N)$ I/Os, and can be built in $O((N/B)\log_B N)$ I/Os. Combining these results gives a static index for NN search[3].

- Extreme-point search. In RAM, for static data, an extreme-point query can be answered in logarithmic time utilizing a linear-space index. This can be achieved by first computing the dataset's convex hull in $O(N\log N)$ time (with, for example, *Graham scan* [22]), and then, building a binary tree on the points of the hull. The dynamic case has been solved by Brodal and Jacob [13], whose structure achieves linear space and logarithmic query and amortized update time. Their data structure is rather complex such that the authors themselves listed a practical solution with the same complexity as an open problem.

  In EM, replacing the binary tree with a B-tree gives a static structure that occupies linear space and an-

[3]Arge et al. [9] gave an alternative linear-size index that solves a point-location query in $O(\log_B N)$ I/Os. Unfortunately, their structure is expensive to construct, and thus is not suitable as the base structure for FIFO conversion.

swers a query in $O(\log_B N)$ cost. The structure can be constructed in $O((N/B)\log_B N)$ I/Os by externalizing Graham scan [21].

1.5 Our results

Our first main result is a generic framework that converts a static solution of any decomposable problem Π into an index for solving the FIFO version of Π:

THEOREM 1.1 (THE FIFO THEOREM). *Assume that there is a static structure solving a decomposable problem Π in RAM that consumes no more than $S(N)$ space, answers a query within $Q(N)$ time, and can be built within $N \cdot U(N)$ time. Then, there is a FIFO index for Π with the following performance guarantees:*

> *If the dataset currently has N objects, the index consumes $O(S(N))$ space, solves a query in $O(Q(N)\log N)$ time, and supports an update in $O(U(N)\log N)$ amortized time.*

The same result holds in EM except that (i) the construction time of the static index should be bounded above by $(N/B)\cdot U(N)$, and (ii) each $\log N$ should be replaced with $\log_B N$.

Our technique extends the logarithmic method of Bentley and Saxe [12] with a *2-phase-indexing* idea. More specifically, we partition the dataset D into $2\log N$ subsets with different sizes, based on objects' insertion times. Among these subsets, $\log N$ of them contain objects ready for deletion — these objects are in the *deletion phase*. The rest of the objects are in the *insertion phase*. We give algorithms to merge and split the subsets, so that the amortized cost of an update can be minimized. The performance bounds are derived through an analysis that is substantially different from that of [12].

Table 1 compares the FIFO indexes produced by Theorem 1.1 to those by the existing frameworks. In particular, regarding the dyadic approach [8], the proposed technique (i) reduces the space by a logarithmic factor in both RAM and EM, and (ii) improves the query efficiency by a factor of $O(B)$ in EM.

Theorem 1.1 immediately gives non-trivial FIFO indexes for all the problems in Section 1.3, by combining with the state-of-the-art static structures surveyed in Section 1.4. Each of those FIFO indexes treats the underlying static structure as a *black box*. Our second contribution is to show that we can sometimes obtain more efficient FIFO access methods by *hacking* into those black boxes. More specifically, in RAM, the objective of hacking is to achieve a query bound of $o(Q(N)\log N)$ and/or an update bound of $o(U(N)\log N)$, thus improving the results in Theorem 1.1. An analogous objective exists in EM as well.

Table 2 summarizes our results. It also includes the performance of the best corresponding fully-dynamic structure, to echo the motivation mentioned in Section 1.2 that better solutions may be derived by leveraging the special properties of the FIFO scheme. For extreme-point search in RAM, although having the same performance as that of [13], our structure is simpler, and amenable to practical implementation. All logarithmic query cost (with linear output time) in Table 2 is optimal in the comparison model of computation.

2. MAKING A STATIC STRUCTURE FIFO

Given a static structure for settling a decomposable problem with performance characterized by $S(N)$, $Q(N)$ and $U(N)$, we will present a generic framework to obtain a FIFO index. Section 2.1 describes the framework for the RAM model, which will be extended to the EM model in Section 2.2.

2.1 The RAM model

We say that an object is *active* if it has been inserted but not deleted yet. At any moment, we assign a *phase* to every object o. Specifically, when o is inserted, it enters the *insertion phase*. Sometime later, we will switch o into the *deletion phase*, in which the object stays until its deletion.

Denote by D the set of all objects currently active. Set $N = |D|$. The objects of D can be naturally ordered by their insertion timestamps. We say that an object is *newer* (*older*) than another, if the former one has greater (lower) insertion time. Let D^+ (D^-) be the subset of D that includes all the objects in the insertion (deletion) phase. We partition D^+ (D^-) into subsets with the following properties:

- D^+ is divided into D_0^+, D_1^+, ..., $D_{h^+}^+$, where h^+ is an integer that is $O(\log N)$ and varied by our algorithm. The size of D_i^+ ($0 \le i \le h^+$) is either 0 or 2^i. Furthermore, for any $i < j$, each object in D_i^+ is newer than all the objects in D_j^+.
- D^- is divided into $D_{h^-}^-$, ..., D_1^-, D_0^-, where $h^- = O(\log N)$ is also varied by our algorithm. The size of D_i^- ($0 \le i \le h^-$) is either 0 or 2^i. For any $i < j$, each object in D_i^- is older than all objects in D_j^-.
- $h^+ \le h^-$.

See Figure 2 for an illustration. We say that a subset is *older* (*newer*) than another if the objects in the former subset are older (newer). Each of the $h^+ + h^- + 2$ subsets is indexed with a static structure. Furthermore, for each subset, we keep a chronological list of its objects.

At the beginning (when D is empty), both h^+ and h^- equal 0. Next, we explain how to perform updates efficiently.

Insertion. Our insertion algorithm is the same as in the logarithmic method [12]. Given an object o, we identify the smallest $i \le h^+$ such that D_i^+ is empty. In case such an i does not exist (i.e., D_0^+, ..., $D_{h^+}^+$ are all non-empty), increment h^+ by 1 and set i to the new h^+. We empty all of D_0^+, ..., D_{i-1}^+, discard the structures on them, add all the objects therein together with o to D_i^+, and create a new static index on D_i^+. Note that the total number of migrated objects is

$$1 + \sum_{j=0}^{i-1} |D_j^+| = 1 + \sum_{j=0}^{i-1} 2^j = 2^i,$$

which is exactly the desired size of D_i^+. The construction takes $O(2^i \cdot U(2^i))$ time. If now h^+ has exceeded h^-, we simply rename $D_{h^+}^+$ as $D_{h^-+1}^-$, after which h^- is increased by 1, and h^+ is reset to 0.

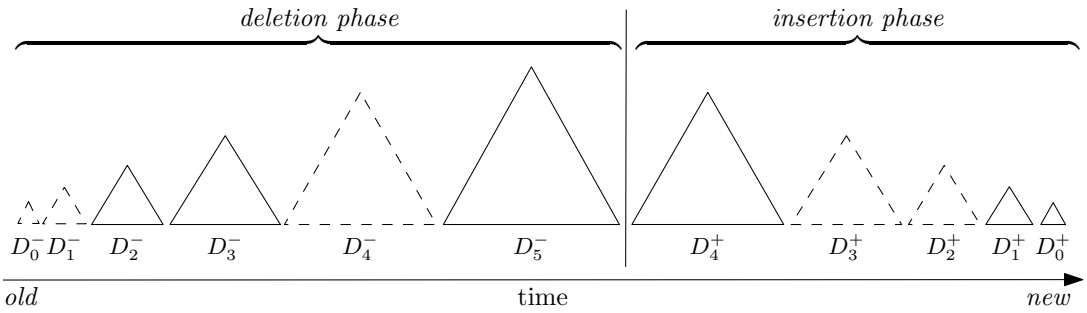

Figure 2: Illustration of 2-phase indexing: Solid triangles represent non-empty structures while dashed triangles represent empty ones; $h^+ = 4$ and $h^- = 5$.

Deletion. To remove the oldest object o in D, we find the smallest $i \leq h^-$ such that D_i^- is non-empty. Find and discard o from the chronological list of D_i^- in constant time. The remaining objects of D_i^- are then broken into D_0^-, ..., D_{i-1}^- in $O(|D_i^-|) = O(2^i)$ time, observing the time-ordering and cardinality constraints mentioned earlier. Build a static structure on each of D_0^-, ..., D_{i-1}^- in totally

$$\begin{aligned}\sum_{j=0}^{i-1} |D_j^-| \cdot U(|D_j^-|) &= \sum_{j=0}^{i-1} 2^j \cdot U(2^j) \\ &= O(2^i \cdot U(2^i))\end{aligned}$$

time. If $i = h^-$, we decrease h^- by 1. It is possible that h^- now drops below h^+. In this case, rename $D_{h^+}^+$ as $D_{h^-+1}^-$, increase h^- by 1, and decrease h^+ to the greatest j such that D_j^+ is non-empty (if no such j exists, $h^+ = 0$).

Query. We answer a query by simply searching the indexes of all subsets and combining their results. Correctness is obvious as the underlying problem is decomposable. The query time is

$$\begin{aligned}\sum_{i=0}^{h^+} Q(2^i) + \sum_{i=0}^{h^-} Q(2^i) &= O\Big(\big(h^+ + h^-\big)\, Q(N)\Big) \\ &= O\Big(Q(N) \log N\Big). \qquad (2)\end{aligned}$$

Analysis. If no deletion ever happens, our 2-phase framework degenerates into the logarithmic method [12]. Unfortunately, the analysis of [12] cannot be extended to prove the performance guarantees (in particular, the update time) of our technique. In fact, in the absence of deletions, the value of N monotonically increases with time, thus allowing an insertion to use more time than each of the earlier insertions. This property can be leveraged to simplify the analysis significantly. In the FIFO context, N can vary *arbitrarily* with time, which necessitates a more delicate discussion, as presented below.

The query cost of 2-phase indexing is already given in Equation 2. Clearly all the $h^+ + h^- + 2$ structures consume

$$\sum_{i=0}^{h^+} S\big(|D_i^+|\big) + \sum_{i=0}^{h^-} S\big(|D_i^-|\big) = O(S(N))$$

space, where the equality used the fact that function $S(N)$ grows at least linearly.

Next, we focus on bounding the update overhead. Initially, D is empty. Consider a FIFO update sequence SEQ with n updates, each of which can be an insertion or a deletion. Denote by N_i ($i \leq n$) the size of D after the i-th update. Our objective is to show that the total cost of handling all the updates is bounded by

$$cost(SEQ) = O\left(\sum_{i=0}^{n} U(N_i) \log N_i\right). \qquad (3)$$

This is what is needed to prove that the i-th update has amortized cost $O(U(N_i) \log N_i)$.

Let n_{del} be the number of deletions in SEQ. It follows that $n_{ins} = n - n_{del}$ objects are inserted. After processing the entire SEQ, the oldest n_{del} objects have been removed from D. We say that they are *short-term* objects. The other $n_{ins} - n_{del}$ objects are *long-term*.

We define the *timestamp* of an update in SEQ as its sequence number (i.e., the first update is at time 1, the second at time 2, and so on). If o is a short-term object, denote by $t^+(o)$ and $t^-(o)$ the timestamps at which o is inserted and deleted, respectively. If o is long-term, we still use $t^+(o)$ to represent its insertion time, and assume a *virtual* deletion of o at time:

$$t^-(o) = n + i \qquad (4)$$

where i is such that o is the i-th oldest long-term object in SEQ. Remember that $cost(SEQ)$, as in Equation 3, does not include the cost of virtual deletions. Regardless of whether o is long- or short-term, we define its *lifespan* as $[t^+(o), t^-(o))$. Note that a lifespan is close on the left and open on the right.

If o is short-term, let $\Delta^+(o)$ be the number of insertions during the lifespan of o, excluding the insertion of o itself. Similarly, let $\Delta^-(o)$ be the number of deletions during the lifespan. If o is long-term, the semantics of $\Delta^+(o)$ is the same, while $\Delta^-(o)$ is defined as:

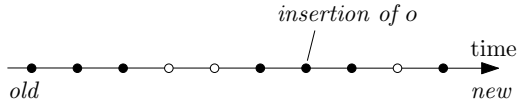

Figure 3: Illustration of the notations of a long-term object: each black (white) dot represents an insertion (deletion) in *SEQ*. Let *o* be the object as shown, which is the 2nd oldest long-term object. We have: $t^+(o) = 7$, $t^-(o) = 10 + 2 = 12$, $s(o) = 5$, $\Delta^+(o) = 2$, $\Delta^-(o) = 1 + 2 - 1 = 2$.

$$\Delta^-(o) = c + i - 1 \tag{5}$$

where c counts the number of deletions in SEQ after the insertion of o, and i is the same as in Equation 4. Finally, no matter whether o is short- or long-term, we define its *spread* as

$$s(o) = t^-(o) - t^+(o).$$

which is essentially the number of timestamps in the lifespan of o. By the definitions of $\Delta^+(o)$ and $\Delta^-(o)$, it is not hard to observe:

$$s(o) = \Delta^+(o) + \Delta^-(o) + 1. \tag{6}$$

Figure 3 illustrates the above notations for a long-term o.

Some useful observations can be made:

- Property 1: $N_{t^+(o)} = \Delta^-(o) + 1$. When o is inserted, D contains only those objects inserted earlier but not deleted yet. All and only such objects have deletion timestamps in the lifespan of o, by definition of FIFO updating.
- Property 2: for a short-term object o, $N_{t^-(o)} = \Delta^+(o)$. This is because only the objects (not counting o itself) inserted during the lifespan of o can still remain in D right after o is deleted.
- Property 3: for a short-term object o, $s(o) = N_{t^+(o)} + N_{t^-(o)}$, which immediately follows the previous two properties and Equation 6.
- Property 4: $N_i \leq s(o)$ for all i satisfying $t^+(o) < i < t^-(o)$. The reason is that, if another object o' coexists with o in D at any moment, their lifespans must intersect. Hence, either $t^+(o')$ or $t^-(o')$ falls within the lifespan of o, making $\Delta^+(o) + \Delta^-(o)$ an upper bound of the size of $D - \{o\}$ for all D containing o.

LEMMA 2.1.

$$cost(SEQ) = \sum_{i=1}^{n_{ins}} O\Big(U\big(s(o_i)\big) \log s(o_i)\Big)$$

where o_i is the i-th oldest object ever inserted.

PROOF. Every time a static structure in the insertion (deletion) phase is constructed (destroyed), the cost of processing SEQ increases by at most $x \cdot U(x)$, where x is the number of objects in the structure. We charge $O(U(x))$ on every object o in the structure. By Property 4, $x \leq s(o)$; hence, $O(U(x))$ is bounded above by $O(U(s(o)))$. Every time o is charged, it migrates to an older subset. Since o cannot appear in a D with more than $s(o)$ objects (otherwise, N would be greater than $s(o)$), it can only be charged $O(\log s(o))$ times. Therefore, the total cost of processing the whole SEQ that is amortized over o is bounded by $O\left(U\big(s(o)\big) \log s(o)\right)$. □

We are finally ready to establish Equation 3.

LEMMA 2.2.

$$\sum_{i=1}^{n_{ins}} U\big(s(o_i)\big) \log s(o_i) = O\left(\sum_{i=0}^{n} U(N_i) \log N_i\right) \tag{7}$$

PROOF. Introduce $f(x) = U(x) \log x$. From Equation 1, we have $f(O(x)) = O(f(x))$. Therefore, it holds that:

$$f(x + y) = \Theta(f(x) + f(y)) \tag{8}$$

To prove this, assume, without loss of generality, $x \leq y$. Hence, $f(x+y) = f(O(y)) = O(f(y)) = O(f(x)+f(y))$. On the other hand, since function f is non-decreasing, $f(x) + f(y) \leq 2f(x + y) = O(f(x + y))$.

Let $\mathcal{S}$ ($\mathcal{L}$) be the set of short- (long-) term objects. Each object o ever inserted by SEQ contributes a term $f(s(o))$ in the left hand side (LHS) of Equation 7. On the other hand, every object in $\mathcal{S}$ and $\mathcal{L}$ contributes two and one term in the right hand side (RHS), respectively.

For each object $o \in \mathcal{S}$, we have

$$f(s(o)) = O\Big(f\big(N_{t^+(o)}\big) + f\big(N_{t^-(o)}\big)\Big) \tag{9}$$

To see this, notice that Property 3 implies that either $N_{t^+(o)}$ or $N_{t^-(o)}$ is $\Omega(s(o))$. Without loss of generality, suppose that $N_{t^+(o)} \leq N_{t^-(o)}$. Then, $f(s(o)) = f(O(N_{t^-(o)})) = O(f(N_{t^-(o)}))$, validating Equation 9.

There are $n_{ins} - n_{del}$ long-term objects, among which let o_i be the i-th oldest. In the sequel, we will show

$$\sum_{i=1}^{n_{ins}-n_{del}} f\big(s(o_i)\big) = \sum_{i=1}^{n_{ins}-n_{del}} O\Big(f\big(N_{t^+(o_i)}\big)\Big) \tag{10}$$

Let δ_i be the number of *short-term* objects deleted after the insertion of o_i. We observe:

$$N_{t^+(o_i)} = i + \delta_i$$

which is because, right after o_i is inserted, D consists of exactly i *long-term* objects and δ_i short-term objects. We also observe:

$$s(o_i) = n_{ins} - n_{del} + \delta_i$$

due to the fact that the lifespan of o_i covers (i) either the insertion or deletion time of every long-term object, and (ii) the deletion timestamps of δ_i short-term objects.

Therefore:

$$\begin{aligned}\text{LHS of (10)} &= \sum_{i=1}^{n_{ins}-n_{del}} f\big(n_{ins}-n_{del}+\delta_i\big)\\ &= \sum_{i=1}^{n_{ins}-n_{del}} O\Big(f\big(n_{ins}-n_{del}-i\big)+f(i)+f(\delta_i)\Big)\\ &= \sum_{i=1}^{n_{ins}-n_{del}} O\Big(2f(i)+f(\delta_i)\Big)\\ &= \sum_{i=1}^{n_{ins}-n_{del}} O\Big(f(i+\delta_i)\Big) = \text{RHS of (10)}\end{aligned}$$

where the 2nd and 4th equalities used Equation 8.

From Inequalities 9 and 10, we know that

$$\begin{aligned}\text{LHS of (7)} &= \sum_{o\in\mathcal{S}} f(s(o)) + \sum_{o\in\mathcal{L}} f(s(o))\\ &= \sum_{o\in\mathcal{S}} O\Big(f\big(N_{t^+(o)}\big)+f\big(N_{t^-(o)}\big)\Big)+\\ &\quad \sum_{o\in\mathcal{L}} O\Big(f\big(N_{t^+(o)}\big)\Big) = \text{RHS of (7)}\end{aligned}$$

thus completing the proof. □

This proves the part of Theorem 1.1 in the RAM model.

2.2 The EM model

In external memory, our 2-phase indexing divides D into D_0^+, ..., $D_{h^+}^+$, $D_{h^-}^-$, ..., D_0^- in the same manner as in the RAM model, except for two differences:

- Both h^+ and h^- are $O(\log_B N)$.
- For each $i \le h^+$ or h^-, $|D_i^+|$ or $|D_i^-|$ should
  - be either 0 or
  - fall in the range $[B^i/2, B^{i+1}]$.

Insertion of an object o is similar to that in the external logarithmic method [10]. We first find the smallest i such that the union of all D_j^+ $(0 \le j \le i)$ has less than B^{i+1} objects. Then, empty D_0^+, ..., D_i^+, collect all their objects together with o into D_i^+, and construct a static index on D_i^+. In case the requirement $h^+ \le h^-$ is violated, carry out renaming and adjust h^- and h^+ as in the RAM model.

To delete an object o, we identify the lowest i such that D_i^- is non-empty, and discard o from D_i^-. The remaining objects of D_i^- are then divided into D_0^-, D_1^-, ..., D_i^- (obeying the time-ordering constraint) such that D_j^- $(0 \le j \le i-1)$ gets exactly $B^{j+1}/2$ objects. If there are not enough objects to achieve the purpose, we allow D_{i-1}^- to include less than $B^i/2$ objects; otherwise, the outstanding objects are put back in D_i^-. In any case, let $k \le i$ be the maximum integer such that that D_k^- is non-empty (k must be either $i-1$ or i). In case $|D_k^-| \le B^k/2$, empty D_k^- and move all its objects to D_{k-1}^-, whose size is now $B^k/2 + |D_k^-| \le B^k$. Construct a static index on D_0^-, ..., D_{k-1}^- respectively, and also on D_k^- if it is non-empty. If h^+ now exceeds h^-, fix it in the same way as in the RAM model

The query algorithm is identical to the RAM counterpart.

Analysis. We discuss only the update complexity as the space and query cost can be derived easily. All the notations in the analysis of the RAM model can be re-used here. The following is the EM version of Lemma 2.1.

LEMMA 2.3.

$$cost(SEQ) = \sum_{i=1}^{n_{ins}} O\Big(U\big(s(o_i)\big)\log_B s(o_i)\Big)$$

where o_i is the i-th oldest object ever inserted.

PROOF. In the insertion (deletion) algorithm, every time the static structure on a D_i^+ (D_i^-) is constructed (destroyed), the cost of processing SEQ grows by at most $(x/B)\cdot U(x)$ I/Os, where x is the number of objects in the structure. Note that $(x/B)\cdot U(x) = O(B^i \cdot U(x))$. Next, we will show that $\Omega(B^i)$ objects will migrate to an older subset. This allows us to charge $O(U(x))$ I/Os over each of those objects, which will complete the proof as an object o can be charged $O(\log_B s(o))$ times.

For the insertion case, all the objects in D_0^+, ..., D_{i-1}^+ will migrate to D_i^+. The selection of i ensures that there are at least B^i such objects. For the deletion case, consider the situation after the deletion has finished. If D_i^- becomes empty, all the $|D_i^-| \ge B^i/2$ objects in D_i^- have migrated to older subsets. Otherwise, at least $B^i/2$ objects have migrated to D_{i-1}^-. □

Changing each log to $\log_B$, Lemma 2.2 still holds (in fact, the proof directly applies after the changes). This indicates that the i-th update is handled with $O(U(N_i)\log_B N_i)$ I/Os amortized. We thus conclude the proof of Theorem 1.1.

3. SOLVING CONCRETE PROBLEMS

In this section, we present FIFO indexes for the problems defined in Section 1.3. Integration of Theorem 1.1 and the static solutions reviewed in Section 1.4 immediately gives the results in the fourth column of Table 2. The subsequent discussion will show how to obtain the results in the last column of that table. Towards this, We will first give two general techniques for improving the FIFO structures yielded by Theorem 1.1, and then, apply them to solve each individual problem.

FIFO fractional cascading. Recall that our 2-phase indexing framework divides the set D of N active objects into $h^+ + h^- + 2$ subsets D_0^+, ..., $D_{h^+}^+$, $D_{h^-}^-$, ..., D_0^-. Each subset is managed by a static structure. Next, we consider the following problem. Let G_i^+ $(0 \le i \le h^+)$ be a set of $O(|D_i^+|)$ real values. G_i^+ has the property that it can be retrieved from D_i^+ into the sorted order using linear time. Define G_i^- $(0 \le i \le h^-)$ similarly on D_i^-. Given a real value q, a *combined predecessor query* returns, from each possible G_i^+ (G_i^-), the predecessor of q, namely, the greatest value in G_i^+ (G_i^-) not exceeding q.

LEMMA 3.1. *In RAM, a FIFO index can be augmented such that a combined predecessor query can be answered*

in $O(\log N)$ time. The augmentation does not alter the space, query, and update complexities of the original index. The same result holds in EM by replacing $O(\log N)$ with $O(\log_B N)$.

PROOF. We achieve this with a manipulation of *fractional cascading* (FC) [18]. In RAM, we build two FC structures F^+ and F^-, where the former indexes G_0^+, ..., $G_{h^+}^+$, and the latter indexes G_0^-, ..., $G_{h^-}^-$. Specifically, F^+ has $h^+ + 1$ sorted lists H_0^+, ..., $H_{h^+}^+$ where

- $H_{h^+}^+$ includes a value out of every 4 consecutive values in $G_{h^+}^+$;
- for each $i = h^+ - 1, ..., 0$, H_i^+ includes a value out of every 4 consecutive values in the sorted list of $G_i^+ \cup H_{i+1}^+$.

Note that an H_i^+ can be non-empty even if G_i^+ is empty, as H_i^+ may contain values from G_j^+ with $j > i$. We refer to the constant 4 as the *sample interval*. An analogous definition applies to F^-. By standard techniques [18], F^+ and F^- can be deployed to answer a combined predecessor query in $O(\log N + h^+ + h^-) = O(\log N)$ time.

The two FC structures occupy linear space. In fact, regarding F^+, the following holds for any i satisfying $0 \leq i \leq h^+ - 1$:

$$|H_i^+| \leq \left(|H_{i+1}^+| + |G_i^+|\right)/4 \leq |H_{i+1}^+|/4 + c2^{i-2} \qquad (11)$$

for some constant c. Obviously, $|H_{h^+}^+| \leq c2^{h^+-2}$. In general, given $H_{i+1}^+ < c2^i$, Equation 11 shows:

$$|H_i^+| < c2^i/4 + c2^{i-2} < c2^{i-1}.$$

Hence, we can create H_i^+ from G_i^+ and H_{i+1}^+ in $O(|G_i^+| + |H_{i+1}^+|) = O(2^i)$ time. Similar arguments apply to F^- as well.

To maintain the FC structures, we extend our insertion algorithm in Section 2.1 as follows. Consider the moment when the index on D_i^+ has been rebuilt (for some legal i), namely, D_0^+, ..., D_{i-1}^+ have been emptied. We derive the new G_i^+ in $O(|D_i^+|)$ time, compute H_i^+ in $O(2^i)$ time, and then create the new $H_{i-1}^+, ..., H_0^+$ (in this order) using $\sum_{j=0}^{i-1} O(2^j) = O(2^i)$ time. Hence, the total insertion cost is the same as before if the requirement $h^+ \leq h^-$ is not violated. In case h^+ exceeds h^-, denote by x the number of objects currently in $D_{h^+}^+$. We perform renaming and adjust h^+, h^- as in the original algorithm. This is followed by rebuilding both F^+ and F^- from scratch. As all G_i^+ and G_i^- are sorted, F^+ and F^- can be constructed in $O(N)$ time. We amortize the cost over the x objects that just migrated from the insertion phase to the deletion phase. As $x = \Omega(N)$, each object bears only constant time. Since every object is charged at most once this way, the amortized update cost is unaffected. The deletion algorithm in Section 2.1 can be modified based on analogous ideas without any extra overhead asymptotically.

A similar approach also works in EM, using a sample interval of $2B$. Adapting the above analysis in a straightforward manner, we can show that the resulting FC structures occupy linear space, answer a combined predecessor query in $O(\log_B N)$ I/Os, and their maintenance demands no extra cost on top of the original update overhead. □

Linear reorganization. Let S be a set of active objects. Consider a *chronological partition* S_1, S_2 of S, namely, (i) S_1, S_2 form a partition of S, and (ii) every object in S_1 is older than all objects in S_2. We say that the underlying static structure is *linearly reorganizable* if it satisfies both conditions below:

- given a structure on S, we can create the structures on S_1 and S_2 respectively in total time linear to $|S|$.
- given the structures on S_1 and S_2 respectively, we can create the structure on S in time linear to $|S|$.

LEMMA 3.2. *If the underlying static structure is linearly reorganizable, the update cost of 2-phase indexing is $O(\log N)$ and $O(\log_B N)$ amortized in RAM and EM, respectively.*

PROOF. We prove the lemma only in RAM because the discussion extends to external memory easily. The key is to accelerate each insertion and deletion by a factor of $O(U(N))$, compared to the algorithms in Section 2.1.

Recall that insertion requires constructing the index of a D_i^+ on the union of D_j^+ for all $0 \leq j \leq i-1$. This can be done in $O(2^i)$ time (as opposed to $O(2^i \cdot U(2^i))$ in Section 2.1) as follows. First, build an index T from the structures on D_0^+ and D_1^+. Then, incrementally, create a new T from the previous T and the structure on D_j^+, for $j = 2, ..., i-1$. The final T is the index on D_i^+. By the definition of linear reorganizable, this process takes

$$\sum_{j=1}^{i-1} O(2^j + 2^{j-1}) = O(2^i)$$

time. A similar, but reverse, approach works for deletion to achieve the claimed efficiency. □

Applications. The above techniques serve different purposes. Specifically, FIFO fractional cascading may be leveraged for reducing the query overhead, whereas linear reorganization for lowering the update cost. They are independent in the sense that sometimes they can be applied together to enhance query and update efficiency simultaneously. Next, we demonstrate this by improving the FIFO indexes directly obtained from Theorem 1.1.

- Half-plane search: In RAM, Chazelle et al. [19] showed that a half-plane query can be reduced to finding the predecessor of a value (computed based on the original query) in a set G of real numbers, such that $|G|$ is no greater than the cardinality of the dataset. Once the predecessor is found, all the result objects can be reported in $O(K)$ time. G can be extracted into a sorted order in linear time after constructing the (static) index of [19]. In 2-phase indexing, query processing boils down finding the predecessors of a query value, in $h^+ + h^- + 2$ sets of real values respectively, i.e., one for each of D_0^+, ..., $D_{h^+}^+$, $D_{h^-}^-$, ..., D_0^-. By Lemma 3.1, the query cost is $O(\log N + K)$.

- NN search: In RAM, the static version of the problem can be settled by constructing a point-location structure on the Voronoi diagram (VD) of the dataset. One such structure is due to Kirkpatrick [24]. Given a VD, his structure can be built in linear time. In the FIFO context, let S be a set of points, and S_1, S_2 a chronological partition of it. Using the techniques of Chazelle [16] and Chazelle et al. [17], we can (i) create the VDs of S_1 and S_2 respectively from that of S in $O(|S|)$ expected time, and (ii) create the VD of S from those of S_1 and S_2 also in $O(|S|)$ time. Therefore, the static NN structure is linearly reorganizable. By Lemma 3.2, the update cost can be lowered to $O(\log N)$ amortized.
- Extreme-point search: In RAM, the static structure is merely a binary tree indexing the points on the convex hull of the dataset, sorted in clockwise order. A query can be reduced to predecessor search on the tree. To make the structure linearly reorganizable, we keep a separate *x-list* that sorts *all* data points by their x-coordinates. By feeding the x-list to Graham scan, we can compute the convex hull in time linear to the dataset size. Now, consider a set S of points, and a chronological partition S_1, S_2. To obtain the structure of S_1 (the case of S_2 is similar) from S, we scan the x-list of S once to generate the x-list of S_1 in $O(|S|)$ time, after which the structure on S_1 can be built in $O(|S_1|)$ time. Conversely, to create the structure of S from those of S_1, S_2, we obtain the x-list of S by merging those of S_1 and S_2 in $O(|S|)$ time, after which the index on S is constructed with linear cost.

  Given the FIFO index from Theorem 1.1, extreme-point search is performed by first retrieving the predecessors of a query value in $h^+ + h^- + 2$ sets of numbers, and then returning the best one among the $O(\log N)$ objects corresponding to those predecessors. By Lemma 3.1, this can be done in $O(\log N)$ time overall. Finally, Lemma 3.2 allows us to improve the update time to $O(\log N)$ amortized. A similar approach works in the EM model to achieve logarithmic query and amortized update cost.

4. CONCLUSIONS

We have presented a general framework of designing FIFO data structures for decomposable problems, provided that an index solving the static version of the underlying problem is available. Furthermore, as long as the static index can be efficiently constructed, the resulting FIFO structure is simple enough for practical use. We have utilized our framework to develop FIFO indexing solutions with non-trivial performance guarantees to half-plane search, nearest neighbor search, and extreme-point search (all in 2-d space).

5. ACKNOWLEDGEMENTS

This work was supported by grants GRF 4173/08, GRF 4169/09, and GRF 4166/10 from HKRGC. We are grateful to the anonymous reviewers for their comments on how to improve the paper.

6. REFERENCES

[1] P. Afshani and T. M. Chan. Optimal halfspace range reporting in three dimensions. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 180–186, 2009.

[2] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–20, 1999.

[3] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Journal of Computer and System Sciences (JCSS)*, 66(1):207–243, 2003.

[4] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences (JCSS)*, 61(2):194–216, 2000.

[5] P. K. Agarwal, L. Arge, J. Yang, and K. Yi. I/O-efficient structures for orthogonal range-max and stabbing-max queries. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 7–18, 2003.

[6] P. K. Agarwal and J. Matousek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.

[7] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[8] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.

[9] L. Arge, A. Danner, and S.-M. Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8, 2003.

[10] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2):147–162, 2004.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[12] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[13] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 617–626, 2002.

[14] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *Journal of the ACM (JACM)*, 57(3), 2010.

[15] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The Onion technique:

Indexing for linear optimization queries. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 391–402, 2000.

[16] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM Journal of Computing*, 21(4):671–696, 1992.

[17] B. Chazelle, O. Devillers, F. Hurtado, M. Mora, V. Sacristan, and M. Teillaud. Splitting a delaunay triangulation in linear time. *Algorithmica*, 34(1):39–46, 2002.

[18] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[19] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT Numerical Mathematics*, 25(1):76–90, 1985.

[20] D. P. Dobkin and S. Suri. Maintenance of geometric extrema. *Journal of the ACM (JACM)*, 38(2):275–298, 1991.

[21] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[22] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters (IPL)*, 1(4):132–133, 1972.

[23] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal*, 13(1):49–70, 2004.

[24] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12(1):28–35, 1983.

[25] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 634–645, 2005.

[26] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 331–342, 2000.

[27] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[28] N. Shivakumar and H. Garcia-Molina. Wave-indices: Indexing evolving databases. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 381–392, 1997.

[29] S. Sioutas, D. Sofotassios, K. Tsichlas, D. Sotiropoulos, and P. Vlamos. Canonical polygon queries on the plane: A new approach. *Journal of Computers*, 4(9):913–919, 2009.

[30] J. S. Vitter. Algorithms and data structures for external memory. *Foundation and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[31] O. Wolfson, L. Jiang, A. P. Sistla, S. Chamberlain, N. Rishe, and M. Deng. Databases for tracking mobile units in real time. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 169–186, 1999.