White Paper

Pilosa: A Technical Overview

December 2017

Introduction

Pilosa is a standalone index for big data. Its goal is to help big data storage solutions support real time, complex queries without resorting to pre-computation or approximation. Pilosa achieves this goal by implementing a distributed bitmap index which provides a compact representation not of the data itself, but of the relationships present in the data. By encoding only relationships in an efficient manner, Pilosa is able to operate in-memory using low-level machine instructions to process queries at unparalleled speed on commodity hardware. The remainder of this paper gives a technical overview of Pilosa covering the data model, clustering strategy, typical usage, and advanced features.

Binary Matrices

The core data abstraction in Pilosa is the binary matrix. Each matrix is centered around some "items of interest" which are represented by the *columns* of the matrix. For example, the items of interest could be people, web servers, taxi rides, etc. Each *row* in the matrix represents something with which a column might have some relation. While all of the columns generally represent the same type of item, the rows typically represent many different types of things. For example, in the case where the items of interest are people, the rows could represent foods which people like (or don't), ages, links clicked, or even other people with whom they are related. In a given matrix (or *index* in Pilosa parlance), all of these different types of rows might be present.

Roaring Bitmaps

Each row in an index is stored in a format called

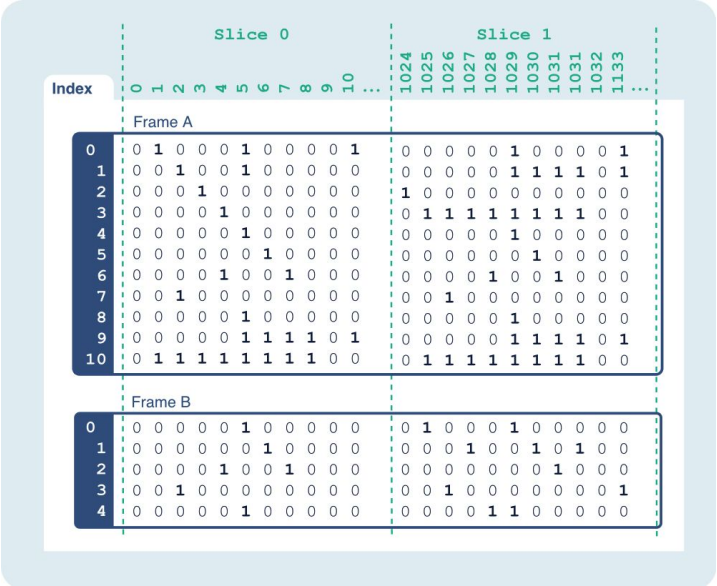

Figure 1: Illustration of the Pilosa data model, a binary index, showcasing groupings of columns called "frames."

Roaring,[1] which both compresses the bitmaps and allows for efficient computation on them. Roaring can compress bitmaps in three different ways, and every basic operation supported on the bitmaps is implemented separately for each pair of compression schemes. Because of this, queries are performed without first decompressing the data, saving both CPU and memory.

Roaring Compression

Roaring stores each bitmap in one of three ways. Two metrics of a bitmap determine the compression method: a count of the set bits in the bitmap, or *cardinality*, and a measure of bit grouping in the bitmap, or *run count*. Based on these values, the storage format is chosen—to minimize size— with either uncompressed bitmap encoding, array encoding, or run-length encoding.

[1] Roaring Bitmaps. http://roaringbitmap.org/.

Uncompressed bitmap encoding stores each bit directly. It is often used for dense bitmaps representing unsorted data. Array encoding stores the locations of set bits, making it appropriate for sparse data. Run-length encoding stores contiguous groups of set bits, and is most beneficial for sorted data. This approach achieves significant compression relative to the uncompressed format, while allowing random bit access in all three storage formats.

Roaring Operations

Roaring supports the unary *count* operation and the binary operations *intersect*, *union*, *difference*, and *XOR*. These bitwise operations are possible on a bitmap, or pair of bitmaps, regardless of their storage format, even when those storage formats are not the same. The compression-aware algorithms for these operations are fast, not only because they avoid decompression, but because the storage format can directly reduce the amount of computation required. For example, when computing the union of two run-length encoded bitmaps, only the endpoints of the contiguous intervals of set bits need to be examined. Every combination of operation and storage format must be accounted for, gaining a significant speed and storage improvement in exchange for a one-time effort in software complexity.

Pilosa Roaring

Although several reference implementations of Roaring Bitmaps exist, support for high-cardinality data in Pilosa necessitated a new library. Pilosa Roaring is based on larger 64-bit integers, practically eliminating the upper limit on the number of columns. There is an official specification for the Roaring storage format, which the Pilosa library follows in spirit, but exact compatibility is not possible.

Logical Operations

The most basic query in Pilosa (and the basis for most of the more advanced queries) is the so-called segmentation query. Given some arbitrary boolean combination of rows (e.g. Row 1 and Row 3, but not Row 7), return the set of columns who have set bits in those rows such that they satisfy the boolean statement. Pilosa executes these queries by performing bitwise logic on the rows - the result of each logical operation on two rows is another bitmap which can be used for the next stage of the computation. These operations can be executed independently, and in parallel by nodes in the cluster.

Frames and TopN

Rows in Pilosa indexes may be grouped into categories called frames. Frames may have various ranking strategies associated with them, where each ranking keeps an ordered list of the rows in the frame based on the strategy. (The most common strategy is simply the count of set bits in the row.) These ranked lists are accessed with "TopN" queries. While segmentation queries return lists of columns, TopN queries return lists of rows. A TopN query can actually take any segmentation query as an argument and will return the ranked set of rows based only on the columns which satisfy the segmentation query.

BSI

Since each row/column pair in Pilosa can encode a relationship as only a single bit, one might wonder how Pilosa could efficiently store scalar numeric data like mass, duration, speed, etc. which can have many possible values. There are a number of possible strategies available, but the most flexible method that Pilosa supports is a technique called bit-sliced indexing (BSI). With BSI, an integer value is encoded using several rows in Pilosa in binary format. For example, a 32-bit number (up to 4 billion or so) could be encoded using 32 Pilosa rows. With some clever query generation, Pilosa can support queries over arbitrary ranges of a value. These queries simply return a bitmap (the set of columns which contain a value in the given range), and this result can be combined with other segmentation and TopN queries. BSI gives Pilosa enormous power in storing data compactly and supporting complex queries.

Bit-sliced Index Concepts

In order to represent scalar data using BSI, the value being encoded is broken up into different components based on a specific scheme. For example, one might decide to encode integer values ranging from 0 to 999 as a base-10, three-component bit-sliced index. This entails representing each component of a value—the ones, tens, and hundreds digits—as a set of 10 bitmaps representing the possible values 0 to 9. If a value to be encoded is 392, then a bit would be set in bitmap 2 for component-0, bitmap 9 for component-1, and bitmap 3 for component-2. This encoding scheme would require 30 bitmaps. Depending on the values being encoded, it may be more efficient to use a different BSI encoding scheme. For example, the same range of integer values from 0 to 999 can be encoded using a base-2, ten-component bit-sliced index. In this case, the example base-10 value 392 would become the base-2 value 0110001000. Here, a bit would be set in bitmap 1 for components 4, 8, and 9. This encoding scheme would require only 20 bitmaps.

Range-Encoding Concepts

In order to support range and aggregate queries, it's necessary to apply range-encoding methods to bitmaps. This entails setting bits per component as described in the bit-sliced index concepts above, but in addition to setting the bit corresponding to the component's value, all bit values greater than the bit are also set. In the previous example using the base-10 value 392, where bit 3 is set in component 2, the range-encoded representation would require setting bit 3 as well as setting bits 4 through 9. This encoding method supports range queries; for example, one could easily query for all values greater than 300.

	component-2										component-1										component-0									
	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
392	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
417	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
56	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
897	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Figure 2: *Example of a base-10, three-component bit-sliced index*

	c-9	c-8	c-7	c-6	c-5	c-4	c-3	c-2	c-1	c-0
	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0
(392) 0110001000	0 1	1 0	1 0	0 1	0 1	0 1	1 0	0 1	0 1	0 1
(417) 0110100001	0 1	1 0	1 0	0 1	1 0	0 1	0 1	0 1	0 1	1 0
(56) 0000111000	0 1	0 1	0 1	0 1	1 0	1 0	1 0	0 1	0 1	0 1
(897) 1110000001	1 0	1 0	1 0	0 1	0 1	0 1	0 1	0 1	0 1	1 0

Figure 3: *Example of a base-2, ten-component bit-sliced index*

An interesting aspect of a range-encoded, bit-sliced index is that the most significant bit of each component is always set to 1; in the base-10 example, bitmap 9 is always set to 1. Because of this, the most significant bit doesn't need to be stored. In the case of a base-10, three-component bit-sliced index, only 27 bitmaps would be required to range-encode values from 0 to 999. Applying this to a base-2 encoding bit-sliced index, the most significant bit (which is 1) will always be set to 1, so it can be omitted. This means that range-encoding values using a base-2, ten-component scheme will only require 10 bitmaps.

Range-Encoded Bit-sliced Indexes

Pilosa uses a base-2, range-encoded bit-sliced index strategy to encode scalar values. Integer values are stored in fields which have been configured with a minimum and maximum value range. This possible range determines the number of bitmaps that Pilosa uses internally to represent values in the field. For example, if a field is configured to accept integers in the range from -10 to 110, Pilosa will determine that seven bitmaps can represent all possible values within the range. In addition to the seven bitmaps representing the values, Pilosa uses an additional bitmap to indicate "not null" values. For every column containing a value, the "not null" bit is set. This additional bitmap allows Pilosa to distinguish between a column having a value of 0 and one that has no value at all, and it also provides some performance enhancements by allowing Pilosa to bypass certain computations depending on values in the "not null" bitmap.

Per-Bit Time Quantums

While one could use BSI to associate a very precise timestamp, or indeed, multiple timestamps with each column in Pilosa, it is also possible to associate a coarse time value with any single bit in Pilosa. Upon creating a frame, one may specify a supported time granularity as coarse as a year, or as granular as an hour. Said another way, the timestamps associated with bits in a given frame can have year, month, day, or hour precision. The reason for this restriction lies in the carefully efficient way that Pilosa handles timestamp support. Instead of naively writing a timestamp alongside the set bit, or in a separate key-value store, Pilosa actually maintains multiple bit matrices behind the scenes. If a bit is set with the time January 12, 2017 at 16:00, Pilosa will actually set 5 bits in 5 different matrices. The first is the main bit matrix where all set bits are stored regardless of whether they have an associated timestamp. The second is for the year "2017", the third is for the month in the year "January 2017", the fourth is for the day "January 12, 2017", and the last is for the full time, down to the hour. By structuring per-bit timestamp support in this way, Pilosa can support efficient queries over arbitrary time ranges. For example if there were a query for the time range from January 1st of 2016 to February 28th of 2017, Pilosa would only need to run the query in three bit matrices, the "2016" matrix, the "January 2017" matrix, and the "February 2017" matrix. These time range queries return bitmaps like other queries in Pilosa, so they can naturally be combined with boolean logic, and used as filters in TopN queries.

Software Architecture

Pilosa is written in the Go programming language. The functionality described in this section is in the top level "pilosa" package in the repository at github.com/pilosa/pilosa. There are several sub-package which contain important functionality, such as "roaring" (Pilosa's implementation of roaring bitmaps) and "pql" (the parser for the query language), but these are not discussed in detail here.

Data

This subsection will discuss the parts of Pilosa's software which implement its data model.

At the top level in Pilosa is a structure called a *holder,* which holds references to all the indexes that Pilosa is managing. Each *index* represents a separate binary matrix which is not necessarily related to any other *index* in Pilosa. Column attributes are managed at the *index* level since

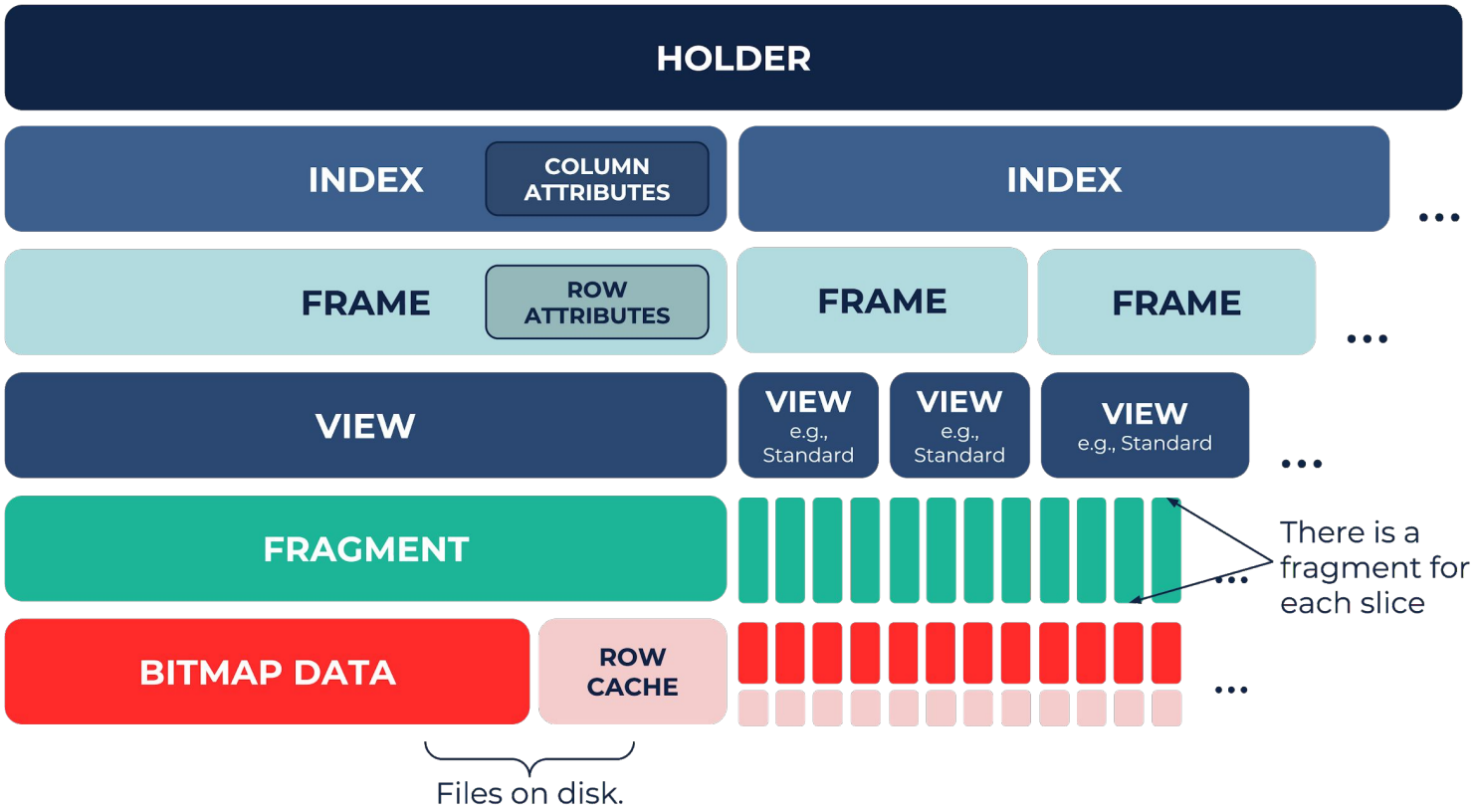

Figure 4: *Visualization of Pilosa data architecture*

columns are shared among all *frames*. Each *index* has one or more *frames*, each of which holds some of the *index's* rows. *Frames* are user created, and an *index* might only have one *frame*, or it might have hundreds of *frames*, depending on a user's needs. The TopN section, below, describes more about the motivation for categorizing rows into different *frames*. Row attributes are managed at the *frame* level since rows are not further partitioned beyond *frames*. Each *frame* may have one or more *views*. The most common reason for a *frame* to have more than one view is if it is *inverseEnabled*. *Frames* which are configured to be *inverseEnabled* store the matrix both in its standard format and in the inverted format, which allows efficient queries on columns. A **frame**'s BSI *fields* are also stored as separate *views* under that *frame*.

Below the *view* is the *fragment* - *views* are broken up into *fragments* based on *slices*. A *slice* is a contiguous group of columns (2^{20} by default), described in further detail in following sections. Each *fragment* corresponds to two files stored on disk. One file contains the serialized bitmap data for the *fragment* consisting of all rows in the *fragment's frame* by all columns in the *fragment's slice*. The other file contains the row rank cache for the *fragment*, which is an in-order list of the rows in the *slice* with the ranking score for each row. There are different possible ranking types, but the most common is based on the number of set bits in the row.

Logic

This subsection will discuss the parts of Pilosa's software which are not directly related to the data model.

Pilosa contains a *server* structure which manages the various routines that comprise PIlosa's running state. These include an HTTP handler for Pilosa's main API, a number of internal communication

handlers, and several background monitoring tasks.

The HTTP handler has direct access to the *holder*, and can therefore answer requests about what indexes and frames are available, create new indexes, frames, and fields, etc. One can access the full API documentation to see what endpoints are implemented.

When the *handler* receives a PQL query, it uses the "pql" subpackage to parse it, and then passes the parsed structure along to the *executor*. Each PQL call in a query is processed serially. The *executor* behaves somewhat differently, based on which call it is processing, but generally it processes the call for all relevant slices on the local node, and concurrently issues requests to process the call for slices which reside on remote nodes in the cluster. Once all local and remote processing is finished, it performs any aggregation or reduction work which is required and returns the results back to the *handler*. The coordination of remote and local execution is handled by a lightweight map/reduce framework inside Pilosa.

The *server* also runs a number of background processes to keep data in sync across the cluster, send optional telemetry data back to Pilosa HQ, and collect various metrics.

Clustering

Pilosa runs as a cluster of one or more nodes - there is no designated master node. All nodes run the same standalone Pilosa binary which has no external dependencies.

Slices

Pilosa performs parallelized query execution over all cores of all nodes in a cluster. The data unit of parallelism is called a *slice*. A slice is a group of 1,048,576 (2^{20}) columns and all rows within an index. The data is sliced vertically in this way so that the processing of segmentation queries can occur independently on each slice. Each slice is wholly contained on a single node (though it may have replicas on other nodes), and a node may contain multiple slices.

Replication

Data within Pilosa is replicated to multiple nodes in the cluster based on the replication factor specified in the configuration. In a five-node cluster made up of nodes 1 to 5, with a replication factor of two, if data is determined—based on a hashing algorithm—to be on node 3, a copy of that data will also reside on node 4. Under normal conditions, read queries will always retrieve the data from node 3. In the case where node-3 is unavailable, the query executor will continue trying replicas (node 4 in this example) until the data is successfully retrieved. If no replicas are available, an error is returned.

Gossip

Pilosa uses HashiCorp's memberlist protocol to manage cluster state and to broadcast internode messages within the cluster. Memberlist is a gossip implementation based on the SWIM protocol that supports both synchronous and asynchronous communication. By providing a gossip seed—the address of a node in the cluster—memberlist determines cluster membership for all nodes, and delivers node membership events to all nodes in the cluster. For example, when a new node joins the cluster, memberlist ensures that all other nodes in the cluster receive a NodeJoin event along with information about the joining node. In addition to management node membership, memberlist also provides Pilosa with the ability to send messages between nodes in the cluster. Pilosa uses this functionality to broadcast messages about cluster state and schema changes. For example, when a Frame is created on a node, Pilosa broadcasts a FrameCreation message to all other nodes in the cluster. This ensures that every node is aware of the latest schema and can therefore appropriately respond to future queries.

Guarantees

Pilosa uses memory mapped files, allowing writes to be persisted to disk while still providing fast, in-memory reads. Memory mapped files offer several benefits. Restarting a Pilosa node is fast because the mapped files are loaded lazily into

memory. Data can be paged in and out of memory as needed, so in the case where a Pilosa cluster has been memory-constrained it can still operate on the data. Pages are located in the operating system page cache, so they remain cached even between process restarts. As an additional safeguard against data loss, all writes are written to an append-only log on disk, and then applied to the in-memory representation.

Attributes

Pilosa can store metadata pertaining to rows and columns in an embedded BoltDB key/value database. The intention is that attributes should be used to store information that is specific to a row or column. There are a couple of uses for attributes. First, row and column attributes can be returned in query results. In an example where the columns of an index represent individual people, one might store each person's name in the column attributes. In that case, a query could be specified to return the list of *columnIds* that represent the query, as well as the list of names that make up that result set. Second, attributes on rows can be used as filters in TopN queries. If a Pilosa frame contains rows that represent movies, and each row is given an attribute called genre with values like "comedy" and "drama", then one could perform a TopN query on the movie frame filtered by genre.

API

Communication with a Pilosa cluster happens over an HTTP API supporting both JSON and protocol buffers serialization.

Client Libraries

Application developers can include one of the supported client libraries in their application in order to interact with the Pilosa API. Currently, client libraries are available in Go, Python, and Java.

PDK

As a general purpose, database agnostic index, some external tooling is generally required to make good use of Pilosa within a given organization. The Pilosa Development Kit (PDK) is a set of tools and libraries which make it easier to fit Pilosa into existing stacks and derive its full benefits.

The PDK contains tooling specifically for working in data streaming pipelines using Kafka, and in some cases, it may be possible to start streaming data from Kafka into Pilosa without writing any code or configuration. The PDK's libraries will reflectively examine data and try to automatically determine the best possible Pilosa representation for that data. By default, any record coming from Kafka would be assigned a new column in the index, and the various fields and their values would be broken out into frames and rows respectively. While using this automatic tooling may be a great way to get started with Pilosa, it's almost always possible to gain efficiency and flexibility benefits with some application specific configuration.

Conclusion

We've seen how Pilosa's data model, clustering strategy, typical usage, and advanced features allow it to operate in-memory using low-level machine instructions to process queries at unparalleled speed on commodity hardware.

Pilosa is, first and foremost, an open source project built and maintained by a community of dedicated engineers. We are always looking to make improvements, and appreciate any feedback you may have. Please visit our GitHub repository to learn more about our software. For details and instructions on how to download and install the open source version of our software, please visit our website.

Pilosa is also available as an enterprise-level solution. Pilosa Enterprise automates the staging of enterprise data for real-time retrievability and consumption, greatly reducing the time DataOps spends preparing it. If you believe Pilosa may be a good fit for your project or business, please visit our Enterprise page, or reach out to our team at info@pilosa.com.