CS-E4640 - Big Data Platforms
Lecture 6

Keijo Heljanko

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

17.10-2018

Storage Technologies for the Cloud

The following storage technologies are widely used in the cloud:

- RAM (Random Access Memory)
- Flash (also called SSD - Solid State Drive)
- Hard Disk
- Tape

Flash Storage

- Currently one of the trends is the introduction of Flash memory SSDs (solid state disks) are replacing hard disks in many applications
- Flash capacity per Euro is increasing faster than hard disk capacity per Euro
- Currently Flash capacity is around 5% of all storage capacity but is growing quickly
- Random SSD read (and often also write) IOPS are more than 100 $\times$ those of high end hard disk read and write IOPS

Flash Storage (cnt.)

- Sequential read and write speeds are faster than those of the best hard disks
- As SSDs have no moving parts they are fail quite a bit less frequently than hard disks
- As such SSDs are a very good match for typical client laptop and desktop usage patterns

Flash Storage

- Flash has both strengths and limitations compared to hard disks, they are not a direct hard disk replacement for all server workloads
- Especially the write endurance of SSDs (the amount of data that the SSD is specified to write reliably during its lifetime) is quite often very small compared to hard disks
- For write intensive server workloads hard disks might still be the only economically viable option due to SSDs having to be replaced once write endurance is exhausted

Flash Organization

- A flash chip is organized in pages of some KBs in size (e.g., 2KB)
- A page read can load the data of a page into buffers that can be quickly randomly accessed
- A page write can only flip bits of the page from 0 to 1
- In order to change the bits from 1 to 0 and erase operation operating on a large number of blocks needs to be performed
- An erase operation works on blocks that consist of several flash pages at the time (e.g., 128KB of data)
- If the blocks to be erased contain some data, that data must be written elsewhere before the block is erased

Flash Types

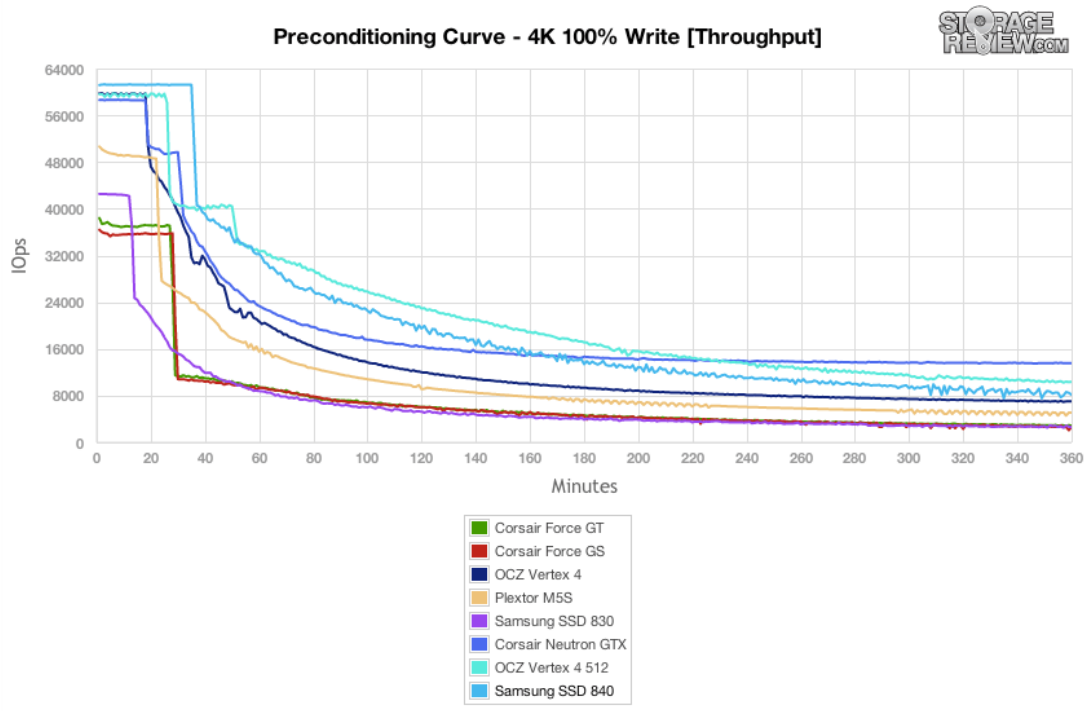
- Flash memory comes in several flavors, the main ones being NAND flash in varieties: triple-level cell TLC (cheapest), multi-level cell MLC (cheap), and single-level cell SLC (expensive)
- Flash blocks can be erased: around 1000 times (TLC flash), 1000-10000 times (MLC flash), or 100000-1000000 times (SLC flash)

Flash Wear Levelling Algorithms

- Highly sophisticated algorithms are used for wear leveling, where the goal is to write each flash cell as evenly as possible
- When flash disk becomes almost full, the free space can become fragmented which can result in performance loss on writes as the SSD moves blocks around to make space for the written data
- Flash disks often use “spare capacity” set aside to make the fragmentation problems less severe

Sustained Flash Write Performance

- Because of the RAM write caches and the write leveling algoritms, the sustained Flash memory write speeds can sometimes be compromised:
  `http://www.storagereview.com/samsung_ssd_840_pro_review`



Optimizing for Flash

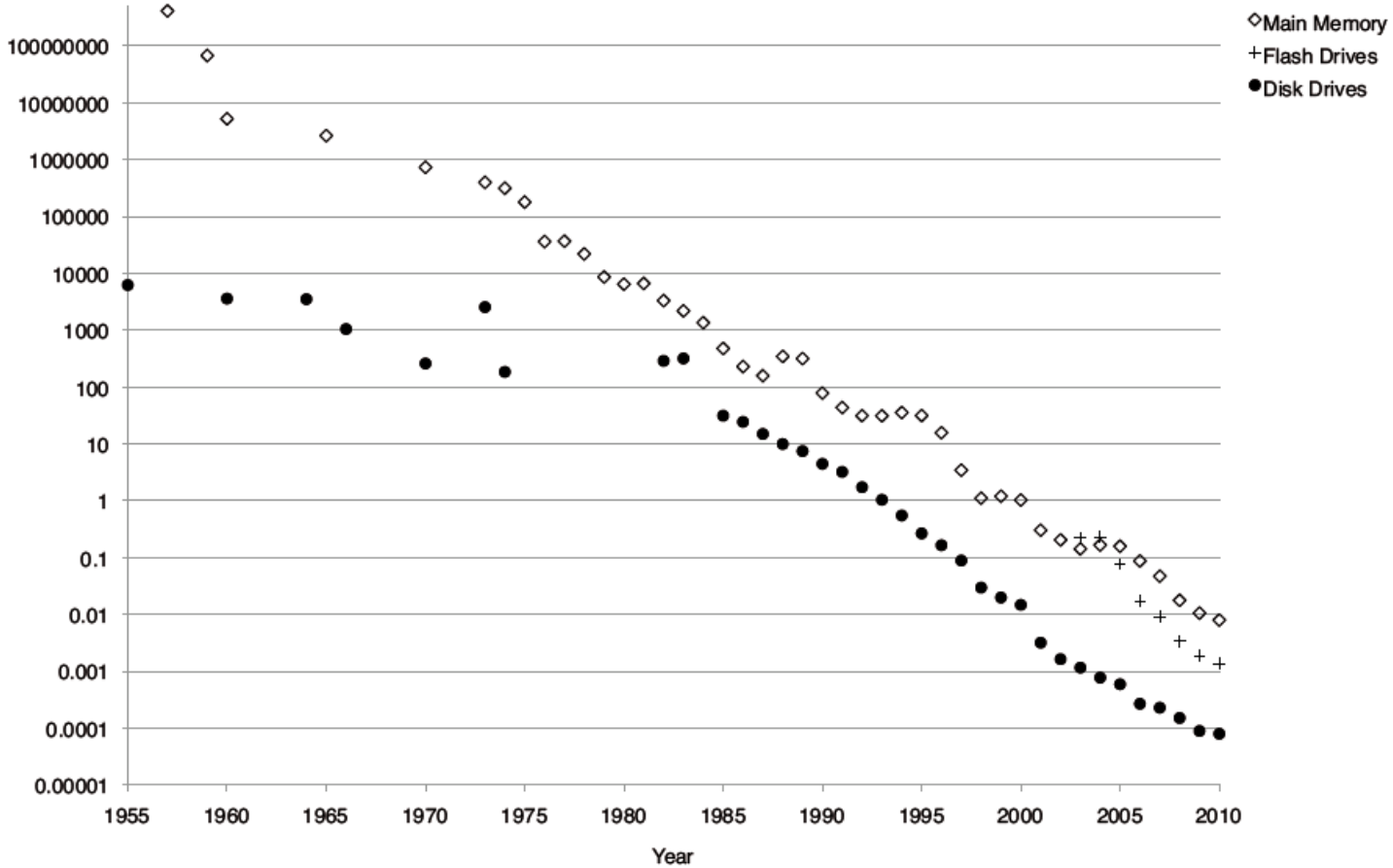
- One should minimize the number of bytes written to Flash
  - Less bytes written means less wear and longer lifetime expectancy of Flash
  - Less bytes written means less frequent slow block erases, and this improves the overall Flash IOPS
- Heavy sequential write workloads (e.g., database logs) can be efficiently handled by arrays of hard disks without any problems with write endurance
- Operating systems support such as the TRIM command which tells to the SSD which data blocks can be safely discarded is very useful

Some Rule of Thumb Numbers for Flash

For a single random access read, the following rules of thumb numbers apply:

- RAM memory reference: 100 ns
- Flash drive access: 100 000 ns (1000x slower than memory)
- Disk seek: 10 000 000 ns (100 000x slower than memory)

Storage Price Trends



- Trends of RAM, Flash, and HDD prices. From: H. Plattner and A. Zeier: In-Memory Data Management: An Inflection Point for Enterprise Applications

Storage Usage Scenarios

- Tapes are still being used for backup purposes as they are cheap per Terabyte
- RAM (and Flash) are radically faster than HDDs: One should use RAM/Flash whenever possible
- RAM is roughly the same price as HDDs were a decade earlier
  - Workloads that were viable with hard disks a decade ago are now viable in RAM
  - One should only use hard disk based storage for datasets that are not yet economically viable in RAM (or Flash)
  - The Big Data applications (HDD based massive storage) should consist of applications that were not economically feasible a decade ago using HDDs

Cost of Storage: RAM vs Flash vs Disk

The paper: John K. Ousterhout et. al: The case for RAMCloud. Commun. ACM 54(7): 121-130 (2011) summarizes the Total Cost of Ownership for Storage Technologies:

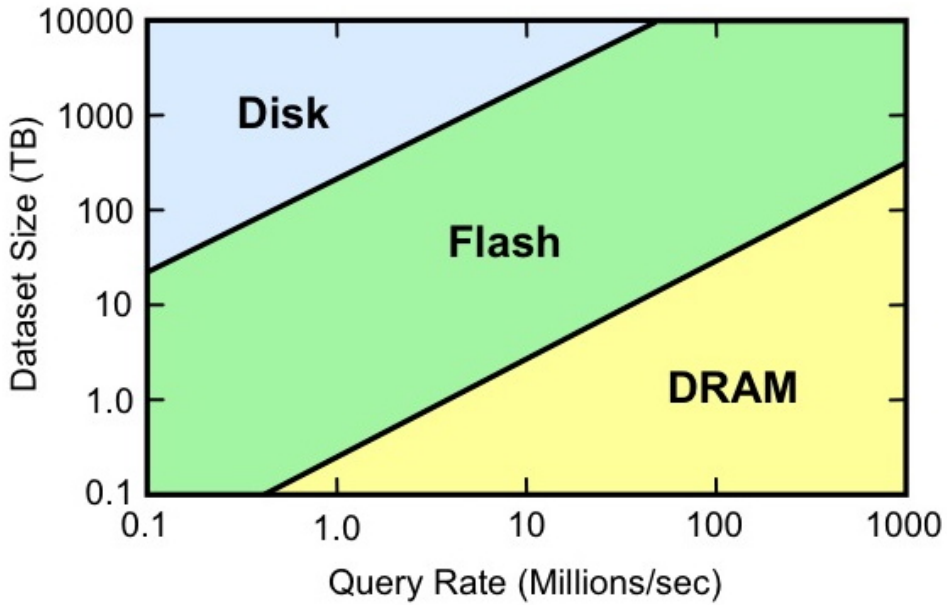


Figure 2. This figure (reproduced from Andersen et al. [1]) indicates which storage technology has the lowest total cost of ownership over 3 years, including server cost and energy usage, given the required dataset size and query rate for an application (assumes random access workloads).

Cost of Storage: RAM vs Flash vs Disk

- Most practical system contain data sets with different “temperatures”
- Hot Storage: When a large number of IOPS/TB is needed, RAM is the cheapest way to obtain lots of IOPS
- Cold Storage: When a large amount of TB/IOPS is needed, Hard Disks are the cheapest way to obtain lots of storage
- Warm Storage: Flash is a compromise between cost per IOPS and cost per TB
- When the pricing of RAM/Flash/HDD changes, the exact optimal selection of best storage technology changes

Example: Facebook Storage

- Facebook has done research on the warmt of their datasets in paper: Subramanian Muralidhar et. al: f4: Facebooks Warm BLOB Storage System

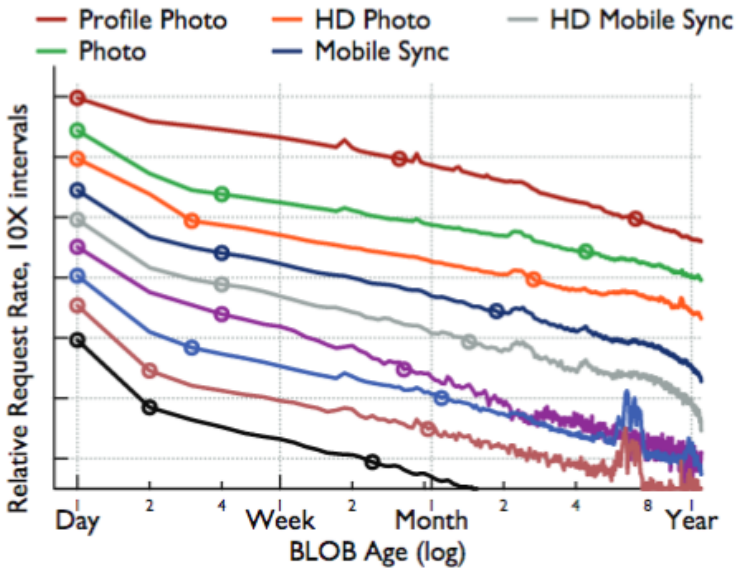


Figure 3: Relative request rates by age. Each line is relative to only itself, absolute values have been denormalized to increase readability, and points mark an order-of-magnitude decrease in request rate.

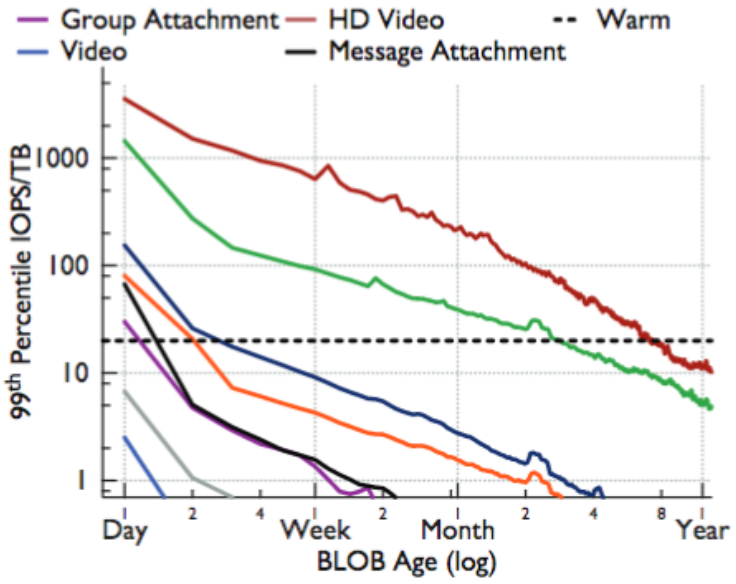


Figure 4: 99th percentile load in IOPS/TB of data for different BLOB types for BLOBs of various ages.

Example: Facebook Storage

- The data stored at Facebook cools as it gets older: The optimal way to store the data is different for new and old files at Facebook
- Facebook has separate storage solutions for Hot, Warm, and Cold data storage
- As data gets older and thus colder, it is migrated from one storage system to another to save costs

Storage Technologies - Jim Gray view

A mantra from Turing Award Winner Jim Gray of Microsoft in his very visionary (from year 2006!) presentation: `http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt`:

- Tape is Dead
- Disk is Tape
- Flash is Disk
- RAM Locality is King

Jim Gray on Disks (in 2006)

- Disk are cheap per capacity
- Sequential access full disk reads of take hours
- Random access full disk reads take weeks
- Thus most of disk should be treated as a cold-storage archive

Jim Gray on Flash (in 2006)

- Lots of IOPS
- Expensive compared to disks (but improving)
- Limited write endurance
- Slow to write (compared to reading)

Jim Gray on RAM (2006 numbers)

- Flash/disk is 100000-1000000 cycles away from the CPU
- RAM is 100 cycles away from the CPU
- Thus Jim Gray concludes that main memory databases are going to be common

Caching to Improve Hard Disk Performance

- Caching the hot part of the dataset in RAM is a well known technique for improving the performance of a hard disk based storage system
- The large difference in RAM and hard disk random access latencies makes this sometimes difficult, as caches need to have extremely good hit rates for the hard disk accesses not to dominate

RAMCloud

John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazires, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, Ryan Stutsman: The case for RAMCloud. Commun. ACM 54(7): 121-130 (2011).

- Facebook in 2009 has been running in 2009 with enough RAM to fit 75% of their dataset (not counting images or video) in RAM
- If they would use enough RAM cache to hit 99% of the dataset, the random disk seek latency would still kill their average latency performance: (99% $\times$ 100 ns + 1% $\times$ 10 000 000 ns) = 100099 ns, which is way more that 100ns!

RAMCloud - Discussing the Numbers

- With Flash disks and 99% cache hit rate we get latencies: (99% $\times$ 100 ns + 1% $\times$ 100 000 ns) = 1099 ns, which is still 10x more that 100ns.
- The above concerns only average latency, not throughput, but similar reasoning also applies to aggregate IOPS numbers of RAM, Flash, and Disk
- If a system can have enough memory to have 100% of the working set in RAM, instead of Flash / Hard disk, way better latency can be obtained, and more IOPS can be served with the same number of servers.

RAMCloud - Limiting Factors

- The main problem in RAMcloud style designs to guarantee data persistence in case of power failure + quick recovery in case of server crash / power outage
- New non-volatile memory technologies such as Intel/Micron 3D-Xpoint memories (`https://en.wikipedia.org/wiki/3D_XPoint`) that promise even much faster performance than Flash might be able to help realize this
- The second problem is to have enough low latency networking hardware and OS to keep RAM busy
- For discussion, see: Luiz Barroso et. al: Attack of the killer microseconds, Communications of the ACM, Volume 60 Issue 4, April 2017, Pages 48-54 `https://doi.org/10.1145/3015146`

Improving Disk Based Database Performance

- Sometimes we need to create databases that are much larger than can fit the RAM
- RAM is a very expensive resource, and thus should be utilized as well as possible
- Using a normal cache of hot data in RAM can improve query latency for items that are in the database
- RAM can also be used to improve performance for queries of items not in the database
  - RAM can hold an index of all the items on hard disk. However, sometimes the index size can be larger than RAM
  - Probabilistic data structures use RAM even more efficiently than normal indexes - Example: Bloom filters

Motivating Problem: Caching on Repeated Access

- 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 in memory in addition to having them stored in the hard drive
- 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

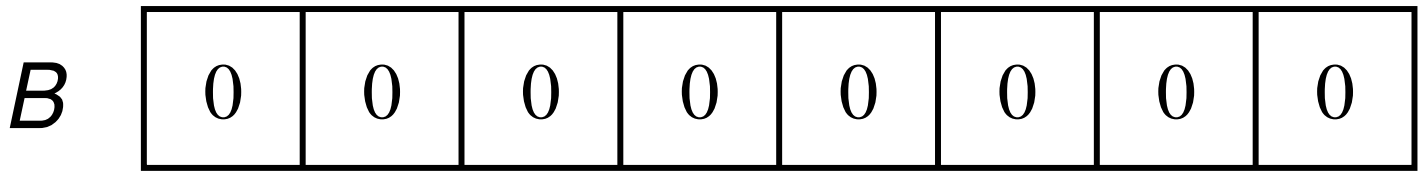
- Bloom filters are a highly memory-efficient probabilistic data structure to store sets
  - The Bloom filter consist of a bitarray B of m bits $B[0], B[1], \ldots, B[m-1]$, with all bits initialized to zeros
  - To insert a data item d to this set, first k independent hash functions $h_0(d), h_1(d), \ldots h_{k-1}(d)$ 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 $B[0] = 0, \ldots, B[7] = 0$:



- Typical Bloom filter sizes for many applications are in the order of megabytes, 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.)

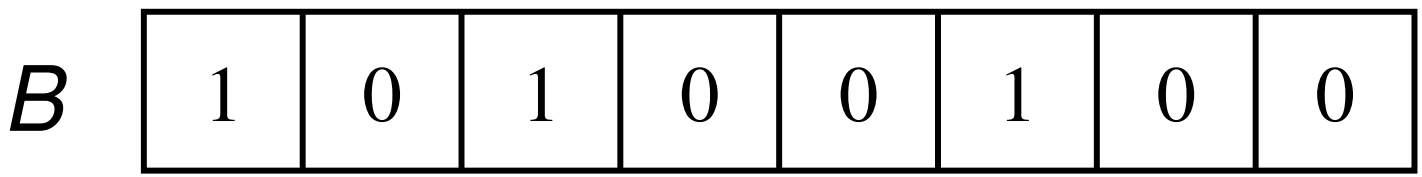
- Suppose we want to next add the data item `http://www.aalto.fi/` into the Bloom filter B
- Calculate $k = 2$ hash functions, assume we get:
  - $i_0 = h_0(\texttt{http://www.aalto.fi/}) = 5$
  - $i_1 = h_1(\texttt{http://www.aalto.fi/}) = 0$
- As $B[i_1] = B[0] == 0$ before insertion, we know `http://www.aalto.fi/` 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:



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 access to a large 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 store 10 million 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 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, 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
- 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