Indexing for Summary Queries: Theory and Practice

KE YI, Tsinghua University and Hong Kong University of Science and Technology
LU WANG, Hong Kong University of Science and Technology
ZHEWEI WEI, MADALGO and Aarhus University

Database queries can be broadly classified into two categories: reporting queries and aggregation queries. The former retrieves a collection of records from the database that match the query's conditions, while the latter returns an aggregate, such as count, sum, average, or max (min), of a particular attribute of these records. Aggregation queries are especially useful in business intelligence and data analysis applications where users are interested not in the actual records, but some statistics of them. They can also be executed much more efficiently than reporting queries, by embedding properly precomputed aggregates into an index.

However, reporting and aggregation queries provide only two extremes for exploring the data. Data analysts often need more insight into the data distribution than what those simple aggregates provide, and yet certainly do not want the sheer volume of data returned by reporting queries. In this article, we design indexing techniques that allow for extracting a statistical summary of all the records in the query. The summaries we support include frequent items, quantiles, and various sketches, all of which are of central importance in massive data analysis. Our indexes require linear space and extract a summary with the optimal or near-optimal query cost. We illustrate the efficiency and usefulness of our designs through extensive experiments and a system demonstration.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Theory, Experimentation

Additional Key Words and Phrases: Indexing, summary queries

ACM Reference Format:
Ke Yi, Lu Wang, and Zhewei Wei. 2014. Indexing for summary queries: Theory and practice. ACM Trans. Datab. Syst. 39, 1, Article 2 (January 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2508702

1. INTRODUCTION

A database system's primary function is to answer users' queries. These queries can be broadly classified into two categories: reporting queries and aggregation queries. The former retrieves a collection of records from the database that match the query's conditions, while the latter only produces an aggregate, such as count, sum, average or max (min), of a particular attribute of these records. With reporting queries, the database is simply used as a data storage-retrieval tool. Many modern business intelligence applications, however, require ad hoc analytical queries with a rapid execution time. Users issuing these analytical queries are interested not in the actual records, but some statistics of them. This has therefore led to extensive research on how to perform aggregation queries efficiently. By augmenting a database index (very often a

K. Yi and L. Wang are partially supported by HKRGC under grant GRF-621413.
K. Yi and L. Wang; email: {yike, luwang}@cse.ust.hk; Z. Wei; email: zhewei@cs.au.dk.

DOI: http://dx.doi.org/10.1145/2508702

B-tree) with properly precomputed aggregates, aggregation queries can be answered efficiently at query time without going through the actual data records.

However, reporting and aggregation queries provide only two extremes for analyzing the data, by returning either all the records matching the query condition or one (or a few) single-valued aggregates. These simple aggregates are not expressive enough, and data analysts often need more insight into the data distribution. Consider the following queries.

Q1 *In a company's database*. What is the distribution of salaries of all employees aged between 30 and 40?

Q2 *In a search engine's query logs.* What are the most frequently queried keywords between May 1 and July 1, 2010?

The analyst issuing the query is perhaps not interested in listing all the records in the query range one by one, while probably not happy with a simple aggregate, such as average or max, either. What would be nice is some summary on the data, which is more complex than the simple aggregates, yet still much smaller than the raw query results. Some useful summaries include the frequent items, the ϕ-quantiles for, say, $\phi = 0.1, 0.2, \ldots, 0.9$, or a sketch (e.g., the Count-Min sketch [Cormode and Muthukrishnan 2005] or the AMS sketch [Alon et al. 1999]). All these summaries are of central importance in massive data analysis and have been extensively studied for offline and streaming data. Yet, to use the existing algorithms, one still has to first issue a reporting query to retrieve all query results, and then construct the desired summary afterward. This is time consuming and wasteful. One possible way to avoid retrieving all results is to get a sample, typically done with *block-level sampling* in database systems [Chaudhuri et al. 2004]. But such a sample has very low accuracy compared with these summaries just mentioned, as evident by abundant work on data summarization, as well as our own experimental study in Section 5.

In this article, we propose to add a native support for *summary queries* in a database index such that a summary can be returned in time proportional to the size of the summary itself, not the size of the raw query results. The problem we consider can be defined more precisely as follows. Let $\mathcal{D}$ be a database containing N records. Each record $r \in \mathcal{D}$ is associated with a *query attribute* $A_q(r)$ and a *summary attribute* $A_s(r)$, drawing values possibly from different domains. A *summary query* specifies a range constraint $[q_1, q_2]$ on A_q and the database returns a summary on the A_s attribute of all records whose A_q attribute is within the range. For example, in the query (Q1), A_q is "age" and A_s is "salary". Note that A_s and A_q could be the same attribute, but it is more useful when they are different, as the analyst is exploring the relationship between two attributes. Our goal is to build an index on $\mathcal{D}$ so that a summary query can be answered efficiently. As with any indexing problem, the primary measures are the query time and the space the index uses. The index should work in external memory, where it is stored in blocks of size B, and the query cost is measured in terms of the number of blocks accessed (I/Os). Finally, we also would like the index to support updates, that is, insertion and deletion of records.

1.1. Previous Work on Indexing for Aggregation Queries

In one dimension, most aggregates can be supported easily using a binary tree (a B-tree in external memory). At each internal node of the binary tree, we simply store the aggregate of all the records below the node. This way, an aggregation query can be answered in $O(\log N)$ time ($O(\log_B N)$ I/Os in external memory).

In higher dimensions, the problem becomes more difficult and has been extensively studied in both the computational geometry and the database communities. Solutions are typically based on space-partitioning hierarchies, like partition trees, quadtrees and R-trees, where an internal node stores the aggregate for its subtree. There is a large body of work on spatial data structures; please refer to the survey by Agarwal and Erickson [1999] and the book by Samet [2006]. When the data space forms an array, the data cube [Gray et al. 1997] is a classical structure for answering aggregation queries.

However, all the past research, whether in computational geometry or databases, has only considered queries that return simple aggregates like count, sum, max (min), distinct count [Tao et al. 2004], top-k [Afshani et al. 2011], and median [Brodal et al. 2011; Jørgensen and Larsen 2011]. The problem of returning complex summaries has not been addressed.

1.2. Previous Work on (Non-Indexed) Summaries

There is also a vast literature on various summaries in both the database and algorithms communities, motivated by the fact that simple aggregates cannot well capture the data distribution. These summaries, depending on the context and community, are also called *synopses*, *sketches*, or *compressed representations*. However, all past research has focused on how to construct a summary, either offline, in a streaming fashion, or over distributed data [Shrivastava et al. 2004; Agarwal et al. 2012], on the entire dataset. The indexing problem has not been considered, where the focus is to intelligently compute and store auxiliary information in the index at precomputation time, so that a summary on a *requested subset* of the records in the database can be built quickly at query time. The problem of how to maintain a summary as the underlying data changes, namely, under insertions and deletions of records or under the sliding window semantics [Datar et al. 2002], has also been extensively studied. But this shall not be confused with our dynamic index problem. The former maintains a single summary for the entire dynamic dataset, while the latter aims at maintaining a dynamic structure from which a summary for any queried subset can be extracted, which is more general than the former. Of course for the former, there often exist small-space solutions, while for the indexing problem, we cannot hope for sublinear space, as a query range may be small enough so that the summary degenerates to the raw query results.

Next we review some of the most fundamental and most studied summaries in the literature. Let D be a bag of items, and let $f_D(x)$ be the frequency of x in D.

Heavy Hitters. An (approximate) heavy hitters summary allows one to extract all frequent items approximately, that is, for a user-specified $0 < \phi < 1$, it returns all items x with $f_D(x) > \phi|D|$ and no items with $f_D(x) < (\phi - \varepsilon)|D|$, while an item x with $(\phi - \varepsilon)|D| \leq f_D(x) \leq \phi|D|$ may or may not be returned. A heavy hitters summary of size $O(1/\varepsilon)$ can be constructed in one pass over D, using the MG algorithm [Misra and Gries 1982] or the SpaceSaving algorithm [Metwally et al. 2006].

Quantiles. The quantiles (a.k.a. the *order statistics*), which generalize the median, are important statistics about the data distribution. Recall that the *ϕ-quantile*, for $0 < \phi < 1$, of a set D of items from a totally ordered universe is the one ranked at $\phi|D|$ in D (for convenience, for the quantile problem it is usually assumed that there are no duplicates in D). A *quantile summary* contains enough information so that for any $0 < \phi < 1$, an ε-approximate ϕ-quantile can be extracted, that is, the summary returns a ϕ'-quantile, where $\phi - \varepsilon \leq \phi' \leq \phi + \varepsilon$. A quantile summary has size $O(1/\varepsilon)$ and can be easily computed by sorting D, and then taking the items ranked at $\varepsilon|D|, 2\varepsilon|D|, 3\varepsilon|D|, \ldots, |D|$. In the streaming model where sorting is not possible, one could construct a quantile summary of the optimal $O(1/\varepsilon)$ size with $O((1/\varepsilon)\log \varepsilon N)$ working space, using the GK algorithm [Greenwald and Khanna 2001].

Sketches. Various sketches have been developed as a useful tool for summarizing massive data. In this article, we consider the two most widely used ones: the *Count-Min* *sketch* [Cormode and Muthukrishnan 2005] and the *AMS sketch* [Alon et al. 1999]. They summarize important information about D and can be used for a variety of purposes. Most notably, they can be used to estimate the join size of two datasets, with self-join size being a special case. Given a precision parameter δ, we can use the Count-Min sketches (resp. AMS sketches) of two datasets D_1 and D_2 to estimate $|D_1 \bowtie D_2|$ within an additive error of $\varepsilon F_1(D_1)F_1(D_2)$ (resp. $\varepsilon\sqrt{F_2(D_1)F_2(D_2)}$) with probability at least $1-\delta$ [Cormode and Muthukrishnan 2005; Alon et al. 2002], where F_k is the kth frequency moment of D: $F_k(D) = \sum_x f_D^k(x)$. Note that $\sqrt{F_2(D)} \le F_1(D)$, so the error of the AMS sketch is no larger. However, its size is $O((1/\varepsilon^2)\log(1/\delta))$, which is larger than the Count-Min sketch's size $O((1/\varepsilon)\log(1/\delta))$, so they are not strictly comparable. Which one is better will depend on the skewness of the datasets. In particular, since $F_1(D) = |D|$, the error of the Count-Min sketch does not depend on the skewness of the data, but $F_2(D)$ could range from $|D|$ for uniform data to $|D|^2$ for highly skewed data.

All the aforementioned work studies how to construct or maintain the summary on the given D. In our case, D is the A_s attributes of all records whose A_q attributes are within the query range. Our goal is to design an index so that the desired summary on D can be constructed efficiently without actually going through the elements of D.

Note that all these summaries are parameterized by an error parameter ε, which controls the trade-off between the summary size and the approximation error. As ε changes from large to small, the summary gradually gets larger and more accurate. When the summary is as large as the size of the raw query results, a summary query degenerates into a reporting query. Thus, summary queries provide a middle ground between the two extremes of aggregation queries and reporting queries. However, in our proposed index structures, this error parameter ε needs to be fixed before the index is built and cannot be changed at query time. It remains an interesting open question how to allow ε to be decided on-the-fly.

1.3. Our Results

This article mainly consists of two components. In Section 2 and 3, we ask the theoretical question whether optimal indexing is possible for summary queries. Recall that the B-tree occupies linear space, supports a range aggregation query in $O(\log_B N)$ I/Os, and a range reporting query in $O(\log_B N + K/B)$ I/Os, where K is the number or records reported, both of which are optimal (in a comparison model). For summary queries, the best obtainable query time is thus $O(\log_B N + s_\varepsilon/B)$, where s is the size of the summary returned, which is parameterized by ε, the error parameter. Note that for not-too-large summaries $s_\varepsilon = O(B)$, the query cost becomes just $O(\log_B N)$, the same as that for a simple aggregation query or a lookup on a B-tree. Meanwhile, we would like to achieve this optimal query time with a linear-size index.

We first observe that the optimal query time can be easily achieved if the summary is *subtractive*, that is, given the summary of a data set A and the summary of $B \subseteq A$, we can get the summary of $B \setminus A$ by subtracting the two summaries. All sketches that are linear projections of the data, such as the Count-Min sketch and the AMS sketch, have this property. We describe this simple solution in Section 2.1. However, for nonsubtractive summaries, such as heavy hitters and quantile, the problem becomes nontrivial. To achieve optimality for these summaries, we define an *exponentially decomposable* property enjoyed by these these summaries. We first show how this property leads to an optimal internal memory index in Section 2.3, and then convert it to an external memory index with some nontrivial data structuring techniques in Section 2.4.

In Sections 4 and 5, we turn to the practical side of the problem. Our theoretically optimal index is unlikely to yield a practical implementation, and the hidden constants in the big-O's are quite large. The one for subtractive summaries is simple, but it is inherently static. Nevertheless, some ideas in the development of the theory are still useful in designing a simpler and practical index structure for summary queries. In Section 4, we present a simplified, practical version of our index. In doing so, we have to settle for a slightly worse asymptotic running time in favor of better constants and easier implementation. In addition, the practical version of the index is fully dynamic, that is, supporting insertion and deletion of records efficiently. In Section 5, we conduct extensive experiments to study the practical efficiency of the indexes and showcase their usefulness with a few example queries and a Web-based demonstration.

1.4. Other Related Work

A few other lines of work also head in the general direction of addressing the gap between reporting all query results and returning some simple aggregates. Lin et al. [2007] and Tao et al. [2009] propose returning only a subset of the query results, called representatives. But the representatives do not summarize the data as we do. They also only consider skyline queries. The line of work on *online aggregation* [Hellerstein et al. 1997; Jermaine et al. 2008] aims at producing a random sample of the query results at early stages of long-running queries, in particular, joins. A random sample indeed gives a rough approximation of the data distribution, but it is much less accurate than the summaries we consider: For heavy hitters and quantiles, a random sample of size $\Theta(1/\varepsilon^2)$ is needed [Vapnik and Chervonenkis 1971] to achieve the same accuracy as the $O(1/\varepsilon)$-sized summaries we mentioned earlier; for estimating join sizes, a random sample of size $\Omega(\sqrt{N})$ is required to achieve a constant approximation, which is much worse than using the sketches [Alon et al. 2002]. Furthermore, the key difference is that they focus on query processing techniques for joins rather than indexing issues. Correlated aggregates [Gehrke et al. 2001] aim at exploring the relationship between two attributes. They are computed on one attribute subject to a certain condition on the other. However, this condition has to be specified in advance, and the goal is to compute the aggregate in the streaming setting, thus the problem is fundamentally different from ours. Buccafurri et al. [2008] study how to use indexing techniques to improve the accuracy of a histogram summary, which is in some sense “dual” to the problem considered in this article.

This article extends the earlier work of Wei and Yi [2011], where the results are purely theoretical. To put the result into practice, we have introduced many simplifications as well as new ideas in the original design. In addition, the practical versions of the indexes presented in this article are fully dynamic, that is, supporting insertion and deletion of records efficiently, whereas the external memory index structures in Wei and Yi [2011] are static. Sections 4 and 5, which comprise half of the material in this article, are new.

2. OPTIMAL INDEXING FOR SUMMARY QUERIES

In this section, we will describe our structures without instantiating into any particular summary. Instead we just use “ε-summary” as a placeholder for any summary with error parameter ε. Let $\mathcal{S}(\varepsilon, D)$ denote an ε-summary on data set D. We use s_ε to denote the size of an ε-summary.[1]

2.1. Optimal Indexing for Subtractive Sketches

We present a simple indexing that achieves optimal space usage and query cost for subtractive summaries. The subtractive property can be formally defined as follow. Consider multiset D_1 and a subset $D_2 \subseteq D_1$. For any item $x \in D_2$, we have $x \in D_1$ and

[1]Strictly speaking, we should write $s_{\varepsilon,D}$. But as most ε-summaries have sizes independent of D, we drop the subscript D for brevity.

$f_{D_2}(x) \leq f_{D_1}(x)$. Let $D_1 \setminus D_2$ denote the *multiset difference* of D_1 and D_2, that is, for any item $x \in D_1 \setminus D_2$, the frequency of x in $D_1 \setminus D_2$ is $f_{D_1}(x) - f_{D_2}(x)$.

Definition 2.1. A summary $\mathcal{S}$ is subtractive if for a multiset D_1 and its subset D_2, given summaries $\mathcal{S}(\varepsilon, D_1)$ and $\mathcal{S}(\varepsilon, D_2)$, one can obtain an ε-summary for $D_1 \setminus D_2$.

All linear sketches, including the Count-Min sketch and the AMS sketch, are subtractive. In this section, we demonstrate the subtractive property for the Count-Min sketch as an example; the analysis for other linear sketches is similar. A Count-Min sketch with error parameter ε and precision parameter δ is represented by a two-dimensional array of counters of width w and depth d: $C[1,1], \ldots, C[d,w]$, where $w = \lceil \frac{e}{w} \rceil$ and $d = \lceil \frac{1}{\delta} \rceil$. The counters are initialized to all zeros. The ith row of counters is associated with a pairwise independent hash function h_i which maps the items uniformly to the range $\{1, 2, \ldots w\}$, for $i = 1, \ldots, d$. When a new update (x, c) comes, meaning that the count of item x is updated by quantity c, we add c to counter $C[i, h_i(x)]$, for $i = 1, \ldots, d$. To query the frequency of item x, we retrieve all counters $C[i, h_i(x)]$, $i = 1, \ldots, d$, and use the minimum count as an estimation. In other word, the frequency of item x is estimated by $\hat{f}_D(x) = \min_{1 \leq i \leq d} C[i, h_i(x)]$. Previous analysis shows that with probability $1 - \delta$, the Count-Min sketch provides additive error $\varepsilon F_1(D)$: $f_D(x) \leq \hat{f}_D(x) \leq f_D(x) + \varepsilon F_1(D)$. To show the subtractive property for the Count-Min sketch, consider multiset D_1 and its subset $D_2 \subseteq D_1$. Suppose we are given two Count-Min Sketches $\mathcal{S}(\varepsilon, D_1)$ and $\mathcal{S}(\varepsilon, D_2)$, and they use the same hash functions. One can simply subtract the counters of $\mathcal{S}(\varepsilon, D_2)$ from the corresponding counters of $\mathcal{S}(\varepsilon, D_1)$, and the resulting summary is a Count-Min Sketch for $D_1 \setminus D_2$.

Suppose we are given a subtractive summary $\mathcal{S}$, and let s_ε denote the size of an ε-summary. Given a dataset with N records, we sort the N records by their A_q attributes and partition them into N/s_ε groups, each of size s_ε. Let $G_1, \ldots, G_{N/s_\varepsilon}$ denote the N/s_ε groups. For the ith group G_i, we store a summary for the A_s attributed of the records in groups $G_1, \ldots, G_i$, that is, summary $\mathcal{S}(\varepsilon, G_1 \uplus \ldots \uplus G_i)$. One can verify that these summaries can supply a summary query. Given a query range R, let G_i and G_j be the two groups that intersect with the end points of R. For the groups contained completely in R, that is, groups $G_{i+1}, \ldots, G_{j-1}$, the index can supply a summary by subtracting summary $\mathcal{S}(\varepsilon, G_1 \uplus \ldots \uplus G_{j-1})$ and $\mathcal{S}(\varepsilon, G_1 \uplus \ldots \uplus G_i)$. For the records in $G_i \cap R$ and $G_j \cap R$, we treat their A_s attributes as updates and add them to the resulting summary. The cost of assembling these summaries and groups is $O(s_\varepsilon)$, since the index only touches two summaries and two groups, each of size s_ε. The search cost is $O(\log N)$. If we apply a binary tree on top of the N/s_ε groups, therefore, the total query cost is $O(\log N + s_\varepsilon)$. The space usage is linear, since the space to store the summaries is $N/s_\varepsilon * s_\varepsilon = N$.

This approach also works for the I/O model. If block size $B \leq s_\varepsilon$, we simply replace the binary tree with an B-tree; otherwise, we pack B/s_ε groups into a single block and build a B-tree on top of these blocks. By the properties of the B-tree and the fact that all summaries and groups are stored consecutively, the I/O cost for a query is bounded by $O(\log_B N + s_\varepsilon/B)$. The space usage is clearly linear. Finally, we note that this approach is inherently static, since an update may touch all N/s_ε summaries, which leads to very high update cost in both internal and external memory.

2.2. Decomposability and Exponential Decomposability

Many useful summaries, like the heavy hitters and quantiles, are not subtractive. To deal with these summaries, we need exploit some other properties. We start with the *decomposable* property that is enjoyed by almost all known summaries. This property states that if we are given the ε-summaries for t datasets (bags of elements) $D_1, \ldots, D_t$, then we can combine them together into an $O(\varepsilon)$-summary on $D_1 \uplus \cdots \uplus D_t$. This property has been exploited in many other works on maintaining summaries in the streaming context or on distributed data [Arasu and Manku 2004; Beyer et al. 2007; Cormode and Muthukrishnan 2005]. We can also make use of this property for our indexing problem. Build a binary tree $\mathcal{T}$ on the N data records in the database on the A_q attribute. It has N leaves, each corresponding to a data record. At each internal node u of $\mathcal{T}$, we attach an ε-summary on the A_s attribute of all records stored in the subtree below u. It is well known that any range query can be decomposed into $O(\log N)$ *dyadic* intervals, each corresponding to a subtree in $\mathcal{T}$. Thus, to answer a summary query, we simply retrieve the ε-summaries attached to the nodes corresponding to these dyadic intervals and then combine them together. Assuming we can combine these summaries in linear time (for some summaries, the combine step actually takes slightly more than linear time), the total query time would be $O(s_\varepsilon \log N)$.

To improve upon this solution, we observe that its main bottleneck is not the search cost, but the fact that it needs to assemble $O(\log N)$ summaries, each of size s_ε. In the absence of additional properties of the summary, it is impossible to make further improvement. Fortunately, we identify a stronger decomposable property for many of the F_1-based summaries that we call *exponential decomposability*, which allows us to assemble summaries of exponentially decreasing sizes. This turns out to be the key to optimal indexing for these summaries.

Definition 2.2 (*Exponentially Decomposable*). For $0 < \alpha < 1$, a summary $\mathcal{S}$ is *α-exponentially decomposable* if there exists a constant $c > 1$ (which can depend on α), such that for any t multisets $D_1, \ldots, D_t$ with their sizes satisfying $F_1(D_i) \le \alpha^{i-1} F_1(D_1)$ for $i = 1, \ldots, t$, given $\mathcal{S}(\varepsilon, D_1), \mathcal{S}(c\varepsilon, D_2) \ldots, \mathcal{S}(c^{t-1}\varepsilon, D_t)$, the following hold:

(1) We can combine them into an $O(\varepsilon)$-summary on $D_1 \uplus \cdots \uplus D_t$;
(2) The total size of $\mathcal{S}(\varepsilon, D_1), \ldots, \mathcal{S}(c^{t-1}\varepsilon, D_t)$ is $O(s_\varepsilon)$ and they can be combined in $O(s_\varepsilon)$ time; and
(3) For any multiset D, the total size of $\mathcal{S}(\varepsilon, D), \ldots, \mathcal{S}(c^{t-1}\varepsilon, D)$ is $O(s_\varepsilon)$.

Intuitively, since an F_1-based summary $\mathcal{S}(\varepsilon, D)$ provides an error bound of $\varepsilon|D|$, the total error from $\mathcal{S}(\varepsilon, D_1)$, $\mathcal{S}(c\varepsilon, D_2)$, $\ldots$, $\mathcal{S}(c^{t-1}\varepsilon, D_t)$ is

$$\begin{aligned}&\varepsilon|D_1| + c\varepsilon|D_2| + \cdots + c^{t-1}\varepsilon|D_t|\\ &\quad \le \varepsilon|D_1| + (c\alpha)\varepsilon|D_1| + \cdots + (c\alpha)^{t-1}\varepsilon|D_1|.\end{aligned}$$

If we choose c such that $c\alpha < 1$, then the error is bounded by $O(\varepsilon|D_1|)$, satisfying (1). Meanwhile, the F_1-based summaries usually have size $s_\varepsilon = \Theta(1/\varepsilon)$, so (2) and (3) can be satisfied, too. In Section 3, we will formally prove the α-exponentially decomposable property for all the F_1-based summaries mentioned in Section 1.2.

2.3. Optimal Internal Memory Index Structure

In this section, we show how the exponentially decomposable property leads to an optimal internal memory index structure of size $O(N)$ and a query time of $O(\log N + s_\varepsilon)$. This will serve as the first step towards an optimal index in external memory.

Let $\mathcal{T}$ be the binary tree as before. For any node u of $\mathcal{T}$, we denote by $\mathcal{T}_u$ the subtree below u. We first note that if $\mathcal{T}_u$ contains less than s_ε records, there is no need to attach any summary to u, since the summary will be just the same as all the records in $\mathcal{T}_u$. So equivalently, after sorting the N records by the A_q attribute, we partition them into N/s_ε groups of size s_ε, and build $\mathcal{T}$ on top of these groups. Thus each leaf of $\mathcal{T}$ becomes a *fat leaf* that stores s_ε records. Without loss of generality, we assume $\mathcal{T}$ is a complete binary tree; otherwise, we can always add at most N dummy records to make N/s_ε a power of 2 so that $\mathcal{T}$ is complete.

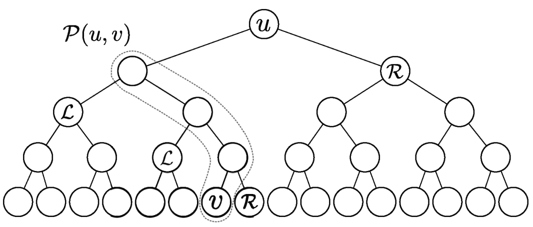

Fig. 1. An illustration of $\mathcal{P}(u, v)$, $\mathcal{L}(u, v)$ and $\mathcal{R}(u, v)$.

We introduce some more notation on $\mathcal{T}$. We use $\mathcal{S}(\varepsilon, u)$ to denote the ε-summary on the A_s attribute of all records stored in $\mathcal{T}_u$. Consider an internal node u and one of its descendants, say v. Let $\mathcal{P}(u, v)$ be the set of nodes on the path from u to v, excluding u. Define the *left sibling set* of $\mathcal{P}(u, v)$ to be

$$\mathcal{L}(u, v) = \{w \mid w \text{ is a left child and } w\text{'s right sibling node} \in \mathcal{P}(u, v)\},$$

and similarly the *right sibling set* of $\mathcal{P}(u, v)$ to be

$$\mathcal{R}(u, v) = \{w \mid w \text{ is a right child and } w\text{'s left sibling node} \in \mathcal{P}(u, v)\}.$$

Figure 1 shows an example illustrating u, v, and the corresponding $\mathcal{P}(u, v)$, $\mathcal{L}(u, v)$ and $\mathcal{R}(u, v)$.

To answer a query $[q_1, q_2]$, we first locate the two fat leaves a and b in $\mathcal{T}$ that contain q_1 and q_2, respectively, by doing two searches on $\mathcal{T}$. Let u be the lowest common ancestor of a and b. We call $\mathcal{P}(u, a)$ and $\mathcal{P}(u, b)$ the left and, respectively, the right query path. We observe that the subtrees rooted at the nodes in $\mathcal{R}(u, a) \cup \mathcal{L}(u, b)$ make up the dyadic set for the query range $[q_1, q_2]$.

Focusing on $\mathcal{R}(u, a)$, let $w_1, \dots, w_t$ be the nodes of $\mathcal{R}(u, a)$ and let $d_1 < \cdots < d_t$ denote their depths in $\mathcal{T}$ (the root of $\mathcal{T}$ is said to be at depth 0). Since $\mathcal{T}$ is a balanced binary tree, we have $F_1(w_i) \le (1/2)^{d_i - d_1} F_1(w_1)$ for $i = 1, \dots, t$. Here we use $F_1(w)$ to denote the first frequency moment (i.e., size) of the dataset stored below w. Thus, if the summary is $(1/2)$-exponentially decomposable, and we have $\mathcal{S}(c^{d_i - d_1}\varepsilon, w_i)$ for $i = 1, \dots, t$ at our disposal, we can combine them and form an $O(\varepsilon)$-summary for all the data covered by $w_1, \dots, w_t$. We do the same for $\mathcal{L}(u, b)$. Finally, the two fat leaves can always supply the exact data (it is a summary with no error) of size $O(s_\varepsilon)$ in the query range. Plus the initial $O(\log N)$ search cost for locating $\mathcal{R}(u, a)$ and $\mathcal{L}(u, b)$, the query time now improves to the optimal $O(\log N + s_\varepsilon)$.

It only remains to show how to supply $\mathcal{S}(c^{d_i - d_1}\varepsilon, w_i)$ for each of the w_i's. In fact, we can afford to attach to each node $u \in \mathcal{T}$ all the summaries: $\mathcal{S}(\varepsilon, u), \mathcal{S}(c\varepsilon, u), \dots \mathcal{S}(c^q\varepsilon, u)$, where q is an integer such that $s_{c^q\varepsilon} = O(1)$. Nicely, these summaries still have total size $O(s_\varepsilon)$ by the exponentially decomposable property, thus the space required by each node of $\mathcal{T}$ is still $O(s_\varepsilon)$. Since $\mathcal{T}$ has $O(N/s_\varepsilon)$ nodes, the total space is linear. A schematic illustration of the overall structure is shown in Figure 2. The grayed nodes form the dyadic decomposition of the query range, and the grayed summaries are those we combine into the final summary for the queried data. In this example, we use $c = \frac{3}{2}$.

THEOREM 2.3. *For any $(1/2)$-exponentially decomposable summary, a database $\mathcal{D}$ of N records can be stored in an internal memory structure of linear size so that a summary query can be answered in $O(\log N + s_\varepsilon)$ time.*

Example. We present a simple example to demonstrate how our index can be used to answer the queries in Section 1. Suppose the goal is to build an index to support (Q1): In a company's database: what is the distribution of salaries of all employees aged between 30 and 40?

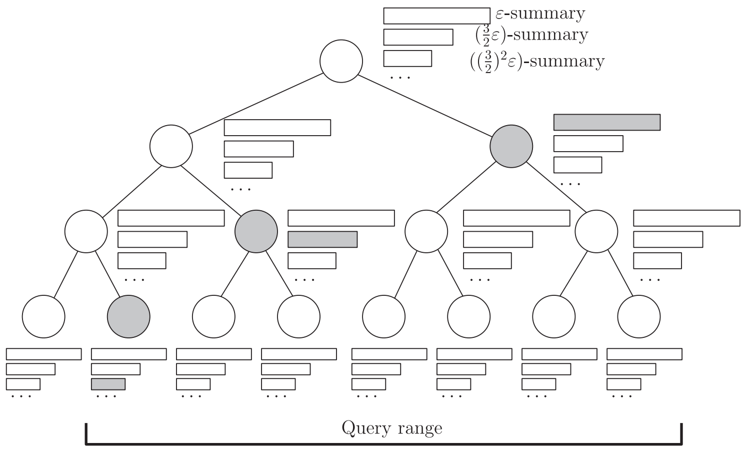
ε-summary
$(\frac{3}{2}\varepsilon)$-summary
$((\frac{3}{2})^2\varepsilon)$-summary
Query range

Fig. 2. A schematic illustration of our internal memory structure.

First we need to choose a proper summary for this particular application. To characterize a data distribution, one of the most commonly used method is the histograms. In this example, we use the *equidepth histogram*, which seeks to partition the distribution equally in terms of the cumulative distribution function. More precisely, given a data set of N tuples, a k-bucket equidepth histogram selects $k-1$ tuples from the data set such that the number of tuples between any two consecutive tuples is approximately N/k. The equidepth histogram can be used as an approximation to the data distribution and is relatively easy to maintain; therefore, it is widely used in streaming and data analytics applications. It is in general more accurate and can adapt better to skewed data distribution than the equiwidth histogram. To adapt our indexing framework, we note that a quantile summary can serve as an approximate equi-depth histogram: given a quantile summary with error parameter ε, the ϕ-quantiles for $\phi = \frac{1}{k}, \ldots, \frac{k-1}{k}$ can serve as the boundaries in the equi-depth histogram with error at most εN. For detailed discussion on the quantile summary, we refer the reader to Section 3.

Now we have a concrete problem that falls into our framework: given a set of N records, each consisting of an age attribute and a salary attribute, build an index such that given a range on the age attribute, the index can return a quantile summary for the salary attributes of the records whose age attributes are within the query range. Following the construction in Section 2.3, we sort the N records by the age attribute and partition them into fat leaves of size s_ε. We then build the binary tree $\mathcal{T}$ on top of the fat leaves. Recall that for each internal node u, we need to construct summaries of various sizes for records in the subtree rooted by u. The summary of size $s_{c^j\varepsilon}$ is simply constructed by selecting and storing the ϕ-quantiles together with their ranks, for $\phi = c^j\varepsilon, 2c^j\varepsilon, \ldots, 1-c^j\varepsilon$. These summaries are stored separately from the records and the binary tree $\mathcal{T}$, and we store a pointer pointing from u to the first summary. Given a query range, the index finds the internal nodes that make up for the range, and extracts the corresponding summaries from each node. By the analysis in Section 2.3, we can merge them into one quantile summary of size $O(s_\varepsilon)$. Finally, we add the records in the fat leaves into the resulting summary and obtain a quantile summary for all records within the query range. The quantile summary can be used to construct an equidepth histogram to approximate the distribution of the salaries of all employees in the query range. Figure 3 gives an illustration of a possible query result, where each bucket contains approximately 20% of employees in the query range. The bucket

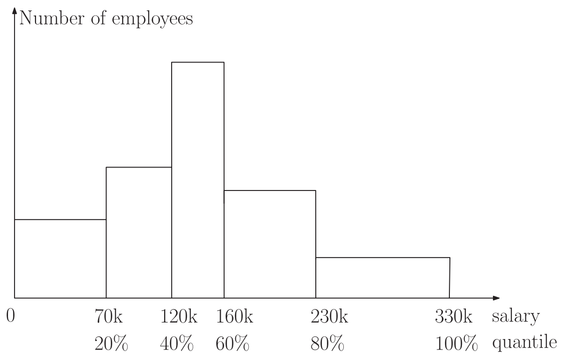

Fig. 3. An illustration of the equidepth histogram. The dashed line represents the actual data.

boundaries are thus the 20%-, 40%-, 60%-, and 80%-quantiles, and the area of each bucket (rectangle) is the same.

2.4. Optimal External Memory Index Structure

In this section, we show how to achieve the $O(\log_B N + s_\varepsilon/B)$-I/O query cost in external memory still with linear space. Here, the difficulty that we need to assemble $O(\log N)$ summaries lingers. In internal memory, we managed to get around it by the exponentially decomposable property so that the total size of these summaries is $O(s_\varepsilon)$. However, they still reside in $O(\log N)$ separate nodes. If we still use a standard B-tree blocking for the binary tree $\mathcal{T}$, for $s_\varepsilon \geq B$ we need to access $\Omega(\log N)$ blocks; for $s_\varepsilon < B$, we need to access $\Omega(\log N/\log(B/s_\varepsilon))$ blocks, neither of which is optimal. We first show how to achieve the optimal query cost by increasing the space to superlinear, then propose a packed structure to reduce the space back to linear.

Consider an internal node u and one of its descendants v. Let the sibling sets $\mathcal{R}(u, v)$ and $\mathcal{L}(u, v)$ be as previously defined. In the following, we only describe how to handle the $\mathcal{R}(u, v)$'s; we will do the same for the $\mathcal{L}(u, v)$'s. Suppose $\mathcal{R}(u, v)$ contains nodes $w_1, \ldots, w_t$ at depths $d_1, \ldots, d_t$. We define the *summary set* for $\mathcal{R}(u, v)$ with error parameter ε to be

$$\mathcal{RS}(u, v, \varepsilon) = \left\{\mathcal{S}(\varepsilon, w_1), \mathcal{S}\left(c^{d_2-d_1}\varepsilon, w_2\right), \ldots, \mathcal{S}\left(c^{d_t-d_1}\varepsilon, w_t\right)\right\}.$$

The following two facts easily follow from the exponentially decomposable property.

Fact 2.1. The total size of the summaries in $\mathcal{RS}(u, v, \varepsilon)$ is $O(s_\varepsilon)$.

Fact 2.2. The total size of all the summary sets $\mathcal{RS}(u, v, \varepsilon)$, $\mathcal{RS}(u, v, c\varepsilon), \ldots, \mathcal{RS}(u, v, c^t\varepsilon)$ is $O(s_\varepsilon)$.

2.4.1. The Indexing Structure. We first build the binary tree $\mathcal{T}$ as before with a fat leaf size of s_ε. Before attaching any summaries, we block $\mathcal{T}$ in a standard B-tree fashion so that each block stores a subtree of $\mathcal{T}$ of size $\Theta(B)$, except possibly the root block, which may contain 2 to B nodes of $\mathcal{T}$. The resulting blocked tree is essentially a B-tree on top of N/s_ε fat leaves, and each internal node occupies one block. Please see Figure 4 for an example of the standard B-tree blocking.

Consider an internal block $\mathcal{B}$ in the B-tree. We next describe the additional structures we attach to $\mathcal{B}$. Since there are $O(N/(Bs_\varepsilon))$ internal nodes, if the additional structures attached to each $\mathcal{B}$ have size $O(Bs_\varepsilon)$, the total size will be linear. Let $\mathcal{T}_\mathcal{B}$ be the binary subtree of $\mathcal{T}$ stored in $\mathcal{B}$ and let $r_\mathcal{B}$ be the root of $\mathcal{T}_\mathcal{B}$. To achieve the optimal query cost, the summaries attached to the nodes of $\mathcal{T}_\mathcal{B}$ that we need to retrieve for answering any query must be stored consecutively, or in at most $O(1)$ consecutive chunks. Therefore, the idea is to store all the summaries for a query path in $\mathcal{T}_\mathcal{B}$ together, which is the

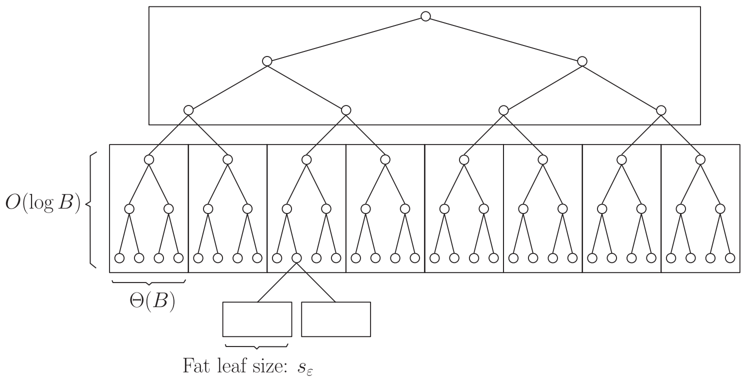

Fig. 4. The standard B-tree blocking of the binary tree $\mathcal{T}$.

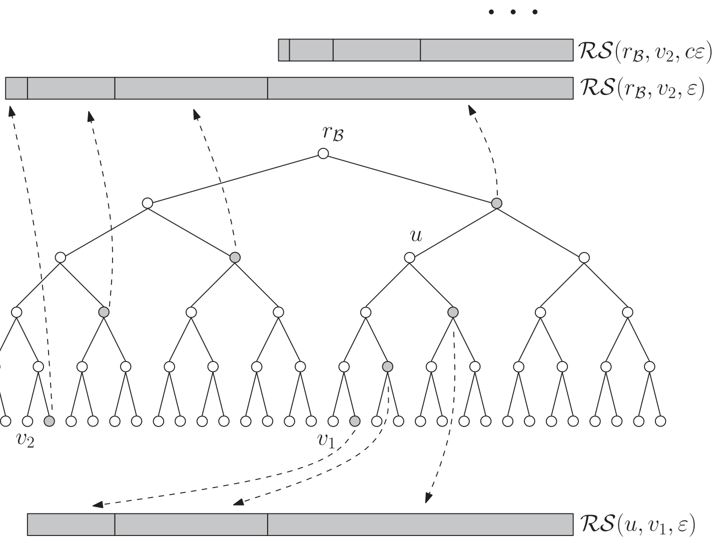
$\mathcal{RS}(r_\mathcal{B}, v_2, c\varepsilon)$
$\mathcal{RS}(r_\mathcal{B}, v_2, \varepsilon)$
$r_\mathcal{B}$
u
v_2
v_1
$\mathcal{RS}(u, v_1, \varepsilon)$

Fig. 5. The summaries attached to an internal block $\mathcal{B}$.

reason we introduced the summary set $\mathcal{RS}(u, v, \varepsilon)$. The detailed structures that we attach to $\mathcal{B}$ are as follows.

(1) For each internal node $u \in \mathcal{T}_\mathcal{B}$ and each leaf v in u's subtree in $\mathcal{T}_\mathcal{B}$, we store all summaries in $\mathcal{RS}(u, v, \varepsilon)$ sequentially.
(2) For each leaf $v \in \mathcal{T}_\mathcal{B}$, we store the summaries in $\mathcal{RS}(r_\mathcal{B}, v, c^j\varepsilon)$ sequentially, for all $j = 0, \ldots, q$. Recall that q is an integer such that $s_{c^q\varepsilon} = O(1)$.
(3) For the root $r_\mathcal{B}$, we store $\mathcal{S}(c^j\varepsilon, r_\mathcal{B})$ for $j = 0, \ldots, q$.

An illustration of the first and the second type of structures is shown in Figure 5. The summary set $\mathcal{RS}(u, v_1, \varepsilon)$ forms the first type of structure for node u and its leaf v_1, and the summary sets $\mathcal{RS}(r_\mathcal{B}, v_2, \varepsilon)$, $\mathcal{RS}(r_\mathcal{B}, v_2, c\varepsilon)$, ... form the second type of structure for leaf v_2.

The size of these additional structures can be determined as follows.

(1) For each leaf $v \in \mathcal{T}_\mathcal{B}$, there are at most $O(\log B)$ ancestors of v, so there are in total $O(B \log B)$ such pairs (u, v). For each pair we use $O(s_\varepsilon)$ space, so the space usage is $O(s_\varepsilon B \log B)$.

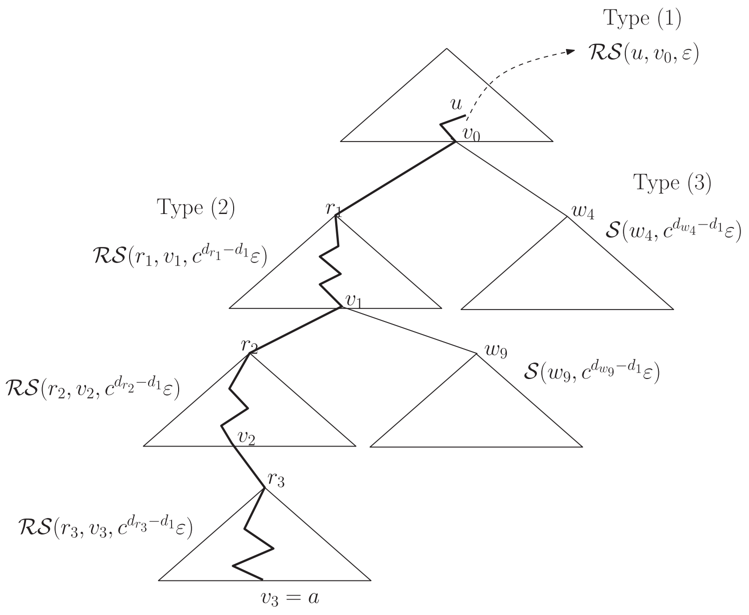
Type (1)
$\mathcal{RS}(u, v_0, \varepsilon)$
u
v_0
Type (2)
$\mathcal{RS}(r_1, v_1, c^{d_{r_1}-d_1}\varepsilon)$
r_1
w_4
Type (3)
$\mathcal{S}(w_4, c^{d_{w_4}-d_1}\varepsilon)$
v_1
r_2
w_9
$\mathcal{RS}(r_2, v_2, c^{d_{r_2}-d_1}\varepsilon)$
$\mathcal{S}(w_9, c^{d_{w_9}-d_1}\varepsilon)$
v_2
r_3
$\mathcal{RS}(r_3, v_3, c^{d_{r_3}-d_1}\varepsilon)$
$v_3 = a$

Fig. 6. An illustration of the query procedure.

(2) For each leaf $v \in \mathcal{T}_\mathcal{B}$ we use $O(s_\varepsilon)$ space, so the space usage is $O(s_\varepsilon B)$.
(3) For the root $r_\mathcal{B}$, the space usage is $O(s_\varepsilon)$.

Summing up these cases, the space for storing the summaries of any internal block $\mathcal{B}$ is $O(s_\varepsilon B \log B)$, namely, the type (1) structures are the bottleneck. Since there are $O(N/(Bs_\varepsilon))$ internal blocks, thus the total space usage is $O(N \log B)$. Next, we first show that these additional structures suffice to answer queries in the optimal $O(\log_B N + s_\varepsilon/B)$ I/Os, before trying to reduce the total size back to linear.

2.4.2. Query Procedure. Given a query range $[q_1, q_2]$, let a and b be the two leaves containing q_1 and q_2, respectively. We focus on how to retrieve the necessary summaries for the right sibling set $\mathcal{R}(u, a)$, where u is the lowest common ancestor of a and b; the left sibling set $\mathcal{L}(u, b)$ can be handled symmetrically. By the previous analysis, we need exactly the summaries in $\mathcal{RS}(u, a, \varepsilon)$. Recall that $\mathcal{R}(u, a)$ are the right siblings of the left query path $\mathcal{P}(u, a)$. Let $\mathcal{B}_0, \ldots, \mathcal{B}_l$ be the blocks that $\mathcal{P}(u, a)$ intersects from u to a. The path $\mathcal{P}(u, a)$ is partitioned into $l + 1$ segments by these $l + 1$ blocks. Let $\mathcal{P}(u, v_0), \mathcal{P}(r_1, v_1), \ldots, \mathcal{P}(r_l, v_l = a)$ be the $l + 1$ segments, with r_i being the root of the binary tree $\mathcal{T}_{\mathcal{B}_i}$ in block $\mathcal{B}_i$ and v_i being a leaf of $\mathcal{T}_{\mathcal{B}_i}$, $i = 0, \ldots, l$. Please see Figure 6 for an illustration. Let $w_1, \ldots, w_t$ be the nodes in $\mathcal{R}(u, a)$, at depths $d_1, \ldots, d_t$ of $\mathcal{T}$. We claim that w_i is either a node of $\mathcal{T}_{\mathcal{B}_k}$ for some $k \in \{0, \ldots, l\}$, or a right sibling of r_k for some $k \in \{0, \ldots, l\}$, which makes w_i a root of some other block. This is because by the definition of $\mathcal{R}(u, a)$, we know that w_i is a right child whose left sibling is in some $\mathcal{B}_k$. If w_i is not in $\mathcal{B}_k$, it must be the root of some other block. Recall that we need to retrieve $\mathcal{S}(c^{d_i - d_1}\varepsilon, w_i)$ for $i = 1, \ldots, t$. We next show how this can be done efficiently using our structure.

For the w_i's in the first block $\mathcal{B}_0$, since we have stored all summaries in $\mathcal{RS}(u, v_0, \varepsilon)$ sequentially for $\mathcal{B}_0$ (type (1)), they can be retrieved in $O(1 + s_\varepsilon/B)$ I/Os.

For any w_i being the root of some other block $\mathcal{B}'$ not on the path $\mathcal{B}_0, \ldots, \mathcal{B}_l$, since we have stored the summaries $\mathcal{S}(c^j \varepsilon, w_i)$ for $j = 0, \ldots, q$ for every block (type (3)), the required summary $\mathcal{S}(c^{d_i - d_1}\varepsilon, w_i)$ can be retrieved in $O(1 + s_{c^{d_i - d_1}\varepsilon}/B)$ I/Os. Note that the number of such w_i's is bounded by $O(\log_B N)$, so the total cost for retrieving summaries for these nodes is at most $O(\log_B N + s_\varepsilon/B)$ I/Os.

The rest of the w_i's are in $\mathcal{B}_1, \ldots, \mathcal{B}_l$. Consider each $\mathcal{B}_k, k = 1, \ldots, l$. Recall that the segment of the path $\mathcal{P}(u, a)$ in $\mathcal{B}_k$ is $\mathcal{P}(r_k, v_k)$, and the w_i's in $\mathcal{B}_k$ are exactly $\mathcal{R}(r_k, v_k)$. We have stored $\mathcal{RS}(r_k, v_k, c^j \varepsilon)$ for $\mathcal{B}_k$ for all j (type (2)), so no matter at which relative depths $d_i - d_1$ the nodes in $\mathcal{R}(r_k, v_k)$ start and end, we can always find the required summary set. Retrieving the desired summary set takes $O(1 + s_{c^{d'-d_1}\varepsilon}/B)$ I/Os, where d' is the depth of the highest node in $\mathcal{R}(r_k, v_k)$. Summing over all blocks $\mathcal{B}_1, \ldots, \mathcal{B}_l$, the total cost is $O(\log_B N + s_\varepsilon/B)$ I/Os. Finally, we scan all the records in the fat leaves a and b to get all the remaining records in $[q_1, q_2]$ not covered by the summaries retrieved above. This takes $O(1 + s_\varepsilon/B)$ I/Os, which does not affect the overall asymptotic query cost.

2.4.3. Reducing the Size to Linear. The previous structure has a superlinear size $O(N \log B)$. Next we show how to reduce its size back to $O(N)$ while not affecting the optimal query time.

Observe that the extra $O(\log B)$ factor comes from the type (1) structures, where we store $\mathcal{RS}(u, v, \varepsilon)$ for each internal node u and each leaf v in u's subtree in u's block $\mathcal{B}$. Focus on one internal block $\mathcal{B}$ and the binary tree $\mathcal{T}_\mathcal{B}$ stored in it. Abusing notation, we now use $\mathcal{T}_u$ to denote the subtree rooted at u in $\mathcal{T}_\mathcal{B}$. Assume $\mathcal{T}_u$ has height h in $\mathcal{T}_\mathcal{B}$ (the leaves of $\mathcal{T}_\mathcal{B}$ are defined to be at height 0). Our idea is to pack the $\mathcal{RS}(u, v, \varepsilon)$'s for some leaves $v \in \mathcal{T}_u$ to reduce the space usage. Let u_l and u_r be the left and right child of u, respectively. The first observation is that we only need to store $\mathcal{RS}(u, v, \varepsilon)$ for each leaf v in u_l's subtree. This is because for any leaf v in u_r's subtree, the sibling set $\mathcal{R}(u, v)$ is the same as $\mathcal{R}(u_r, v)$, so $\mathcal{RS}(u, v, \varepsilon) = \mathcal{RS}(u_r, v, \varepsilon)$, which will be stored when considering u_r in place of u. For any leaf v in u_l's subtree, observe that the highest node in $\mathcal{R}(u, v)$ is u_r. This means for a node $w \in \mathcal{R}(u, v)$ with height i in tree $\mathcal{T}_u$, the summary for w in $\mathcal{RS}(u, v, \varepsilon)$ is $\mathcal{S}(c^{h-i-1}\varepsilon, w)$. Let u' be an internal node in u_l's subtree, and suppose u' has k_h leaves below it. We will decide later the value of k_h, hence the height $\log k_h$ at which u' is chosen. We do the following for each u' at height $\log k_h$ in u_l's subtree. Instead of storing the summary set $\mathcal{RS}(u, v, \varepsilon)$ for each leaf v in u''s subtree, we store $\mathcal{RS}(u, u', \varepsilon)$, which is the common prefix of all the $\mathcal{RS}(u, v, \varepsilon)$'s, together with a summary for each of the nodes in u''s subtree. More precisely, for each node w in u''s subtree, if its height is i, we store a summary $\mathcal{S}(c^{h-i-1}\varepsilon, w)$. All these summaries below u' are stored sequentially. A schematic illustration of our packed structure is shown in Figure 7. The grayed subtree denotes the nodes whose summaries are packed together in order to save space.

Recall that all the type (1) summary sets are used to cover the top portion of the query path $\mathcal{P}(u, v_0)$ in block $\mathcal{B}_0$, that is, $\mathcal{RS}(u, v_0, \varepsilon)$. Clearly the packed structure still serves this purpose: we first find the u' which has v_0 as one of its descendants. Then we load $\mathcal{RS}(u, u', \varepsilon)$, followed by the summaries $\mathcal{S}(c^{h-i-1}, w)$ required in $\mathcal{RS}(u, v_0, \varepsilon)$. Loading $\mathcal{RS}(u, u', \varepsilon)$ still takes $O(1 + s_\varepsilon/B)$ I/Os, but loading the remaining individual summaries may incur many, I/Os, since they may not be stored sequentially. Nevertheless, if we ensure that all the individual summaries below u' have total size $O(s_\varepsilon)$, then loading any subset of them does not take more than $O(1 + s_\varepsilon/B)$ I/Os. Note that there are $k_h/2^i$ nodes at height i in u's subtree, the total size of all summaries below u' is

$$\sum_{i=0}^{\log k_h} \frac{k_h}{2^i} s_{c^{h-i-1}\varepsilon}. \tag{1}$$

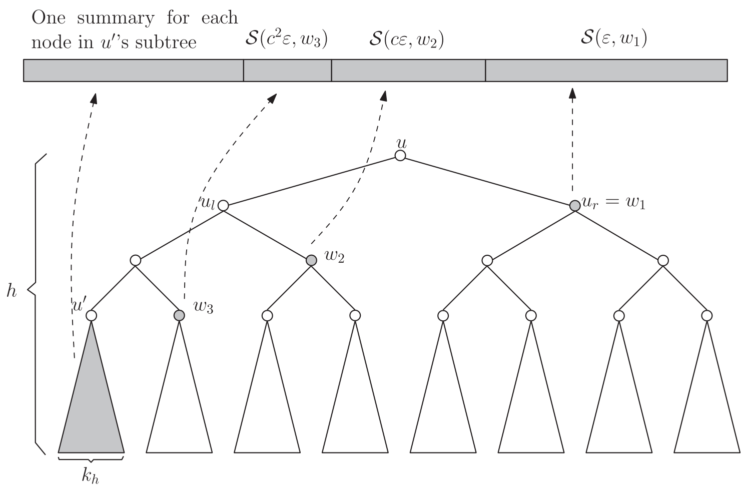
One summary for each node in u''s subtree
$\mathcal{S}(c^2\varepsilon, w_3)$
$\mathcal{S}(c\varepsilon, w_2)$
$\mathcal{S}(\varepsilon, w_1)$
u
u_l
$u_r = w_1$
w_2
u'
w_3
h
k_h

Fig. 7. An illustration of our packed structure.

Thus it is sufficient to choose k_h such that (1) is $\Theta(s_\varepsilon)$. Note that such a k_h always exists:[2] when $k_h = 1$, (1) is $s_{c^{h-1}\varepsilon} = O(s_\varepsilon)$; when k_h takes the maximum possible value $k_h = 2^{h-1}$, the last term (when $i = h$) in the summation of (1) is s_ε, so (1) is at least $\Omega(s_\varepsilon)$; and every time k_h doubles, (1) increases by at most $O(s_\varepsilon)$.

It only remains to show that by employing the packed structure, the space usage for a block is indeed $O(Bs_\varepsilon)$. For a node u at height h in $\mathcal{T}_\mathcal{B}$, the number of u''s at height $\log k_h$ under u is $2^h/k_h$. For each such u', storing $\mathcal{RS}(u, u', \varepsilon)$, as well as all the individual summaries below u', takes $O(s_\varepsilon)$ space. So the space required for node u is $O(2^h s_\varepsilon/k_h)$. There are $O(B/2^h)$ nodes u at height h. Thus the total space required is

$$O\left(\sum_{h=1}^{\log B} 2^h s_\varepsilon/k_h \cdot B/2^h\right) = O\left(\sum_{h=1}^{\log B} Bs_\varepsilon/k_h\right).$$

Note that the choice of k_h implies that

$$s_\varepsilon/k_h = O\left(\sum_{i=0}^{\log k_h} \frac{1}{2^i} s_{c^{h-i-1}\varepsilon}\right) = O\left(\sum_{i=0}^{h-1} \frac{1}{2^i} s_{c^{h-i-1}\varepsilon}\right),$$

so the total size of the packed structures in $\mathcal{B}$ is bounded by

$$\begin{aligned}\sum_{h=1}^{\log B} Bs_\varepsilon/k_h &\le B\sum_{h=0}^{\log B}\sum_{i=0}^{h-1} \frac{1}{2^i} s_{c^{h-i-1}\varepsilon}\\ &= B\sum_{h=0}^{\log B}\sum_{i=0}^{h-1} \frac{1}{2^{h-i-1}} s_{c^i\varepsilon}\end{aligned}$$

[2]We define k_h in this implicit way for its generality. When instantiating into specific summaries, there are often closed forms for k_h. For example when $s_\varepsilon = \Theta(1/\varepsilon)$ and $1 < c < 2$, $k_h = \Theta(c^h)$.

$$\begin{aligned}
&\le B\sum_{i=0}^{\log B} s_{c^i\varepsilon}\sum_{h=i}^{\log B}\frac{1}{2^{h-i-1}}\\
&\le 2B\sum_{i=0}^{\log B} s_{c^i\varepsilon}\\
&= O(Bs_\varepsilon).
\end{aligned}$$

THEOREM 2.4. *For any* $(1/2)$*-exponentially decomposable summary, a database* $\mathcal{D}$ *of* N *records can be stored in an external memory index of linear size so that a summary query can be answered in* $O(\log_B N + s_\varepsilon/B)$ *I/Os.*

Remark. One technical subtlety is that the $O(s_\varepsilon)$ combining time in internal memory does not guarantee that we can combine the $O(\log N)$ summaries in $O(\lceil s_\varepsilon/B\rceil)$ I/Os in external memory. However, if the merging algorithm only makes linear scans on the summaries, then this is not a problem, as we shall see in Section 3.

3. SUMMARIES

In this section, we demonstrate exponentially decomposable properties for the heavy hitters and quantile summary. Thus, they can be used in our optimal index in Section 2.4.

3.1. Heavy Hitters

Given a multiset D, let $f_D(x)$ be the frequency of x in D. The MG summary [Misra and Gries 1982] is a popular counter-based summary for the frequency estimation and the heavy hitters problem. We first recall how it works on a stream of items. For a parameter k, an MG summary maintains up to k items with their associated counters. There are three cases when processing an item x in the stream: (1) If x is already maintained in the summary, we increase its counter by 1. (2) If x is not maintained and there are fewer than k items in the summary, we add x into the summary with its counter set to 1. (3) If the summary maintains k items and x is not one of them, we decrement all counters by 1 and remove all items with 0 counts. For any item x in the counter set, the MG summary maintains an estimated count $\hat{f}_D(x)$ such that $f_D(x) - F_1(D)/(k+1) \le \hat{f}_D(x) \le f_D(x)$; for any item x not in the counter set, it is guaranteed that $f_D(x) \le F_1(D)/(k+1)$. By setting $k = \lfloor 1/\varepsilon\rfloor$, the MG summary has size $\lfloor 1/\varepsilon\rfloor$, and provides an additive $\varepsilon F_1(D)$ error: $f_D(x) - \varepsilon F_1(D) \le \hat{f}_D(x) \le f_D(x)$ for any x. The SpaceSaving summary [Metwally et al. 2006] is very similar to the MG summary except that the SpaceSaving summary provides an $\hat{f}_D(x)$ overestimating $f_D(x)$: $f_D(x) \le \hat{f}_D(x) < f_D(x) + \varepsilon F_1(D)$. Thus they solve the heavy hitters problem.

The MG summary is clearly decomposable. We next show that it is also α-exponentially decomposable for any $0 < \alpha < 1$. The same proof also works for the SpaceSaving summary.

Consider t multisets $D_1, \dots, D_t$ with $F_1(D_i) \le \alpha^{i-1}F_1(D_1)$ for $i = 1, \dots, t$. We set $c = 1/\sqrt{\alpha} > 1$. Given a series of MG summaries $\mathcal{S}(\varepsilon, D_1), \mathcal{S}(c\varepsilon, D_2), \dots, \mathcal{S}(c^{t-1}\varepsilon, D_t)$, we combine them by adding up the counters for the same item. Note that the total size of these summaries is bounded by

$$\sum_{j=0}^{t-1} s_{c^j\varepsilon} = \sum_{j=0}^{t-1}\frac{1}{c^j\varepsilon} = O(1/\varepsilon) = O(s_\varepsilon).$$

In order to analyze the error in the combined summary, let $f_j(x)$ denote the true frequency of item x in D_j and $\hat{f}_j(x)$ be the estimator of $f_j(x)$ in $\mathcal{S}(c^{j-1}\varepsilon, D_j)$. The combined

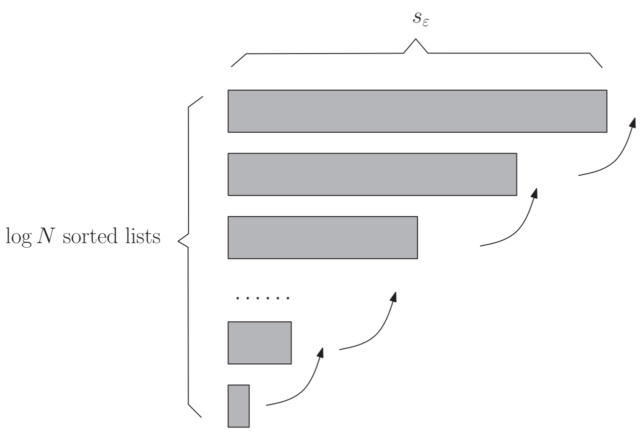

Fig. 8. An illustration of the merging procedure. The cost is bounded by the size of the largest summary.

summary uses $\sum_{j=1}^t \hat{f}_j(x)$ to estimate the true frequency of x, which is $\sum_{j=1}^t f_j(x)$. Note that

$$f_j(x) \geq \hat{f}_j(x) \geq f_j(x) - c^{j-1}\varepsilon F_1(D_j)$$

for $j = 1, \ldots, t$. Summing up the first inequality over all j yields $\sum_{j=1}^t f_j(x) \geq \sum_{j=1}^t \hat{f}_j(x)$. For the second inequality, we have

$$\begin{aligned}\sum_{j=1}^t \hat{f}_j(x) &\geq \sum_{j=1}^t f_j(x) - \sum_{j=1}^t c^{j-1}\varepsilon F_1(D_j)\\ &\geq \sum_{j=1}^t f_j(x) - \sum_{j=1}^t \left(\frac{\alpha}{\sqrt{\alpha}}\right)^{j-1} \varepsilon F_1(D_1)\\ &\geq \sum_{j=1}^t f_j(x) - \varepsilon F_1(D_1)\sum_{j=1}^t (\sqrt{\alpha})^{j-1}\\ &= \sum_{j=1}^t f_j(x) - O(\varepsilon F_1(D_1)).\end{aligned}$$

Therefore the error bound is $O(\varepsilon F_1(D_1)) = O(\varepsilon(F_1(D_1 \uplus \cdots \uplus D_t))$.

To combine the summaries, we require that each summary maintains its (item, counter) pairs in the increasing order of items (we impose an arbitrary ordering if the items are from an unordered domain). In this case, each summary can be viewed as a sorted list, and we can combine the t sorted lists into a single list, where the counters for the same item are added up. Note that if each summary is of size s_ε, then we need to employ a t-way merging algorithm and it takes $O(s_\varepsilon t \log t)$ time in internal memory and $O(\frac{s_\varepsilon t}{B} \log_{M/B} t)$ I/Os in external memory. However, when the sizes of the t summaries form a geometrically decreasing sequence, we can repeatedly perform two-way merges in a bottom-up fashion with linear total cost. Figure 8 provides an illustration of the merging procedure. The algorithm starts with an empty list. Then at step i, it merges the current list with the summary $\mathcal{S}(\varepsilon_{t+1-i}, D_{t+1-i})$. Note that in this process every counter of $\mathcal{S}(\varepsilon_j, D_j)$ is merged j times, but since the size of $\mathcal{S}(\varepsilon_j, D_j)$ is $\frac{1}{c^{j-1}\varepsilon}$, the total running time is bounded by

$$\sum_{j=1}^{t} \frac{j}{c^{j-1}\varepsilon} = O\left(\frac{1}{\varepsilon}\right) = O(s_\varepsilon).$$

In external memory, we can use the same algorithm and achieve the $O(s_\varepsilon/B)$ I/O bound if the smallest summary $\mathcal{S}(c^{t-1}\varepsilon, D_t)$ has size $\frac{1}{c^{t-1}\varepsilon} > B$; otherwise, we can take the smallest k summaries, where k is the maximum number such that the smallest k summaries can fit in one block, and merge them in main memory. In either case, we can merge the t summaries in $O(\lceil s_\varepsilon/B\rceil)$ I/Os.

THEOREM 3.1. *The MG summary and the SpaceSaving summary have size $s_\varepsilon = O(1/\varepsilon)$ and are α-exponentially decomposable for any $0 < \alpha < 1$.*

3.2. Quantiles

Recall that in the ε-approximate quantile problem, we are given a set D of N items from a totally ordered universe, and the goal is to have a summary $\mathcal{S}(\varepsilon, D)$ from which for any $0 < \phi < 1$, a record with rank in $[(\phi - \varepsilon)N, (\phi + \varepsilon)N]$ can be extracted. It is easy to obtain a quantile summary of size $O(1/\varepsilon)$: we simply sort D and take an item every εN consecutive items. Given any rank $r = \phi N$, there is always an element within rank $[r - \varepsilon N, r + \varepsilon N]$.

We now show that quantile summaries are α-exponentially decomposable. Suppose we are given a series of such quantile summaries $\mathcal{S}(\varepsilon_1, D_1), \mathcal{S}(\varepsilon_2, D_2), \ldots, \mathcal{S}(\varepsilon_t, D_t)$, for datasets $D_1, \ldots, D_t$. We combine them by sorting all the items in these summaries. We claim this forms an approximate quantile summary for $D = D_1 \cup \cdots \cup D_t$ with error at most $\sum_{j=1}^{t} \varepsilon_j F_1(D_j)$, that is, given a rank r, we can find an item in the combined summary whose rank is in $[r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j), r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)]$ in D. For an element x in the combined summary, let y_j and z_j be the two consecutive elements in $\mathcal{S}(\varepsilon_j, D_j)$ such that $y_j \le x \le z_j$. We define $r_j^{\min}(x)$ to be the rank of y_j in D_j and $r_j^{\max}(x)$ to be rank of z_j in D_j. In other words, $r_j^{\min}(x)$ (resp. $r_j^{\max}(x)$) is the minimum (resp. maximum) possible rank of x in D_j. We state the following lemma that describes the properties of $r_j^{\min}(x)$ and $r_j^{\max}(x)$.

LEMMA 3.2. *(1) For an element x in the combined summary,*

$$\sum_{j=1}^{t} r_j^{\max}(x) - \sum_{j=1}^{t} r_j^{\min}(x) \le \sum_{j=1}^{t} \varepsilon_j F_1(D_j).$$

(2) For two consecutive elements $x_1 \le x_2$ in the combined summary,

$$\sum_{j=1}^{t} r_j^{\min}(x_2) - \sum_{j=1}^{t} r_j^{\min}(x_1) \le \sum_{j=1}^{t} \varepsilon_j F_1(D_j).$$

PROOF. Since $r_j^{\max}(x)$ and $r_j^{\min}(x)$ are the local ranks of two consecutive elements in $\mathcal{S}(\varepsilon_j, D_j)$, we have $r_j^{\max}(x) - r_j^{\min}(x) \le \varepsilon_j F_1(D_j)$. Taking summation over all j, part (1) of the lemma follows. We also note that if x_1 and x_2 are consecutive in the combined summary, $r_j^{\min}(x_1)$ and $r_j^{\min}(x_2)$ are local ranks of either the same element or two consecutive elements of $\mathcal{S}(\varepsilon_j, D_j)$. In either case we have $r_j^{\min}(x_2) - r_j^{\min}(x_1) \le \varepsilon_j F_1(D_j)$. Summing over all j proves part (2) of the lemma. □

Now for each element x in the combined summary, we compute the global minimum rank $r^{\min}(x) = \sum_{j=1}^{t} r_j^{\min}(x)$. Note that all these global ranks can be computed by scanning the combined summary in sorted order. Given a query rank r, we find the smallest element x with $r^{\min}(x) \geq r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$. We claim that the actual rank of x in D is in the range $[r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j), r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)]$. Indeed, we observe that the actual rank of x in set D is in the range $[\sum_{j=1}^{t} r_j^{\min}(x), \sum_{j=1}^{t} r_j^{\max}(x)]$, so we only need to prove that this range is contained by $[r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j), r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)]$. The left side trivially follows from the choice of x. For the right side, let x' be the largest element in the new summary such that $x' \leq x$. By the choice of x, we have $\sum_{j=1}^{t} r_j^{\min}(x') < r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$. By Lemma 3.2, we have $\sum_{j=1}^{t} r_j^{\min}(x) - \sum_{j=1}^{t} r_j^{\min}(x') \leq \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$ and $\sum_{j=1}^{t} r_j^{\max}(x) - \sum_{j=1}^{t} r_j^{\min}(x) \leq \sum_{j=1}^{t} \varepsilon_j F_1(P_j)$. Summing up these three inequalities yields $\sum_{j=1}^{t} r_j^{\max}(x) \leq r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$, so the claim follows.

For α-exponentially decomposability, the t datasets have $F_1(D_i) \leq \alpha^{i-1} F_1(D_1)$ for $i = 1, \ldots, t$. We choose $c = 1/\sqrt{\alpha} > 1$. The summaries $\mathcal{S}(\varepsilon_1, D_1), \mathcal{S}(\varepsilon_2, D_2), \ldots, \mathcal{S}(\varepsilon_t, D_t)$ have $\varepsilon_i = c^{i-1}\varepsilon$. Therefore we can combine them with error

$$\begin{aligned}
\sum_{j=1}^{t} c^{j-1} \varepsilon F_1(D_j) &\leq \sum_{j=1}^{t} \left(\frac{\alpha}{\sqrt{\alpha}}\right)^{j-1} \varepsilon F_1(D_1) \\
&= \varepsilon F_1(D_1) \sum_{j=1}^{t} (\sqrt{\alpha})^{j-1} \\
&= O(\varepsilon F_1(D_1)) \\
&= O(\varepsilon F_1(D_1 \cup \cdots \cup D_t)).
\end{aligned}$$

To combine the t summaries, we notice that we are essentially merging k sorted lists with geometrically decreasing sizes, so we can adapt the algorithm in Section 3.1. The cost of merging the t summaries is therefore $O(s_\varepsilon)$ in internal memory and $O(\lceil s_\varepsilon / B \rceil)$ I/Os in external memory. The size of combined summary is

$$\sum_{j=1}^{t} \frac{1}{c^{j-1}\varepsilon} = O\left(\frac{1}{\varepsilon}\right) = O(s_\varepsilon).$$

THEOREM 3.3. *The quantile summary has size* $s_\varepsilon = O(1/\varepsilon)$ *and is* α*-exponentially decomposable for any* $0 < \alpha < 1$.

4. PRACTICAL INDEX STRUCTURES FOR SUMMARY QUERIES

Though being a nice theoretical result, our asymptotic optimal index in Section 2 suffers from the following problems when it comes to practice. The simple index for subtractive summaries is inherently static. For the optimal index for nonsubtractive summaries, (1) it is too complex to be implemented, as it uses various complex packing tricks to reduce the size of the structure to $O(N)$; (2) the hidden constants in the big-O's are quite large (the size of the index is roughly $23N$ and the query cost is about $2\log_B N + 7s_\varepsilon/B$ I/Os); and (3) the index is static, although it might be theoretically possible to make it dynamic, as hinted in Wei and Yi [2011], with some heavy data structuring machinery, but the procedure will be highly complex and painful.

In view of these deficiencies, in this section, we present a simplified and practical version of the index, which addresses all of these concerns while still has some theoretical guarantees on its performance, albeit weaker than the optimal. We first describe the static structure; later we show how to make it dynamic.

4.1. The Static Structure

Our practical index structure will only make use of the decomposable property of the summaries. It is conceptually very simple. We just build the binary tree $\mathcal{T}$ as in Section 2 and attach an ε-summary at each node of $\mathcal{T}$. For a given query range $[q_1, q_2]$, we locate the $O(\log N)$ dyadic nodes of $\mathcal{T}$ that make up the range and combine the attached summaries together.

There are still some technical choices we have to make when it comes to actual implementation. First, we decide not to build the binary tree $\mathcal{T}$ directly but to attach the required summaries to the nodes of a B-tree. The rationale is that, most likely, there is already a B-tree on the data (on the A_q attribute) to support lookup, range reporting, and aggregation queries, so we can reuse this existing index for summary queries. Let l be the maximum number of records that can be stored in a leaf block of the B-tree, and b the maximum branching factor of an internal block. Note that both l and b are on the order of $\Theta(B)$ but may differ by a constant factor depending on the actual implementation of the B-tree blocks. The N data records are stored in the leaves of the B-tree. Each leaf block stores between $l/4$ and l records; each internal block of the B-tree has fanout between $b/4$ and b except the root, which may have 2 to b children.

At each internal block $\mathcal{B}$ of the B-tree, we store an extra pointer pointing to a separate *summary pool* on disk where all the necessary summaries for this block are stored. In the summary pool, we build a binary tree $\mathcal{T}_\mathcal{B}$ on the $\leq b$ children of $\mathcal{B}$, with each leaf corresponding to a child of $\mathcal{B}$. We attach an ε-summary to each node u of $\mathcal{T}_\mathcal{B}$, summarizing all records stored below u, except if the number of records below u is smaller than βs_ε, where $\beta \geq 1$ is a constant. If $s_\varepsilon \geq B$, each ε-summary will occupy at least one disk block, so we can store the summaries in the pool separately. If $s_\varepsilon < B$, we pack multiple summaries into one disk block using the standard B-tree blocking so that each block stores a subtree of $\mathcal{T}_\mathcal{B}$ of height $\log(B/s_\varepsilon)$.

Assuming that the B-tree is already available, all the summary pools can be constructed efficiently in a bottom-up fashion. We first build the summaries at the bottom of the binary tree $\mathcal{T}$; once the two children of a node have their summaries built, we merge them into a summary for the parent. Note that there is a subtle, but critical, difference between this merge and the merge when we talk about the decomposable property of the summary. For the latter, the merged summary is required to be an $O(\varepsilon)$-summary (see the beginning of Section 2), and there may be a hidden constant in the big-O. For linear sketches like the Count-Min sketch or the AMS sketch, this hidden constant is 1. But for many other summaries, like the heavy hitters and quantile summary, the merged summary is actually a 2ε-summary, that is, the error doubles after the merge (this has also been noted as early as in [Munro and Paterson 1980]). If we insist on getting an ε-summary, the size of the merged summary must be the total size of all the summaries being merged. Getting a doubled ε is not a problem for the query procedure, because the merging there is a one-time operation. One can simply rescale ε down by a factor of 2 beforehand to compensate. However, in our level-by-level construction of the index, we need to carry out these merges repeatedly, and the error will accumulate. Thus, we need a merge operation on the summaries that preserves both the error and the size. For linear sketches like the Count-Min sketch or the AMS sketch, this is trivially doable; we will show in Section 4.2 how this can be done for heavy hitters and quantile summaries, which turns out to be crucial in the dynamic index as well. To answer a query $[q_1, q_2]$, we first search the B-tree for the two leaf blocks a and b that contains q_1 and q_2, respectively. Then for each internal block on the search path, we find the dyadic nodes in the corresponding summary pools that make up the query range (see Figure 6 for an illustration) and combine their associated ε-summaries together by the decomposable property. If a dyadic node does not have an associated summary, that means its subtree contains less than βs_ε records. In this case, we go down to the leaf blocks of the B-tree to retrieve the actual records and insert them into the summary. Finally, we scan the leaf blocks a and b to retrieve the records there that fall into the query range and insert them into the summary.

Analysis. It should be quite straightforward to analyze the size and query cost of this index. The total size of all the ε-summaries in all the summary pools is at most $2N/\beta = O(N)$, since we only attach a summary to a binary tree node u if its subtree contains at least βs_ε records, and there are $2N/(\beta s_\varepsilon)$ such nodes.

For the query cost, the initial search in the B-tree takes $O(\log_B N)$ I/Os. Then we retrieve and combine $O(\log N)$ summaries that make up the query range. If $s_\varepsilon > B/2$, these summaries are stored in $O((s_\varepsilon/B)\log N)$ blocks. If $s_\varepsilon \leq B/2$, then B/s_ε summaries are packed in a block, which corresponds to a subtree of height $\log(B/s_\varepsilon)$. So the $O(\log N)$ summaries that make up the query range are stored in $O(\log N/\log(B/s_\varepsilon))$ blocks.

Note that if a dyadic node does not have an associated summary, retrieving the actual βs_ε records will be the same as if there were a summary of size s_ε, with a constant-factor difference. Thus, the parameter β controls the trade-off between the size of the index and the query cost.

THEOREM 4.1. *Our practical index for summary queries uses linear space, answers a query in $O((s_\varepsilon/B)\log N)$ I/Os for $s_\varepsilon \geq B$, and $O(\log N/\log(B/s_\varepsilon))$ I/Os for $s_\varepsilon < B$. Given a B-tree on the A_q attribute, the index can be constructed in $O(N/B)$ I/Os and linear time.*

4.2. The Dynamic Structure for Linear Sketches

We first consider the easy case when the summary is a *linear sketch*, a sketch that is simply a linear transformation of the data frequency vector. The Count-Min sketch and the AMS-sketch are both linear sketches. Linear sketches are nice in that they are self-maintainable, that is, they support insertions and deletions without accessing the underlying data.

In this case, it is relatively easy to make the index structure dynamic. To insert or delete a record, we first do a normal insertion or deletion in the B-tree. Then we update each summary that contains the record being inserted or deleted. There are $O(\log N)$ such summaries, and they are stored in the summary pools attached to the $O(\log_B N)$ B-tree blocks on a root-to-leaf path. For $s_\varepsilon \geq B$, they can be updated in $O(\log N)$ I/Os; for $s_\varepsilon < B$, they can be updated in $O(\log N/\log(B/s_\varepsilon))$ I/Os.

As the structure of the B-tree will change as updates are performed, we also use a dynamic binary tree $\mathcal{T}_\mathcal{B}$ to organize the summaries in the summary pool attached to each internal B-tree block. For reasons that will become clearer later, we choose to use the BB(α)-tree [Baeza-Yates and Gonnet 1991]. The BB(α)-tree has the following α-balance property: for any internal node u, the number of leaves in its either subtree is at least a fraction of α of the number of leaves below u. It has been suggested to use an α between $2/11$ and $1 - \sqrt{2}/2 \approx 0.29$, and we choose $\alpha = 1/4$. The α-balance property ensures that a BB(α)-tree with n nodes has height $O(\log n)$.

When a B-tree block $\mathcal{B}$ gets split, we also split its summary pool. When $\mathcal{T}_\mathcal{B}$ is implemented as a BB(α) tree, splitting it is very easy: we just split at the root, and the α-balance property trivially still holds for both subtrees of the root. See Figure 9. Next, we insert a new child at $\mathcal{B}$'s parent block. This corresponds to inserting a new leaf in the BB(α)-tree $\mathcal{T}_{\mathcal{B}'}$, where $\mathcal{B}'$ is the parent block of $\mathcal{B}$ in the B-tree. We may need to perform a number of rotations in $\mathcal{T}_{\mathcal{B}'}$ to restore its α-balance property, while each rotation will affect the corresponding summaries as well. There are a total of four cases of rotations: two of them are shown in Figure 10, while the other two cases are similar. After each

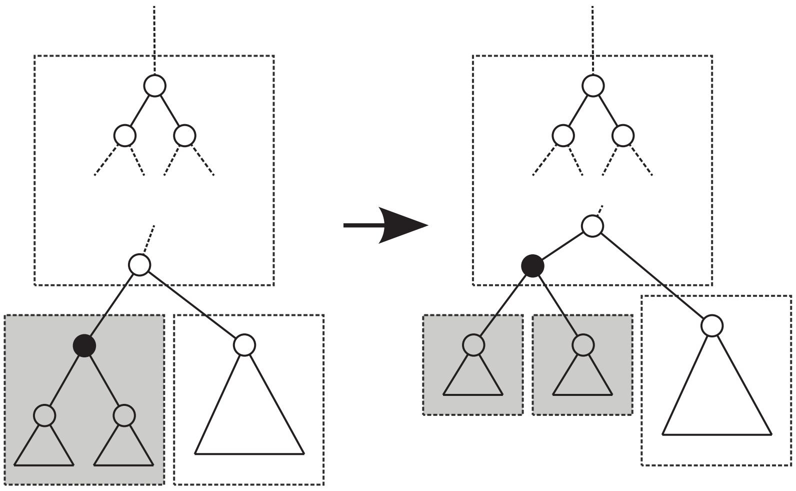
Fig. 9. Block splitting in a BB(α) tree. The solid node was the root of the block to be split and is to be inserted into the parent block after splitting.

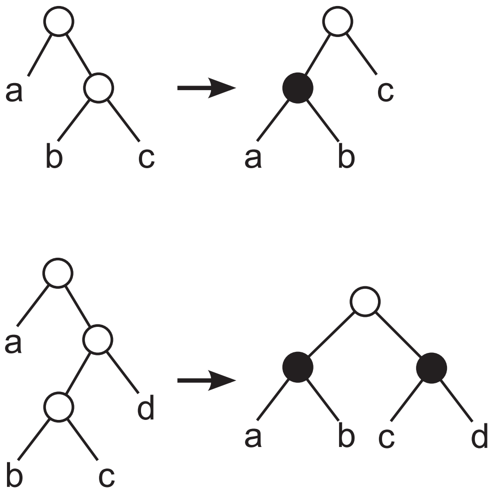

Fig. 10. Two cases of node rotation in BB(α) trees. The other two cases are symmetric. The solid nodes indicate summaries to be recalculated after rotation.

rotation, we need to recompute one or two summaries, as shown in the figure. We can recompute these summaries by merging the summaries at its two children.

After inserting a new child at $\mathcal{B}'$, it may in turn overflow, and needs to be split again. In the worst case, all the $O(\log_B N)$ B-tree blocks on a leaf-to-root path may get split, but the amortized number of block splits per insertion is merely $O(1/B)$ [Vitter 2008].

When two B-tree sibling blocks are merged, we also need to merge the corresponding summary pools. This means that we need to merge two BB(α)-trees. To do so, we first check if the α-balance property is still maintained if we simply create a new root and take these two trees as its subtrees. If so, we only need to compute the summary at the new root. If not, we rebuild the whole BB(α)-tree for the merged B-tree block. Similarly, a merge of two B-tree blocks may cause repeated merges all the way up, but the amortized number of block merges per deletion is $O(1/B)$ [Vitter 2008]. With some standard amortization arguments, we obtain the following.

THEOREM 4.2. *Our practical index for linear sketches supports insertion and deletion of records in amortized $O(\log N + \lceil s_\varepsilon/B\rceil)$ I/Os for $s_\varepsilon \geq B$, or $O(\log N/\log(B/s_\varepsilon))$ I/Os for $s_\varepsilon < B$.*

From Theorem 4.1 and 4.2, we see that there is little benefit of using an $s_\varepsilon < B$: the index size is independent of s_ε; the query and update costs only get slight improvements for $s_\varepsilon \ll B$, which also requires a careful packing of multiple small summaries. This packing is particularly difficult to maintain as the tree $\mathcal{T}_B$ dynamically changes. We feel that any benefit introduced by using an $s_\varepsilon < B$ does not warrant its additional overhead, and thus, in our implementation, we will only use $s_\varepsilon \geq B$. We will also assume $s_\varepsilon \geq B$ in the rest of this article.

4.3. The Dynamic Structure for Heavy Hitters and Quantile Summaries

Unfortunately, most heavy hitters and quantile summaries are not self-maintainable [Misra and Gries 1982; Metwally et al. 2006; Greenwald and Khanna 2001]; they support only insertions but not deletions. Using terms from the streaming literature, these summaries work only in the *cash register model*, but not the *turnstile model* [Muthukrishnan 2005]. In addition, even to just support insertions, the summary size often increases. For example, the quantile summary of the optimal size $O(1/\varepsilon)$ supports neither insertions or deletions. The GK summary [Greenwald and Khanna 2001] supports insertions but the size increases to $O((1/\varepsilon)\log N)$, with a fairly complicated structure and insertion algorithms. Note that the size increase is due to the need of extra working space; the actual summary itself in the final output still has size $O(1/\varepsilon)$. There are some randomized quantile summaries in the turnstile model [Gilbert et al. 2002; Cormode and Muthukrishnan 2005], but their sizes are quite large, and they only work when the data items are drawn from a fixed universe (thus will not work for floating-point numbers). Letting u be the universe size, the best quantile summary in the turnstile model has size (assuming a constant failure probability) $O((1/\varepsilon)\log^2 u \log\log u)$ [Cormode and Muthukrishnan 2005]. Although logarithmic factors are usually tolerable in theory, but in this case they cause major problems in practice: With the data being 32-bit integers, $\log^2 u$ is 1,024, leading to a huge space blowup of the index.

Fortunately, our use of summaries is different from the streaming model in the sense that the underlying data being summarized is in fact always available (it is stored in the leaves of the B-tree), though it may be costly to access. Whereas in the streaming model, the data is thrown away after passing through the sketch. This allows us to use a summary that is "almost" self-maintainable, that is, we can still access the underlying data when necessary as updates are being handled, but of course we should do so only occasionally. Another crucial feature in our setting is that we are maintaining many summaries that form a hierarchical structure, as opposed to a single sketch in the streaming model. Thus, information from one summary may be useful for other summaries, and it is possible to exploit this interdependence to update the summaries more efficiently.

It turns out that a new quantile summary recently proposed by Huang et al. [2011] serves our purpose well. As we will see in a moment, it is a summary that is self-maintainable for most updates; only with a small probability do we need to access outside information in order to maintain it. It is very simple, and has the optimal size $O(1/\varepsilon)$ with a small hidden constant. We first briefly describe this summary, then show how to use it in our dynamic index structure.

4.3.1. The Quantile Summary. Let D be the dataset being summarized. The summary simply consists of a list of $\langle$item, rank$\rangle$ pairs. For a probability $p = \Theta(1/(\varepsilon|D|))$, we sample each data item in D with probability p into the list. For each sampled item, we also store its rank in D (by the A_s attribute). Recall that the rank of x is the number of items in D smaller than x. Thus the summary has (expected) size $s_\varepsilon = p|D| = O(1/\varepsilon)$. To build the summary, we first sort D, and then make a scan. During the scan, we sample every item with probability p into the summary, while its rank is simply its position in the sorted list.

The summary can be used to answer two types of queries: item-to-rank queries, and rank-to-item queries (i.e., finding quantiles). Given any item x, we can find its approximate rank in D as

$$\hat{r}(x, D) = \begin{cases} r(pred(x), D) + 1/p, & \text{if } pred(x) \text{ exists;} \\ 0, & \text{else.} \end{cases} \tag{2}$$

Here, $r(x, D)$ denotes the rank of x in D, and $pred(x)$ denotes the predecessor of x (i.e., the largest item smaller than or equal to x) in the summary. It is shown in Huang et al. [2011] that $\hat{r}(x, D)$ is an unbiased estimator of $r(x, D)$ with variance $\le 1/p^2$. It should be emphasized that for this guarantee to hold, the queried item x should be picked independent of the randomization within the summary.

The quantiles of D, which are what we really care about, can be found as follows. To return the ϕ-quantile of D, we simply find the item in the summary whose rank is closest to $\phi|D|$. It is shown [Huang et al. 2011] that this will be an ε-approximate ϕ-quantile of D with at least constant probability, and this constant can be made arbitrarily small by increasing p by an appropriate constant factor. To find the item with the closest rank to $\phi|D|$, one can do a binary search in $O(\log(1/\varepsilon))$ time. But often we want to return all the quantiles for $\phi = \varepsilon, 2\varepsilon, 3\varepsilon, \ldots, 1 - \varepsilon$ so as to get an approximate distribution of D. In this case, we can scan the summary once to compute all of them, in just $O(1/\varepsilon)$ time.

This summary can also be used to find the heavy hitters with error ε using the standard reduction: for the heavy hitters problem, there is no ordering on the items, but we impose an arbitrary ordering and break the ties using any consistent tie breaker. Then we find the ε-approximate ϕ-quantiles for $\phi = \varepsilon, 2\varepsilon, 3\varepsilon, \ldots, 1 - \varepsilon$. For any item x, its frequency in D, $f_D(x)$, can be estimated by counting the number of these quantiles that are the same as x, multiplied by $\varepsilon|D|$. If all the quantiles are exact, this will give us an estimate of $f_D(x)$ with error at most $2\varepsilon|D|$. But since each of these quantiles may be off by $\varepsilon|D|$ in terms of rank, this adds another error of $\varepsilon|D|$. So we can estimate $f_D(x)$ with error $4\varepsilon|D|$. Rescaling ε can reduce the error to $\varepsilon|D|$ and thus solve the heavy hitters problem.

Finally, we note that even if the rank of an item in the summary is not its exact rank in D, but an unbiased estimator with variance at most $(c\varepsilon|D|)^2$ for some constant c, the preceding procedures still return the approximate quantiles and heavy hitters. In particular, this simply adds $(c\varepsilon|D|)^2$ to the variance of Equation (2)[3], hence introducing an additive $c\varepsilon$ error term, which can be compensated by rescaling ε beforehand. For this reason, the summary is decomposable. Specifically, given the summaries for t datasets $D_1, \ldots, D_t$ that have been constructed pairwise independently, we can merge them into a summary for $D = D_1 \cup \cdots \cup D_t$ as follows. We first decide the sampling probability $p = \Theta(1/(\varepsilon|D|))$ for the merged summary, which must be smaller than the sampling probability p_i for each individual summary of D_i. Then for each sampled item in the summary of D_i, we subsample it into the merged summary with probability p/p_i. If it is sampled, we need to compute its rank in D, which is the sum of its ranks in

[3]There is a technical condition for this to hold, that the estimator of the rank of any item x in the summary must be independent of the sampling of all items in D that are greater than x, while it is allowed to depend on the sampling decisions for items smaller than x. This condition is satisfied in all our uses of the summary.

$D_1, \dots, D_t$. Its rank in D_i is already available, while its rank in D_j, $j \neq i$, is estimated using Equation (2) in the summary of D_j. Thus, the variance of the estimated rank is

$$\sum_{1 \le j \le t, j \neq i} \frac{1}{p_i^2} \le \sum_{j=1}^{t} O\big((\varepsilon |D_j|)^2\big) \le O\big(\varepsilon^2 (|D_1| + \cdots + |D_t|)^2\big) = O(\varepsilon |D|^2),$$

as desired. Note that the merging can be done in time linear to the total size of the t summaries.

4.3.2. Maintaining the Summaries in the Index. We will now see that this summary is an "almost" self-maintainable summary that suits our needs. For cases when it cannot be self-maintained, we exploit the fact that all the summaries stored in our index form a binary tree $\mathcal{T}$. In particular, the dataset summarized at a node $u \in \mathcal{T}$ is the union of the datasets summarized by the two children of u. We define the *weight* of a node u (summary) in $\mathcal{T}$ as the number of records stored below u (summarized by the summary), denoted by $w(u)$. For now we will assume that $\mathcal{T}$ is *weight-balanced*. More precisely, this requires that, for any node u, the weight of its either child is at least a fraction of γ of $w(u)$, for some constant $0 < \gamma < 1/2$. Clearly, this constraint can be easily met when we build the index; we will show later how it can be maintained dynamically.

First, for a node $u \in T$, the sampling probability p_u at u will not always change as $w(u)$ changes; p_u just needs to kept on the order of $\Theta(1/(\varepsilon w(u)))$. In our implementation, we maintain the following invariant.

Invariant 4.1. For any $u \in \mathcal{T}$, $\frac{1}{\varepsilon w(u)} \le p_u \le \frac{4}{\varepsilon w(u)}$.

Initially, p_u is set to $\frac{2}{\varepsilon w(u)}$.

There are two more invariants that we maintain. During the updates, we will not maintain the ranks of the sampled items in the summaries exactly, as that will be costly. Instead, we make sure of the following.

Invariant 4.2. The rank of every sampled item in the summary at any node $u \in \mathcal{T}$ is an unbiased estimator with variance at most $(c\varepsilon w(u))^2$, where c is a constant that depends on γ.

Finally, we need the summaries to be pairwise independent unless they have an ancestor-descendant relationship. This is needed for answering a summary query when we merge $O(\log N)$ summaries together that correspond to the dyadic nodes of $\mathcal{T}$ that make up the query range.

Invariant 4.3. For any two nodes $u, v \in \mathcal{T}$ that have no ancestor-descendant relationship, the summaries at u and v are independent.

When we insert or delete an item x, we need to insert it into or delete it from all the $O(\log N)$ summaries that contain x. These summaries are on a root-to-leaf path of $\mathcal{T}$, and we update them in a bottom-up fashion.

Handling Insertions. Let us first consider insertions, and let x be the new item inserted below a node $u \in \mathcal{T}$. Recall that each item is sampled into the summary with probability p_u, so x should be sampled into the summary with probability p_u as well. If it is not sampled, the summary can be easily maintained: We simply increment the ranks of all the existing items in the summary that are greater than x. If it is sampled, besides updating the ranks of the existing items, we also need to compute the rank of x. This is a case where the summary is not self-maintainable. To compute the rank of x, we look at the summaries at the two children of u, denoted v_1 and v_2, since the rank of x at u is simply the sum of its ranks at v_1 and v_2. If v_1 (respectively

v_2) does not have an associated summary, that means the number of records below is less than βs_ε. In this case, we simply retrieve the $\leq \beta s_\varepsilon$ records from the B-tree and compute the rank of x directly. Otherwise, we estimate its rank using Equation (2), which gives an unbiased estimator with variance at most $1/p_{v_1}^2$ (respectively $1/p_{v_2}^2$). However, Equation (2) assumes $r(pred(x), D)$ is an accurate rank. In our case, it may also have been estimated from summaries further down. But by Invariant 4.2, it is an unbiased estimator with variance at most $(c\varepsilon w(v_1))^2$ (respectively $(c\varepsilon w(v_2))^2$). So the estimated rank of x at u is an unbiased estimator with variance at most

$$\frac{1}{p_{v_1}^2} + (c\varepsilon w(v_1))^2 + \frac{1}{p_{v_2}^2} + (c\varepsilon w(v_2))^2. \tag{3}$$

To meet Invariant 4.2 for u, we need Eq. (3) to be no more than $(c\varepsilon w(u))^2$. As the tree is weight-balanced, we have

$$\gamma w(u) \leq w(v_1) \leq (1-\gamma)w(u) \text{ and } \gamma w(u) \leq w(v_2) \leq (1-\gamma)w(u).$$

By Invariant 4.1 on v_1 and v_2, we have

$$\frac{1}{p_{v_1}^2} \leq (\varepsilon w(v_1))^2 \text{ and } \frac{1}{p_{v_2}^2} \leq (\varepsilon w(v_2))^2.$$

Thus, Eq. (3) is at most

$$\begin{aligned}
&(\varepsilon w(v_1))^2 + (\varepsilon w(v_2))^2 + (c\varepsilon w(v_1))^2 + (c\varepsilon w(v_2))^2 \\
&\quad = \varepsilon^2(c^2+1)\big(w(v_1)^2 + w(v_2)^2\big) \\
&\quad \leq \varepsilon^2(c^2+1)((\gamma w(u))^2 + ((1-\gamma)w(u))^2) \qquad \text{(maximized when most unbalanced)} \\
&\quad = \varepsilon^2(c^2+1)(1-2\gamma+2\gamma^2)(w(u))^2.
\end{aligned}$$

So, it suffices to have $(c^2+1)(1-2\gamma+2\gamma^2) = c^2$, i.e., $c = \sqrt{\frac{1-2\gamma+2\gamma^2}{2\gamma(1-\gamma)}}$.

Finally, after a number of insertions, $w(u)$ may increase to a point such that $p_u > \frac{4}{\varepsilon w(u)}$, violating Invariant 4.1. When this happens, we set $p_u \leftarrow p_u/2$ and simply subsample each item in the current summary with probability 1/2. Also, if u did not have a summary, but $w(u)$ has become βs_ε, we build a new summary at u.

Handling Deletions. Deletions are handled using similar ideas, though the detailed procedures are different. When an item x below u is deleted, we also delete it from the summary at u. If x is currently one of the sampled items in the summary, we delete it from the summary. Then we decrement the ranks of the other items in the summary that are greater than x.

After a number of deletions, $w(u)$ may decrease to a point such that $p_i < \frac{1}{\varepsilon w(u)}$. When this happens, we will build a new summary for u from the summaries at v_1 and v_2. The new summary has $p_u = \min\{\frac{2}{\varepsilon w(u)}, p_{v_1}, p_{v_2}\}$, where v_1 and v_2 are the two children of u. Since the tree is weight-balanced, $w(v_1) \leq (1-\gamma)w(u)$, which means that

$$\begin{aligned}
p_{v_1} &\geq \frac{1}{\varepsilon w(v_1)} \qquad \text{(by Invariant 4.1)} \\
&\geq \frac{1}{\varepsilon(1-\gamma)w(u)} > \frac{1}{\varepsilon w(u)}.
\end{aligned}$$

The same holds for p_{v_2}, so the new p_u must satisfy Invariant 4.1 for u. Then we subsample each item in the summary at v_1 into the new summary at u with probability p_u/p_{v_1}. If v_1 does not have a summary, we scan the $\leq \beta s_\varepsilon$ records below u and sample each one with probability p_u. For each subsampled item x, we estimate its rank in the summary of v_2 using Equation (2). Then we add it to the rank of x in v_1 to become the rank of x at u. We then perform the same subsampling for items in the summary of v_2 and estimate their ranks in v_1. Finally, when $w(u) < \beta s_\varepsilon$ after a deletion, we delete the summary at u.

It remains to show that the newly constructed summary for u meets Invariant 4.2. We will only consider an item x subsampled from v_1; the same analysis works for items from v_2. The variance of the rank of x at u consists of three components: its original variance at v_1, which is at most $(c\varepsilon w(v_1))^2$, the estimation error from Equation (2), which is $1/p_{v_2}^2$, and the variance of the rank of x's predecessor at v_2, which is $(c\varepsilon w(v_2))^2$. Thus, the total variance is

$$(c\varepsilon w(v_1))^2 + \frac{1}{p_{v_2}^2} + (c\varepsilon w(v_2))^2,$$

which is less than Eq. (3), so must be smaller than $(c\varepsilon w(u))^2$, as desired.

Also note that this algorithm for building a new summary at u from those at v_1 and v_2 is exactly the error-preserving merge operation needed for the bottom-up construction of the whole index, as required in Section 4.1.

Finally, as we only consult the children of u when updating the summary at u, Invariant 4.3 is automatically preserved.

4.3.3. Maintaining Weight Balance. Now, the only remaining task is to maintain the weight balance of the binary tree $\mathcal{T}$ that organizes all our summaries. Recall that we use a BB(α)-tree for $\mathcal{T}_\mathcal{B}$ of each B-tree block, which is already weight balanced, so we only need to make the B-tree itself weight balanced.

Weight-Balanced B-Tree. A weight-balanced B-tree is structurally the same as a B-tree and differs only in the merge/split rules of the tree blocks. We define the *weight* of a B-tree block $\mathcal{B}$ to be the number of records stored in the subtree below $\mathcal{B}$. We store and maintain the weight of each block $\mathcal{B}$ at $\mathcal{B}$'s parent block. Thus we need to store three fields for each child of an internal B-tree block: the routing key, a pointer, and the weight.

To maintain the weight balance, we require a B-tree block at level i (the leaves are at level 0), except the root, to have weight between $\frac{1}{4}l(\frac{1}{2}b)^i$ and $l(\frac{1}{2}b)^i$; the weight of the root is only subject to the upper bound but not the lower bound. We impose the usual constraint that each internal block should have at most b children so that they fit in one disk block. We no longer impose a lower bound on the fanout, but the weight constraint implicitly sets a $\frac{\frac{1}{4}(\frac{1}{2}b)^i}{(\frac{1}{2}b)^{i-1}} = \frac{1}{8}b$ lower bound. Note that the weight constraint increases by a factor of $\frac{1}{2}b$ every level, so the average fanout is $\frac{1}{2}b$. Such a B-tree is called a *weight-balanced B-tree*. It was first proposed by Arge and Vitter [2003], but to the best of our knowledge, it has not been implemented in practice yet, probably due to a lack of application. Interestingly, it turns out this weight balance is crucial for our purpose, as we have seen earlier. Our description of the weight-balanced B-tree here is slightly different from Arge and Vitter [2003], with improved constants (e.g., the average fanout is $\frac{1}{4}b$ in Arge and Vitter [2003]).

The leaf blocks of a weight-balanced B-tree are maintained in exactly the same way as in a normal B-tree. Specifically, when a leaf block gets overflowed, it is splits into two, each of weight $\frac{1}{2}l$ (or $\frac{1}{2}l + 1$). When a leaf block underflows, we first try to merge with one of its two siblings, if the merged block is not full; otherwise, we merge it with a sibling, followed by an immediate split. In the latter case, each block from the split must have weight between $\frac{1}{2}l$ and $\frac{5}{8}l$.

The internal blocks of a weight-balanced B-tree are maintained similarly. More precisely, there are the following four cases.

(1) When the weight of a block at level i is above the upper bound $l(\frac{1}{2}b)^i$, we do a weight-split of the block into two, so that each resulting block has weight at least $\frac{1}{3}l(\frac{1}{2}b)^i$.

(2) When the fanout of a block is more than b but its weight is within the limits, we do a fanout-split of the block so that each resulting block has fanout $\frac{1}{2}b$. Note that the weight of each resulting block is at least $\frac{1}{4}l(\frac{1}{2}b)^{i-1} \cdot \frac{1}{2}b = \frac{1}{4}l(\frac{1}{2}b)^i$, so it satisfies the weight constraint.

(3.1) When an internal block has weight below $\frac{1}{4}(\frac{1}{2}b)^i$, we first try to merge it with one of two siblings, if the merged block has weight no more than $l(\frac{1}{2}b)^i$.

(3.2) Otherwise, we merge it with a sibling, followed by an immediate weight-split. In this case, each resulting block must have weight between $\frac{1}{2}l(\frac{1}{2}b)^i$ and $\frac{5}{8}l(\frac{1}{2}b)^i$. (More precisely, the weight must be between $\frac{1}{2}l(\frac{1}{2}b)^i - l(\frac{1}{2}b)^{i-1}$ and $\frac{5}{8}l(\frac{1}{2}b)^i + l(\frac{1}{2}b)^{i-1}$, since a child may have weight $l(\frac{1}{2}b)^{i-1}$ and we cannot split a child.)

After an insertion/deletion in the B-tree, we perform these splits/merges of blocks in a bottom-up fashion, for all blocks that either violate the weight or the fanout constraint. After a split, we add one more child in the parent block; after a merge, we delete a child in the parent block. When the root of the B-tree splits, we create a new root node, and the B-tree height increases by one; when the root block has just one child, it is deleted, and the B-tree height decreases by one.

Maintaining the Weight Balance of $\mathcal{T}_\mathcal{B}$. Recall that for each B-tree block $\mathcal{B}$, we organize its summary pool by a binary tree $\mathcal{T}_\mathcal{B}$; all these small binary trees make up the entire $\mathcal{T}$. To make sure the $\mathcal{T}$ is weight-balanced, we also need each individual $\mathcal{T}_\mathcal{B}$ to be weight-balanced. Recall that each $\mathcal{T}_\mathcal{B}$ is implemented as a a BB(α)-tree [Baeza-Yates and Gonnet 1991], with the α-balance property that for any internal node u, the number of leaves in its either subtree is at least a fraction of α of the number of leaves below u. As we choose $\alpha = 1/4$, this is equivalent to saying that the sizes of any two sibling subtrees (in terms of the number of leaves in $\mathcal{T}_\mathcal{B}$) differ by at most a factor of 3. Each leaf of $\mathcal{T}_\mathcal{B}$ corresponds to a child of $\mathcal{B}$, while the weights of the children of $\mathcal{B}$ differ by at most a factor of 4 by the weight-balanced B-tree, therefore the weights of any two sibling nodes in $\mathcal{T}_\mathcal{B}$ differ by at most a factor of $3 \times 4 = 12$.

When we insert or delete a child of $\mathcal{B}$, we also need to insert or delete a leaf in $\mathcal{T}_\mathcal{B}$, which may result in the tree getting out of balance. Then we perform the necessary rotations to restore the α-balance as in Figure 10, and recompute the affected summaries as before, using the same merging algorithm in Section 4.3.2. Note that although the two children of the new summary were not siblings in the tree before the rotation, their weight still only differs by a constant factor as they share a common ancestor within three hops. Thus the merging algorithm still works correctly, though it requires a smaller γ.

Whenever an internal B-tree block $\mathcal{B}$ splits into $\mathcal{B}'$ and $\mathcal{B}''$, under cases (1) and (3.2), we also need to split $\mathcal{T}_\mathcal{B}$. We first check if splitting at the root of $\mathcal{T}_\mathcal{B}$ satisfies the required weight constraint. If so, we can split $\mathcal{T}_\mathcal{B}$ easily. Otherwise, we rebuild two new trees $\mathcal{T}_{\mathcal{B}'}$ and $\mathcal{T}_{\mathcal{B}''}$, from the children of $\mathcal{B}'$ and $\mathcal{B}''$. During the rebuilding, we always try to keep the subtrees as balanced as possible. When the tree is built, we compute all the summaries in a bottom-up fashion. Similarly, when two B-tree blocks $\mathcal{B}'$ and $\mathcal{B}''$ merge under case (2) and (3.1), we also merge $\mathcal{T}_{\mathcal{B}'}$ and $\mathcal{T}_{\mathcal{B}''}$. We first check if their sizes meet the BB(α) tree constraint, if they are to become siblings. If so, we simply create a new root and put $\mathcal{T}_{\mathcal{B}'}$ and $\mathcal{T}_{\mathcal{B}''}$ as its two subtrees. Otherwise, we build a new tree from all the children of the merged block.

Putting Everything Together. Now, everything has been in place for a dynamic index for heavy hitters and quantile summaries. Here we give a brief review of the update procedure and analyze its I/O cost.

(1) We first search down the B-tree and insert or delete it in one of the leaf blocks of the B-tree. Along the way, we also increment or decrement the weights of all blocks above (and including) this leaf block. This takes $O(\log_B N)$ I/Os.
(2) In a bottom-up fashion, we update the $O(\log N)$ summaries that contain the newly inserted or deleted item. These summaries are on a leaf-to-root path on the binary tree $\mathcal{T}$. Updating each summary takes $O(s_\varepsilon/B)$ I/Os.
(3) Starting from the leaf block, we split or merge the B-tree blocks in a bottom-up fashion to maintain the weight and fanout constraints as necessary. After each split or merge of a B-tree block, we also split or merge the associated summary pools. As there are $O(B)$ summaries in a summary pool, and in the some cases, we may need to rebuild them all, so it takes $O(Bs_\varepsilon/B) = O(s_\varepsilon)$ I/Os. However, one can check that, for a B-tree block, its weight has to change by a constant factor before another weight split/merge, so the amortized number of weight splits/merges per update for a level i block is $O(\frac{1}{l(b/2)^i})$. Summing over all levels, it is just $O(1/B)$. Also, the amortized number of fanout splits is still $O(1/B)$, the same as in a normal B-tree. Thus, the amortized cost for splitting or merging the summary pools is $O(s_\varepsilon/B)$ I/Os.
(4) After a split/merge of the B-tree block, we insert/delete a child in its parent block. This corresponds to inserting/deleting a leaf in the BB(α) tree in the parent block. This insertion/deletion may lead to $O(\log B)$ rotations in the tree. Each rotation requires recomputing one or two summaries, which takes $O(s_\varepsilon/B)$ I/Os.

Summing over all costs, we can conclude.

THEOREM 4.3. *Our practical index for heavy hitters and quantile summaries supports insertion and deletion of records in amortized $O(s_\varepsilon/B \cdot \log N)$ I/Os for $s_\varepsilon \geq B$.*

5. EXPERIMENTS

We have implemented the index structure described in the previous section in C++. The summaries we support include the Count-Min sketch, AMS sketch, heavy hitters, and quantiles (the latter two are supported by the same summary described in Section 4.3.1). Our index is fully dynamic, supporting both insertion and deletion of records. We have also built a Web interface at `http://i4sq-demo.appspot.com` to demonstrate a subset of the functions supported by our index. In this section, we report our experimental findings on the effectiveness and efficiency of our index structure.

5.1. Datasets

We used two real datasets for our experimental study.

The first dataset is the WorldCup98 dataset.[4] It consists of 1.3 billion requests made to the 1998 World Cup website. Each request consists of several attributes. We took the time stamps of the request as the key (i.e., the A_q attribute) for our index, and built the summaries on the requested URLs (the A_s attribute). There are about 90,000 distinct URLs in total. They have been anonymized to distinct integers, which are called the objectID. Each time stamp and objectID can be fit into a 32-bit integer. We used the

[4]`http://ita.ee.lbl.gov/html/contrib/worldcup.html.`

Table I. Parameters Used in the Experiments

β	$\{1, 2, 4\}$
ε	0.01 for Count-Min, 0.1 for AMS, 0.005 for quantile
Block size	4096 bytes

Table II. Stats about the Indexes

Data set	WorldCup			MPCAT-OBS		
β	1	2	4	1	2	4
# records	1,352,804,107			87,688,123		
Block size (bytes)	4,096					
# leaf blocks	3,789,367			245,625		
# index blocks	56,558			3,666		
# summary blocks	3,789,366	2,006,134	891,615	245,624	130,040	57,791

time stamp as the key and built heavy hitters, AMS and Count-Min sketch summaries, respectively, on objectID.

The second dataset we used is the MPCAT-OBS dataset.[5] It contains observation records of minor planets submitted to the Minor Planet Center from Jan 26, 1802 until Apr 27, 2012. We only used the optical observation records, and there are 87 million of them. Each record includes attributes, such as time stamp, observatory, right ascension (RA), and declination (Decl) of the planet.[6] We used the timestamp as the key, and built summaries on the other three attributes, respectively. The observatories have been mapped to 32-bit integers, upon which we built the heavy hitters, AMS, and Count-Min sketch summaries. RA and Decl are stored as double precision floating-point numbers, and we built quantile summaries on them.

5.2. Experiment Setup

The experiments were performed on a PC with 2G memory and a 3GHz CPU. The block size was set to 4,096 bytes. We set the accuracy parameters appropriately (see Table I) for various summaries so that a single summary fits in a block.

We also tested the index with different values of β. Recall that β controls the trade-off between the size of the index and its query efficiency: the larger β, the smaller the index becomes, but queries also take longer time. Table II lists some statistics about the size of the indexes over the two datasets. Recall that our index is built on top of a B-tree. So we first built a B-tree on these records (with time stamp as the key) with a load factor of 70%. Then we constructed the all necessary summary pools and attach them to the internal blocks of the B-tree. The total size of all the summaries is controlled by the parameter β: it is roughly $1/\beta$ of the B-tree size.

5.3. Use Cases

We first showcase a few example queries supported by our index on how they convey much richer information about the records in the query range than simple aggregates like count or average.

We built the index on the MPCAT-OBS dataset with the time stamp as the key. Summaries were built on the observatory and right ascension attributes, respectively. We tested four query ranges, as shown in Table III, where query length is the number of

[5] `http://www.minorplanetcenter.net/iau/ecs/mpcat-obs/mpcat-obs.html`.

[6] Right ascension (RA) and declination (Decl) are astronomical terms of coordinates, which are used in the dataset to mark the location of the observed planets. RA ranges from 0 to 24 hours and Decl ranges from −90 to 90 degrees.

Table III. Queries

#	Start Date	End Date	Length
Q1	1800.01.01	1900.01.01	9,993
Q2	1900.01.01	2000.01.01	4,827,585
Q3	2000.01.01	2100.01.01	82,850,545
Q4	1800.01.01	2100.01.01	87,688,123

Table IV. Top Observatories in Q1

Observatory	Est. rank	Est. occurrences %	True rank	True occurences %	Error
Leipzig (since 1861)	1	6.68	1	6.81	0.13
Vienna (since 1879)	2	6.18	2	6.34	0.16
U.S. Naval Obs., Washington (before 1893)	3	5.16	4	5.36	0.20
Arcetri Obs., Florence	4	5.11	5	5.15	0.04
Geocentric	5	4.93	3	5.36	0.43
Berlin (1835–1913)	6	4.70	6	4.83	0.13
Leiden	7	4.50	7	4.42	0.08
Marseilles	8	3.64	8	3.74	0.10
Kremsmunster	9	3.47	10	3.26	0.21
Hamburg (before 1909)	10	3.46	9	3.47	0.01

Table V. Top Observatories in Q2

Observatory	Est. rank	Est. occurrences %	True rank	True occurences %	Error
Lincoln Laboratory ETS, New Mexico	1	36.36	1	36.69	0.33
Steward Obs., Kitt Peak-Spacewatch	2	16.99	2	17.21	0.22
Lowell Obs.-LONEOS	3	7.18	3	7.32	0.14
European Southern Obs., La Silla	4	4.43	4	4.52	0.09
Palomar Mountain	5	4.08	5	4.14	0.06
Catalina Sky Survey	6	3.12	6	3.32	0.20
Haleakala-NEAT/GEODSS	7	1.88	7	1.75	0.13
Klet Obs., Ceske Budejovice	8	1.67	11	1.13	0.54
Caussols-ODAS	9	1.37	12	0.88	0.49
Peking Obs., Xinglong Station	10	1.35	10	1.22	0.13

records in the range. For each query range, the heavy hitters summary were extracted on the observatory attribute, while the quantile summary were extracted on the right ascension attribute. Note that the first three query ranges are disjoint, while Q4 is the union of the first three queries, which is in fact the entire dataset. The lengths of the four queries are significantly different (much more observations have been made in this century), so as to see how they affect the accuracy of the returned summaries (they should not).

Tables IV–VII reports the heavy hitters summaries for (Q1)–(Q4) on the observatory attribute, namely, the observatories with the highest number of records. In the tables, we give both estimated numbers of occurrences extracted from the summary, as well as the true numbers, which we computed from the data directly. Recall that we are actually using the quantile summary to estimate the occurrences, so an ε error in the quantiles translates to an 4ε error in terms of occurrences (see Section 4.3.1). In practice, we see from the tables that the errors are well below the $4\varepsilon = 3.2\%$ guarantee. Note that these errors are absolute errors; the relative errors could be high when the

Table VI. Top Observatories in Q3

Observatory	Est. rank	Est. occurrences %	True rank	True occurences %	Error
Lincoln Laboratory ETS, New Mexico	1	36.30	1	36.35	0.05
Mt. Lemmon Survey	2	12.96	2	13.68	0.72
Catalina Sky Survey	3	12.82	3	12.94	0.12
Steward Obs., Kitt Peak-Spacewatch	4	9.93	4	10.14	0.21
Lowell Obs.-LONEOS	5	6.23	5	6.05	0.19
Palomar Mountain/NEAT	6	5.06	6	4.70	0.37
Pan-STARRS 1, Haleakala	7	3.62	7	3.17	0.45
Siding Spring Survey	8	2.22	8	2.13	0.09
Loomberah	9	1.49	212	0.00	1.49
OAM Obs., La Sagra	10	1.28	10	1.40	0.11

Table VII. Top Observatories in Q4

Observatory	Est. rank	Est. occurrences %	True rank	True occurences %	Error
Lincoln Laboratory ETS, New Mexico	1	36.08	1	36.37	0.29
Mt. Lemmon Survey	2	12.77	2	12.93	0.16
Catalina Sky Survey	3	12.62	3	12.41	0.22
Steward Obs., Kitt Peak-Spacewatch	4	10.30	4	10.53	0.23
Lowell Obs.-LONEOS	5	6.48	5	6.12	0.37
Palomar Mountain/NEAT	6	4.44	6	4.44	0.00
Pan-STARRS 1, Haleakala	7	3.06	7	3.00	0.07
Purple Mountain Obs., XuYi Station	8	1.65	12	0.73	0.92
Haleakala-AMOS	9	1.49	9	1.46	0.02
Siding Spring Survey	10	1.42	8	2.01	0.59

true occurrences are low, for which no guarantee is provided. Also, we observe from the tables that the query length does not have any impact on the accuracy of the extracted summaries, noting that the percentages reported are all with respect to the length of the query range.

Next, we extracted the quantile summaries on the RA attribute for these queries. In Figure 11, we plot both the quantiles in summary and true value-rank curve (i.e., the cdf of the data distribution). We can see that the quantiles in the summary fit the real data really well with almost negligible differences.

The reader is invited to check out our Web demo to play with more queries and see how the system responds.

We will not demonstrate the effectiveness of the Count-Min and the AMS sketch. Since they are linear sketches, the sketch returned by our index is exactly the same as if the sketch would have been computed from the underlying data directly. They can thus be used in exactly the same way as how they would have been used otherwise. But these summaries will be included in our efficiency study of the index structure.

5.4. Baseline Solutions

Summary queries, such as those previously showcased, can also be answered by the following two baseline solutions, which we also implemented for comparison.

Standard B-Tree. We simply build a B-Tree on the dataset to support range-reporting queries. To obtain a summary, we retrieve all elements in the given range and feed them into a streaming algorithm. For heavy hitters, we used the classical

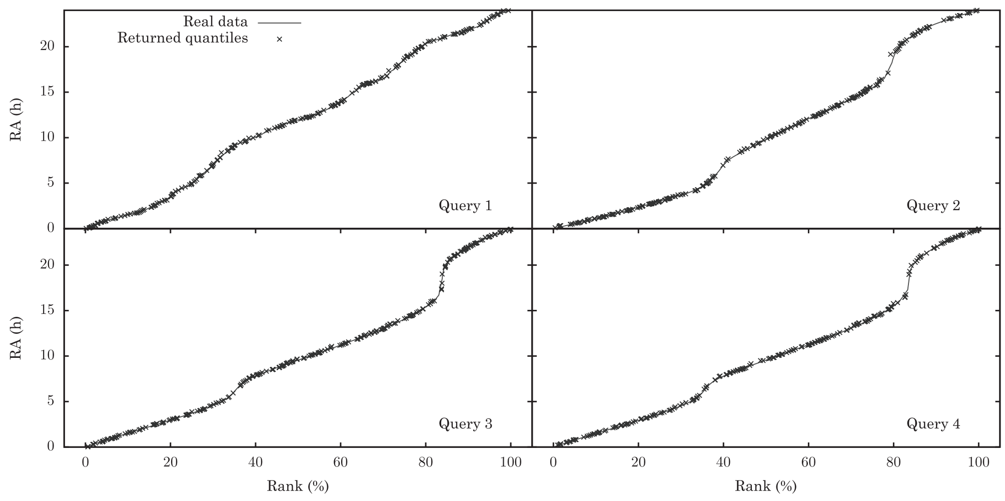

Fig. 11. Quantiles of RA.

MG algorithm [Misra and Gries 1982], which has been shown to be one of the best performers [Cormode and Hadjieleftheriou 2008]; for quantiles, we used the GK algorithm [Greenwald and Khanna 2001]; the AMS sketch and the Count-Min sketch are easily constructed in a streaming fashion. The error parameter ε used in these streaming algorithms is set as in Table I.

Random Sampling. A random sample can be considered as a general-purpose summary that can be used for any estimation problem. For heavy hitters, we simply find the heavy hitters in the sample (also using a streaming algorithm) and compute their frequencies, which are unbiased estimators for their true frequencies in the whole dataset. The same is done for quantiles. Note that, however, the F_2 (self-join size) of the sample is not an unbiased estimator of the F_2 in the whole population; a more careful formula needs to be used [McGregor et al. 2012; Bar-Yossef 2002].

For data stored in external memory, block-level sampling is always used to make full use of a block access [Chaudhuri et al. 1998, 2004], namely, when we sample a block, we put all elements in the block into the sample. To sample one block, one way is to start from the root of the B-tree and follow the pointers downward until we reach a leaf block. At each internal node, among all pointers within the query range, we randomly pick one to follow. In our experiments, we used a more efficient implementation, where all leaf blocks are stored consecutively on disk (which may not be true in reality due to fragmentation). This way, we just need to do two root-to-leaf traversals in the B-tree to identify the first and the last block of the query range. Then, for a desired sample size, we randomly decide the locations of the blocks to be sampled, and then directly seek these blocks in the location order. This not only saves multiple root-to-leaf traversals but also improves access locality when the sampled blocks are consecutively stored on disk.

As we read the sampled blocks, we feed the records into a streaming algorithm to construct the desired summary as in the standard B-tree method. However, since both the sampling and the streaming algorithm will contribute errors, the error parameter ε for the streaming algorithm used here is set to half of the values in Table I. So the sampling method uses more memory at query time than the standard B-tree and our index.

The last but most crucial issue with the sampling method is how much we should sample. This problem is easier if the sample is uniformly random but much more difficult when block-level sampling is used. A naive approach is to just randomly pick

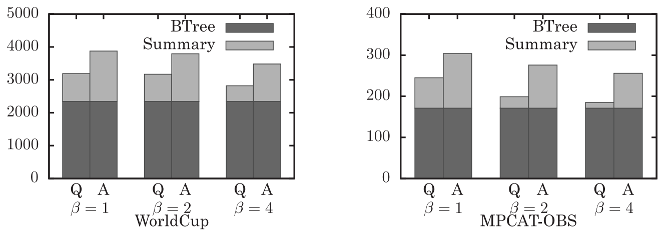

Fig. 12. Construction time (sec) for different summaries and different β's. Q and A stand for Quantile and AMS, respectively.

one record from every sampled block, which gives out a uniformly random sample, but this is obviously a big waste. In fact, the problem of determining the "quality" of all the records in a sampled block so as to decide the right sample size for block-level sampling is highly nontrivial and has received much attention in the literature. Prior work has looked at the problems of histogram construction and distinct count estimation [Chaudhuri et al. 2004, 1998], but not for the problems we are dealing with here (heavy hitters, quantiles, and F_2). To avoid using a suboptimal sample size determination algorithm invented by us, we used in our experiments an ideal calibration process that determines the right sample size for achieving a desired accuracy. More precisely, we first answer the query using our index, measure how accurate it is, and then gradually increase the sample size until the sampling method achieves (roughly) the same accuracy. Since the accuracy of the sample is measured by comparing with the full query results, this calibration process is idealistic and expensive, but it obviously does a better job than any realistic method that determines whether the sample is already enough by looking at only the sampled records, as done in Chaudhuri et al. [2004, 1998]. In reporting the query cost, we exclude this calibration process and only measure the actual sampling cost after the ideal sample size has been determined. The rationale is that this will expose the limit of the sampling method itself, assuming that its prerequisite, the sample size determination problem, can be solved perfectly at no cost.

It remains to describe how we measure the accuracy for various summaries in the calibration process. For F_2, accuracy is measured simply by the (relative) difference between the true F_2 and the estimated one. For heavy hitters, we find the true frequency (as a fraction of the total query result size) of every distinct element and compute the difference from its estimated frequency. Note that the estimated frequency could be 0, when this element is not included in the summary at all. Then, over all elements, we take the one with largest error. For quantiles, we extract the 1%-quantile, 2%-quantile, . . . , 99%-quantile from the returned summary and see how far their true ranks in the underlying dataset deviate from the required ones. From these 99 quantiles, we take the one with the maximum error (i.e., largest deviation).

5.5. Construction Time

Figure 12 shows the times for constructing the indexes. Recall that our indexes are built on top of a B-tree, and we recorded the time spent on building the B-tree and on building the summary blocks separately. From the figure, we see that the extra time spent on building the summary blocks is not much compared with building the B-tree itself, which is needed for the two baseline solutions as well. The value of β affects the construction time similarly as it affects the summary index size: the larger β is, the less summaries need to be built, and hence the faster the construction process becomes.

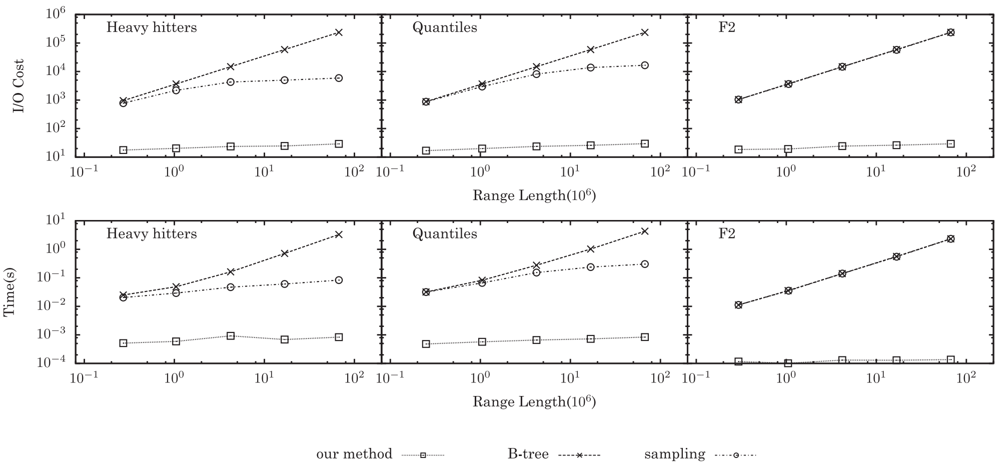

Fig. 13. Comparison of different methods in terms of query performance.

5.6. Query Performance

5.6.1. Comparison with Baseline Methods. We compare the query efficiency of our index with the baseline solutions for achieving the same accuracy. We used the MPCAT-OBS dataset for this set of experiments. The query ranges are on the time-stamp attribute of different lengths. For the summary attribute, we used the observatory attribute to compute the heavy hitters and F_2 (using the AMS sketch), and the RA attribute to compute the quantiles. For our method, we used the index with $\beta = 2$.

Figure 13 shows the performance of the three methods in terms of either I/O cost or wall-clock time, as the query range varies from containing 0.2 million records to 87 million, which is the full dataset. Each point in the figure is the average of 30 queries of the same length. We did not try ranges that are too small, since the primary application domain of summary queries is data analytics on the scale; for small ranges, full results can be retrieved and analyzed easily. We did not clear the cache (i.e., warm cache) between consecutive queries, so as to simulate a real environment. Note that this only affects the wall-clock time but not the number of I/Os performed by the algorithms.

Figure 13 left shows the results for heavy hitters. The standard B-tree obviously incurs a high I/O cost, which is linear in the query range. The sampling method also does poorly. For small ranges, it almost has to sample all the records in order to achieve the same accuracy as our method. For large ranges, the proportion of data that needs to be sampled reduces to about 1% of all the data in the range, which is still a fairly large sample size. This is mainly due to the high correlation among the records within a block, which is the case for most real datasets, so the sample obtained by block-level sampling is far from a truly uniform sample. Compared with the sampling method, our method is about 100 times more efficient, and the cost is quite flat across different query ranges. In terms of wall-clock time, the gap is slightly smaller, since the two baseline methods simply feed all data into a streaming algorithm (MG in this case), which is very simple and efficient, while we need to merge the retrieved summaries together.

Figure 13 middle shows the results for quantiles, which exhibits similar behavior for all three methods, except that the sampling method performs even worse than for heavy hitters. This is because the dataset is skewed, with a few heavy hitters of high frequency. As long as the sampling method gets their estimations right, the error will be low. On the other hand, for the quantile problem, we in some sense need a more

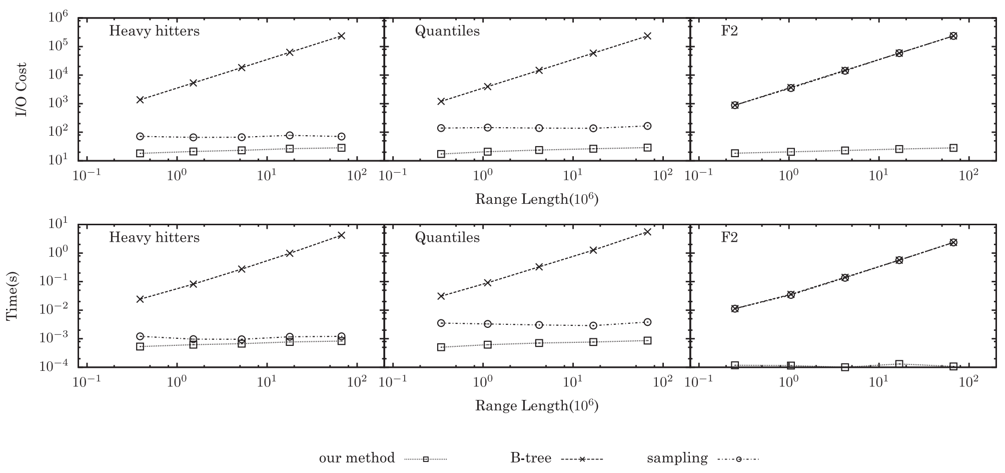

Fig. 14. Comparison on the randomly shuffled dataset.

uniformly good estimation for all quantiles, which is more difficult for the sampling method.

Figure 13 right shows the results for estimating the F_2 of the data in the query range. Here we see that the sampling method almost has to sample all the data in the range in order to give an accurate estimation. This agrees with the earlier result in the non-indexing setting that sampling is not an effective way for F_2 estimation [Alon et al. 2002].

In these experiments, the sampling method performs poorly due to the high correlation between the time stamp and the observatory attribute (for computing heavy hitters), and also between the time stamp and the RA attribute (for computing quantiles) in the MPCAT-OBS dataset. To explore the limit of the sampling method, we generated another dataset by changing the time-stamp attribute to random numbers. This effectively gives us a dataset with no correlation at all, so block-level sampling will actually give us a truly uniformly random sample. This dataset is thus the best-case input for the sampling method. Note that such a randomly shuffled dataset is not so meaningful in reality, since the data distribution is essentially the same for any query range. Anyway, the idea here is to see how sampling can do in its best case.

The results on the randomly shuffled data set are shown in Figure 14. We can indeed see a substantial improvement of the sampling method, especially in terms of I/O cost, which is essentially flat across all query ranges. This is consistent with the theoretical analysis that a (truly uniformly) random sample of size $O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$ suffices to estimate the heavy hitters and quantiles with ε error, and this is independent of the underlying data size. But still, even on this best-case data, the sampling method is outperformed by our method. Finally, sampling still cannot estimate the F_2 effectively on this randomly shuffled dataset.

5.6.2. Query Scalability. To investigate the query performance of our index to the scale, we conducted experiments on the WorldCup98 dataset, which consists of 1.6 billion records. We issued 50,000 random queries and measured the I/O cost and wall-clock time of answering these queries. For each query, the two endpoints of the query range were randomly chosen from the dataset. Here, instead of just showing the average, we plot the full results in the query length-cost space in Figure 15, in order to also see the variation in performance. We did not test the baseline methods since they are too slow on this large dataset.

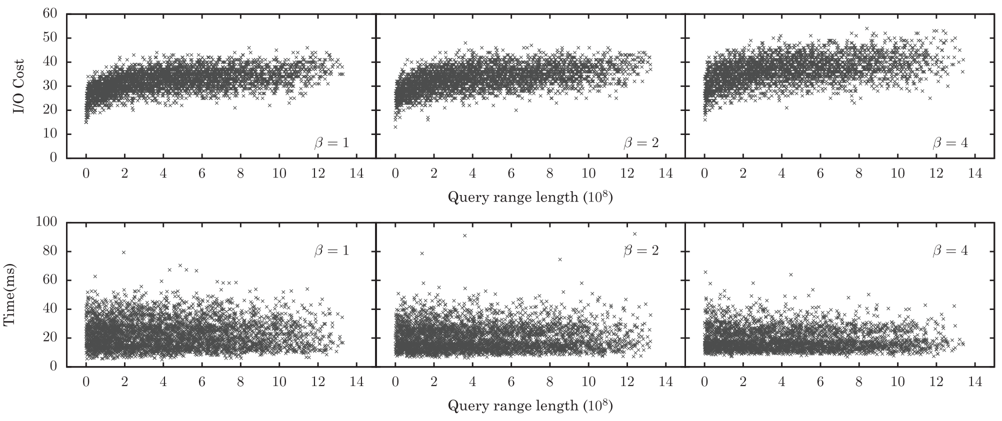

Fig. 15. Query scalability.

From the results, we see that the query cost of our index is not significantly affected by the query range. This agrees with the theoretical analysis that it depends only logarithmically on the query range. Meanwhile, for a fixed query length l, depending on the exact position of the query range boundaries, the query I/O cost may vary from $\log(N/s_\varepsilon) + \log(l/s_\varepsilon)$ to $2\log(N/s_\varepsilon)$. This is why we see a band of different costs for the same query length. As β increases, the query cost also increases, since with a larger β, we do not store summaries for shorter dyadic intervals. Thus, the query algorithm will have to retrieve the raw data in these intervals. The situation of query time is similar.

5.7. Update Performance

To see how well our update algorithms work, we carried out the following update operations. We used the WorldCup98 dataset for this set of experiments.

—*Sequential Insertion.* For a fixed random key, insert it 50,000 times to the index. Due to the default tie-breaking rule, this results in 50,000 records inserted sequentially.
—*Random Insertion.* Insert 5,000 random keys to the data structure.
—*Sequential Deletion.* Randomly choose 50,000 consecutive keys in the index, delete them one by one.
—*Random Deletion.* Delete 5,000 random keys from the index.
—*Mixed Updates.* Randomly choose 5,000 keys from the index. For each key, either insert another copy or delete it, each with probability 1/2.

For each of the five batches of update operations, we repeated it three times, starting from the same initial index, and report the average I/O cost and wall-clock time of all the operations in Table VIII. In the table, we have separated the cost of updating the B-tree and updating the attached summaries.

Recall that we use a weight-balanced B-tree, which adopts slightly different rules in splitting/merging the nodes than the standard B-tree. In the experiments, we see little effect of the new splitting/merging rules, since the cost is dominated by the root-to-leaf traversal. For random insertions and deletions, the I/O cost is exactly 5, which is equal to the height of the B-tree, meaning that no merging/split has ever happened (recall that we used a load factor of 70% when we built the B-tree initially). Sequential insertions/deletions are concentrated in one region of the B-tree, so merges/splits did happen, but the additional I/O cost is insignificant. In fact, sequential insertions/deletions incur less wall-clock time than random insertions/deletions, due to much better access locality.

Table VIII. Insertion and Deletion Performance on the Summaries and the B-Tree

		#Block accessed			Time (ms)		
		$\beta = 1$	$\beta = 2$	$\beta = 4$	$\beta = 1$	$\beta = 2$	$\beta = 4$
AMS	Seq Ins	27.00	25.27	23.08	2.61	2.72	2.13
	Rnd Ins	22.03	21.09	19.91	42.63	29.86	20.40
Quantile	Seq Ins	28.83	26.66	25.34	2.37	2.70	2.19
	Rnd Ins	22.97	22.04	20.82	38.50	38.01	19.51
B-Tree	Seq Ins	5.02			0.76		
	Rnd Ins	5.00			12.63		
AMS	Seq Del	19.79	19.33	18.84	1.61	1.55	1.47
	Rnd Del	22.04	21.07	19.91	43.52	30.06	20.00
Quantile	Seq Del	20.04	20.00	18.78	1.83	1.48	1.38
	Rnd Del	22.03	21.08	19.91	37.85	37.67	21.56
B-Tree	Seq Del	5.03			0.82		
	Rnd Del	5.00			12.49		
AMS	Mixed	22.02	21.08	19.90	42.78	30.24	19.52
Quantile	Mixed	22.51	21.56	20.39	38.48	35.39	21.24
B-Tree	Mixed	5.00			12.56		

The cost of updating the summaries is in general about four times the cost of updating the B-tree. This is because the summaries are organized in a binary tree, whose height is about four times that of the B-tree. The I/O cost for sequential insertions is slightly higher than that of random insertions, as splitting is more likely to happen in the former case. On the other hand, the I/O cost for sequential deletions is slightly lower than for random deletions. This is due to the fact, although the height of the B-tree remains constant, the height of the binary trees $\mathcal{T}_{\mathcal{B}}$ that organize the summaries on a root-to-leaf path decreases as more sequential deletions are performed, which means that less summaries need to be updated. In terms of wall-clock time, sequential insertions/deletions are much faster than random ones due to access locality. We do not observe any significant difference between the two summary types. Finally, as β increases, the update cost decreases, as we have fewer summaries to update.

From these results, we indeed see a higher update cost of our index compared with the baseline solutions, which only use a B-tree. However, considering the huge improvement over the standard B-tree and the (idealized) sampling method, our method is still clearly the better choice for answering summary queries, especially for large query ranges and when there are not too many updates to the dataset.

6. CONCLUSION

In this article, we presented both theoretical and practical indexing structures for supporting summary queries efficiently. We have demonstrated that summary queries contain much richer information about the query results than simple aggregates, and believe that they will become a useful tool for large-scale data analytics. There are many interesting directions to explore.

(1) One interesting theoretical question is if optimal indexing is also possible for F_2-based sketches like the AMS sketch. In fact, we can partition the data in terms of F_2 so that the F_2-based summaries are also exponentially decomposable (in terms of F_2), but we meet some technical difficulties, since the resulting tree $\mathcal{T}$ is not balanced.
(2) We have only considered the case where there is only one query attribute. In general, there could be more than one query attribute and the query range could be any spatial constraint. For example, one could ask the following queries.

(Q3) Return a summary on the salaries of all employees aged between 20 and 30 with ranks below VP.

(Q4) Return a summary on the household income distribution for the area within 50 miles from Washington, DC.

In the most general and challenging case, one could consider any `SELECT-FROM-WHERE` aggregate SQL query and replace the aggregate operator with a `summary` operator.

(3) Likewise, the summary could also involve more than one attribute. When the user is interested in the joint distribution of two or more attributes, or the spatial distribution of the query results, a multidimensional summary would be very useful. An example is as follows.

(Q5) What is the geographical distribution of households with annual income below $50,000?

Note how this query serves the complementing purpose of (Q4). To summarize multidimensional data, one could consider using multidimensional quantiles or histograms, as well as geometric summaries, such as ε-approximations and various clusterings.

REFERENCES

P. Afshani, G. S. Brodal, and N. Zeh. 2011. Ordered and unordered top-k range reporting in large data sets. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*.

P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. 2012. Mergeable summaries. In *Proceedings of the ACM Symposium on Principles of Database Systems*.

P. K. Agarwal and J. Erickson. 1999. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*. American Mathematical Society, 1–56.

N. Alon , P. B. Gibbons, Y. Matias , and M. Szegedy. 2002. Tracking join and self-join sizes in limited storage. *J. Comput. Syst. Sci.* 64, 3, 719–747.

N. Alon, Y. Matias, and M. Szegedy. 1999. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.* 58, 1, 137–147.

A. Arasu and G. Manku. 2004. Approximate counts and quantiles over sliding windows. In *Proceedings of the ACM Symposium on Principles of Database Systems*.

L. Arge and J. S. Vitter. 2003. Optimal external memory interval management. *SIAM J. Comput.* 32, 6, 1488–1508.

R. A. Baeza-Yates and G. H. Gonnet. 1991. *Handbook of Algorithms and Data Structures*. Addison-Wesley.

Z. Bar-Yossef. 2002. The complexity of massive data set computations. Ph.D. Dissertation, University of California at Berkeley.

K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. 2007. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. 2011. Towards optimal range medians. *Theor. Comput. Sci.* 412, 1, 2588–2601.

F. Buccafurri, G. Lax, D. Sacca, L. Pontieri, and D. Rosaci. 2008. Enhancing histograms by tree-like bucket indices. *VLDB J.* 17, 5, 1041–1061.

S. Chaudhuri, G. Das, and U. Srivastava. 2004. Effective use of block-level sampling in statistics estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

S. Chaudhuri, R. Motwani, and V. Narasayya. 1998. Random sampling for histogram construction: How much is enough? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

G. Cormode and M. Hadjieleftheriou. 2008. Finding frequent items in data streams. In *Proceedings of the International Conference on Very Large Data Bases*.

G. Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* 55, 1, 58–75.

M. Datar, A. Gionis, P. Indyk, and R. Motwani. 2002. Maintaining stream statistics over sliding windows. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*.

J. Gehrke, F. Korn, and D. Srivastava. 2001. On computing correlated aggregates over continual data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. 2002. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of the International Conference on Very Large Data Bases*.

J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining Knowl. Discov.* 1, 1, 29–53.

M. Greenwald and S. Khanna. 2001. Space-efficient online computation of quantile summaries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

J. Hellerstein, P. Haas, and H. Wang. 1997. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Z. Huang, L. Wang, K. Yi, and Y. Liu. 2011. Sampling based algorithms for quantile computation in sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. 2008. Scalable approximate query processing with the dbo engine. *ACM Trans. Datab. Syst.* 33, 4.

A. Jørgensen and K. Larsen. 2011. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*.

X. Lin , Y. Yuan, Q. Zhang, and Y. Zhang. 2007. Selecting stars: The k most representative skyline operator. In *Proceedings of the IEEE International Conference on Data Engineering*.

A. McGregor, A. Pavan, S. Tirthapura, and D. P. Woodruff. 2012. Space-efficient estimation of statistics over sub-sampled streams. In *Proceedings of the ACM Symposium on Principles of Database Systems*.

A. Metwally, D. Agrawal, and A. Abbadi. 2006. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Datab. Syst.* 31, 3, 1095–1133.

J. Misra and D. Gries. 1982. Finding repeated elements. *Sci. Comput. Program.* 2, 2, 143–152.

J. I. Munro and M. S. Paterson. 1980. Selection and sorting with limited storage. *Theor. Comput. Sci. 12,* 3, 315–323.

S. Muthukrishnan. 2005. *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers.

H. Samet. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. 2004. Medians and beyond: New aggregation techniques for sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04).* ACM.

Y. Tao, L. Ding, X. Lin, and J. Pei. 2009. Distance-based representative skyline. In *Proceedings of the IEEE International Conference on Data Engineering*.

Y. Tao, G. Kollios, J. Considine, F. Li, and Papadias. 2004. Spatio-temporal aggregation using sketches. In *Proceedings of the IEEE International Conference on Data Engineering*.

V. N. Vapnik and A. Y. Chervonenkis. 1971. On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.* 16, 2, 264–280.

J. S. Vitter. 2008. *Algorithms and Data Structures for External Memory*. Now Publishers.

Z. Wei and K. Yi. 2011. Beyond simple aggregates: Indexing for summary queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*.

Received July 2012; revised March, July 2013; accepted July 2013