COMPSCI 590S Systems for Data Science **Fall 2017**

Lecture 13: Bloom Filters and BigTable

Lecturer: Emery Berger *Scribe(s): Abhiram Eshwaran, Akul Siddalingaswamy*

13.1 Counting Bloom Filters (Continued)

As discussed earlier, bloom filters are used for fast approximate lookup where we trade accuracy for space. Counting bloom filters are another variation which support deletion operation as well.

Consider a sample bloom filter of 8 bits long [0 0 0 0 0 0 0 0] with 0-based indexing. Consider the operations of add(x), delete(x), ismember(x) where x is the input object. For now, let's take 2 strings- "foo" and "bar"

Also, consider 2 hash functions $h_1(x)$ and $h_2(x)$ which spit out the bit positions (one each) for the input x. Let's assume,
h_1("foo") = 1 and h_2("foo") = 2
h_1("bar") = 2 and h_2("bar") = 4

Initially, bloom = [0 0 0 0 0 0 0 0]
add("foo") increments the bits at positions 1 and 2 by 1 making bloom = [0 1 1 0 0 0 0 0]
Similarly, add("bar") increments the bits at positions 2 and 4 by 1 making bloom = [0 1 2 0 1 0 0 0]
Now, if we execute ismember("foo"), we check positions 1 and 2. Since both are non-empty, the bloom filter outputs "MAYBE"
Next, if we execute delete("foo"), we decrement the positions 1 and 2 by 1 making bloom = [0 0 1 0 1 0 0 0]
However, if we now execute ismember("foo"), we check positions 1 and 2. Since both are empty, the bloom filter outputs "NO"
This way, the counting bloom filter supports the delete operation.

13.2 Web Search, Indexing and Spidering

13.2.1 A Primitive Web Search

A simple Google query must crunch through petabyte of data to return the results of the query. There are two ways to do this,

- Naive approach: Linear Search
- Better approach: Use indices to data, store this in Btree and this can be achieved in log time big O complexity.

But consider a query like "Traffic In Boston For Friday", this query will need to find an intersection of all results in Traffic, Boston and Friday to give the results. A perfect solution to the above problem would be an inverted index.

Token	Document Id
Harry	1, 2
Potter	1, 2
And	1, 2
The	1, 2
Half	1
Blood	1
Prince	1
Deathly	2
Hallows	2

Inverted index

Figure 13.1: Inverted Index

13.2.2 Inverted Index

Consider the image above. Now a query like "Harry Potter and The Half Blood Prince" would take an intersection of all documents in the inverted index and return document 1 as the result.

But what if there are millions of results? The next challenge would be to order these millions of documents. This is know as **ranking**.

"**Link Span**" was one concept which determined the rank of a web page. If most results of a query pointed to a particular web page then that web page ended up getting a higher rank on the search results. This "link span" was later removed due to malicious websites exploiting this concept.

But we are still left with the task of creating a database for the web. This leads us to Spidering.

13.2.3 Spidering

We have the web, we have a spider, all we have to do now is crawl. So this what a web spider does. A web spider crawls the web. By crawl we mean, it takes a web page, get all the links in the web page and recursively follows all the links. (We use marking to prevent revisiting a web page)

The web spider crawls the web continuously. Pages are cached with different frequencies and this is done really fast. (Google News leverages this to present all the breaking news). Few important characteristics of the web spider,

- Needs to store only the most recent version.
- The Web Spider is basically just writes. Writes need to be super efficient and new versions should be managed.
- Disk storage is crucial. Seek time for random access is a pain and the memory has to be defragmented. We solve this particular problem by GFS which does only bulk writes.

13.3 BigTable

13.3.1 Motivation for BigTable

As discussed above, periodic crawling could help keep the pages up-to date. However, if there are millions of pages, it becomes expensive to do these writes/updates one at a time (seek time + rotational latency + transfer time for each write). A solution to this is to just write sequentially in the form of a log, a.k.a Log-Structured File System. Write throughput on optical and magnetic disks improves because the writes can be batched into large sequential runs and costly seeks are kept to a minimum. However, when we try to read them, we hit a bottleneck since the data is sequential. As a way of providing random access to the data, an index is created for this log-structured file system and this index is basically BigTable.

13.3.2 Data model

Generally, NoSQL datadases store data as simple *key* $\rightarrow$ *value* pairs. Bigtable's data model is a generalization of such data storage systems. Here, the map is indexed by a row key, column key, and a timestamp: *(row:string, column:string, time:int64)* $\rightarrow$ *(value:string)*

Column1	Column2	...	ColumnN	Value

A logical overview of Bigtable rows

A general way of key-value stores is to map all the columns to a single row key. But BigTable key is a triple where they have (row, column, timestamp) and allow accessing a column directly. Bigtable is a dense data store, which means it has cells with non-empty data. Using timestamps, different versions of a cell are maintained in most-recent-first order. The number of versions depend on a threshold set that specifies only the last n versions of a cell be kept. The columns are grouped as families, and each column is identified as a pair *(family: qualifier)*. The row IDs are sorted and split into different groups called *tablets*. This lets BigTable take advantage of locality of data.

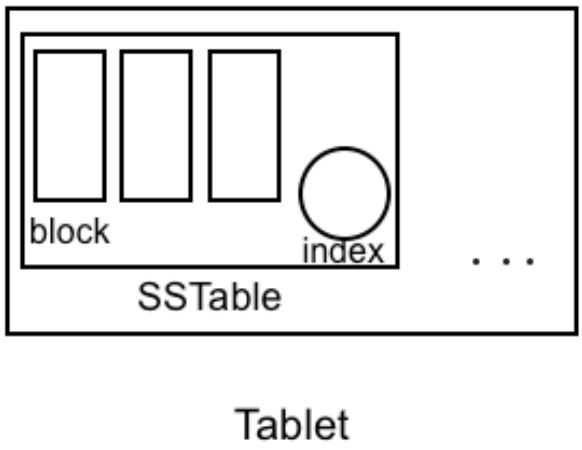

Figure 13.2: Tablet structure

13.3.3 Tablet

Each tablet contains one or more SSTables. SSTable (Sorted String Table) is a file format that provides an immutable, ordered key-value map. An SSTable contains multiple blocks (or chunks) and an index to locate the blocks. SSTables use binary search within the block for finding rows. Additionally, the recently committed updates are stored in memory in a sorted buffer called *memtable*. When the memtable size reaches a threshold, it is frozen and converted to an SSTable.