Abstract—Collections of time-series data appear in a wide variety of contexts. To gain insight into the underlying phenomenon (that the data represents), one must analyze the time-series data. Analysis can quickly become challenging for very large data (~terabytes or more) sets, and it may be infeasible to scan the entire data-set on each query due to time limits or resource constraints. To avoid this problem, one might pre-compute partial results by scanning the data-set (usually as the data arrives). However, for complex queries, where the value of a new data record depends on all of the data previously seen, this might be infeasible because incorporating a large amount of historical data into a query requires a large amount of storage.

We present an approach to performing complex queries over very large data-sets in a manner that is (i) practical, meaning that a query does not require a scan of the entire data-set, and (ii) fixed-cost, meaning that the amount of storage required only depends on the time-range spanned by the entire data-set (and not the size of the data-set itself). We evaluate our approach with three different data-sets: (i) a 4-year commercial analytics data-set from a production content-delivery platform with over 15 million mobile users, (ii) an 18-year data-set from the Linux-kernel commit-history, and (iii) an 8-day data-set from Common Crawl HTTP logs. Our evaluation demonstrates the feasibility and practicality of our approach for a diverse set of complex queries on a diverse set of very large data-sets.

I. INTRODUCTION

Several organizations today are focused on collecting data about their systems and their operations. These organizations aim to understand and ultimately improve their systems through the analysis of this data. Such organizations range from retailers collecting analytics about shopper behavior in order to make pricing decisions [1], to geological monitoring systems for proactive earthquake warnings [2].

While the data sets are increasingly becoming diverse, ranging from scientific observations to social media streams, queries executed upon them are becoming increasingly more complex. Queries range from simple counts of matching records, more complex ones like distinct-value and frequency estimation. These query types are useful in the analysis of many data sets.

The difference between simple and complex queries is the amount of additional storage required to execute the query. The counting query only needs to maintain a single counter as it scans the data set, but the distinct-value query needs to maintain a set of each distinct value previously seen (allowing the uniqueness of any subsequent value can be determined). Put more concisely, simple queries have a fixed auxiliary storage cost, while the additional storage for complex queries may grow with the size of the data set.

Many large-scale data processing approaches today focus on obtaining exact query results using abundant processing power. The prime example of this approach is MapReduce [3]. By partitioning the data and the query across an entire cluster of computers, MapReduce can execute complex queries on data sets of arbitrary size. However, larger data sets require larger clusters to maintain the same level of performance. Studies [4] have shown that hundreds of MapReduce nodes might be required to process a multi-terabyte data set within seconds.

Other approaches include query preprocessing and stream summarization as described in section VI. Incremental materialized views can provide fast query results, but these approaches may require unbounded storage, and some do not permit arbitrary range queries (which are crucial in the analysis of time-series data). Sketch data structures provide succinct summaries of arbitrary-size data sets with bounded error, but don’t provide a way to take a closer look at arbitrary subsets of the data.

Contributions of this paper. We present an approach to performing complex queries over very large data-sets in a manner that is (i) practical, meaning that a query does not require a scan of the entire data-set, and (ii) fixed-cost, meaning that the amount of storage required only depends on the time-range spanned by the entire data-set (and not the size of the data-set itself). We evaluate our approach with three different data-sets: (i) a 4-year commercial analytics data-set from a production content-delivery platform with over 15 million mobile users, (ii) an 18-year data-set from the Linux-kernel commit-history, and (iii) an 8-day data-set from Common Crawl HTTP logs. Our evaluation demonstrates the feasibility and practicality of our approach for a diverse set of complex queries on a diverse set of very large data-sets.

Our unique insight into this problem is the synthesis of existing approaches for stream summarization, combined with a query preprocessing method that stores partial query results, such that they can be reused by multiple queries over arbitrary subsets of a single time-series data set. This combination allows our approach to execute arbitrary range queries in sub-linear time, while also maintaining fixed storage costs by trading off the exactness of results for bounded approximations.

The remaining sections of this paper are organized as follows. Section II presents the motivation for our work and describes some use cases for our approach. Section III clearly defines the scope of our work, including our requirements.
assumptions, goals and non-goals. Section IV describes our approach, including the data structure and query algorithm. Section V presents an evaluation of our approach, showing its fixed storage and execution costs for queries on a variety of data sets. Section VI discusses related work. Finally, we conclude in section VII.

II. ANECDOTAL MOTIVATION

Over the past five years, we have been fortunate to be able to work closely with a large-scale content-delivery platform for mobile applications [5], [6]. Since its initial launch in 2009, the platform has grown to support over 15 million users. Much of this work was motivated by the challenges faced in supporting this platform in its initial years as well as its evolution into the large-scale system that it is today.

One of the core components of this content-delivery platform is an analytics system. This system collects user-action reports from mobile applications, stores the data, and allows data analysts to execute various queries on the data. These user-action reports, sorted by their timestamps, form a time series.

The initial version of the analytics system was simple: A relational database stored each action in a single table, one row per action. Aggregate queries were expressed as SQL statements executed on this table, and the relational database handled the query execution. Later, the system was augmented with an in-stream aggregator that would update counters stored in the database. Each query matched a set of counters that were incremented as matching data arrived in the system, effectively pre-computing the query results. For example, each time that an "application open" event is reported, the aggregator would update the "application open" counter. The system could now simply read the latest value of this counter to answer queries about the number of "application open" events, rather than needing to scan through the data set. This approach reduced query execution time considerably – queries that once took days to execute could now be answered in a matter of seconds.

Figure 1 shows a high-level overview of the system architecture. Mobile applications send usage reports to multiple aggregators, which are servers that perform in-stream aggregation of the reports. Both the raw data and all of the aggregates are stored in a MySQL database. When needed for real-time or batch queries, the query service fetches this data from the database and presents it to data consumers.

A. Challenges

Unfortunately, extending the in-stream aggregation concept beyond simple counting is a challenging problem. When executing some queries, like distinct-value counting, determining value of a new piece of data requires information about all data previously seen. We call these complex queries.

One approach to applying in-stream aggregation to complex queries is to simply store all of the information previously seen, referencing it when new data is encountered. This is practical for small data sets, but as the size of the data set increases it becomes expensive to store and query all previous information. For example, a data set with 10 billion records may well have 1 billion distinct values for a particular field. Supposing that each value has a size of 10 bytes, after seeing all 10 billion records the system would be querying about 10 GB of data each time that a new value is seen.

The goal of this work is to address the problem of making it practical to execute complex queries over multi-terabyte data sets with fixed storage cost. Here, practical means that it does not require a linear scan of the data set to answer the query, and queries can be executed over the entire data set or any subset thereof. Fixed cost refers to the requirement that the additional storage used by the system does not vary based on the size of the data set alone; rather, it is proportional to the time spanned by the data set. The following section will define these goals in more detail.

III. PROBLEM STATEMENT

A. Requirements and Goals

We want to begin by formalizing the requirements that were mentioned in our motivation for this work. To meet our requirements, our approach must achieve the goals below.

1) **Data-Set Scalability.** There must not be a limit on the size of the data set upon which queries are executed. It may not be possible to predict the size of the data set before it is collected, and many data sets grow continuously over time. A size limit would make our approach impractical.

2) **Query Complexity.** A query is complex if the value of any new data to the result of the query is based on all of the data previously seen. Our approach must be able to execute complex queries.

3) **Storage Practicality.** Given a time series of arbitrary size that spans a fixed time range, the amount of storage used by our approach must remain fixed. This does not include the storage of the data set itself.

4) **Query Practicality.** Time and computational resources are finite. Given a time series of arbitrary size that spans a fixed time range, the execution time of the query must remain fixed.

B. Assumptions

We want to clearly define the assumptions we are making about the data sets under study and the queries made upon those data sets. The assumptions of our work are as follows.

1) **Time-series data.** The data set takes the form of a time series. It is composed of a sequence of records, with each record consisting of any number of key-value pairs. One of the values in each record must be a timestamp.
2) **Query execution by linear scan is not practical.** There is some resource limitation that makes it impractical to execute queries by scanning the entire data set in question. This may be due to a large data set size, limited computational resources, a time limit on query execution, or a combination of these factors.

3) **Uniform random-access latency.** We assume that the overall execution time of a query is proportional to the number of read operations executed by the query, and the latency of all read operations are equal.

4) **Query results can have bounded error.** It is acceptable for the results of queries to have some degree of error, as long as that error is bounded and the maximum error is known.

5) **Queries are known in advance.** We assume that there is a set common queries that are known prior to applying our approach. The time range over which the query is executed may vary, but other query parameters must be known in advance.

### C. Non-Goals

We also want to clearly define goals that are not in the scope of our work. While some of the below are desirable, we reserve them for future work.

1) **Maximum Query Latency.** We do not aim to achieve a query times below some absolute threshold.

2) **Maximum Update Latency.** We do not guarantee a maximum latency between the time that a new data point arrives and the time that it is included in subsequent query results.

3) **High Availability.** Our approach does not attempt to provide high availability at this time.

4) **Data-Loss Protection.** Our approach does not include mechanisms to prevent data loss due to system failure.

### D. Research Question

Now that we have clearly defined our requirements and goals, assumptions, and non-goals, here is the research question that we seek to address:

*Given a data set in the form of a time series of arbitrary size spanning a fixed time range, how can we execute complex queries over any time range covered by the data set within a fixed amount of time and using a fixed amount of auxiliary storage?*

### IV. Approach

This section provides a detailed description of our approach, beginning with a high-level overview and then a deeper description of the components of our approach.

#### A. Overview

Our approach defines both a data structure and a query algorithm that operates upon that data structure. Given a time-series data set, we first perform a scan of each data record. For each record, we update our data structure according to the values in that record. Later, when queries are made, our query algorithm reads certain values out of the data structure and combines them to produce the query result.

The scan of the data records is called the pre-processing phase. This can be done either as the data arrives, or at a later time by replaying the data set. The purpose of the pre-processing phase is to compute partial results that can later be used to answer queries without requiring a re-scan of the data set for each query. These partial results are stored in the data structure for later retrieval by the query algorithm. The format of the partial results is described later in this section.

Once the data set has been pre-processed, it is ready for query. Depending on the time range requested, the query algorithm retrieves some set of partial results from the data structure. By combining the partial results, the query algorithm produces the final answer to the query. The details of the query algorithm are described later in this section.

#### B. Data Structure

1) **Buckets:** Our data structure is composed of elements called buckets. Buckets are a represent a summary of all of the time-series records that occur over a particular time period. This time period is defined by a start timestamp and a duration. They are so named because aggregation can be thought of as putting individual items into a bucket and then looking at the bucket’s contents as a whole.

Despite the metaphor, the buckets in our system do not actually manage individual items. Instead, our buckets only contain aggregate information about a collection of items. When an item is placed into a bucket, the information is updated based on the values within that item.

The data within the bucket can take different forms depending on the type of query desired. For simple queries counting the number of matching items, an integer counter will suffice. For complex queries such as distinct-value and frequency-estimation, we use fixed-size sketches – HyperLogLog (HLL) and the Count-Min Sketch (CM), accordingly. The parameters of these sketches are chosen based on error tolerance and storage constraints. Generally, larger sketch sizes translate to smaller errors. These sketches also have the property that they can be merged in constant time, and the result of merging two instances does not change the error bounds. More precisely, given a set $S$ partitioned into two disjoint subsets $S_1$ and $S_2$.
\[ S_2, \text{sketch}(S_1) + \text{sketch}(S_2) = \text{sketch}(S), \] where \( \text{sketch}(s) \) denotes creating a sketch of all the items in \( s \), and \( + \) is the merge operation.

In addition to adding individual items, one entire bucket can be merged with another. To do this, the information in both buckets are combined depending on the type of data in the bucket. Integer counters are simply added, and sketches are combined using the appropriate merge algorithm (both HLL and CM define this operation). In any case, this can be done without any knowledge of the individual items originally added to the bucket. Since our data structures are simply a structured collection of buckets, our entire data structure can be merged with another instance, bucket-by-bucket. Merge operations are important in our system, as the merge operation forms the core of our query algorithm as discussed later.

In summary, buckets are small data structures containing a start timestamp, a duration, and some aggregate information about the items in the bucket. Buckets support an update operation, taking a new item and updating the aggregate information stored within. Buckets can also be merged, where the information stored in one bucket is combined with the information in a second bucket to produce a new bucket.

2) Levels: In our data structure, buckets are organized into sequences called levels, as shown in figure 2. All buckets on a level have the same duration. The first bucket on a level has a start time of zero, and subsequent buckets are described by \( s_{i+1} = s_i + d_i \), where \( d_i \) is the common duration of all buckets on the level. Note that there are no holes in this sequence and no buckets overlap, so each timestamped item falls in to exactly one bucket on each level.

Levels are implemented as a sparse mapping from start times to buckets using a binary search tree. When looking up appropriate bucket for an item, the timestamp of the item is rounded down to the nearest multiple of the bucket duration for the level. This time will match exactly one of the start times for a bucket on the level. The tree is then used to look up the bucket corresponding to the start time. If the mapping does not exist, a new empty bucket is automatically created and inserted into the tree, and the item is added to this new bucket.

3) Operations and Storage Costs: The entire data structure is composed of an sequence of \( L \) levels in increasing order of bucket duration. Each level has a unique bucket duration \( d = (d_0, d_1, d_2, \ldots, d_{L-1}) \), with the same arbitrary duration \( A > 0 \) for the first level \( (d_0) \) and each subsequent bucket duration being \( M > 1 \) times the previous one \( (d_{i+1} = Md_i) \). Using the branching factor \( M \), this sequence can be written as \( d = (A, AM, AM^2, \ldots, AM^{L-1}) \).

Adding a new item to the data structure simply requires updating the proper bucket on each level. For each of the \( L \) levels, the updater first finds the bucket corresponding to the time span that includes the timestamp of the item. These buckets are then updated with the item, using the update procedure described above.

We define four parameters of our data structure, \( A, M, L, \) and \( Q \). The base duration \( A \) is the duration of the smallest bucket in the data structure. The branching factor \( M \) is the ratio between the bucket durations of any level \( l > 1 \) and its previous level \( l-1 \) in the sequence. By definition, for every bucket on level \( l \), there are \( M \) buckets on level \( l-1 \). The parameter \( L \) defines the total number of levels in the data structure. All three of these parameters are closely related to the scale factor \( Q \), the total time range covered by the data structure, by the equation \( Q = AM^L \).

One may calculate storage costs for a particular time range, \( r \), based on the description of our parameters above. The total number of buckets on level 0 \( (b_0) \) is simply \( r/A \). For each subsequent level, \( b_{i+1} = b_i/M \). Thus, the total number of buckets we need to consider in this time range is given by the function

\[ B(r) = \sum_{i=0}^{L-1} \frac{r}{AM^i} = \frac{r}{A} \left( \frac{M^L - 1}{ML - M^{L-1}} \right) \]

Since \( Q \) is often very large or possibly even unknown when planning for storage costs, we can simplify this formula by letting \( Q \) (and therefore \( L \), related by the formula above) go to infinity:

\[ B_{\text{max}}(r) = \sum_{i=0}^{\infty} \frac{r}{AM^i} = \frac{r}{A} \left( \frac{M}{M-1} \right) \]

This gives us the maximum number of buckets possible for any value of \( L \) or \( Q \). When \( Q \gg A \), as would be the case for very large time series, \( B(r) \approx B_{\text{max}}(r) \) for all values of \( r \).

The complex relationships between these parameters dictate the various properties of our data structure. The base duration \( A \) primarily affects the maximum precision of the results returned by queries. Query start and end times must be rounded to a multiple of \( A \), and more precise information is lost. However, a larger \( A \) decreases the number of levels in the data structure, decreasing both query latency and storage costs. The branching factor \( M \) affects the number of levels \( L \) required to meet \( Q \). A larger \( M \) requires a smaller \( L \), and decreases storage costs at the expense of higher query latency. \( Q \) is the general scale factor, and increasing it requires either a larger \( M \) or \( L \), the choice being one between storage costs and query latency.

C. Query Algorithm

The goal of the query algorithm is to find the least number of buckets that cover a given time interval. In this section, we show that the query algorithm uses the data structure we just described to provide query results using asymptotically fewer read operations than a linear scan of the data set.

Input. In the context of an instance of our data structure, the input to the algorithm is a time range \( q \). This range is defined by a pair of start and end times, \( q_s \) and \( q_e \). These values are interpreted as inclusive and exclusive respectively, or in interval notation \( q = [q_s, q_e) \). The length of this range is defined as \( |q| = q_e - q_s \), and the algorithm ensures that \( |q| > 0 \) before proceeding.

Step 0. If \( |q| \leq A \), output the partial result contained in the first bucket on level 0 with start time less than or equal to \( q_s \). To find the bucket, the algorithm performs the bucket lookup as described above.

Step 1. Find the longest-duration bucket(s) in the data structure entirely within \( q \). These will be a (consecutive) sequence of buckets on some level \( l \geq 0 \). We know this will be at least 2 buckets on level 0, or at least 1 bucket on some higher level.

Step 2. Merge all of the buckets found in step 1 into a single bucket. The time range covered by this bucket is at most equal
to the time range of the entire query, since the time range covered by the buckets is entirely within the interval defined by $q$.

**Step 3.** Let $a$ be the start time of the first bucket in the sequence found in step 1, and $b$ be the start time of the next consecutive bucket after the sequence found in step 1 (on level $l$). Recursively find the partial results for $q_{left} = |qa, a)$ and $q_{right} = [b, qa)$, starting from step 0.

**Output.** Merge the bucket from step 2 with the left and right buckets found in step 3. The partial result within this merged bucket is the complete result for $q$.

We now analyze the run-time cost of our algorithm. First, we need to recall some of the parameters defined in the above description of the data structure. These include $d$, the monotonically increasing sequence of level bucket durations, $A = d_0$, and $M$, the ratio between successive elements in $d$.

We first want to establish bounds on the first level examined by the first iteration of the algorithm. Let $i$ be the largest positive integer such that $d_i \leq |q| < d_{i+1}$. The buckets found in step 1 of the algorithm are on level $i$ or lower, since all buckets on level $i + 1$ or higher exceed $|q|$ in duration. Since the sequence $d$ is monotonically increasing (sorted), we can find the proper level in time $c_1 = O(|\log L|)$ with binary search. Furthermore, from the bounds on $|q|$ it follows that $i \leq \log_M A^{-1} |q| = O(\log_M |q|)$.

At most $c_2 = 2M - 2$ bucket lookups are required for the first iteration of the algorithm. In the simplest case, a bucket on level $i$ exists such that the time range of the bucket is entirely within $q$. Here, the algorithm performs at most $M - 1$ bucket lookups, since querying $M$ more buckets would violate the constraint $|q| < d_{i+1}$.

If no such bucket exists on level $i$, then $q$ is contained within the combined time range of a sequence of buckets on level $i$ with length exactly two. It cannot be more than two, since the buckets not on the ends of the sequence would have time ranges entirely within $q$, contradicting the assertion that such a bucket does not exist on level $i$. It cannot be less than two, since this would violate the constraint $d_i \leq |q|$.

There are at most $M - 1$ buckets on level $i - 1$ within the time range of the left bucket in the sequence, and there are at most $M - 1$ buckets within the time range of the right bucket in the sequence. This is because a sequence of $M$ or more buckets aligned to the start of the left or right bucket would be covered by a bucket on level $i$, contradicting the assertion that a bucket covered entirely by $q$ does not exist on level $i$.

Thus, at most $2(M - 1) = 2M - 2$ buckets are queried in this case. Overall, at most $2M - 2$ buckets are queried in the first iteration.

At most $c_3 = M - 1$ bucket queries are generated for each of the (at most) $i$ lower levels. If algorithm is considering a level $l$, step-3’s sub-queries are always left- or right-aligned with buckets on level $l$. This means that the bucket sequences considered during the sub-query will also be so aligned. Therefore, a sub-query will never look up more than $M - 1$ buckets on the next lower level, since $M$ or more buckets would have been covered already by querying a single bucket on the higher level. Thus, at most $2(M - 1) = 2M - 2$ buckets are queried in every additional iteration on lower levels.

Putting these results together, we can form a cost estimate for the algorithm in terms of the maximum number of bucket read operations. This is simply the sum of the bucket cost for the first iteration plus the bucket costs the subsequent iteration, or $c_2 + c_3i$. By substituting the above results, this quantity is less than or equal to $2M - 2 + (2M - 2) \log_M (A^{-1} |q|)$ and so the number of read operations is in $O(\log_M |q|)$.

**V. Evaluation**

This section details the evaluation of our approach, based on both a real-world data-aggregation system as well as results obtained from the analysis of publicly-available data sets. The goal of our evaluation is to show that our approach meets the four requirements and goals set forth in section III. The first two, data-set scalability and query complexity, will be shown through the breadth of our evaluation across different data set sizes and query types. The second two, storage practicality and query practicality, are explicitly shown by two different sets of experiments.

**A. Data Sets**

We studied four data sets that vary both in the time range spanned and records per unit time (density). The four data sets are as follows:

- **Linux kernel commit history (KERNEL).** This is the data set containing the Git commit history of the Linux kernel from 1991 to 2010. This time series contains...
<table>
<thead>
<tr>
<th>Data Set</th>
<th>Time Span (days)</th>
<th>Density (B/day)</th>
<th>DV-NAIVE</th>
<th>DV-HLL</th>
<th>FREQ-CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAWL</td>
<td>8</td>
<td>3,249,392,998,000</td>
<td>did not finish</td>
<td>19,473,436</td>
<td>31,611,136</td>
</tr>
<tr>
<td>YINZCAM-48</td>
<td>24</td>
<td>113,223,308</td>
<td>5,252,941,201</td>
<td>192,661,462</td>
<td>502,574,848</td>
</tr>
<tr>
<td>YINZCAM-205</td>
<td>33</td>
<td>296,583,478</td>
<td>11,447,493,044</td>
<td>200,459,669</td>
<td>1,492,130,048</td>
</tr>
<tr>
<td>YINZCAM (full)</td>
<td>1,170</td>
<td>267,040,051</td>
<td>did not finish</td>
<td>8,843,105,274</td>
<td>53,050,190,848</td>
</tr>
<tr>
<td>KERNEL</td>
<td>6,707</td>
<td>2,170</td>
<td>43,808,538</td>
<td>324,466,488</td>
<td>3,636,521,442</td>
</tr>
</tbody>
</table>

Fig. 4. A table showing the relationship between our data sets and the storage required to aggregate them. The data sets are ordered the duration of their time span.

- **YinzCam mobile app usage data** (YINZCAM). This is a time series of mobile-application user-action records collected by a commercial content-delivery platform. Each record contains information about a user action along with metadata about the runtime environment. This time series contains 312,436,859,499 bytes (291 GB) of data in 6,063,000,000 records spanning 1,170 days. We also identify two subsets of this data set: YINZCAM-42, containing 2,717,359,397 bytes (2.5 GB) of data in 48,111,712 records spanning 24 days, and YINZCAM-205, containing 9,787,254,765 bytes (9.1 GB) of data in 205,185,198 records spanning 33 days.

- **Common Crawl logs** (CRAWL). This data set contains the timestamped web pages downloaded and archived by Common Crawl in 2013. Each record contains a HTTP request and response pair, including headers and crawler metadata. This time series contains 24,790,901 MB (24 TB) of data in 1,796,256,584 records spanning 24 days, and 948,130,048 bytes (9.1 GB) of data in 2,717,359,397 bytes (2.5 GB) of data in 48,111,712 records spanning 24 days, and 9,787,254,765 bytes (9.1 GB) of data in 205,185,198 records spanning 33 days.

B. Query Types

For our tests, we focused on two common complex aggregate queries on time series data: distinct-value and frequency-estimation. For distinct-value, we implemented both the naïve approach of maintaining the entire distinct-value set within each bucket, as well as maintaining a single HyperLogLog counter in each bucket. For frequency-estimation, we used the Count-Min sketch. These queries are described in more detail below.

- **Distinct-value with naïve approach (DV-NAIVE).** For this data structure, in each bucket we maintained the complete set of distinct values seen during that time period. To serialize these sets, we used the native collection serializer bundled with the Kryo serialization library for Java (version 3.0.0).

- **Distinct-value with HyperLogLog (DV-HLL).** This data structure also maintains a set of distinct values in each bucket, but does so using a HyperLogLog implementation. Our implementation is based on open-source code found in the AggregateKnowledge Github repository. We used version 1.6.0 of the net.agkn.hll Maven artifact. The HLL in each bucket used 2,147 5-bit registers.

- **Frequency-estimation with Count-Min (FREQ-CM).** This data structure can answer frequency queries for each of the distinct items seen in the data set. Our implementation is based on open-source code for the Count-Min sketch found in the Clearspring analytics Github repository. We used version 2.7.0 of the com.clearspring.analytics Maven artifact. The CM sketch in each bucket used a depth of 8, a width of 128, and a seed of 0.

1) Hadoop Cluster Testbed: For analyzing both storage and query costs, we used a Hadoop cluster running on Amazon EC2 using the Elastic MapReduce (EMR) cluster-management tool. We chose Hadoop primarily for the ease with which we could plug in various data inputs, including both the commercial content delivery platform’s proprietary format as well as various public formats. Hadoop also has the advantage of built-in counters for a variety of useful statistics.

Our cluster consisted of a group of standard EC2 instances. For the cluster’s master node (running the Hadoop management services) we used a m3.2xlarge instance type. The remaining worker nodes used the c3.2xlarge instance type. All cluster nodes ran Hadoop 2.4.0 on an Amazon Linux machine image with kernel 3.14.20-20.44.amzn1.x86_64.

We used Amazon S3 as storage for our test data. The version of Hadoop available through Amazon EMR has been extended to support reading data directly from S3, without requiring a pre-load onto HDFS. HDFS is still used to store intermediate results as needed, but the input and final output data is stored on S3. When we measure data sizes, we are measuring the data transferred in and out of S3 during the lifetime of the Hadoop task.

2) Live-System Testbed: To demonstrate our approach on a real-world system working with live data, we also analyzed the query cost of our approach on a production system used by a commercial content-delivery platform for mobile applications. The function of the system is to collect, analyze, and report on user activity across 90+ mobile applications. This system has been in production since September 2011 and has collected over 6.5 billion user-action records totaling nearly 3 TB of data (raw + aggregates).

C. Storage Practicality

In this subsection, we show that our approach meets our storage practicality requirement (A.3). As defined in section III this means that the auxiliary storage used by our approach scales as a function of the time range spanned by the data, not the size of the data set itself. To demonstrate this, we first constructed a instance of our data structure for each combination of data set and query type. We then measured and recorded the size of the resulting data structures. Details of our implementation and a discussion of the results are below.
We implemented the algorithm for building our data structure in Java as a Hadoop task. To fit within the MapReduce framework, we split our implementation into mapper and reducer tasks as follows. The mapper task consumes records from the input set and extracts from them the data of interest to the query. This information is then inserted into a set of buckets, one for each level of the data structure. These single-item buckets are then output to the reducer, keyed by a combination of the level and bucket timestamp. The reducer then produces multi-item buckets by combining multiple single-item (and multi-item) buckets, grouped by level and timestamp. The data structure is complete when no two output buckets share the same level and timestamp key.

The data structures we built were parametrized as follows. Our smallest bucket duration was 15 minutes. We used 6 levels total, with a branching factor of 5 between levels. As a result, the largest bucket duration was approximately 32 days, or just over 1 month.

Our output format was a set of flat plain-text files with one bucket recorded per line. The buckets were written by first writing a single digit representing the level (0-5), followed by a pipe character, followed by the Unix timestamp of the bucket in milliseconds, followed by a space. The remainder of the bucket record was a hexadecimal dump of the bucket contents, two hex digits per byte. An example of an output record would be 0|1299928500000 0c 03 01 31.

Once the MapReduce job completed, we read the "Bytes Written" output counter from Hadoop to measure the size of the resulting data structure.

Both implementationsallback our approach meets our query practicality requirement (A.4). The experiments in this section use both a real-world implementation of our approach that is used to provide mobile-application usage-analytics for a commercial content delivery platform, as well as an implementation based on Hadoop. We show that the execution of queries in both implementations matches the bucket-read cost shown by our algorithm analysis. We also show actual query run times of queries of varying lengths on live production data.

1) Implementation: We implemented the algorithm for building our data structure in Java as a Hadoop task. To fit within the MapReduce framework, we split our implementation into mapper and reducer tasks as follows. The mapper task consumes records from the input set and extracts from them the data of interest to the query. This information is then inserted into a set of buckets, one for each level of the data structure. These single-item buckets are then output to the reducer, keyed by a combination of the level and bucket timestamp. The reducer then produces multi-item buckets by combining multiple single-item (and multi-item) buckets, grouped by level and timestamp. The data structure is complete when no two output buckets share the same level and timestamp key.

The data structures we built were parametrized as follows. Our smallest bucket duration was 15 minutes. We used 6 levels total, with a branching factor of 5 between levels. As a result, the largest bucket duration was approximately 32 days, or just over 1 month.

Our output format was a set of flat plain-text files with one bucket recorded per line. The buckets were written by first writing a single digit representing the level (0-5), followed by a pipe character, followed by the Unix timestamp of the bucket in milliseconds, followed by a space. The remainder of the bucket record was a hexadecimal dump of the bucket contents, two hex digits per byte. An example of an output record would be 0|1299928500000 0c 03 01 31.

Once the MapReduce job completed, we read the "Bytes Written" output counter from Hadoop to measure the size of the resulting data structure.

2) Results and Discussion: The results across all combinations of data set and query type are shown in figure 4. The storage for the DV-NAIVE query scales roughly in proportion with the density of the data set, and we were unable to complete the data structures for the larger data sets due to time and storage constraints. However, DV-HLL and FREQ-CM both scale in proportion to the time span. Most notably, the space required to query the multi-terabyte CRAWL data set is only a few megarays due to its short time span.

The only data set seemingly contrary to our desired trend is KERNEL, which showed a smaller overall data structure size for DV-NAIVE. Also, despite having a length 6 times that of the previous data set, DV-HLL and FREQ-CM are both much smaller in size. The explanation for this is that, while the data set spans a very large time range, the data inside is sparse. As an optimization, our data structure does not store buckets that have no contents - they are simply not recorded on disk. Also, when buckets only contain a few items (as in this case), the DV-NAIVE bucket is often smaller than the DV-HLL bucket due to the data-dependent size of DV-NAIVE versus the fixed size of DV-HLL.

To better show the relationship between data set time span and storage costs, as well as the lack of relationship between data set size and storage costs, we analyzed these relationships across 34 disjoint subsets of our YYNQZCAM data set. These results are shown in figures 5 and 6. The plots show a linear relationship between data set time span and storage cost, and no linear relationship between data set size and storage cost.

D. Query Practicality

In this subsection, we show that our approach meets our query practicality requirement (A.4). The experiments in this section use both a real-world implementation of our approach that is used to provide mobile-application usage-analytics for a commercial content delivery platform, as well as an implementation based on Hadoop. We show that the execution of queries in both implementations matches the bucket-read cost shown by our algorithm analysis. We also show actual query run times of queries of varying lengths on live production data.

1) Implementation: We analyzed the query execution costs for our test data sets using our Hadoop 2.4.0 testbed running on Amazon EMR. To analyze the query cost in terms of reads from the data structure, we first computed level-timestamp keys of all buckets that contain relevant partial query results (using the query algorithm presented earlier). Taking the flat-file version of our data structure as input, our Hadoop mapper searched through the entire data structure for buckets matching these keys, and outputted the count and total size of the matching buckets. Our reducer combined these counts into single values, resulting in the total number of matched buckets for the query, along with the total size of the matched buckets.

For each data set, we ran a queries with lengths of 1 hour, 1 day (24 hours), and 24 days (576 hours). For each data set, the start and end time of each query was adjusted to make the
query fit within the range of the data set. We then measured the
number of bytes read from the data structure for each query
using the total size of the matched buckets output from the
MapReduce job described above.

In addition to the test data sets, we added instrumentation
to the query engine used by the production version of our
system that is currently aggregating app-usage data for a com-
mercial content-delivery platform. Similar to our MapReduce
job above, the instrumentation allows us to measure for each
query: the number of buckets matched, the number of bytes
read, and the total query execution time.

We executed a set of queries for the total count of mo-
BILE app page views with exponentially-increasing lengths.
We recorded the number of bytes read and the total query
execution time for each query. Because this is a live system,
our query execution times can vary due to a number of factors,
so we repeated each of our queries 10 times and report the
average, min, and max execution time for each.

2) Results and Discussion: Figure 7 shows the results of
our experiments on our test data sets. Both clearly show a
logarithmic relationship between query length and bytes read.
Again, due to the sparseness of the KERNEL data set and our
data structure’s optimization to not store zero-item buckets, the
amount of data read for this data structure is significantly less
than the others.

Figures 8 and 9 show the results of our tests on a live
production system. Figure 8 shows how the number of bytes
read from the data structure varies with the size of our
query. This graph shows that there is a strong logarithmic
relationship between the query length and the amount of data
read, consistent with the results on our test data set.

Figure 9 shows the distribution of execution times for the
same set of queries. Ideally, these times should be proportional
to the amount of data read and also show a strong logarithmic
relationship. This is true for lengths up to 256 hours (about
11 days), but the relationship begins to decay past this point.
This is because our current bucket-storage system has not
been optimized for random access over large time periods.
In the future, we hope to better optimize our implementation
for random access, and also improve overall query time by
improving read parallelism.

VI. RELATED WORK

Our work builds on previous research in the areas of large-
scale data processing, query preprocessing, stream summariza-
tion, and stream approximation. In this section, we summarize
the existing work in these domains and discuss how it relates
to our approach.

Large-Scale Data Processing. Some data-processing sys-
tems take a distributed-systems approach to handling very
large data sets. The intuition for these approaches is that
multiple computers working cooperatively can analyze large
data sets better, or more efficiently, than a single computer.
Popular distributed-parallel-processing approaches include dis-
tributed databases, such as HP Vertica\(^1\) and, more recently,
RethinkDB\(^2\), as well as MapReduce [3].

Distributed-parallel-processing approaches excel at coordi-
nating many computers working in parallel to answer queries
over data sets. However, for very large data sets, it may be
cost-prohibitive to build a large enough cluster to quickly
execute queries across the data set. Studies [4] have shown
that hundreds of nodes might be required to process a multi-
terabyte data set within seconds.

---

1\(\text{http://www.vertica.com/}\)
2\(\text{http://www.rethinkdb.com/}\)
Furthermore, if a query is repeated, perhaps only shifting the subset of data under consideration, distributed-parallel-processing systems must regenerate the partial results from the underlying data for each query. Instead of discarding the partial results from previous queries, these results could be stored and reused on subsequent queries. Our approach extends this idea by generating and storing these partial results on demand as the data enters the system.

In addition to query processing, there has been much research into efficient techniques for storing large-scale time-series data. Shafer et. al. [7] describe a data-storage approach that is customized for time series data, improving upon the storage requirements of traditional databases by organizing the data such that it can be more efficiently compressed. Przyymus et. al. [8] show that the compression of time-series data can be made both fast and efficient using GPUs. In contrast to these approaches, our work focuses on the storage of partial query results rather than the time-series data itself. However, by applying these techniques, we may be able to improve the constant-factor storage efficiency of our approach. This is a topic that we intend to explore in future work.

In general, our approach is not a replacement for traditional large-scale data processing; rather, it is an enhancement to existing approaches. For example, our system can (and does) operate alongside MapReduce clusters to provide fast answers to queries where partial results have been precomputed. Indeed, our approach can even be implemented with a MapReduce cluster, as described in section V.

**Query Preprocessing.** Query preprocessing is a fundamentally different approach to large-scale data processing. Instead of speeding up query-time through parallelism, query preprocessing relies on the computation of partial results as the data arrives, and then reaps performance benefits by combining the partial results to produce the final query-result.

Recent work by Fan et. al. [9], [10] formally defines the concept of query practicality through their definition of II-tractability. A query Q is II-tractable if there exists a polynomial-time algorithm to transform a data set D into D' such that Q(D') can be computed in polylog-time (O((log n)^k)) for some k. Our approach is a practical method of making II-tractable many common queries on very large time-series data-sets.

Much of the applied work in this area focuses on the incremental maintenance of materialized views (IMV), D7Toaster [11] and CReaM [12] each provide a framework for building query processors that update materialized views in real-time. However, these approaches do not bound storage or query-execution costs. On the other hand, PIQL [13] aims to bound storage cost, but explicitly ignores holistic aggregates (complex queries) as well as range queries. Our approach bounds the storage cost for holistic aggregates through its use of sketch data-structures, and supports range queries by pre-computing the results needed for queries over arbitrary time-ranges.

**Stream Summarization.** Stream summarization refers to methods that produce aggregate statistics (called a stream summary) about a data stream. Stream summaries are useful to concisely express some property of a potentially large data-stream. Comparing stream summaries is often far easier than comparing the streams themselves. Some examples of stream summaries include counting matching records and counting distinct values.

The one of the most challenging problems faced when summarizing a large data-stream is minimizing storage costs. Simple stream-summaries like match-counting don’t pose a problem (2^{64} matches can be stored in a 64-bit counter), while complex summaries like distinct-value counts may. Much of the literature in stream-summarization has been focused on storing complex summaries of large data-sets.

Using distinct-value counting as an example, we’ve shown that the naïve, but exact, method for counting the number of distinct values can have an enormous storage cost as the number of values grows large, because the size of the summary grows in proportion to the number of distinct values seen. However, the HyperLogLog [14] data-structure shows that it is possible to trade off accuracy in order to obtain fixed-size distinct-value summaries. Similarly, the Count-Min sketch [15] can be used to obtain fixed-size frequency-estimation summaries. These data structures and others that provide compact data-stream summaries form a group called sketches.

Sketches have been used to improve the performance of computational tasks across many fields of study. They feature prominently in the analysis of telecommunications data [16], [17] and distributed databases [18]. Here, the error introduced by the sketch is either more desirable than time/storage costs needed for an exact result, or the erroneous result is later verified for accuracy at greater cost if needed. For example, in telecommunications, the fast detection of an event (such as network intrusion) is more desirable than an accurate analysis of the data flowing in the network.

Our approach provides a general framework for applying stream summaries to answer queries over large-scale time-series data. The existing stream-summarization work strengthens our approach by providing variety to the types of queries that can be executed. While we focused primarily on the distinct-value and frequency-estimation summaries in this paper, we can employ any fixed-size summary without altering the query-cost and storage-cost results.

**Stream Approximation.** Many large-scale time-series analysis approaches store approximations of the time series in order to reduce storage requirements. Some of these approaches may offer benefits beyond our approach for certain types of time series. Amnesic approximation [19] focuses on applications where the accuracy of older data is much less important than the accuracy of very recent data. By allowing more error in older data, the cost of storing the entire data set is reduced. Unstructured time-series data, such as audio or video, may benefit from approximation using transforms. One approach of this type is SWAT [20], which builds a tree of wavelet transforms over subsequences of the data set. These transforms generally lend themselves to higher compression ratios than the time-series data itself, reducing storage costs.

While our approach approximates query results, these approaches approximate the time series itself. They have the advantage that the approximate time series data can then be applied to any query, not just queries known in advance. However, when querying a time-series approximation, one must show that the query result approximates the actual result that would be obtained by querying the original time series.
Our approach avoids this problem altogether by operating on the original time series data.

VII. SUMMARY AND CONCLUSION

In summary, we have demonstrated an approach to time-series data analysis that can execute complex queries on multi-terabyte data sets with practical storage costs and query execution time. We take an approach similar to database indexing, where a data structure of partial query results is constructed and stored as data enters the system. This data structure is later used to answer queries by reading and combining a subset of these fixed results, based on the time range of the query. Furthermore, we have shown both qualitatively and empirically that our approach meets our storage and query practicality requirements. In summary, both the storage cost and the query cost grow as a function of the time range spanned by the data set, and not the size of the data set itself. Furthermore, queries can be completed in time that is logarithmic in the time span of the query.

In conclusion, we believe that our approach is an excellent complement to existing large-scale data analysis approaches. While other systems exist that can answer complex queries, our approach can do so on very large data sets using limited storage and computational resources. To achieve this, our approach requires queries to be known in advance, and it introduces a bounded amount of error into query results. If storage and computational resources are not a constraint, other approaches such as MapReduce may be a better fit, as they can give exact answers to nearly any query given enough time and storage. However, if storage or compute time is limited, our approach can give approximate results for the same queries while staying within these resource constraints.

REFERENCES


