Gloria: A Batch Friendly Indexing And Storage Framework

Rachid Kachemir
Apple Inc.
1 Infinite Loop
Cupertino 95014, CA, USA
rkachemir@apple.com

Brad Kellett
Apple Inc.
1 Infinite Loop
Cupertino 95014, CA, USA
bck@apple.com

Krishna Behara
Apple Inc.
1 Infinite Loop
Cupertino 95014, CA, USA
kbehara@apple.com

ABSTRACT

Indexing and delivering spatial data to a massive user base composed of over a billion devices around the world stretches the limits of traditional infrastructure and operational tools. For instance, offline bulk indexing and loading fall short of viable solutions when it comes to data at scale; Integration with distributed systems such as Apache Hadoop[®] or Spark[®] is sparse, while data loading is often performed in a sub-optimal fashion by relying on intermediate file formats.

We present in this paper an approach toward a hybrid on-line/offline indexing framework called Gloria that has been running in production settings for the past year at over 350k requests per seconds with lookup latencies under 5 μ s. The resulting output is an in-memory key-value store and we show that by leveraging higher level MapReduce [7] constructs as defined in FlumeJava [5], Gloria can achieve large scale *key-value* offline indexing in a fraction of the time required by traditional datastores while maintaining similar operational performance. Gloria also provides a spatial layer based on improvements to pointer-less quadrees [12] and locational identifiers we call *shift key* that reduces the nearest neighbor problem in spatial data to simple *key-value* lookups. Shift keys have shown to outperform well established solutions such as Google S2 with locational key operations.

CCS Concepts

• **Information systems** $\rightarrow$ *MapReduce-based systems; Record layout alternatives; Geographic information systems;*

Keywords

Offline Indexing, Distributed Bulk Indexing, Spatial Index, Key Value Storage, Locational Identifiers

1. INTRODUCTION

Corporate thirst for data is seeing a tremendous demand that keeps increasing at an ever higher rate. At the same

time, price per unit of storage on commodity hardware has been seeing a significant decrease as suggested in [15], enabling new use cases.

With new volumes of data coming in, comes also the need to design appropriate solutions that traditional databases were not thought for decades ago. The recent years saw the ascension of new database paradigms and technologies including, but not limited to, distributed databases, master-master architecture, distributed transactions, data streaming pipelines and various methodologies for consistent replication. While many solutions are available today in the market, there are still areas where coverage is sparse with few attempts at solving recurring problems.

We found ourselves limited with datastores lacking support for batch indexing of large amounts of data, eventually leading to long ingestion time and high maintenance costs. This limitation was further constrained with no or very little integration with distributed infrastructures provided by the likes of Hadoop/Spark, and sparse literature related to distributed batch indexing, often narrowed to niche cases. LinkedIn's attempt at solving the problem with Voldemort [28] proposes a batch indexing strategy for read-only datasets using Hadoop. While the setup presents great performance, it unfortunately shows a strong coupling to the rest of the Voldemort stack with no intent of turning the read-only datastore to support a read-write scenario without resorting to third party processes.

Our quest to find a fast, batch indexing and Hadoop/Spark friendly datastore lead us to design Gloria, an embedded in-memory key-value datastore that exposes a framework for large scale distributed batch indexing with spatial support. Gloria aims to cover a spectrum of use cases we saw as important to our core indexing logic, including the ability to index billions of keys, low serving latency, low storage overhead and a strong decoupling of the offline indexing process from its serving client. Gloria also presents spatial awareness with an indexing strategy based on pointer-less quadrees [12] enabling a nearest neighbor lookup in a unified fashion for heterogeneous geometries.

The system has been running in production for over a year on different large scale setups with promising metrics:

- 2 Billion keys cumulative
- 500 Billion GET/day
- 150 TB/day delivered to devices
- 0.8-5 μ s GET latency time
- Index build time: 2 to 10 mins

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGSPATIAL'16 October 31 - November 03, 2016, Burlingame, CA, USA

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4589-7/16/10.

DOI: <http://dx.doi.org/10.1145/2996913.2997013>

The goal of this paper is to present the internal architecture of Gloria and show how we accomplished distributed batch indexing with simple, yet efficient methods built upon previous research. The next section will present prior related efforts and will be followed by a description of the design, concepts and support for spatial indexing. We will then present performance numbers from a selection of benchmarks performed under various conditions and finish with a conclusion to this effort. The key contributions brought by this paper are:

- A framework that decouples the indexing process from its serving logic, enabling a scalable offline batch indexing built on top of Hadoop/Spark
- Two storage engines: one favoring read-only data and the other providing the full flexibility of key value datastore for read-write scenario
- Spatial support and a cache-miss layer to handle sparse geo data via Bloom filters

2. RELATED WORK

2.1 Initial Requirements

Given a large corpus of documents, our goal is to design a system able to perform batch indexing with geo-spatial awareness. The system should let us retrieve documents by keys or by locations *via* a nearest neighbor search and additionally provide a decoupling of the indexing infrastructure with the end delivering platform. This indexing problem is well known and has received a particular attention over the years with dedicated systems and approaches. For instance, Silberstein et al. describe a system for batch insertions in [27] using a map/reduce job in Hadoop to perform records insertions. Another approach was suggested by Barbuzzi et al. [3] and consists in building HFiles offline that are later sent to HBase. Arguably, the former solution is heavily involved and not ideal for large batch indexing due to network latencies whereas the latter strengthen the coupling with Hadoop’s ecosystem, which goes against our requirements.

Our investigations presented thereafter led us to look for prior efforts in large scale batch indexing of spatial and non spatial data. We share some of our findings in the next section.

2.2 Prior Work

As mentioned earlier in Section 1, offline batch indexing systems targeting spatial and non spatial data is a topic tackled from multiple angles but with few solutions that intend to minimize coupling with the indexing infrastructure. We however present here systems that provides attempts at solving parts of the problem as a starting point.

A simple solution to perform batch indexing nowadays can be done entirely with Hadoop and very little effort. From the initial Map/Reduce presentation from Dean & Ghemawat in [7], Apache Hadoop [16] has evolved all the way down to provide custom output formats delivering indexes out of the box in a single map/reduce job. The MapFile format provides the possibility to generate a range based dense index to allow for an efficient random access pattern. We found this approach to be unfortunately constraining: the MapFile format operates on a 32 bit addressing space while providing additional coupling to the Hadoop stack. Furthermore, the

range based index strategy is prone to data skew that can significantly affect the payload retrieval latency.

Another similar solution, albeit more evolved, is LinkedIn’s attempt at solving distributed offline indexing with project Voldemort, a distributed key-value datastore [28] modeled after Amazon’s Dynamo [8]. Voldemort aims at providing a datastore with high availability, fault tolerance, load balancing as well as large scale batch indexing using Hadoop. The latter suggests leveraging a set of map/reduce jobs in a similar fashion as described in the previous section. The difference resides in the indexing done over hashed keys using a hash function, MD5, which provides the benefit of adding entropy and distributing keys in a uniform fashion. The resulting output is a dense index of ordered hashed keys which are then memory mapped during start time and accessed via interpolation search [24] for an average $\mathcal{O}(\log \log n)$ access time. With Voldemort, we are getting to a solution that meets our requirements but the drawbacks we found, such as the lack of data partitioning, no plans to batch generate a read-write index and a 32 bits addressing space scheme made it hard to adopt.

On the spatial indexing side, various systems were suggested to provide batch indexing of spatial data in Hadoop. Among these systems, our attention was focused on SpatialHadoop presented by El Dawy et al. in [9] and Hadoop-GIS [1] from Aji et al. Hadoop-GIS acts as an extension of Apache Hive, a layer on top of HDFS that supports data queries in a SQL-like fashion. While far from our requirements, it shows the possibility to address spatial queries via Hadoop using the standard API exposed. SpatialHadoop goes one step further with a deeper integration with Hadoop’s internal architecture, relying on HDFS’ block indexing scheme in three steps: partitioning, local indexing and global indexing. This allows map/reduce jobs to access spatially indexed data during runtime without having to re-compute the index. These approaches are unfortunately too tied to Hadoop’s stack to be used in our production use case, but it shows with a great extent how spatial data can be batch indexed and accessed in a distributed setting.

3. GLORIA

3.1 Design Overview

Figure 1: Gloria Pipeline and Client

Our previous investigation showed that systems combining efforts to provide offline batch indexing for spatial and non spatial data focuses on either one of the problems or provide a solution deeply integrated with the indexing platform, in this case Hadoop. Nonetheless, our findings inspired

us to develop Gloria, an in-memory datastore designed with a strong separation of concerns between the offline indexing platform and the querying clients. We also wanted Gloria to be flexible enough to accept updates during online queries and provide a safeguard against data skew, a naturally occurring artifact of geo-spatial data from dense and sparse regions.

As a framework, Gloria comprises two distinct parts as shown in Figure 1:

- A pipeline that performs the offline indexing
- A client that exposes standard key-value API to retrieve spatial and non spatial documents.

The pipeline lets users generate two type of indexes, with or without Bloom filters, and provides a trade-off between space and latency with a read-only and a read-write data structure. On the one hand, the read-only index, a simple static b-tree, shows a low memory footprint, on the other hand, the read-write index, designed with a chained hash table, provides a latency optimized data structure. The spatial layer comes atop any one of these indexes to provide the spatial awareness component in Gloria. In order to best fit the key-value paradigm, Gloria uses a modified linear quadtree scheme derived from Samet and Gargantini’s efforts in [26, 12] that we further describe in Section 5.

3.2 Physical Layout

Gloria’s file layout approach revolves around using a dense index layout separating the search key space from the associated payload. Since the payload is a plain arbitrary byte sequence, this effectively makes Gloria operate as a blob store where the unit of retrieval is a tuple (index, data). We describe thereafter two flavors exposed in the pipeline: a read-only and a chained hash table layout.

Read Only.

The read-only index layout uses a simple static b-tree structure where the index file each contains a list of lexicographically ordered pairs of (key, offset) and where each key are unique. The offset points to the location of the payload in the associated data file. Looking for a key becomes then a simple binary search or interpolation search when the keys are numerically ordered. The advantage of a read-only layout in Gloria is the space overhead: only 8 to 12 bytes are required for the offset and the payload size. Additionally the key format is user defined and strongly typed providing further information to perform the indexing step which is then passed onto Gloria during index computation and query runtime. Another aspect of the read-only use case is the fast batch indexing time: only a single map/reduce job is needed to perform the indexing. This read-only structure is particularly suited for data with significant churn where batch indexing is faster than individual record updates. On the other hand large static assets can also greatly benefit from this indexing logic. However, when *ad-hoc* updates are required, we recommend using a chained hash table layout described in the next paragraph.

Chained Hash Table.

Documents stored in a datastore may occasionally need updates but still require an initial batch indexing step. For this purpose, Gloria exposes the possibility to generate an

index layout representing a chained hash table structure allowing users to perform updates on individual documents without having to regenerate the entire index. To the best of our knowledge at the time of writing, this approach of batch generating a large hash table offline in a map/reduce context hasn’t been explored before despite the value this can bring. This opens up the possibility to provide a truly hybrid system not only capable of large scale offline batch indexing, but also online indexing for *ad-hoc* updates during query runtime. HBase provides similar online/offline indexing semantic by letting users generate HFiles and perform incremental updates with the caveat of potential compactions and further splits being performed during load time. Our comparison stops here since Gloria’s indexes are readily consumable with no additional logic to be performed at start time.

We investigated multiple hash table resolution strategies and structures before adopting a chained hash table implementation. The main criteria considered in the context of Gloria were: load factor, performance decay, overhead of the structure, read-write latencies, and last but not least, index persistence. The traditional linear and quadratic probing collision resolutions are commonly used in the industry - e.g. Google Sparsehash [20] or Java’s *IdentityHashMap* [22] - due to their well understood properties. Building on top of these schemes, we found current literature with new resolution strategies that showed attractive design properties such as *cuckoo hashing*, *hopscotch* and *hash array mapped trie* (HAMT) respectively presented in [23, 17, 2]. The foundations for each of these novel methods present substantial improvements over traditional hash table design, including, but not limited to, a near ideal load factor for HAMT, low overhead and cache-consciousness design for cuckoo hashing and hopscotch. Given the sophistication level brought in, we found implementation details to be impractical with our requirements such as index persistence and batch indexing friendliness. Cuckoo hashing for instance requires to maintain for each bucket a collision graph (or *Cuckoo Graph*) in order to be able to proceed to write the keys into their associated slots. On the other hand, implementing a linear HAMT in a map/reduce context seemed highly impractical: the multi-branching index as presented in [2] could hardly fit in a map/reduce paradigm.

We opted for a design using a separate chaining hash table for Gloria, or as referred in this paper a chained hash table. Chained hashing presents practical and desirable *for-giving* properties when reaching a high load factor with a predictable average cache miss without incurring an unreasonable latency penalty. In practice, hash tables are subject to the birthday paradox and we can thus determine, given a set of n keys and k buckets the expected number of collisions to be:

$$n - k + k(1 - 1/k)^n$$

and as shown by Gonnet in [13] the expected maximum bucket length when $n = k$ to be:

$$\mathcal{O}(\log n / \log \log n)$$

The next section, Section 3.3, describes the internal structure of Gloria.

3.3 Internal Structure

Many storage engines seek to maintain complex data structures in core memory with an inevitable use of OS based

Figure 2: Gloria Chained Hash Physical Layout

page cache mechanism when growing in size, effectively adding an indirection layer. To overcome this issue, we define with Gloria a dedicated serializable file structure to directly take advantage of the OS caching policy with the following benefits: fast cold startup, contiguous memory span and better cache control with direct OS settings. In this paragraph, we will only focus on the chained hash table data structure description. The read-only case is simple enough: the dense index is composed of an index file containing a tuple (key, offset) ordered lexicographically by keys. The offset points to a location in the data file containing the payload starting with its size and followed with the expected data as a contiguous array of bytes. The foundation for both read-only and chained hash format shows a similar design, composed of:

Header: a structured payload carrying internal information. The header is a fixed byte array of 4KB that contains information such as index type (read-only or chained hash), endianness, file type (data or index) and other useful meta-data describing the overall artifacts. The header is added to every files that comprises Gloria’s artifacts.

Figure 3: Header Format

Index File: the serialized index used by Gloria to retrieve documents.

Data File: file storing the payload in an append-only fashion.

Index Structure.

The index file in the chained hash store as described in Figure 2 contains information related to the offset position of the *last* inserted record in the collision chain. Each bucket being of a fixed size of $8 + 1$ bytes, the offset location is directly accessible with the standard:

$$\text{mod}(\text{hash}(\text{key}), n) * \text{bucketSize}$$

where n represents the number of buckets. An extra byte prefixes each bucket for book-keeping purpose. It is worth

noting at this stage, that the index is unaware of the structure of the data file: the intention is to be able to isolate both data structures with low coupling, leaving enough flexibility for future implementation strategies.

Append-Only Layout.

The append-only file layout is a temporally ordered and structured file for which new records are added at the end of the file. This design organization exempts Gloria from using an additional *Write Ahead Log* (WAL) file to ensure atomicity and persistence semantics. The append-only file records both the data as well as the actions last performed in a linear fashion. This format was designed with a multi version concurrency control (MVCC) policy in mind ¹. In comparison, Voldemort’s [28] read-only storage engine provides a similar attempt at defining a structured index for bulk indexing but is not targeting the writable use case: new data that needs to be added to the datastore will have to wait for the next bulk indexing cycle.

The following section provides a description of the bulk indexing process and shows how we leverage Hadoop and Apache Crunch to orchestrate our data pipeline to generate and organize Gloria’s assets.

4. BATCH OFFLINE INDEXING

4.1 High Level Constructs

The offline indexing process in Gloria is built on top of Hadoop which provides a well understood and industry proof distributed compute and storage paradigm over map/reduce with hardware failure resiliency. The data orchestration layer organizing each step of the offline indexing process is done with Apache Crunch [6], a library implemented from FlumeJava [5] that provides high level data abstraction operation over collections such as *join*, *groupBy*, *map*, *sample*, *filter*, etc. A very simple, yet fundamental core idea behind Gloria lies in the concept of *transformers*, a straightforward block of logic designed to break an input record R into a tuple (key, value). This idea is powerful enough to fit spatial

¹this will be the subject of an additional publication describing spatial data transactions in Gloria

records in: as we will see later in Section 5, our spatial index uses the same (key, value) data model. The transformer plays a crucial role of interfacing the user with the pipeline logic with three different variations:

- **KeyValueStreamTransformer** A transformer providing the necessary logic to break an initial input into a (key, value) stream. The data issued from this transformer is immediately processed and flushed by the pipeline in a streaming fashion.
- **KeyValueTransformer** Similar to the stream transformer but collects the data into an intermediate buffer that is then processed by the pipeline.
- **SpatialKeyValueTransformer** is similar to KeyValueStreamTransformer with the additional logic of exposing a locational identifier, the tile ids, as defined in Section 5 and the associated payload, the tile content.

The user may chose to implement one these transformers in order to generate an index. Both read-only and chained-hash pipeline will use the same transformation logic giving the possibility for the user to alternatively use one or the other layout with no, or very little, code changes.

Shards.

More often than not, partitioning is beneficial when dealing with distributed tasks, especially when it comes to workload balancing. In Gloria, a partitioned dataset is called a shard and constitute a *query-able* unit composed of a tuple (index, data) with an optional accompanying Bloom filter. *Sharding* happens by hashing each key with a universal hash function dispatching the tuple (key, value) to a specific reducer for serialization and guaranteeing at the same time a balanced number of records per shards, thanks to the uniform distribution property of universal hash functions used in Gloria. Furthermore, sharding also provides the benefit of smoothing hotspots: keys are uniformly assigned to a shard with no consideration with regards to their payload, making a dense spatial areas (e.g. New York City, London, Paris) span across different shards. When querying Gloria's artifacts, the client uses the same hash function to find the associated shard and return its payload.

4.2 Indexing Pipeline

Read Only Pipeline.

The read-only storage engine was designed to take care of high volume of data that are either static or for which the churn rate is too high to consider an incremental update approach. It optimizes the storage space by minimizing the datastore bookkeeping footprint and requiring records to be *typed* via a record definition *à-la* POJO. By implementing these classes, the user gets the additional benefit of relying on a known pseudo-schema that can be shared through the application stack and help formalizing a description of the natural key for each record. The intention behind this abstraction level is to simplify the index generation process by hiding any internal details related to the structure or lower level pieces such as mapper, reducer, emitter, etc. These objects can be passed to the pipeline driver *via* its option settings which will then be injected throughout the flow as described in Figure 4. The read-only index pipeline flow can

Figure 4: Gloria Pipeline Flow

also be seen through the intermediate output of the main steps as described below:

0	[INPUT] => [(key, value)]
1	=> (shuffle/partition/sort)
2	=> [(bucket, (key, value))]
3	=> [{bucket, [(key, value)]}]

- Step (0), ExtractKeyValue: Records are individually streamed from the input collection to the transformer T, resulting in a tuple (key, value).
- Step (1,2), Partition: Once the records are broken down, Hadoop/Spark will perform a standard shuffle/partition/sort phase, lexicographically ordering the previous tuples by keys. The partitioner is configured to assign each key to a given reducer through hashing. Each reducer becomes effectively responsible for

processing a shard.

- Step(3), GroupByKey: The records are provided to each reducer as an ordered collection. The purpose of the groupBy operation in the context of the read-only pipeline is to make sure the appropriate action is to be taken if duplicate keys are detected. Gloria then takes care of formatting both index and data payloads.

Chained Hash Pipeline.

The chained hash pipeline uses a similar flow as the one presented previously, albeit adding few more steps to take care of the bucketing logic that governs a general purpose hash table. One major step consists in computing buckets on a per-shard basis, which requires counting the numbers of records for each shard. The flow can be simplified with the following intermediate output:

```

0  INPUT => [(key, value)]
1          | => [(shardId, (key, value))]
2          | => [(shardId, COUNT)]
3          => [(shardId, bucketId), (key, value)]
4          => (shuffle/sort/partition)
5          => [(shardId, bucketId), [(key, value)]]

```

- Step (0) ExtractKeyValue: Similar to the one described in the read-only pipeline, with the purpose of breaking records into (key, value)
- Step (1,2) CountPerShard: These steps gathers the count of records per shards by invoking the partitioner's hashing logic.
- Step(3) HashBucket: Knowing the count of records for a given shard, we are now able to compute a bucket from the provided key and shard id.
- Step(4): Hadoop/Spark steps shuffling, sorting and partitioning the data with the provided partitioning logic
- Step(5) GroupByKey: The keys are grouped: the grouping is performed on the (shardId, bucketId) tuple as opposed to the key itself as done in the read-only pipeline. This step performs the collision resolution and serializes the data following Gloria's format.

The following diagram depicts the end layout produced from the pipeline and a set of expected operations during query time.

4.3 Fail Fast: Bloom Filters

In order to mitigate cache misses, Gloria provides the possibility to generate an accompanying Bloom filter for each shard during offline indexing. A Bloom filter is a space-efficient randomized data structure suitable for membership testing. Original Bloom filters as described by Bloom in [4] were not designed to support deletion, but flavored versions were proposed to overcome this issue. Gloria uses a particular Bloom filter implementation called Dynamic Counting Bloom Filter (*DCBF*) based on [10, 14].

Bloom filters in Gloria are configured and generated within the pipeline in a stream-fashion over a given shard key space. The data structure is maintained in memory during the last step of index formatting in Gloria with no added costs to the

Figure 5: Gloria High Level Layout

pipeline. The configuration is done dynamically to parameterize each filter with an optimal number of bits and hashes in order to match the expected false positive probability p provided by the user, also approximated with:

$$p \approx \left(1 - e^{-kn/m}\right)^k \quad (1)$$

where k is the number of hashes, m is the number of bits, and n represents the number of items in the Bloom filter. Given p and n for each shard, we may infer both k and m from the derivative of equation 1 - with a more formal description provided in [21] - such that:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

$$k = \frac{m}{n} \ln 2$$

5. SPATIAL INDEXING

5.1 Quadtree, Shift Key & Shift Offset

A quadtree is a hierarchical data structure first described by Finkel and Bentley in [11] which has since then lead to multiple versions as listed by Samet in [26]. Its construction is obtained with a regular disjointed space decomposition from recursive splits into cells, also known as quadrants; each quadrant is identified with a Morton code (or Z-order code). In this section, the depth of decomposition into sub-quadrant is alternatively called zoom level or granularity.

Gloria's spatial layer uses a quadtree based approach for geo-spatial indexing. More specifically, it leverages a pointerless linear quadtree to fit its key-value paradigm and preserve the data structure's hierarchical properties. Each node in the quadtree is uniquely identified with a locational code derived from its Morton sequence. This locational code, also denoted as quadkey in this article, can be seen as a base 4 representation for which each number in the sequence represents a child identifier in the hierarchy and encoded using two bits (NW=00_b=0, NE=01_b=1, SW=10_b=2, and SE=11_b=3). The length of the code represents its depth within the tree - the *zoom level*.

Gloria exposes utility methods to convert a (latitude, longitude, zoom) tuple into a fixed length quadkey encoding using a 64 bits integer. However, as mentioned by Hjaltason and Samet in [18], a problem arises with keys colliding at different zoom level: 000000_b and 00_b, which are keys respectively defined for level 3 and 1, will both translate to 0,

going against the notion of unicity of locational identifiers. This problem stems from the fact that the zoom level is not part of the key encoding when using a 64 bits integer.

Hjaltason suggests to store the tuple (key, level) in order to palliate the problem. Other methods have been employed in industrial settings to overcome this issue: Google S2, Google’s spherical geometry library, defines a key space over 64 bits integers for quadkeys, also called *cellid*, reserving 3 bits to identify the face for a cube projection and 1 bit to determine the level of a cell id. Finding the level of a *cellid* consists then in finding the most significant bit position that defines the zoom level for a particular key. While this method shows great performance in practice, we can argue that the definition of the keys as such, yields a non contiguous key space and additionally requires storing keys in big endian in order to leverage lower level instruction sets. RocksDB, Facebook’s key-value datastore, exposes a spatial layer using quadtrees that relies on *namespacing* through column families to isolate each zoom level and prevent key collisions. This isolation can unfortunately end with a potentially unbalanced set of column families both on the storage and the access pattern.

The method we present below is intended to provide a compact fixed encoding of locational identifiers in quadtrees, with a guaranteed unicity across all nodes and zoom levels ranging from 0 to 31. This identifier, we call shift key, requires a minimal amount of computation while maintaining all the properties of quadtrees, allows a constant time conversion quadkeys to shift keys and directly embeds the zoom level information for each key. In order to achieve this, we define the concept of *shift offset*, an offset used to position shift keys with regards to previously identified nodes in their hierarchy. Shift offsets are defined with respect to their zoom level z in the range $[0..31]$ such that:

$$offset(z) = \begin{cases} \sum_{i=1}^{i=z-1} 4^i, & \text{if } z \geq 1 \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

which can be simplified to:

$$offset(z) = \frac{4 * (4^z - 1)}{3} \quad (3)$$

Shift offsets being defined for a bounded zoom level range, we can easily cache the results in a standard array addressable by zoom level. We reuse the notion of shift offsets thereafter to define a shift key K , such that for any (lat, lng) defined in the WGS84 datum, and zoom level z in the range $[0..31]$, we have:

$$\underbrace{K(lat, lng, z)}_{\text{shift Key}} = \underbrace{offset(z)}_{\text{shift offset}} + \underbrace{QK_{10}(lat, lng, z)}_{\text{quadkey base 10}} \quad (4)$$

This method enables the definition of a unique locational identifier over a 64 bits key space, covering every area on earth with a 4 cm^2 precision. The following section will show how common operations can be performed with simple bit-wise operators in constant time.

5.2 Key Operations

We show herein a subset of common spatial methods performed in constant time, namely: *Ancestor*, *Children* and

Contains which will be additionally discussed in the Performance section with comparable solutions. For the remaining part, we may assume a lookup table representing the shift offsets as described in equation (2).

Ancestor: The ancestor of a shift key represents the locational identifier of the immediate parent node in the quadtree hierarchy and obtained as such:

Algorithm 1 Ancestor Shift Key

```
function ANCESTOR(shiftKey, depth)
    normalizedKey ← (shiftKey - offset(depth))
    return normalizedKey >> (depth * 2)
end function
```

Children: Similarly defined, the *Children* method returns the four immediate child of a node in the quadtree hierarchy such that:

Algorithm 2 All Four Children Of A Shift Key

```
function CHILDREN(shiftKey)
    base ← (shiftKey << 2) + offset(1)
    return [base, base + 1, base + 2, base + 3]
end function
```

Contains: The Contains operation describes a containment relation between two keys for which a key $s1$ is said to contain another key $s2$ if $s1$ is an ancestor of $s2$. This operation can be performed in near constant time by leveraging the structure of a shift key: the number of leading zeroes in the binary representation of a shift key can be used to determine its zoom level, which can be further optimized at the CPU level to leverage instruction sets such as *CLZ*, *BSF* or *BSR* available on various CPU architectures. The difference of level in between $s1$ and $s2$ will give us information to compute the ancestor of $s2$. We may assume the *level()* method provided to describe the Contains logic thereafter:

Algorithm 3 Shift Key S1 Contains S2

```
function CONTAINS(s1, s2)
    if s1 == s2 then
        return True
    end if
    level1 ← level(s1)
    level2 ← level(s2)
    if level1 >= level2 then
        return False
    end if
    delta ← level2 - level1;
    ancestor ← ANCESTOR(s2, delta)
    if ancestor == s1 then
        return True
    end if
    return False
end function
```

5.3 Spatial Indexing

With shift keys, we have the possibility to uniquely identify areas in the world with a desired granularity. In this section we will consider three different cases of indexing: point, line and polygon indexing.

Point Indexing.

Indexing a collection of points consists in defining the locational key for each point and handling multiple points falling within the same tile. This operation is straightforward and can be performed in a single indexing pass.

Fetching the nearest points given a location and a radius boils down to a nearest neighbor search: we first compute the locational identifier for the location tuple (lat, lng, zoom) and then infer the neighboring tiles that encompasses the radius. Further description can be found in [25] (Chapter 6 listing LQ-POINTQUERY).

Line & Shape Indexing.

For lines and shapes, the problem of offline indexing and retrieval requires additional steps to ensure a proper granularity level. As such, we define an interval of zoom levels where the search space is valid which allows us to confine the search scope and prevent a harmful tile explosion if we were to index at a high zoom level. The offline indexing proceeds with the following steps:

- **Case 1** If a given shape can fit within a single tile within the defined zoom interval, the object is then indexed based on that tile, ending the indexing procedure for the record
- **Case 2** If no single tile was found fitting the object, we proceed with a shape decomposition to find the set of tiles that can best fit the shape/line within the zoom level boundary

For the latter case, we assume a parametric constraint to ensure a proper tessellation: the decomposition is performed in a bounded zoom level range, and a coverage of 80% is used to determine whether or not an additional split needs to be performed. Samet suggests a similar approach in [26] with the *split and merge* procedure from Horowitz [19]. For line indexing, the tiling logic to transform a line segment into its associated quadkeys is roughly the same: instead of using a coverage relationship in between a tile and a shape, we resolve to computing the containment of a line segment within a tile.

The nearest neighbors search for both lines and shapes is similar to the one described for points with the exception of repeating the operations for the predefined zoom level range used for shape decomposition. To alleviate cache misses problem that may arise, we suggest in addition to using reasonable parameters, the usage of Bloom filters as mentioned in section 4.3.

Spatial Integration.

The spatial integration with Gloria is done *via* its *KeyValueSpatialTransformer*: the transformer’s interface exposes methods to extract and/or decompose records into shift keys. The transformer is then injected in the pipeline flow initial *ExtractKeyValue()* phase, after which the indexing path is identical to a standard key value indexing. The spatial data can then leverage a read-only or read-write storage strategy as described in Figure 4 to fit the target application needs.

6. PERFORMANCE

We propose in this section a comparative evaluation focusing on three parts to assess the overall system. We evaluate herein the performance of shift keys, the data build time and

finally the systems’ lookup throughput. All tests were run on a 64 bit platform composed of 32 CPUs, 192GB of RAM and common SSD disks for persistence. The Hadoop cluster used to generate each index for Gloria, is running a similar configuration using standard spinning disks instead of SSDs, and running Hadoop 2.5.0. The cluster is composed of 14 nodes, each with the same configuration.

Shift Keys.

We found during our benchmark setup that very few libraries attempt to provide locational identifiers as a part of their logic. In fact, since Gargantini’s proposal in [12], little attention has been dedicated to the subject. For instance, RocksDB, the key-value datastore from Facebook, is relying on the same method to compute their locational identifiers. We decided to benchmark our work against Gargantini’s and Google S2 implementation which seemed, at the time of writing, the most commonly used solutions. It is worth noting that Google’s locational identifier targets the definition of equi-area *tiles* across the world, hence requiring a substantial computational resource usage. Our benchmark, run over multiple common operations, shows that Gloria’s quadkey consistently outperforms both methods as shown in Figure 6 over a dataset of 50 million distinct points.

Data Build & Lookup Time .

A key feature in Gloria, is to provide distributed batch indexing with a significant gain over traditional datastores without compromising on operational performances. We tested ourselves against solutions commonly used such as RocksDB, LevelDB and MapDB. Our tests highlighted a large latency overhead with Gloria for small datasets containing less than 100M keys. This overhead comes from Hadoop’s scheduling job orchestration, making small datasets unsuitable for Gloria. Starting at 100M records, we see a comparable indexing time with other solutions. At 1B records, Gloria’s data pipeline becomes significantly faster than other datastores: for such a dataset Gloria takes on average 4 minutes to process 1B records spread over 115GB whereas RocksDB was taking on average 4 hours. On the operational side, Gloria outperformed all other datastores compared against it: the benchmark setup to issue random lookups operations (GET) in throughput mode also highlights the logarithmic decay of the performances with regards to the dataset size. While the performance were highly encouraging, it is worth noting that we found a significant discrepancies in our benchmarks when dealing with RocksDB as we could not match the advertised performance of 30 mins indexing for 1 Billion keys. We are still investigating this issue and decided to also remove the lookup latencies that were not matching those numbers.

7. CONCLUSIONS

We presented in this paper a low-latency bulk indexing framework with spatial awareness. By leveraging distributed compute and storage units we are able to perform batch indexing in a fraction of the time that would be required by using common key-value stores, while also bridging the gap in between offline and online indexing. We also showed that by deferring the index build to a distributed infrastructure, we are also able to maintain an operational performance on

Figure 6: Gloria Shift Key vs. Google S2 vs. Gargantini

par with other systems. Making spatial awareness a prime citizen of our framework enables a fast, common spatial indexing pipeline across various entities and services. The framework has been running successfully at scale over the past year with promising results. We will be investigating further index layout structures to accommodate different access patterns.

8. ACKNOWLEDGMENTS

We would like to warmly thank Sylvain Pion, Menelaos Karavelas, Ashok Harikumar, Ahmed Radwan, Fadi Obeid, Nick Puz, Bernard Gallet and Marco Primi for their valuable time, suggestions and expertise. Their help and guidance was key to this project and accomplishments.

9. REFERENCES

- [1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.
- [2] P. Bagwell. Ideal hash trees. Technical report, 2001.
- [3] A. Barbuzzi, P. Michiardi, E. Biersack, and G. Boggia. Parallel bulk insertion for large-scale analytics applications. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, pages 27–31. ACM, 2010.
- [4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [5] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. Flumejava: Easy, efficient data-parallel pipelines. pages 363–375, 2010. URL: <http://dl.acm.org/citation.cfm?id=1806638>.
- [6] A. Crunch. <https://crunch.apache.org>, 2016. [online, accessed 20-April-2016].
- [7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1251254.1251264>.
- [8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.
- [9] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1352–1363. IEEE, 2015.
- [10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder.

Figure 7: Lookup & build time throughput comparing Gloria RW and RO, LevelDB, RocksDB and MapDB

- Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.
- [11] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.
- [12] I. Gargantini. An effective way to represent quadrees. *Commun. ACM*, 25(12):905–910, Dec. 1982. URL: <http://doi.acm.org/10.1145/358728.358741>, doi:10.1145/358728.358741.
- [13] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, Apr. 1981. URL: <http://doi.acm.org/10.1145/322248.322254>, doi:10.1145/322248.322254.
- [14] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The dynamic bloom filters. *Knowledge and Data Engineering, IEEE Transactions on*, 22(1):120–133, 2010.
- [15] P. Gupta, A. Wildani, E. L. Miller, D. Rosenthal, I. F. Adams, C. Strong, and A. Hospodor. An economic perspective of disk vs. flash media in archival storage. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 249–254. IEEE, 2014.
- [16] A. Hadoop. <https://hadoop.apache.org>, 2016. [online, accessed 20-April-2016].
- [17] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Distributed Computing*, pages 350–364. Springer, 2008.
- [18] G. R. Hjaltason and H. Samet. Speeding up construction of pmr quadtree-based spatial indexes. *The VLDB Journal—The International Journal on Very Large Data Bases*, 11(2):109–137, 2002.
- [19] S. L. Horowitz and T. Pavlidis. Picture segmentation by a tree traversal algorithm. *Journal of the ACM (JACM)*, 23(2):368–388, 1976.
- [20] G. Inc. <https://github.com/sparsehash/sparsehash>, 2016. [online, accessed 20-April-2016].
- [21] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
- [22] Oracle. <https://docs.oracle.com/javase/7/docs/api/java/util/IdentityHashMap.html>, 2016. [online, accessed 20-April-2016].
- [23] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
- [24] Y. Perl, A. Itai, and H. Avni. Interpolation search—a log log n search. *Communications of the ACM*, 21(7):550–553, 1978.
- [25] P. Rigaux, M. Scholl, and A. Voisard. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.
- [26] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.
- [27] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 765–778. ACM, 2008.
- [28] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.