Interval Indexing and Querying on Key-Value Cloud Stores

George Sfakianakis, Ioannis Patlakas, Nikos Ntarmos, and Peter Triantafillou
Computer Engineering & Informatics Dept., University of Patras, 26500 Rio, Greece
{sfakianaki,patlakas,ntarmos,peter}@ceid.upatras.gr

***Abstract*— Cloud key-value stores are becoming increasingly more important. Challenging applications, requiring efficient and scalable access to massive data, arise every day. We focus on supporting interval queries (which are prevalent in several data intensive applications, such as temporal querying for temporal analytics), an efficient solution for which is lacking. We contribute a compound interval index structure, comprised of two tiers: (i) the MRSegmentTree (MRST), a key-value representation of the Segment Tree, and (ii) the Endpoints Index (EPI), a column family index that stores information for interval endpoints. In addition to the above, our contributions include: (i) algorithms for efficiently constructing and populating our indices using MapReduce jobs, (ii) techniques for efficient and scalable index maintenance, and (iii) algorithms for processing interval queries. We have implemented all algorithms using HBase and Hadoop, and conducted a detailed performance evaluation. We quantify the costs associated with the construction of the indices, and evaluate our query processing algorithms using queries on real data sets. We compare the performance of our approach to two alternatives: the native support for interval queries provided in HBase, and the execution of such queries using the Hive query execution tool. Our results show a significant speedup, far outperforming the state of the art.**

I. Introduction

The cloud is becoming increasingly more important for data management applications, as it can seamlessly handle huge amounts of data. The elastic and vast processing/storage capacity of clouds has facilitated novel applications requiring efficient and scalable access to massive data. In this picture, cloud key-value stores hold a central position, with relevant offerings including (but not limited to) Google's Bigtable[1] (and its open-source counterpart, Apache HBase[2]), Yahoo! PNUTS[3], Apache (ex-Facebook) Cassandra[4], Amazon Dynamo[5], etc. These systems are in the spotlight, powering applications dealing from data mining issues to social networking, graph algorithms, spatial data processing, image and video processing, bioinformatics, and so on. Along the same lines, MapReduce[6] and the Apache Hadoop[7] project – its open-source implementation – have emerged as the de-facto standard programming paradigm for crunching through massive data, although complex query processing is still better done in a "centralized", coordinator-based manner[8].

We focus on supporting interval queries, an efficient solution for which is lacking in the field of cloud key-value stores. We first establish the difference between range and interval queries through an example, which shall also serve as our champion application: analytics and time-traveling queries over web archives. In this scenario, web crawlers dump new versions

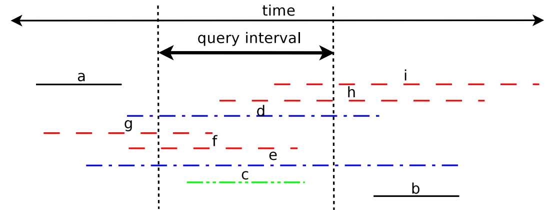


Fig. 1. (a), (b): intervals outside the query interval; (c): interval contained in the query interval; (d), (e): intervals covering the query interval; (c)-(i): intervals intersecting the query interval.

of web pages into a distributed key-value store, keyed by the URL of the page and timestamped by the time of the crawl. Thus, each web page has several "incarnations" at different time points, with any two subsequent such points defining the *interval* during which each version is "alive". These intervals usually are implicit, in the sense that crawlers only record the time of the crawl – that is, the beginning of an interval – but do not update previous crawls to reflect the end time of their validity, mainly for performance reasons. Now consider a query of the form "find all versions of all web pages that existed during a given time interval" (see Fig. 1). A range query on the page timestamps for the query interval, would return only pages that changed or were created during that time interval (i.e., pages (c), (h), (i)); if the crawlers further updated the timestamps of earlier crawls, an appropriate range query might be able to also locate pages (f) and (g); however, both range queries would fail to return pages that were created or modified before the beginning and/or after the end of the query interval (such as (e) and (d)). Such queries prove to be both intriguing and impractical with current state-of-the-art approaches, as we shall see shortly; in addition to their inability to process such queries, standard cloudstore query primitives on the item timestamps are also very inefficient, as the cloudstore engine would have to scan through all crawl key-value pairs across all cloudstore nodes, in order to locate the relevant ones.

Interval queries are prevalent in several data intensive applications, with temporal querying being the prototypical example, and thus arise in a slew of situations, from online social networking services (e.g., find events and notifications that happened within a time interval, find all users that were online during a time interval, etc.), to air control systems (e.g., given the time of entry and exit of flights to/from our

controlled airspace, find all vessels we were responsible for in a given time interval), and so on. Of course, interval queries are not limited to the time dimension; examples can be easily found in spatial and geographical databases (e.g., what streets exist within a user-supplied contour?), scientific applications (e.g., analysis of particle trajectories and collisions), etc.

In all above examples, the "primary keys" for the objects in the database would not be the intervals themselves but some other piece of information (i.e., URLs for crawled web pages, event IDs for notifications, user IDs for online users, flight numbers for the vessels, etc.) Without an appropriate index, all contemporary cloud key-value stores need to check all objects in the database for intersection or containment against the queried intervals. Granted, this process could take advantage of the high parallelism inherent in these systems, as all nodes could be accessed in parallel (e.g., with MapReduce). However, note that (a) not all nodes may contain relevant data, and (b) even on relevant nodes, the processing engine would have to access any and all stored rows to check for compliance with the query predicate constraints.

Challenges: The open challenges we tackle here are the following. First, we wish to identify appropriate indices which can be employed on cloud key-value stores; this requires a matching of the stores' peculiarities and capabilities to the index structures, finding appropriate key-value representations for such indices, and developing novel MapReduce algorithms to build and populate them. Second, specialized interval indices, capable of answering interval queries are typically static structures, which would preclude them from our intended environment; thus, we need to develop strategies that can efficiently handle updates in an efficient manner, appropriate for cloud stores. Third, specialized appropriate interval indices come from fields such as computational geometry and were intended for use as a main-memory index; our adaptation of such indices must address this explicitly. Fourth, we wish to derive new indices, stemming from and exploiting the key performance traits of key-value stores. Finally, given the richness of possible queries, the key question is to study which index can best handle efficiently which query types; then, given a query, it is important to send it to the index that is best suited for the query, or even decompose it into subqueries each one of which utilizes different indices.

Contributions: Our contributions include:

- A compound index structure, comprised of two tiers:
  - The MRSegmentTree (MRST); a distributed segment tree index structure with a key-value representation, bypassing the main-memory limitations of the Segment Tree.
  - The Endpoints Index (EPI); an inverted index on interval endpoints information, as an interval index, naturally arising from the key-value stores' capabilities.
- MapReduce algorithms for efficiently and scalably creating and populating these index tiers.
- Algorithms and additional structures (the UpdatesIndex, UI) for maintaining our indexed data in the face of on-line data insertions and deletions, ensuring up-to-date query results for a negligible overhead during query processing.
- A set of algorithms for interval-query processing over either MRST or EPI, or both, taking advantage of the query processing characteristics of our two index tiers.
- An implementation and thorough performance evaluation of all of our algorithms for index building and interval-query processing using Hadoop and HBase, showing significant speedups against two alternatives: (i) the native support provided by HBase through Hbase filters, and (ii) the Hive[9] query processor.

To our knowledge, this is the first work to address interval indexing and query processing on cloudstores. We plan to make our codebase available and submit it for inclusion in a future release of HBase.

II. Data and Query Model

Our algorithms and data structures are designed for modern cloud key-value stores. Below we outline the common characteristics of these systems that are relevant in our context. As our prototypical implementation is over HBase, we borrow its lingo to describe basic building blocks; however, it should be clear that our approach is applicable to other key-value cloudstores as well.

The smallest unit of data – the equivalent of a single value of some attribute in the relational model – is called a *column* or (abusively) a *key-value pair*. Columns are in essence associative arrays with four keys: rowkey, column name, column value, and timestamp (denoted {*rowkey*: <*column name*=*column value*> @ *timestamp*}). Each such data item is uniquely identified by the triplet {*rowkey*, *column name*, *timestamp*}, also coined the item's *key* (we shall use the term *columnkey* for clarity). Columns are contained within *ColumnFamilies* – the rough equivalent of column-store relational tables – while all columns of a ColumnFamily sharing the same *rowkey* are considered to belong to the same *row* – the equivalent of relational tuples. ColumnFamilies are *sparse*; that is, they can contain any set of columns and are not required to contain any particular column. Within each column family, *columns* are stored sorted on their *columnkeys*, allowing for fast sequential row-level access on the *rowkey* values. Furthermore, several of the cloudstores further utilize stable storage indices to allow for fast random access, although again based only on *rowkeys* (e.g., see HBase's *BlockIndex* and StoreFile Bloom filter options). Last, ColumnFamilies may be logically grouped into *Tables/Relations*, which in turn are horizontally partitioned on their items' *rowkeys* (coined *regions* in HBase) and distributed across the nodes of the cloudstore (coined *regionservers*).

All well-known cloudstores use in-memory caching, write-ahead-logging, and immutable stable storage, to provide high-throughput/low-latency writes. The typical write path consists of: (i) recording mutations in the write-ahead log, (ii) storing the actual data in an in-memory cache, and (iii) lazily writing new data to *storefiles* on disk in an append-only fashion, moving onto new *storefiles* when the current ones exceed some predefined size threshold. Deletions are handled through

tombstone records, while newer versions of a data item (i.e., items with the same *rowkey* and *column name* but larger *timestamps*) may coexist with or hide older ones, according to the system configuration. Write operations are typically either single-row/single-column mutations (*put(rowkey, ...)*) or single-row/single-column deletions (*del(rowkey, ...)*). Read requests can be either for single values (i.e., *get(rowkey, ...)*) or sequential scans (i.e., *scan(start rowkey, end rowkey, ...)*. Read requests first go through the in-memory cache; if the request cannot be served entirely (or at all) from there, the cloudstore goes on to scan its storefiles. All of these operations can further define other information in addition to the rowkey on which to operate, such as column names, timestamps, column families, etc., through what is called "filters". The append-only write scheme means that, over time, data for any given "row" can be stored across several *storefiles*, thus hurting read performance, as a read operation for all items with the same *rowkey* (i.e., a *row read*) would have to scan several storefiles; to this end, there is a periodic operation, coined *compaction*, that scans through the storefiles, applying deletes from tombstone records, and defragmenting the rows spanning multiple storefiles.

As should be obvious from the above, key-value cloudstores are designed for fast point queries and sequential scans on rowkeys. However, operations on other attributes (e.g., column names, column values, etc.) – including implicit key-value timestamps – can be quite inefficient. Even a simple point query on something other than the rowkey (e.g., *SELECT * FROM Table.ColumnFamily WHERE* $a \leq$ *Column.Timestamp* $\leq b$) would have to scan through all columns of the requested ColumnFamily across all regions and all cloudstore nodes. This fact alone makes efficient interval query processing all the more challenging.

On the other hand, intervals are either explicit (e.g., the base data features a column or pair of columns that define the left and right endpoint of the interval) or implicit (e.g., in a web archiving scenario, the timestamp of the crawl of a URL defines the left endpoint of an interval, the next crawl for the same URL defines the right endpoint, etc.) This is all orthogonal to our solutions; our algorithms and structures accept intervals as inputs. In scenarios of the second case, it is up to the application to deal with creating intervals out of the base data to be fed to the index and query engine. Consequently, the base data schema can be assumed to consist of rows identified by a rowkey and containing (among others) two columns storing item's interval; i.e.:

$$\{rowkey\text{: } <begin = *>, <end = *>\}$$

In this work we primarily deal with two types of queries:

- Intersection Queries: Given an interval $I = [a, b]$, return the id's of items whose interval has a non-empty intersection with I; that is items whose interval contains the query interval, plus items with either interval endpoint $\in [a, b]$. More specifically, a row satisfies the intersection query predicate, iff:
$$(begin \leq a \bigcap end \geq b) \bigcup$$
$$((begin \geq a \bigcap begin \leq b) \bigcup (end \geq a \bigcap end \leq b))$$
- Stabbing Queries: Given a point a, return the id's of items that are "alive" at point a; that is, items with $begin \leq a$ and $end \geq a$. This is a special case of Intersection Queries where $a = b$, but such queries arise frequently enough to warrant provision for added optimizations. More specifically, a row satisfies the stabbing query predicate, iff:
$$(begin \leq a \bigcap end \geq a)$$

Let us examine how an interval query of one of the above forms can be answered natively by a cloud key-value store. Remember that each object is stored on a region server, based on its *rowkey*. We can discern the following cases:

1) The rowkey is some information other than the interval data, and there is a separate interval-related column – e.g., in the web archive scenario, the timestamp is recorded along with other data in each row.
2) The rowkey is some information other than the interval data, and there are separate columns recording the item's interval.
3) The interval is explicitly used as (a prefix of) rowkeys (e.g., a concatenation of the left and right endpoint values is prefixed to the rowkey).

In the first case, interval queries are translated into selection queries over the timestamp column. As rows are distributed across regionservers based on the rowkey, the query processing algorithm will have to access all regionservers in search of matching rows. Additionally, any given regionserver will have to search through all of the rows it stores, as there is no index on information other than the rowkeys. The same holds for automatic timestamps, as there is no global index on that information either. What's more, in this case cloud key-value stores can answer *none* of the query types considered in this work *at all*, as each item bears only the left endpoint of its timespan interval. In the second case, interval queries are translated into selection queries over the interval columns; unfortunately, again the cloudstore cannot reduce the processing of the query to a limited number of regionservers and thus all rows on all regionservers will have to be checked against the query predicate. The third case can allow for efficient retrieval of intervals (c), (h), and (i) in Fig. 1 through a range query ($scan()$) over rowkeys, but in order to fully answer an interval query one would have to resort again to accessing all data on all regionservers.

III. Interval Indices

The above discussion showcases the inherent inability of an indexless setup to efficiently process interval queries (if at all). To this end, we have built an index structure, consisting of two tiers: the *Endpoints Index* (EPI), an inverted index on the interval endpoints, and the *MRSegmentTree* (MRST), a novel key-value adaptation of Segment Trees for cloud stores. Our index is stored as a cloudstore table consisting of two column familes: one for the EPI and one for MRST index data. This decision was dictated by our desire to ride on the scalability and performance of the underlying distributed key-value store. We first give an overview of Segment Trees, then describe

our indices and the algorithms for building, maintaining, and querying them.

A. Segment Tree Overview

Segment Trees were proposed to support indexing of intervals with logarithmic complexity querying [10]. Let id : $[begin, end]$ be an interval with endpoints $begin$ and end ($begin \leq end$) identified by id, and let a list of N such intervals be the input to our algorithm. An *elementary interval* is an interval $[e, e']$, so that e and e' appear as endpoints of some interval(s) in the input collection, and there is no input interval endpoint in (e, e'). In other words, elementary intervals can be computed by creating a sorted set of the left and right endpoints of all intervals in the input, and then considering all sequential pairs of endpoints from this set.

For N input intervals, there are $O(N)$ elementary intervals. Given these, the Segment Tree is a balanced binary tree of height $O(\log_2 N)$, using $O(N \log_2 N)$ space, with the following properties (see Fig. 2 for an example):

- Each node in the tree stores the left and right endpoint of the range of values it indexes, along with pointers to its left and right child nodes (where applicable), just like in ordinary binary trees. Let $span(x)$ denote the interval defined by the endpoints stored in the node.
- Elementary intervals are indexed at the leaf level; that is, for every elementary interval in the input collection there is a single leaf node with endpoints equal to the elementary interval's endpoints. Moreover, elementary intervals are indexed in an ordered manner, so that the leftmost leaf node corresponds to the leftmost elementary interval, and in general the i'th leaf node stores information for the i'th elementary interval.
- Internal nodes always have two child nodes and index the union of their ranges; i.e., the left endpoint of an internal node equals the left endpoint of its left child, and its right endpoint equals the right endpoint of its right child. Furthermore, the id of an internal node equals the middle point of its children; i.e., the right endpoint of its left child (or equivalently the left endpoint of its right child).
- Each (leaf or internal) node further stores a subset of the IDs of input intervals, so that each interval id : $[begin, end]$ in this set (i) contains $span(x)$, *and* (ii) does not contain $span(parent(x))$. That is, each node in the Segment Tree stores the intervals that span through its endpoints but do not span through its parent's endpoints.

B. Endpoints Index and MRSegmentTree Index

The creation of EPI and MRST proceeds in phases. First, we compute the EPI data/elementary intervals from the input data, using a MapReduce job. Given these, we build subtrees of the complete tree, merge them, and populate them with interval IDs, using another MapReduce job. We shall also prove some fundamental features of our approach, relating the number of subtrees to the number of map/reduce partitions and also prove that our algorithms indeed produce a tree with the key properties of a Segment Tree.

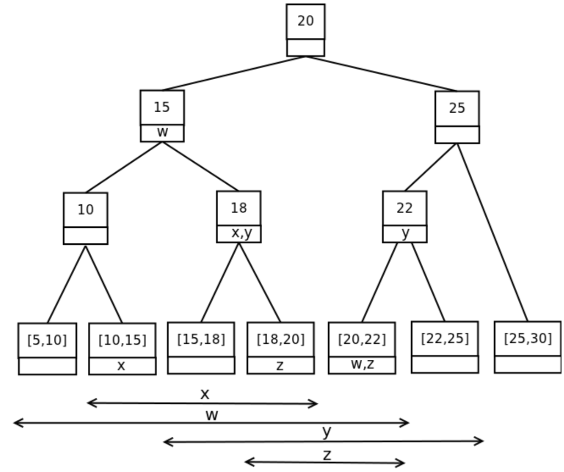


Fig. 2. Example Segment Tree

1) Endpoints Index: The first step in building a Segment Tree is computing the elementary intervals, using a MapReduce algorithm. Remember that items are assumed to be stored in rows with the id of the object (e.g. web page url, etc.) as the rowkey, and a pair of begin/end columns. Each mapper is automatically assigned a subset of the input keys/rows by the MapReduce framework. For each row $\{rowkey : <begin = a>, <end = b>\}$ in its input set, each mapper emits two rows/key-value pairs: $\{begin$: $<$ "r"$|rowkey = end >\}$ and $\{end$: $<$ "l"$|rowkey = begin >\}$, where "x"$|y$ denotes the binary concatenation of the string "x" and the binary representation of y, and "l" or "r" denote that the column value is the left or right endpoint (in which case, the key-value pair's rowkey is the right or left endpoint value) respectively.

If we take a more careful look at this index, we can see how it is in essence an inverted index on the interval endpoints. We coin this the "*Endpoints Index*" (or EPI) and store it in a column family of its own. We choose to treat this data as a separate index tier, as it exhibits different query characteristics than the tree structure discussed shortly, dictating the decomposition of queries to subqueries over the two tiers, as we shall see shortly.

From an implementation point of view, the MapReduce framework can be configured to write the above "emitted" key-value pairs to the appropriate cloudstore column family, either by using a special "null" table reducer that directly stores the emitted key-value pairs into the cloudstore (also called a "map-only job"), or to store it in specially formatted files (called *HFiles* in the case of Hadoop/HBase) that are then bulk-loaded into the cloudstore.

In some cases, the endpoints of the elementary intervals are known or can be computed a priori. For example, consider a web archiving system where crawlers periodically download and store new versions of each archived web page with a fixed frequency (e.g., web pages are updated every say hour, day or week, depending on their popularity and other metrics). Moreover, in real-world systems, interval queries may have relaxed precision requirements, in the sense that one is hardly

Rowkey	← – – – Key-value pairs – – – →				
7L	l=5	r=10			
10	l=5	r=15	L=7L	R=12L	
12L	l=10	r=15			ix10=20
15	l=5	r=20	L=10	R=18	iw5=22
16L	l=15	r=18			
18	l=15	r=20	L=16L	R=19L	ix10=20, iy15=25
19L	l=18	r=20			iz18=22
20	l=5	r=30	L=15	R=25	
21L	l=20	r=22			iw5=22, iz18=22
22	l=20	r=25	L=21L	R=23L	iy15=25
23L	l=22	r=25			
25	l=20	r=30	L=22	R=27L	
27L	l=25	r=30			
R	p=20				

Fig. 3. MRSegmentTree for the example of Fig. 2 (column name prefix "l"/"r": left/right endpoint, "L"/"R": left/right child rowkey, "i": interval IDs, "p": root node rowkey)

ever interested in nanosecond accuracy for an object's crawl time. Instead, we can support an hourly/daily/weekly etc. query granularity, by employing matching fixed elementary intervals (one hour, one day, one week, etc.)

2) Building Subtrees: MRST nodes are stored as rows in a separate column family (see Fig. 3 for an example). Each internal node/row uses the middle point of its children (i.e., the value of the right endpoint of its left child) as its rowkey, while leaf nodes use the median of their interval range (i.e., right and left endpoints) plus a special suffix ("L") to differentiate them from internal nodes. In both cases, the row further consists of columns for the node's left ("l") and right ("r") endpoints, the rowkeys of the node's left ("L") and right ("R") child, plus columns for the IDs of the items/intervals indexed on that row, named "i"$|item\ ID|left\ endpoint$, with the right endpoint as their value. Rows corresponding to leaf nodes will of course have no columns for child node rowkeys, and nodes storing no input interval rowkeys will have no "interval IDs" columns. Last, there is also a single row with rowkey "R" storing a single key-value pair with "p" as the column name and the rowkey of the tree's root as its value. For example, the first line in Fig. 3 (i.e., row {7L: <"l"=5>, <"r"=10>}) corresponds to the left-most leaf node of Fig. 2, indexing the interval $[5, 10]$ storing no interval IDs; the fourth line (i.e., row {15: <"l"=5>, <"r"=20>, <"L"=10>, <"R"=18>, <iw5=22>}) corresponds to the left child of the root, indexing the interval $[5, 20]$, storing interval $w : [5, 22]$.

In order to build and populate this structure, we execute a single map-only MapReduce job. In this job, each mapper reads a sorted partition of EPI rowkeys (i.e., elementary interval endpoints), computes an in-memory subtree corresponding to these intervals, populates its subtree, and then emits individual nodes to the MapReduce framework. We shall first discuss the subtree building phase, then show how the population is performed. Let M be the number of map tasks in this phase, E be the number of elementary intervals each such task should get, and N be the total number of elementary intervals in the EPI. In order to produce a correct Segment tree, we must make sure that M is a power of two (see [10]), and that each of the M map tasks is assigned the proper subset of $E = \frac{N}{M}$ elementary intervals. M, the number of mappers, is user-selectable, based on the desired degree of parallelism (subject to the power-of-2 constraint). Then, before starting the MapReduce job, we retrieve the count of elementary intervals. Then, the number of elementary intervals assigned to each mapper is computed by recursively dividing N by 2 until M parts have been produced; when the number being divided is odd, the "left" part receives a rounded-up and the "right" part a rounded-down half of intervals. The core idea is that each mapper builds a proper segment tree for the elementary intervals it receives, then individual subtrees can be merged to produce the final MRST.

Lemma 1: The k-th mapper of the Tree Building phase will produce the k-th subtree of S with $\log_2 \left\lfloor \frac{N}{M} \right\rfloor$ height.

Proof: Due to the above partitioning, the k-th map task will be assigned the intervals numbered in:

$$\left[1 + \left\lfloor k \cdot \frac{N}{M} \right\rfloor, \left\lfloor (k+1) \cdot \frac{N}{M} \right\rfloor\right]$$

Each mapper will get $i = \left\lfloor \frac{N}{M} \right\rfloor$ elementary intervals. If the division of N over M leaves a remainder then the $N \bmod M$'th mappers will get one more elementary interval. Thus, the subtree built has a $\log_2 i$ height, and the k-th mapper produces the k-th subtree of S. ■

Running the algorithm for the tree building up to this point ensures the building of M subtrees of the Segment tree. For the complete segment tree S we need to assemble the tree from the M subtrees produced during the previous step, up to the root of S. Consider the intervals defined by the roots of the subtrees; then, the segment tree built from their interval endpoints is identical to the top levels of the complete segment tree. We coin this tree the "top-tree". As it is small in size (consisting of $O(M)$ nodes) and the relevant endpoints are already known (computed during the bootstrap stage of this phase, while calculating M and the mapper input partitions), it can be pre-computed and stored during the startup phase of the MapReduce job. Then, each mapper first loads the top-tree, computes the Segment tree for its input data, attaches it at the corresponding position in the top-tree, and then iterates over all tree nodes emitting their information (without interval ID columns at this stage) to the MapReduce framework. Again, the latter can be configured to either write data directly to the cloudstore, or to dump it to flat files and bulk-load them.

Theorem 1: The data structure built in the Tree Building phase is a correct Segment Tree.

Proof: From the definition of the Segment Tree, the algorithm used in the mappers of the first phase builds a Segment Tree locally, so the M subtrees are correct Segment Trees. The textbook approach to build a segment tree is by finding the median of the elementary interval set and then building two new segment trees recursively from the sets defined by splitting up the input set at that median. If only $\log_2 M$ recursions are allowed then there will be M subsets of elementary intervals produced. It can be easily shown that the distribution of the elementary intervals in those subsets is the same as the distribution defined by lemma 1, so the subtrees

created by each mapper would be the same as those created for the correct segment tree. Moreover, the top-tree, having the M subtree roots as its leafs, is also a correct segment tree and equals the top $\log_2 M$ levels of the complete tree. ■

3) Tree Population: Up to now we have created and stored the tree structure in the cloudstore. We now have to populate the MRSegmentTree with interval IDs to make it fully operational. When each mapper has completed building its subtree, it has in essence a completely functional part of the total MRSegmentTree for the range of elementary intervals assigned to it. In order to populate this part, the mapper initiates a scan on the Endpoints Index column family for that same range. For every row of the Endpoints Index retrieved, the mapper computes the input intervals by putting together all combinations of the rowkey and "r"-$*$ columns. For each such row $\{a\text{: } <\text{"}r\text{"}|rowkey = b>\}$ the interval $rowkey$: $[a, b]$ is in turn inserted in the partial tree, with the sole difference that instead of storing interval IDs in the in-memory tree, the mapper emits a $\{node.id\text{: } <\text{"}i\text{"}|rowkey|a = b>\}$ key-value pair (thus leading to this key-value pair being stored in the cloudstore). Note that IDs of intervals with at least one endpoint in a mapper's partition do appear in its subtree, thus intervals spanning multiple partitions (and thus indexed in the top-tree) are added to the appropriate top-tree node(s) by the mappers responsible for their endpoints.

C. Index Updates

The segment tree is by definition a static structure, and (as we shall see) building the MRST consumes considerable time and cloud resources. Thus, accommodating online updates would either result in excessive resource consumption (if the MRST is rebuilt on every update) or in query results which are oblivious to these insertions/deletions (if the tree is rebuilt periodically). This section addresses these problems by exploiting the key-value stores' inherent high write throughput and fast range scans.

We extend our two-tier index scheme by adding a third tier, coined *Updates Index* (or UI). This is in essence a separate, smaller Endpoints Index, storing information on data mutations since the creation of the MRSegmentTree index and implemented as a new column family in the index table. On system startup, the above algorithms are executed to bootstrap the EPI and MRST column families. After that, insertions are recorded to both the Endpoints and Updates Indices, while deletions are handled through deletion of the actual records and insertion of "tombstone" records: key-value pairs with a "-" character suffix in the column name. When present in a query result set, these records translate to a removal of their "positive" counterparts from the final result set. We provide a short example of their usage in the query processing discussion. Last, updates are handled as an insertion of the new values followed by a deletion of the old ones (performed in a single operation, if supported by the cloudstore).

Fig. 4 depicts the changes to the EPI and UI when x's interval is updated from $[10, 20]$ to $[9, 18]$. The starting state of the EPI (Fig. 4(a)) corresponds to the tree of Fig. 2. The update call first inserts the new values in the indices, resulting in the state of Fig. 4(b). Then comes the deletion of the old values, resulting in the state of Fig. 4(c). Notice how the EPI always has fresh information, while the UI only maintains information pertaining to changes (much like a log).

The information recorded in the Updates Index must eventually be integrated in the MRSegmentTree data. This is performed periodically, based on the actual query/update workload; for example, we can choose to rebuild the MRSegmentTree when the Updates Index reaches a predefined size threshold, or when query processing times exceed a predefined time threshold, etc. When the time comes, two new column families are created: one to hold any further updates coming in while the MRSegmentTree is being rebuilt and the other to hold the new MRSegmentTree data. Then, the Updates Index and MRSegmentTree are "frozen"; when the system is in this state, insertions/deletions to/from the index propagate only to the temporary Updates Index. The system then initiates the tree building/population algorithms outlined earlier to create the new tree. When this process completes, the system updates the Endpoints Index with the data in the temporary Updates Index, switches to the new MRSegmentTree and Updates Index column family, and dumps the old Updates Index and MRSegmentTree. As all key-value pairs are timestamped, replaying of the temporary Updates Index consists of merely `put()`/`del()` operations with the same timestamps as the update data, translating key-value pairs with a "-" prefix to deletions and the rest to insertions.

IV. Processing Interval Queries

We first describe how interval queries can be processed by using only EPI or MRST. Then, we describe a hybrid algorithm using both index tiers. Last, we discuss how interval queries are processed in the face of online updates.

A. Interval Queries on Endpoints Index

Given an interval query $[a, b]$, we can proceed as follows:

- Intersection Queries (i.e., $b{\neq}a$): Execute (in parallel) two scans on the EPI column family: (i) a $scan()$ for all items with rowkeys in $[a, b]$, and (ii) a $scan()$ for items with rowkeys in $(-\infty, a]$, filtering for key-value pairs whose column name starts with "r" and value is $\geq b$. What this second scan does is that it returns all those columns corresponding to intervals with a right ("r"-prefix) endpoint $\geq b$, and a left endpoint $\leq a$ (due to the rowkey constraint); i.e., all those items whose interval covers the query interval. Finally return the item IDs (i.e., the column name without the "l"/"r" prefix) of the set union of the two result sets.
- Stabbing Queries (i.e., $b{=}a$): Execute a single scan on the EPI column family for items with keys in $(-\infty, a]$, filter/keep key-value pairs whose column name starts with "r" and value is $\geq a$, and return the item IDs.

Although the Endpoints Index can process all types of interval queries dealt with by our work in a more efficient manner than in an indexless case, there are still scenarios

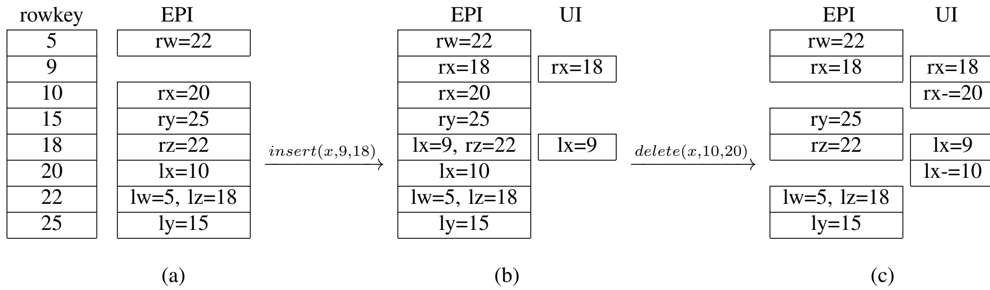


Fig. 4. Example of online updates $(update(x, 10, 20, 9, 18) \rightarrow insert(x, 9, 18) + delete(x, 10, 20))$

where the above scans can be quite slow. More specifically, unless when dealing with queries whose interval spans a large part of the dataset, the first part of Intersection query processing is inherently fast as it uses directly the cloudstore's $scan()$ primitive. It is the second part, also appearing in Stabbing query processing, that proves to be the performance hog and hence a target for optimization.

B. Interval Queries on MRSegmentTree

In general, MRST-only queries proceed as follows:

1) Retrieve the row for the root node of the MRST.
2) If the query interval overlaps with the current node interval, append all interval IDs stored on the current node to the result set.
3) If the query interval overlaps with the interval of the left child node, descend to it and recur to step 2.
4) If the query interval overlaps with the interval of the right child node, descend to it and recur to step 2.

The recursion always stops at the leaf level, so at least $\log_2 N$ nodes are visited in the process. Moreover, if the query interval overlaps with both child nodes, the algorithm descends to both nodes in parallel.

Figure 5 shows the nodes visited by an intersection query for the interval $[17, 21]$, and a stabbing query for 17, on our example MRST. Remember that the MRST nodes are stored in the cloudstore using their node IDs as the rowkey (see Fig. 3). To retrieve a node we just issue a get() to the cloudstore for the rowkey of that node, and the cloudstore responds with all the columns of the MRST column family for that rowkey, including the interval IDs stored in the given node and the rowkeys of its child nodes (if any). Intersection queries are somewhat more complicated than stabbing queries, since if there is a non-empty intersection with both children of the current node, the query descends in parallel to both subtrees.

C. Querying MRST and EPI

As is evident from the above, stabbing queries are very efficient when processed using the MRST. However, Intersection queries can visit an arbitrarily large subset of the nodes in the tree, depending on the query interval. This would pose no problem in a centralized environment where the complete Segment Tree would be available directly to the query processor (possibly stored completely in main-memory); however, in our large-scale, distributed scenario, retrieving

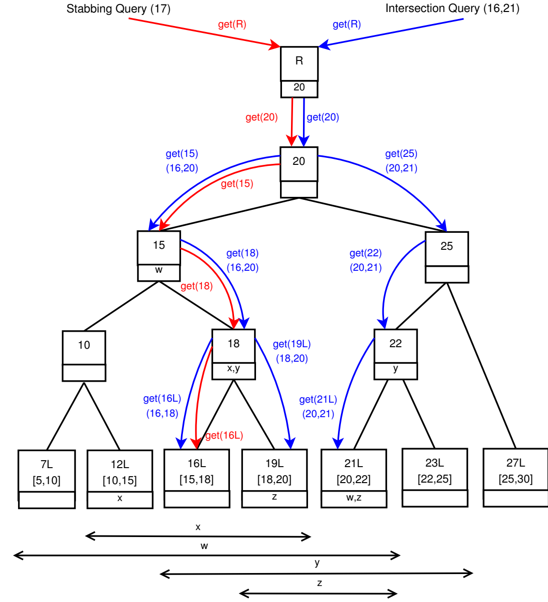


Fig. 5. Example Queries on the MRSegmentTree

parts of the tree one level at a time can prove to be quite costly (we discuss this in more depth in Sec. V). In order to overcome this situation, we proceed by using a hybrid of both the MRST and EPI structures.

Note that the intervals comprising the result set of an Intersection query can be divided in two categories: (i) intervals completely containing the query range, and (ii) intervals with at least one endpoint (i.e., intersecting or completely contained) in the query range. The second kind of intervals is retrieved by a range query over the Endpoints Index. In order to retrieve all intervals with at least one endpoints in query interval $[a, b]$, we perform a $scan()$ for all items with rowkeys in $[a, b]$. The cloudstore will then access just the regionservers storing relevant information, and directly retrieve only the relevant rows (i.e., with $rowkey \in [a, b]$). Recall that for each endpoint, the Endpoints Index also stores the interval IDs and the other endpoint for all matching intervals. Thus, the set of IDs returned by the above operation consists of just the rowkeys of intervals with a non-empty intersection with I. Then, the first kind of intervals can be retrieved by executing a stabbing query on MRST, for any point in the query interval. These two queries are executed in parallel, so that the overall

query processing time is (on average) equal to the maximum of the two.

D. Interval Queries with Online Updates

As outlined above, the MRSegmentTree is used for stabbing queries and the Endpoints Index for a range query for the query interval range. With the Updates Index in place, incoming queries are again broken up into two parts; the range query part is executed only on the Endpoints Index, while the stabbing query part is executed on both the MRSegmentTree and the Updates Index (when the system is in the phase of updating the MRSegmentTree, the temporary Updates Index is also taken into account). The reasoning behind this decision should be obvious: with updates propagating directly to the Endpoints Index, the range query part will always return fresh information. However, the stabbing part, which up to now was executed only on the MRSegmentTree, may return stale data; by taking into account the data from the Updates Index as well, stale results are dropped from the result set and fresh are added. These three queries are executed in parallel, so that the overall query processing time is again (on average) equal to the maximum of the three. An important difference is that in the case of online updates, the stabbing part always queries for the interval's left endpoint, as opposed to a random point in the interval. The reasons for this are twofold: (a) since stabbing queries on the Update Index are in essence $scan()$'s from the minimum key value in UI up to the query value, this leads to fewer rows being examined and thus a faster turnaround time; and (b) choosing the left endpoint guarantees that all relevant tombstone records are located, as we explain below with a short example. Also, note that UI will be small by design, hence running such queries on it will be very fast!

Consider the example of Fig. 4(a) and 4(c). A Stabbing query for value 9 in the original state would be processed by the MRSegmentTree alone, returning only w as the result. With x updated to $[9, 18]$ in the final state, the MRSegmentTree would still miss x; however, by executing the Stabbing query on the Updates Index as well using the method outlined in section IV-A – i.e., scan rows $(-\infty, 9]$ and keep results with "r" column name prefix and a value ≥ 9 – we now add x to the result set. Now, consider an Intersection query for $[19, 21]$. In the original state, we would execute a scan on the Endpoints Index for this interval, thus adding x to the result set, and a Stabbing query on the MRSegmentTree, adding w, y, and z. With the left endpoint of x updated to 18 in the final state, the scan on the Endpoints Index would not add x, but the Stabbing query on the stale MRSegmentTree would still return w, y, z, and $x : [10, 20]$. Now, note that the Stabbing query for 19 on the Updates Index will return the tombstone record $\{$10: <"r"x"-"$= 20$>$\}$; this translates to "omit any record with key x, left endpoint 10, and right endpoint 20", thus removing $x : [10, 20]$ from the final result set.

V. Experimental Evaluation

Our experimental evaluation was performed on the Amazon EC2 infrastructure, using the versions of HBase and Hadoop in the Cloudera CDH3u3 distribution. Our implementation consists of approximately 3.5k (physical) lines of Java code. Our clusters consisted of three, five, and nine m1.large nodes (each with two 64-bit virtual cores with 2 EC2 compute units each, 7.5 GB of memory, and 850 GB of instance storage). The above clusters all comprised one HBase master node (also a Hadoop namenode and jobtracker), and two, four, and eight HBase regionserver nodes respectively (also operating as Hadoop datanodes and tasktracker nodes). Sample results with larger cluster sizes (i.e., 20+1, 40+1, 80+1 nodes) agree with those presented here, with indexing times falling quasi-linearly to the cluster size, and query processing times seeing slight reductions across all cases; we do not report on them, though, as we only got full sets of measurements for the smaller clusters, due to budget limitations with the larger ones. We measured both the time required to build and populate the indexes, and the query processing time with the various algorithms outlined earlier. The figures presented below were averaged over 5 repetitions for the indexing times, and 20 repetitions for the query processing times.

To study the effect of dataset size and interval distributions on our algorithms' performance, we employed two real-world data sets, coined "UKGOV" and "DMOZ". The former is a circa 2.5-years worth web archive of the gov.uk domain, consisting of $\approx$1 million web pages, resulting in $\approx$5.7 million intervals (i.e., distinct page versions) with distinct nanosecond-accuracy endpoints, amounting to $\approx$1TB of raw data. The latter is a circa 2-years worth web archive of the dmoz.org domain, consisting of $\approx$1.9 million web pages, for a grand total of $\approx$2.5 million intervals, again with distinct nanosecond-accuracy endpoints, amounting to $\approx$300GB of raw data. In addition to the difference in size, these two datasets further exhibit totally different distributions of intervals (page versions) across time. The DMOZ dataset increases in abrupt steps around months 6 and 21, with the largest part ($\approx$65%) of web pages being added and subsequently modified in the last couple of months. The UKGOV dataset increases almost linearly, with new pages being added over time, and the number of modifications increasing in tandem with the number of pages, albeit at a lower pace.

For our queries, we used six different sets of query dumps: (i) three samples of 100 queries each drawn from an actual trace of user searches on the UKGOV dataset, corresponding to single points in time, one-week, and one-year ranges respectively, and (ii) three synthetically generated sets of 100 queries over the DMOZ dataset with selectivities of 2%, 25% and 75% of distinct intervals respectively. We compare our approach against both HBase filtered scans and Hive. We further present results on query processing performance in the face of index updates.

A. Index Building

Figure 6 depicts the index building time results for DMOZ and UKGOV, for the three cluster sizes, broken up into the time required by each of the EPI and MRST indices. As is evident, the time required to create/populate our two indices

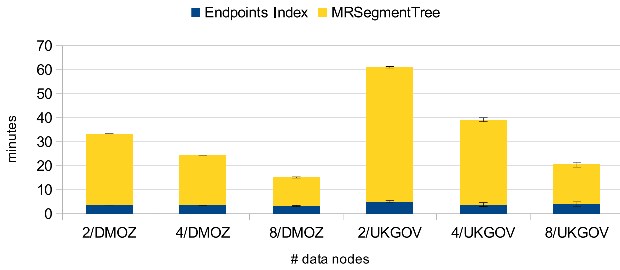


Fig. 6. Indexing time (DMOZ & UKGOV)

shows a linear dependency on the size of the input (remember that UKGOV is around twice the size of DMOZ). The first phase (computation of elementary intervals/building of EPI) takes on average the same (small) time across all three cluster sizes. On the other hand, the building/population of the MRST – the most time-consuming of the two phases – scales well with the number of nodes in the cluster, with the required time decreasing quasi-linearly to the number of nodes. The figure also depicts the standard deviation of the measurements, showcasing the performance stability of our algorithms.

The respective combined MRST+EPI index sizes where circa 1.5GB and 600MB for the UKGOV and DMOZ datasets respectively. In order not to penalize the HBase and Hive query processing results presented next, we reduced the initial datasets down to the bare minimum of timestamp pairs for each page ID for these two contenders; as we shall see shortly, even with this preprocessing on behalf of our contenders, our indices still manage to perform consistently better in query processing times, often by more than a factor of 2×-3×.

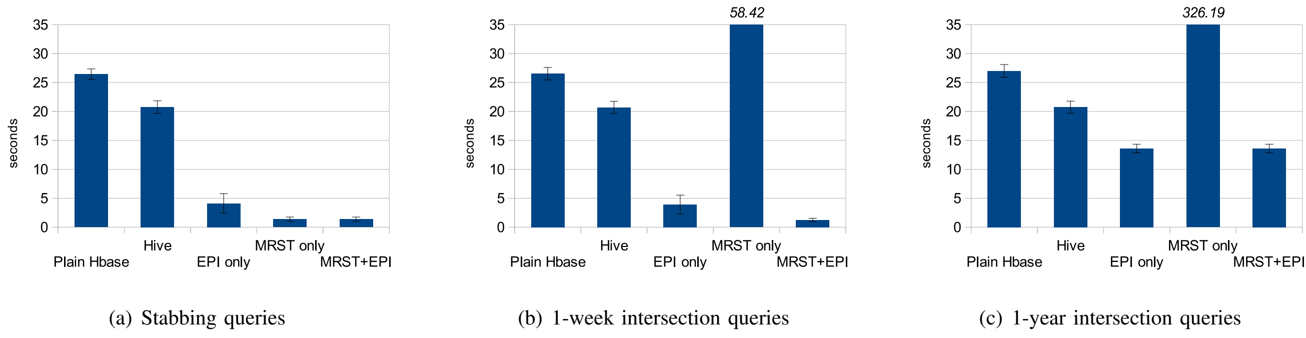
B. Queries

In the execution model of HBase, the client acts as the query coordinator. This is evident in our results, as the query turnaround times were practically the same across the various sized clusters. Hive, on the other hand, executes queries as MapReduce jobs and thus gains in performance with larger clusters. Due to this (and for space reasons), we report results only for the 9-node cluster.

First let us look at Stabbing queries. Figure 7(a) depicts the relevant results over the UKGOV dataset. As we can see, HBase filtered scans are the worst of the pack, requiring around 26”. Hive improved on this figure although not by much, dropping the query time to around 20”. By using only the Endpoints Index the time drops to around 4”, with a standard deviation of around 1.7. This is an already significant improvement over the current approaches. When using only the MRSegmentTree, the time drops even more to ≈1.4” with a standard deviation of ≈0.38, an improvement of more than an order of magnitude compared to our contenders, and almost 3× compared to the EPI-only scenario! Note that since the algorithm utilizing both indices already does a stabbing query for the required range, it takes the exact same time as the MRST-only case (i.e., the scan part is not executed at all).

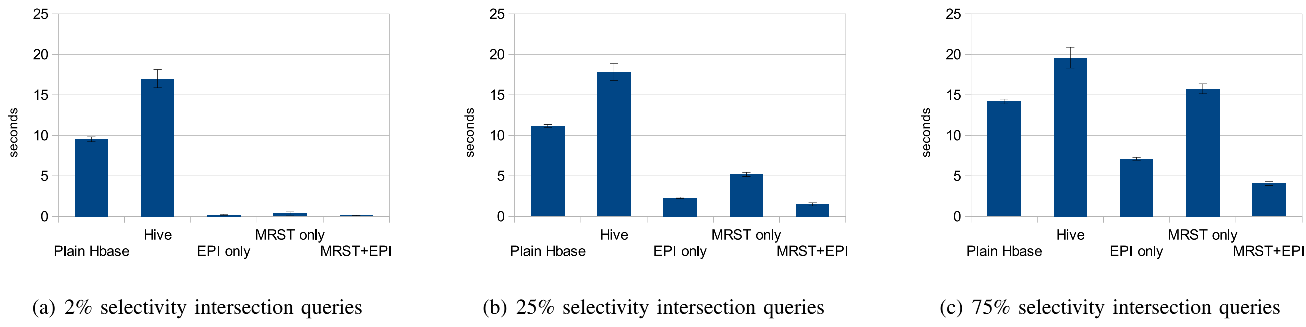
Next, figures 7(b) and 7(c) depict the query processing times for the 1-week and 1-year queries respectively over the UKGOV dataset. HBase and Hive again require around 26” and 20” respectively. Our first query processing algorithm, using only the EPI, requires on average 4” to answer the 1-week queries (with a standard deviation of 1.63), while using only the MRST this figure shoots to around 58”. The high cost of large intersection queries on the MRST is due to inherent characteristics of the MRST/Segment tree, HBase, and Java. For N input intervals there may be up to $2N$ elementary intervals and thus up to $\approx 4N$ nodes in the MRST. For 2.5M intervals this translates to over 10M MRST nodes/rows across 11-13 tree levels. Stabbing queries only retrieve a single node/row from each MRST level (e.g., a stabbing query on the above tree would consist of 11-13 get()s), but an intersection query spanning a large part of the interval space would have to retrieve millions of rows. Our first approach to parallelize this operation was to use a thread pool for the get() invocations. This provided only a constant factor speedup, while for larger pool sizes the context switching and network contention become major hurdles. We thus opted for an HBase-aware solution: pooling/sorting get()s per level and using batch operations (i.e., *HTable.get(List<Get>)*). The HBase client library groups requests by their target regionserver and dispatches them in parallel. This approach achieves a much higher throughput than any threading solution could do, taking advantage of HBase's high sequential read throughput and blockcache. We thus reduced the MRST-only time by an order of magnitude for large intersection queries, but such queries still required tens of thousands of such multi-get()s, again translating to a considerable time overhead, also due to TCP connection overhead, network contention, blockcache thrashing, etc. On the other hand, a $scan()$ for K consecutive rows is significantly faster than K perfectly batched Gets. This was the driving reason why we opted to break up queries into two complementary parts: a stabbing part on the MRST (i.e., a handful of get()s) and a range part on the EPI (via a row-limited $scan()$). In this case the hybrid approach (utilizing MRST and EPI) shines, achieving an average query time of around 1.2” (with a standard deviation of 0.3), again more than an order of magnitude lower than that of current approaches.

For the 1-year queries, the HBase and Hive solutions are again falling behind (≈26” and ≈21” respectively). With an interval this large (spanning nearly half of the dataset) the $scan()$-based operations start dominating the query processing cost. Thus, the EPI-only case achieves an average query response time of around 14”, the larger part of which is attributed to the scan (i.e., not the stabbing) component of the intersection query. The large interval effect is even more severe in the MRST-only case, leading to an average query processing time of around 326”. Last, as the scan part is also present in the MRST+EPI approach, it also dominates its processing time, hence the almost identical processing time compared to the EPI-only case. Despite this, this approach requires almost half the query processing time of plain HBase and around 65% compared to Hive.



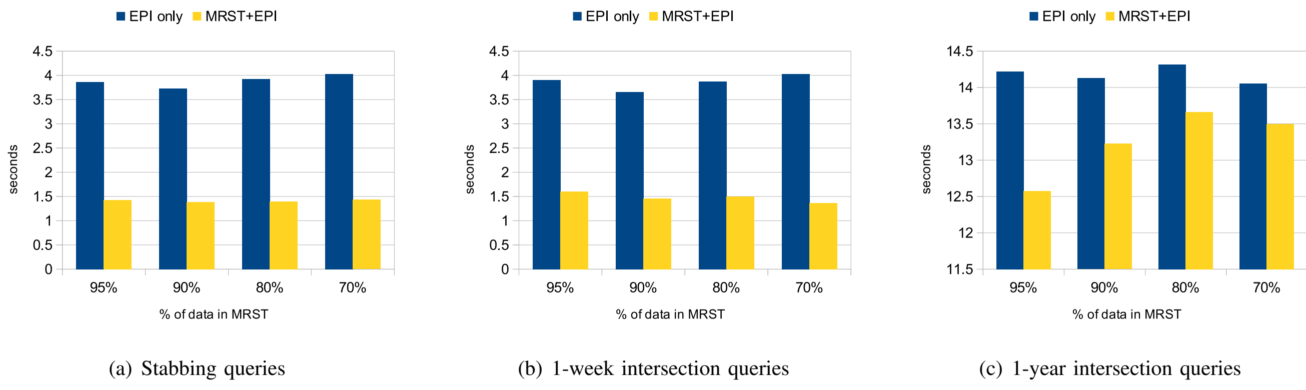
(a) Stabbing queries (b) 1-week intersection queries (c) 1-year intersection queries

Fig. 7. Query times for the UKGOV dataset



(a) 2% selectivity intersection queries (b) 25% selectivity intersection queries (c) 75% selectivity intersection queries

Fig. 8. Query times for the DMOZ dataset



(a) Stabbing queries (b) 1-week intersection queries (c) 1-year intersection queries

Fig. 9. Query times for the UKGOV dataset w/ online updates

Fig. 8(a), 8(b), and 8(c) present the query processing times for queries over the DMOZ dataset, for selectivities of 2%, 25%, and 75% of the intervals respectively. The situation here is quite different with regard to the HBase vs. Hive performance, with the former performing considerably better. As far as our algorithms are concerned, for small selectivities (2%) they are all clear winners, with all of them having more than an order of magnitude lower average query turnaround times, and the MRST+EPI approach being again the best of the pack ($\approx$0.1" average query processing time, compared to $\approx$0.2" for the EPI-only case and $\approx$0.35" for the MRST-only case). For medium selectivities (25%) the cost of the scan component increases again, thus leading to an increase of the overall query processing time of all of our algorithms ($\approx$2.3" for EPI-only, $\approx$5.2" for MRST-only, and less than 1.5" for MRST+EPI); even then, our algorithms achieve considerable improvements over the current approaches ($\approx$11" for the plain HBase approach and $\approx$18" for Hive). Last, for large selectivities (75%), the tree-only algorithm is worst; on the other hand, EPI-only requires (on average) only half the time of plain HBase, while MRST+EPI needs less than one third the time of plain HBase.

Finally, we look into query processing performance in the face of online updates. We show only results for the EPI-only and MRST+EPI algorithms, as index updates are either irrelevant or inapplicable to the other approaches. In these

experiments we inserted a sample of the original dataset into EPI and MRST at the beginning of each run, then treated the remaining dataset as a set of updates. Figure 9 depicts the query processing times for UKGOV (results for DMOZ are omitted for space reasons), for various percentages of the original dataset already having been inserted into MRST. Our algorithms incur little to no extra cost in all cases but one: the 1-year query set for the UKGOV dataset. For all other query sets, the stabbing part of the query dominates the processing time. The reason the MRST+EPI cost is increasing with the number of updates is that in these queries the scan component of queries greatly dominates the stabbing part, and it is this cost component that scales with the number of updates, as stabbing queries on the Updates Index are answered using a $scan()$ operation (see section IV-A). We have omitted showing figures for the cost of index updates; this is due to the fact that this cost is miniscule, as all index update operations are either $put()$'s or $del()$'s, which are by design very fast (i.e., few msecs) in HBase.

VI. Related Work

Interval indexing and querying in key-value cloud stores has not been previously addressed. Work related to ours includes efforts to (i) develop B-tree and R-tree variants for indexing objects in (bi-)temporal DBs, and (ii) to develop B-tree and R-tree indices with MapReduce.

We chose Segment Trees as a starting point for our indexing efforts, as (along with Interval trees) it is the prototypical structure for indexing 1-dimensional intervals [10]. Segment Trees require more storage than Interval Trees However, Intervals Trees induce greater query response times since, at each accessed tree node, post processing is required to filter out irrelevant intervals stored at that node. Further, Segment Trees can easily be extended to handle multiple dimensions (with multi-level Segment Trees). Finally, related literature (from temporal DBs) shows that the essentials of Segment Trees have been used to improve previously-inadequate variants of B-tree or R-tree indexes, [11].

The community has also investigated alternative index structures, primarily in the context of (bi-)temporal DBs, [11], with most of these being B-Tree and R-Tree variants. As our data items (intervals) are one-dimensional, R-trees are inappropriate. R-trees have been employed in two-dimensional indexing in bi-temporal DBs, [12], albeit without great success [11] due to lack of clustered data objects in several dimensions and the fact that for some interval queries there is a high overlap among data objects; a fact that is known to cause performance problems for R-Trees. Some of these performance problems can be avoided if Segment R-Trees [13] are used, which utilize essential features of Segment Trees. B-Tree-based variants, have also been employed primarily for 1-dimensional time indices in temporal DBs. One of the better known representatives, the Time Index, [14], results in an index that is too space-consuming ($O(n^2)$), as opposed to the Segment Trees $O(nlogn)$) and too costly for updates. In fact, some suggestions for lowering the above costs borrow ideas again from Segment Trees, as in the TimeIndex+ structure [15]. In general, however, B-tree indexes are also viewed as inadequate for the job [11]. On the other hand, a disadvantage of Segment Trees is that they are static structures (in that updates are supported only as long as elementary intervals do not need to be redefined). We manage to efficiently accommodate insertions and deletions, by employing an auxiliary structure and adopting the rather typical approach for cloud data, involving the periodic rebuilding of the segment tree. As our results show this efficiently overcomes this limitation.

The work in [16] is related to our work as it describes a Distributed Segment Tree. The emphasis is on supporting range and cover queries in a P2P DHT network overlay. The problem dealt with is quite easier than interval query processing and the underlying infrastructure is radically different.

In the cloudstore/MapReduce field, there is a number of works building tree-based indices for query processing speedups. All of these works (discussed shortly) build either B-tree or R-tree based indices, which are not as efficient for interval queries as segment trees, as discussed earlier. All tree-based MapReduce indexing mechanisms (including ours) are similar to the extent that they all proceed in phases of input data partitioning and subtree building/population/integration. This similarity in the MapReduce jobs structure is natural for all tree indices. However, our approach has several striking differences, outlined below.

In [17], authors show how to build R-trees with MapReduce for multi-dimensional data. The authors propose using two MapReduce phases: the first to compute a near-optimal partition function, and the second to let each reducer build a subtree of the R-tree. In [18] an extension is presented, proposing a new way to partition objects using the X-means Partition algorithm. [19] provide an analysis of the cost and benefits of parallel building of an R-tree. The issues in the parallel building of a structure storing spatial data are complex and the approach in this work provides an analytical model encompassing metrics like the number of processors, partitioning, number of objects, etc. Compared to these works: (i) We also first compute the input data partition keys; however, we feed them to the phase 2 partitioner thus freeing up our mappers to do more than just shuffling data around; (ii) Our algorithm does all chores in the mappers of a single M/R job, avoiding costly transfers to reducers and allowing further speed-ups by writing directly to HFiles and bulk-loading them afterwards; and (iii) Our mappers directly affix their subtrees to the appropriate top-tree leaf nodes, rendering the sequential subtree consolidation of [17] obsolete. This makes a huge difference, as storing intervals spanning several subtrees on the subtree roots, then reconstructing upper levels from these populated nodes, proves too time/memory hungry, let alone if done sequentially.

Traverse[20] provides indexing B-tree based mechanisms for MapReduce operations, which can be used later for join operations. The motivation here is similar to ours: to avoid having MapReduce processes loading and processing all the data, even for small-selectivity queries. [21] contribute a

scalable, efficient, distributed B-tree. The main contribution is the efficient maintenance of a B-tree using transactions that minimize the cost and maximize the availability of the B-tree index. Another approach for B-tree based indexing in the cloud is presented in [22]. Compared to these works: (i) Segment trees require computing elementary intervals whereas there is no such need for B-trees; (ii) Segment trees require a top-down population approach; bottom-up tree building/population proves too memory hungry even for m1.xlarge EC2 instances; (iii) [20] use a MapReduce phase for each tree level; this can be too time consuming for very large datasets, even if only 2 levels are built; (iv) The authors show no results for tree building times; (v) Our tree indices are stored in HBase, not on HDFS; we thus harness the lower I/O latency of HBase to provide very low query processing times, while retrieving data directly from HBase without the MapReduce setup/cleanup overhead ([20] ignore this overhead in their performance results); and (vi) We support online updates to our indices, a field not touched by [20] at all; and with our indices built on top of HBase, they are much easier to handle compared to if stored on flat files on HDFS.

Last, [23] address the problem of time-traveling queries over web archives. Their approach is based on creating augmented inverted text indexes (with entries storing timestamped interval information). *Sharding* is used for the augmented text index, along the document id axis. Unlike our work, there is no support (index and querying) for interval data items. Interval queries are time-interval queries and are facilitated only within the context of keyword searching. Moreover, their algorithms are not intended to be used with MapReduce and are not designed to work on cloud key-value stores and no index updates are not addressed.

VII. Conclusions

Clouds and key-value stores are increasingly attracting attention for processing complex queries. A prominent example of such queries, which are useful in several applications, are interval queries. In this work we have addressed the issues associated with indexing and querying intervals over key-value, cloud stores. Starting from the segment tree, as a prototypical interval indexing structure, we first studied how to build it in our environment, yielding the MRST structure. We accompanied the MRST with another index useful for interval queries, coined the *Endpoints Index (EPI)*, which is a new column family index storing the endpoints of the indexed intervals. As segment trees are relatively static and our environment can face heavy insertion loads, we have contributed techniques to maintain our indexes. These utilize an auxiliary structure in a way that, on the one hand, exploits the key-value store's inherent high write throughput, while, on the other, ensures that incoming queries would observe updates, while paying only a negligible cost. We have contributed efficient MapReduce algorithms to build and populate these indexes. We have described algorithms for answering interval queries using our indexes. These algorithms exploit the strengths of each of MRST (i.e., excellent performance for stubbing queries) and EPI (which performs great for the sequential scan part), "routing" subqueries to the most appropriate index tier. Last, we presented extensive experimental results, with real-world datasets, showing that our indexing and query processing algorithms far outperform the naive approaches of a pure HBase/Bigtable-like system, as well as the well-known Hive query processor.

Acknowledgments

This work is supported by the 7th Framework IST programme of the European Union through the focused research project (STREP) on Longitudinal Analytics of Web Archive data (LAWA) under contract no. 258105.

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proc. OSDI*, 2006.

[2] "Apache HBase," http://hbase.apache.org/.

[3] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," in *Proc. VLDB*, 2008.

[4] A. Lakshman and P. Malik, "Cassandra - a decentralized structured storage system," in *Proc. ACM SOSP*, 2007.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. ACM SOSP*, 2007.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. USENIX OSDI*, 2004.

[7] "Apache Hadoop," http://hadoop.apache.org/.

[8] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and parallel DBMSs: Friends or foes?" *Communications of the ACM*, vol. 53, no. 1, 2010.

[9] "Apache Hive," http://hive.apache.org/.

[10] J. Bentley, "Solutions to Klee's rectangle problems," Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1977.

[11] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys*, vol. 31, no. 2, 1999.

[12] A. Kumar, V. J. Tsotras, and C. Faloutsos, "Designing access methods for bitemporal databases," *IEEE Trans. Knowl. Data Eng*, vol. 10, no. 1, 1998.

[13] C. Kolovson and M. Stonebraker, "Segment indexes: Dynamic indexing tech- niques for multi-dimensional interval data," in *Proc. ACM SIGMOD*, 1991.

[14] R. Elmasri, G. Wuu, and Y. Kim, "The time index: An access structure for temporal data," in *Proc. VLDB*, 1990.

[15] V. Kouramajian, I. Kamel, V. Kouramajian, R. El-Masri, and S. Waheed, "The time index+: an incremental access structure for temporal databases," in *Proc. ACM CIKM*, 1994.

[16] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed segment tree: Support of range query and cover query over DHT," in *Proc. IPTPS*, 2006.

[17] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," in *Proc. SSDBM*, 2009.

[18] J. Ballesteros, A. Cary, and N. Rishe, "Leveraging cloud computing in geodatabase management," in *Proc. IEEE GrC*, 2010.

[19] A. Papadopoulos and Y. Manolopoulos, "Parallel bulk-loading of spatial data," *Parallel Comput.*, vol. 29, no. 10, 2003.

[20] H.-C. Yang and S. Parker, "Traverse: Simplified indexing on large map-reduce-merge clusters," in *Proc. DASFAA*, 2009.

[21] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *PVLDB*, vol. 1, no. 1, 2008.

[22] S. Wu, D. Jiang, B. Ooi, and K.-L. Wu, "Efficient B-tree based indexing for cloud data processing," *PVLDB*, vol. 3, no. 1, 2010.

[23] A. Anand, S. Bedathur, K. Berberich, and R. Schenkel, "Temporal index sharding for space-time efficiency in archive search," in *Proc. ACM SIGIR*, 2011.