Big Data Platforms (5 ECTS)
DATA14003
Lecture 7

Keijo Heljanko

Department of Computer Science
University of Helsinki
keijo.heljanko@helsinki.fi

5.10-2021

NoSQL Databases (Cloud Datastores)

- Quite often the term Cloud Datastore is used for database systems developed for the cloud, as they often do not fully support all features of traditional relational databases (RDBMS)
- Other term used often is NoSQL databases, as the original systems did not support SQL. However, some SQL support is getting added also to cloud datastores, so the term is getting outdated quite fast
- The cloud datastores can be grouped by many characteristics

Scalable Cloud Data Store Features

A pointer to comparison of the different cloud datastores is the survey paper: “Rick Cattell: Scalable SQL and no-SQL Data Stores, SIGMOD Record, Volume 39, Number 4, December 2010”.

`http://www.sigmod.org/publications/sigmod-record/1012/pdfs/04.surveys.cattell.pdf`

Scalable Cloud Data Store Features (cnt.)

The Cattell paper lists some key features many of these new datastores include (no single system contains all of these!):

- The ability to horizontally scale “simple operation” throughput over many servers
- The ability to automatically replicate and to distribute (partition/shard) data over many servers
- A simple call level interface or protocol (vs SQL)

Scalable Cloud Data Store Features (cnt.)

- A weaker concurrency model than the ACID transactions of most relational (SQL) database systems
- Efficient use of distributed indexes and RAM for data storage
- The ability to dynamically add new attributes to data records (no fixed data schema)

Grouping of Data Stores - Key-value Stores

- Key-value stores
  - Examples: Redis, Riak, Apache Cassandra (partially), Scalaris
  - A unique primary key is used to access a data item that is usually a binary blob, but some systems also support more structured data
  - Quite often use peer-to-peer technology such as distributed hash table (DHT) and consistent hashing to shard data and allow elastic addition and removal of servers

Grouping of Data Stores - Key-value Stores (cnt.)

- Key-value stores (cnt.)
  - Some systems use only RAM to store data (and are used as memcached replacements), others also persist data to disk and can be used as persistent database replacements
  - Most systems are AP systems but some (Scalaris) support local row level transactions
  - Cassandra is hard to categorize as it uses DHT, is an AP system, but has a very rich data model

Grouping of Data Stores - Document Stores

- Document Stores
  - Examples: Amazon SimpleDB, CouchDB, MongoDB
  - For storing structured documents (think, e.g., XML)
  - Do not usually support transactions or ACID semantics
  - Usually allow very flexible indexing by many document fields
  - Main focus on programmer productivity, not ultimate data store scalability

Grouping - Extensible Record Stores

- Extensible Record Stores (aka “BigTable clones”)
  - Examples: Google BigTable, Google Megastore, Apache HBase, Hypertable, Apache Cassandra (partially)
  - Google BigTable paper gave a new database design approach that the other systems have emulated
  - BigTable is based on a write optimized design: Read performance is sacrificed to obtain more write performance
  - Other notable features: Consistent and Partition tolerant (CP) design, Automatic sharding on primary key, and a flexible data model (more details later)

Grouping of Data Stores - Scalable Relational Systems

- Scalable Relational Systems (aka Distributed Databases)
  - Examples: MySQL Cluster, VoltDB, Oracle Real Application Clusters (RAC)
  - Data sharded over a number of database servers
  - Usually automatic replication of data to several servers supported
  - Usually SQL database access + support for full ACID transactions
  - For scalability joins that span multiple database servers or global transactions should not be used by applications
  - Scalability to very large (100+) database servers not demonstrated yet but there is no inherent reason why this could not be done

Eventual Consistency / BASE

- The term “Eventually Consistent” was popularized by the authors of Amazon Dynamo, which is a datastore that is an AP system - accessible and partition tolerant
- Also the term BASE (basically available, soft state, eventually consistent) is a synonym for the same approach
- Basically these are datastores that value accessibility over data consistency
- Quite often they offer support to automatically resolve some of the inconsistencies using e.g., version numbering or CRDTs - Commutative Replicated Data Types
- Example systems: Amazon Dynamo, Riak, Apache Cassandra

RDBMS Supporter View

- RDBMS can do everything the scalable cloud data stores can do with similar scalability when used properly
- SQL is a convenient expressive declarative query language
- RDBMS have 30 years of engineering experience put into them and are highly tuned
- There are a very large number of specialized RDBMS for various application domains (interactive vs. analytics, RAM vs. disk based data, etc.)
- A standardization around relational schema and SQL brings benefits in design and training, where same set of skills can be effectively employed with another RDBMS system

RDBMS Opponent View

- The RDBMS vendors have not demonstrated scalability matching that of the newly architected scalable cloud data stores (e.g., BigTable)
- Primary key lookup is more easy to understand than SQL, and thus enables a much lower learning curve
- RDBMS force data schema on applications unnecessarily
- SQL makes it “too easy” to write expensive queries by accident such as complicated joins involving many tables and several servers. If these expensive operations are removed from the query language, the complexity of a query evaluation becomes obvious to the programmer
- RDBMS systems have become dinosaurs in their 30 year rule of the database market, and the (hardware) assumptions on which they have been built are obsolete

Scalable Data Stores - Future Speculation

- Long running serializable global transactions are very hard to implement efficiently, for scalability they should be avoided at all costs. Counterexamples: Google Percolator, Google Cloud Spanner
- The requirements of low latency and high availability make AP solutions attractive but they are more difficult to use for the programmer (need to provide application specific data inconsistency recovery routines!) than CP systems
- The time of “one size fits all”, where a RDBMS was a solution to all datastore problems has passed, and scalable cloud data stores are here to stay

Scalable Data Stores - Future Speculation (cnt.)

- ACID transactions are needed for, e.g., financial transactions, and there traditional RDBMS will be dominant
- Many of the new systems are not yet proven in production
- There will be a consolidation to a smaller number of data stores, once the “design space exploration” settles down

NewSQL - Google Cloud Spanner

- A beta release of Google's internal Spanner Database
- SQL support
- ACID transactions that can span multiple servers globally
- Fully consistent database with a single global database image
- Implementation based on Paxos for replication, Transaction commit priority made using physical timestamps
- All database server clocks synchronized very tightly using atomic clocks + GPS based time synchronization
- All database transactions have to wait two times the max clock drift holding write locks before committing a transaction
- Open source alternatives being developed: CockroachDB, Apache Kudu

Probabilistic Datastructures

- Main memory (RAM) is very expensive compared to Hard Disk or SSD based storage
- It is much faster than other forms of storage and thus using this expensive resource as efficiently as possible is important
- Probabilistic datastructures allow much more efficient RAM usage at the expense of some inaccuracy in the results
- Bloom filters are an example of a probabilistic datastructure that is heavily used in several NoSQL databases to minimize I/O using a small RAM based probabilistic index
- They can be used also for many data analytics tasks to do approximate computations

Example: Bloom filter for Web page caching

- Problem: We are designing a caching mechanism for a Web server. We are given a stream of Web page URLs, listing each URL that is being accessed
- We want to find those Web page URLs that are being accessed more than once to cache them
- We do not want to cache any Web pages only accessed once, as this can potentially remove more frequently accessed data from the cache
- We allow for a small number of false positives: It is OK to cache a small percentage of items on first access if that speeds up the algorithm and especially if it lowers the memory requirements needed

Motivating Problem: Caching on Second Access

- Traditional solution: Use a hash table for storing each one of the encountered URLs, cache each URL when it is encountered the second time
  - Simple solution but requires potentially a lot of memory to store the hashed URLs
- Second traditional design: Use an external database to store all encountered URLs
  - Scales beyond main memory hash table but is potentially slow as each cache access needs a database query
- Third design: Use a Bloom filter to record the set of encountered URLs

Bloom Filters

- Bloom filters are a highly memory-efficient probabilistic data structure to store sets
  - The Bloom filter consist of a bitarray B of m bits $B[0], B[1], \ldots, B[m-1]$, with all bits initialized to zeros
  - To insert a data item d to this set, first k independent hash functions $h_0(d), h_1(d), \ldots h_{k-1}(d)$ with domains $0 \leq h_i(d) \leq m-1$ are calculated from the data item d
  - Then the bit-array is updated to contain a one at each of the k hashed positions: $B[h_0(d)] = 1; B[h_1(d)] = 1; \cdots B[h_{k-1}(d)] = 1;$

Bloom Filter Lookup

- To check whether a data item belongs to the set represented by a Bloom filter B first k independent hash functions $h_0(d), h_1(d), \ldots, h_{k-1}(d)$ are calculated from the data item d
  - If B contains a bit set to 1 in all the positions: $B[h_0(d)], B[h_1(d)], \ldots, B[h_{k-1}(d)]$ then it the Bloom filter returns: "the data item d is potentially in the set". This outcome can also happen even if d has not been inserted to the Bloom filter (due to hash collisions), in which case we call it a false positive
  - Otherwise at least one of the positions: $B[h_0(d)], B[h_1(d)], \ldots, B[h_{k-1}(d)]$ contains a zero. In this case the Bloom filter returns "the data item d is not in the set"

Running Example

- Example: We use a Bloom filter B of $m = 8$ bits of memory, initialized to all zeroes $B[0] = 0, \ldots, B[7] = 0$:

B	0	0	0	0	0	0	0	0

- Typical Bloom filter sizes for many applications are in the order of megabytes or more, we use small m for demonstration purposes
- We use $k = 2$ independent hash functions h_0 and h_1 for this Bloom filter

Running Example (cnt.)

- First add data item d =`http://www.cnn.com/` into the Bloom filter B
- Calculate $k = 2$ hash functions, assume we get:
  - $i_0 = h_0(\texttt{http://www.cnn.com/}) = 2$
  - $i_1 = h_1(\texttt{http://www.cnn.com/}) = 5$
- As $B[i_0] = B[2] == 0$ before insertion, we know `http://www.cnn.com/` is not in the set before the insertion
- Insert the item by setting both bits $B[i_0] = 1$ and $B[i_1] = 1$.
- Bloom filter B after the insertion:

B	0	0	1	0	0	1	0	0

Running Example (cnt.)

- Suppose we want to next add the data item `http://www.helsinki.fi/` into the Bloom filter B
- Calculate $k = 2$ hash functions, assume we get:
  - $i_0 = h_0(\texttt{http://www.helsinki.fi/}) = 5$
  - $i_1 = h_1(\texttt{http://www.helsinki.fi/}) = 0$
- As $B[i_1] = B[0] == 0$ before insertion, we know `http://www.helsinki.fi/` is not in the set before the insertion, even when $B[i_0] = B[5] == 1$
- Insert the item by setting both bits $B[i_0] = 1$ and $B[i_1] = 1$.
- Bloom filter B after the insertion:

B	1	0	1	0	0	1	0	0

Running Example (cnt.)

- Suppose we now encounter the URL `http://www.cnn.com/` a second time
- Calculate $k = 2$ hash functions, assume we get:
  - $i_0 = h_0(\texttt{http://www.cnn.com/}) = 2$
  - $i_1 = h_1(\texttt{http://www.cnn.com/}) = 5$
- As $B[i_0] == 1$ and $B[i_1] == 1$, we correctly notice that `http://www.cnn.com/` has been seen before, and thus should be cached in our Web page caching application
- Bloom filter B remains unmodified:

B	1	0	1	0	0	1	0	0

Running Example (cnt.)

- Next we want to add the data item `http://www.yle.fi/` into the Bloom filter B
- Calculate $k = 2$ hash functions, assume we get:
  - $i_0 = h_0(\texttt{http://www.yle.fi/}) = 0$
  - $i_1 = h_1(\texttt{http://www.yle.fi/}) = 2$
- As $B[i_0] = B[0] == 1$ and $B[i_1] = B[2] == 1$ before insertion, we wrongly assume `http://www.yle.fi/` has been added to the set before due to two hash collisions. This is called a false positive
- Thus in our Web caching application we incorrectly decide to cache `http://www.yle.fi/` already on first access due to the hash collisions
- The Bloom filter B contents remain the same:

B	1	0	1	0	0	1	0	0

How to avoid false positives?

- Discussion: What needs to be done to minimize the number of false positives?

Avoiding false positives

- Increasing Bloom filter size decreases the number of false positives. However, Bloom filters are usually used when memory needs to be saved
- A too small a k, for example $k = 1$, suffers from fairly frequent hash collisions (see: Birthday paradox), thus avoiding $k = 1$ should be done if possible
- However, if k is too large, the Bloom filter quickly fills with bits set to one, as k indexes are assigned to one on for each inserted data item
- The optimal number k depends thus on both the memory m available and the number of items n to be inserted to the Bloom filter
- Very large k also slows processing, as each lookup of a bit in Bloom filter is a random memory access

Bloom Filter False Positives

- What is the probability of a false positive?
- Probability that bit b is set to one by $h_i = \frac{1}{m}$
- Probability that bit b is not set to one by any of the k hash functions = $(1 - \frac{1}{m})^k$
- Probability that bit b is still 0 after inserting n elements = $(1 - \frac{1}{m})^{kn}$
- Probability of a false positive appears when all k bits are already set to one before inserting an element, i.e. probability that a bit is already set to one to the power of k = $(1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{-kn}{m}})^k$, where n is the number of unique items inserted to the Bloom filter so far, and e is the Euler's number $e \approx 2.71828$

Bloom Filter Parameters

- One of the parameters of the Bloom filter is the optimal number of hash functions k to use to minimize the false positive probability
- The optimal $k = \frac{m}{n} \ln(2) \approx 0.693 \frac{m}{n}$
- We refer to the literature for the derivation: Andrei Broder and Michael Mitzenmacher (2004) Network Applications of Bloom Filters: A Survey, Internet Mathematics, 1:4, 485-509.

  `http://dx.doi.org/10.1080/15427951.2004.10129096`

Bloom Filter Parameters - Example

- Assume that in our Web caching example we need to process 10 million unique URLs. Assume also that we have 12 megabyte of memory available for a Bloom filter.
  - What is the optimal number *k* of hash functions to use?
  - What is the false positive rate for the last inserted URL with this k?

Bloom Filter Parameters - Example

- Computing optimal k:
  - We have $m = 12 \cdot 8 \cdot 1024 \cdot 1024 = 100663296$ bits available in our Bloom filter
  - The number of elements to insert is $n = 10000000$
  - Substituting to $k = \frac{m}{n} \ln(2) \approx \frac{100663296}{10000000} \cdot 0.693 \approx 6.98$
  - Rounding to the nearest integer, let's use $k = 7$ hash functions for our Bloom filter

Bloom Filter Parameters - Example

- Computing the false positive rate with $k = 7$:
  - We have $m = 12 \cdot 8 \cdot 1024 \cdot 1024 = 100663296$ bits in our Bloom filter
  - The number of elements to insert is $n = 10000000$
  - Substituting
    $Prob = (1 - e^{\frac{-kn}{m}})^k = (1 - e^{\frac{-7 \cdot 10000000}{100663296}})^7 \approx 0.008 = 0.8\%$
  - So, the false positive rate is 0.8% for the last inserted URL

Traditional Hash Table Memory Usage

- Assume that in our Web page caching example the average length of a URL is 25 bytes, and a worst-case scenario that each one of the URLs is unique
- Then the amount of memory to store the URL strings in a hash table is 25 bytes $\cdot 10000000 \approx 238$ megabytes
- Thus using a Bloom filter and allowing 0.8% false positive rate in caching decisions has reduced the memory requirements by at least $238 - 12 = 226$ megabytes

Bloom Filters - Positives

- Bloom filters are a memory efficient probabilistic data structure for storing sets
- Very simple implementation, very fast for small number of hash functions
- Can be combined with a traditional database index: If Bloom filter says a data item is not in a database, a traditional index need not be traversed. If Bloom filter says “maybe”, traditional index (that may be on a hard disk) needs be consulted
- Bloom filters allow for efficient parallelization to create them: Just take the bitwise or of the Bloom filters created in parallel for subsets of the data
- Many variants exist, see for example stable Bloom filters by Deng for stream processing, which start forgetting older items in a data stream in probabilistic fashion

Bloom Filters - Negatives

- No delete operation available in basic version
- Need to tune k based on data volume n
- To get a very small false positive probability a large number of hash functions need to be used, which makes very precise Bloom filters slow
- Large Bloom filters are not cache friendly
- If exposed to user generated data, may need (very slow) cryptographic hash functions to avoid collisions made on purpose
- Computing a large number of hash functions can be expensive
- There are even more memory efficient data structures allowing deletions, see e.g., Cuckoo Filters by Fan et al.

Further References

- Andrei Broder and Michael Mitzenmacher (2004) Network Applications of Bloom Filters: A Survey, Internet Mathematics, 1:4, 485-509.

  http://dx.doi.org/10.1080/15427951.2004.10129096

- Fan Deng, Davood Rafiei: Approximately detecting duplicates for streaming data using stable Bloom filters. SIGMOD Conference 2006: 25-36.

  http://dx.doi.org/10.1145/1142473.1142477

- Bin Fan, David G. Andersen, Michael Kaminsky, Michael Mitzenmacher: Cuckoo Filter: Practically Better Than Bloom. CoNEXT 2014: 75-88.

  http://dx.doi.org/10.1145/2674005.2674994