Scalable Cloud Computing

Keijo Heljanko

Department of Information and Computer Science
School of Science
Aalto University
keijo.heljanko@aalto.fi

10.10-2012

Guest Lecturer

- Guest Lecturer: Prof. Keijo Heljanko, Department of Information and Computer Science, Aalto University,
  - Email: `keijo.heljanko@aalto.fi`
  - Homepage: `http://users.ics.tkk.fi/kepa/`
- For more info into today's topic, attend the course: "T-79.5308 Scalable Cloud Computing"

Business Drivers of Cloud Computing

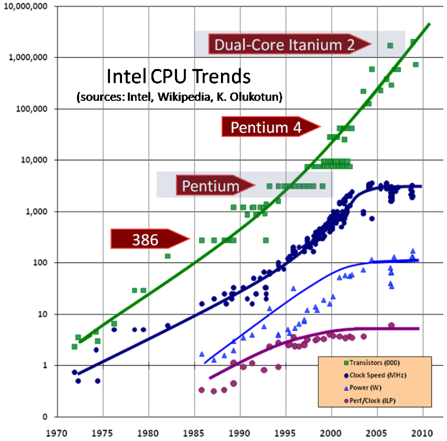
- Large data centers allow for economics of scale
  - Cheaper hardware purchases
  - Cheaper cooling of hardware
    - Example: Google paid 40 MEur for a Summa paper mill site in Hamina, Finland: Data center cooled with sea water from the Baltic Sea
  - Cheaper electricity
  - Cheaper network capacity
  - Smaller number of administrators / computer
- Unreliable commodity hardware is used
- Reliability obtained by replication of hardware components and a combined with a fault tolerant software stack

Cloud Computing Technologies

A collection of technologies aimed to provide elastic “pay as you go” computing

- Virtualization of computing resources: Amazon EC2, Eucalyptus, OpenNebula, Open Stack Compute, ...
- Scalable file storage: Amazon S3, GFS, HDFS, ...
- Scalable batch processing: Google MapReduce, Apache Hadoop, PACT, Microsoft Dryad, Google Pregel, Spark, ...
- Scalable datastore: Amazon Dynamo, Apache Cassandra, Google Bigtable, HBase,...
- Distributed Coordination: Google Chubby, Apache Zookeeper, ...
- Scalable Web applications hosting: Google App Engine, Microsoft Azure, Heroku, ...

Clock Speed of Processors



- Herb Sutter: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3), March 2005 (updated graph in August 2009).

Implications of the End of Free Lunch

- The clock speeds of microprocessors are not going to improve much in the foreseeable future
  - The efficiency gains in single threaded performance are going to be only moderate
- The number of transistors in a microprocessor is still growing at a high rate
  - One of the main uses of transistors has been to increase the number of computing cores the processor has
  - The number of cores in a low end workstation (as those employed in large scale datacenters) is going to keep on steadily growing
- Programming models need to change to efficiently exploit all the available concurrency - scalability to high number of cores/processors will need to be a major focus

Scaling Up vs Scaling Out

- Scaling up: When the need for parallelism arises, a single powerful computer is added with more CPU cores, more memory, and more hard disks
- Scaling out: When the need for parallelism arises, the task is divided between a large number of less powerful machines with (relatively) slow CPUs, moderate memory amounts, moderate hard disk counts
- Scalable cloud computing is trying to exploiting scaling out instead of scaling up

Warehouse-scale Computing (WSC)

- The smallest unit of computation in Google scale is: Warehouse full of computers
- [WSC]: Luiz André Barroso, Urs Hölzle: *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* Morgan & Claypool Publishers 2009

  http://dx.doi.org/10.2200/S00193ED1V01Y200905CAC006
- The WSC book says:
  "…we must treat the datacenter itself as one massive warehouse-scale computer (WSC)."

Jeffrey Dean (Google): LADIS 2009 keynote failure numbers

- LADIS 2009 keynote: “Designs, Lessons and Advice from Building Large Distributed Systems” `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`
- At warehouse-scale, failures will be the norm and thus fault tolerant software will be inevitable
- Typical yearly flakiness metrics from J. Dean (Google, 2009):
  - 1-5% of your disk drives will die
  - Servers will crash at least twice (2-4% failure rate)

Big Data

- As of May 2009, the amount of digital content in the world is estimated to be 500 Exabytes (500 million TB)
- EMC sponsored study by IDC in 2007 estimates the amount of information created in 2010 to be 988 EB
- Worldwide estimated hard disk sales in 2010: $\approx$ 675 million units
- Data comes from: Video, digital images, sensor data, biological data, Internet sites, social media, ...
- The problem of such large data massed, termed Big Data calls for new approaches to storage and processing of data

Google MapReduce

- A scalable batch processing framework developed at Google for computing the Web index
- When dealing with Big Data (a substantial portion of the Internet in the case of Google!), the only viable option is to use hard disks in parallel to store and process it
- Some of the challenges for storage is coming from Web services to store and distribute pictures and videos
- We need a system that can effectively utilize hard disk parallelism and hide hard disk and other component failures from the programmer

Google MapReduce (cnt.)

- MapReduce is tailored for batch processing with hundreds to thousands of machines running in parallel, typical job runtimes are from minutes to hours
- As an added bonus we would like to get increased programmer productivity compared to each programmer developing their own tools for utilizing hard disk parallelism

Google MapReduce (cnt.)

- The MapReduce framework takes care of all issues related to parallelization, synchronization, load balancing, and fault tolerance. All these details are hidden from the programmer
- The system needs to be linearly scalable to thousands of nodes working in parallel. The only way to do this is to have a very restricted programming model where the communication between nodes happens in a carefully controlled fashion
- Apache Hadoop is an open source MapReduce implementation used by Yahoo!, Facebook, and Twitter

MapReduce and Functional Programming

- Based on the functional programming in the large:
  - User is only allowed to write side-effect free functions "**Map**" and "**Reduce**"
  - Re-execution is used for fault tolerance. If a node executing a Map or a Reduce task fails to produce a result due to hardware failure, the task will be re-executed on another node
  - Side effects in functions would make this impossible, as one could not re-create the environment in which the original task executed
  - One just needs a fault tolerant storage of task inputs
  - The functions themselves are usually written in a standard imperative programming language, usually Java

Why No Side-Effects?

- Side-effect free programs will produce the same output irregardless of the number of computing nodes used by MapReduce
- Running the code on one machine for debugging purposes produces the same results as running the same code in parallel
- It is easy to introduce side-effect to MapReduce programs as the framework does not enforce a strict programming methodology. However, the behavior of such programs is undefined by the framework, and should therefore be avoided.

Yahoo! MapReduce Tutorial

- We use a Figures from the excellent MapReduce tutorial of Yahoo! [YDN-MR], available from: `http://developer.yahoo.com/hadoop/tutorial/module4.html`
- In functional programming, two list processing concepts are used
  - Mapping a list with a function
  - Reducing a list with a function

Mapping a List

Mapping a list applies the mapping function to each list element (in parallel) and outputs the list of mapped list elements:

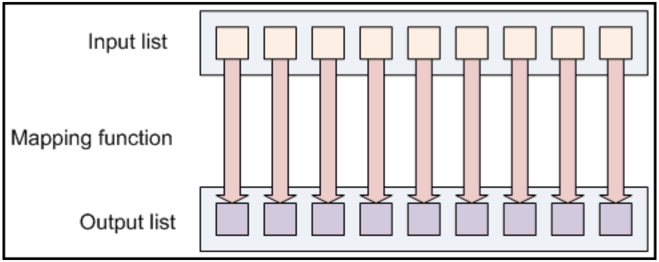


Figure: Mapping a List with a Map Function, Figure 4.1 from [YDN-MR]

Reducing a List

Reducing a list iterates over a list sequentially and produces an output created by the reduce function:

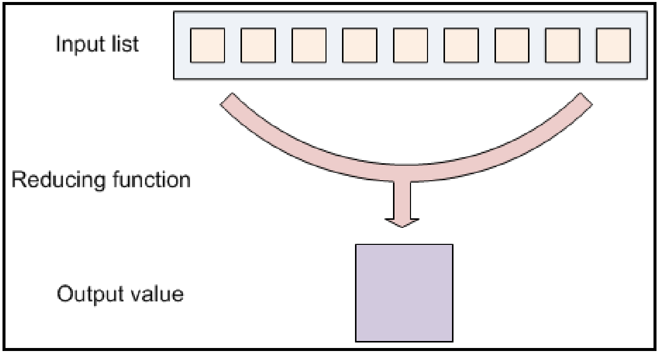


Figure: Reducing a List with a Reduce Function, Figure 4.2 from [YDN-MR]

Grouping Map Output by Key to Reducer

In MapReduce the map function outputs `(key, value)`-pairs. The MapReduce framework groups map outputs by key, and gives each reduce function instance `(key, (..., list of values, ...))` pair as input. Note: Each list of values having the same key will be independently processed:

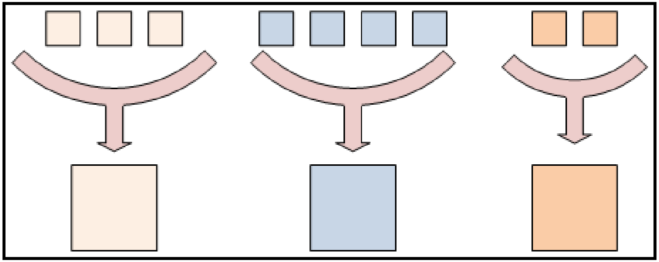
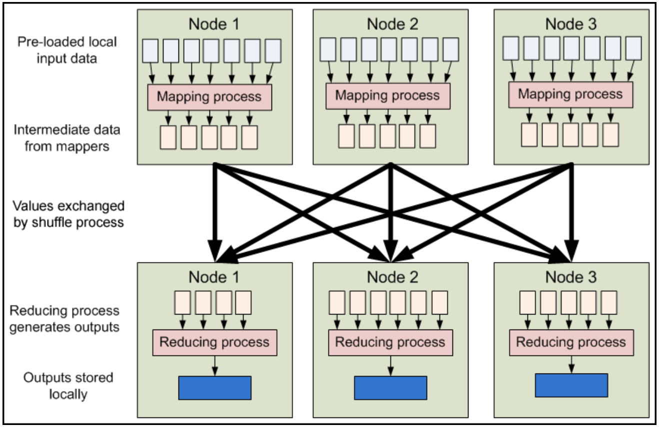


Figure: Keys Divide Map Output to Reducers, Figure 4.3 from [YDN-MR]

MapReduce Data Flow

Practical MapReduce systems split input data into large (64MB+) blocks fed to user defined map functions:



Node 1
Node 2
Node 3
Pre-loaded local input data
Mapping process
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Reducing process
Outputs stored locally

Figure: High Level MapReduce Dataflow, Figure 4.4 from [YDN-MR]

Recap: Map and Reduce Functions

- The framework only allows a user to write two functions: a "**Map**" function and a "**Reduce**" function
- The **Map**-function is fed blocks of data (block size 64-128MB), and it produces `(key, value)` -pairs
- The framework groups all values with the same key to a `(key, (..., list of values, ...))` format, and these are then fed to the **Reduce** function
- A special Master node takes care of the scheduling and fault tolerance by re-executing Mappers or Reducers

Google MapReduce

- The user just supplies the Map and Reduce functions, nothing more
- The only means of communication between nodes is through the shuffle from a mapper to a reducer
- The framework can be used to implement a distributed sorting algorithm by using a custom partitioning function
- The framework does automatic parallelization and fault tolerance by using a centralized Job tracker (Master) and a distributed filesystem that stores all data redundantly on compute nodes
- Uses functional programming paradigm to guarantee correctness of parallelization and to implement fault-tolerance by re-execution

Apache Hadoop

- An Open Source implementation of the MapReduce framework, originally developed by Doug Cutting and heavily used by e.g., Yahoo! and Facebook
- "Moving Computation is Cheaper than Moving Data" - Ship code to data, not data to code.
- Map and Reduce workers are also storage nodes for the underlying distributed filesystem: Job allocation is first tried to a node having a copy of the data, and if that fails, then to a node in the same rack (to maximize network bandwidth)
- Project Web page: `http://hadoop.apache.org/`

Apache Hadoop (cnt.)

- Tuned for large (gigabytes of data) files
- Designed for very large 1 PB+ data sets
- Designed for streaming data accesses in batch processing, designed for high bandwidth instead of low latency
- For scalability: NOT a POSIX filesystem
- Written in Java, runs as a set of user-space daemons

Hadoop Distributed Filesystem (HDFS)

- A distributed replicated filesystem: All data is replicated by default on three different Data Nodes
- Inspired by the Google Filesystem
- Each node is usually a Linux compute node with a small number of hard disks (4-12)
- A single NameNode that maintains the file locations, many DataNodes (1000+)

Hadoop Distributed Filesystem (cnt.)

- Any piece of data is available if at least one datanode replica is up and running
- Rack optimized: by default one replica written locally, second in the same rack, and a third replica in another rack (to combat against rack failures, e.g., rack switch or rack power feed)
- Uses large block size, 128 MB is a common default - designed for batch processing
- For scalability: Write once, read many filesystem

Implications of Write Once

- All applications need to be re-engineered to only do sequential writes. Example systems working on top of HDFS:
  - HBase (Hadoop Database), a database system with only sequential writes, Google Bigtable clone
  - MapReduce batch processing system
  - Apache Pig and Hive data mining tools
  - Mahout machine learning libraries
  - Lucene and Solr full text search
  - Nutch web crawling

Two Large Hadoop Installations

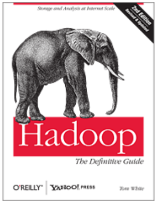
- Yahoo! (2009): 4000 nodes, 16 PB raw disk, 64TB RAM, 32K cores
- Facebook (2010): 2000 nodes, 21 PB storage, 64TB RAM, 22.4K cores
  - 12 TB (compressed) data added per day, 800TB (compressed) data scanned per day
  - A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, H. Liu: *Data warehousing and analytics infrastructure at Facebook*. SIGMOD Conference 2010: 1013-1020.
    `http://doi.acm.org/10.1145/1807167.1807278`

Apache Pig

- Apache Pig is a scripting language that compiles scripting language into a Java MapReduce program
- Apache Pig allows for much improved programmer productivity over writing MapReduce programs in Java
- For more information, see: `http://pig.apache.org/`
- Apache Hive (`http://hive.apache.org/`) is another high-level language for expressing MapReduce programs. Unlike Pig that looks like a scripting language, Hive is closely related to SQL

Hadoop Books

- I can warmly recommend the Book:
  - Tom White: “Hadoop: The Definitive Guide, Second Edition”, O’Reilly Media, 2010. Print ISBN: 978-1-4493-8973-4, Ebook ISBN: 978-1-4493-8974-1. `http://www.hadoopbook.com/`



- An alternative book is:
  - Chuck Lam: “Hadoop in Action”, Manning, 2010. Print ISBN: 9781935182191.

Commercial Hadoop Support

- Cloudera: A Hadoop distribution based on Apache Hadoop + patches. Available from:
  `http://www.cloudera.com/`
- Hortonworks: Yahoo! spin-off of their Hadoop development team, will be working on mainline Apache Hadoop.
  `http://www.hortonworks.com/`
- MapR: A rewrite of much of Apache Hadoop in C++, including a new filesystem. API-compatible with Apache Hadoop.
  `http://www.mapr.com/`

Many systems also support Apache Pig (high-level scripting language for MapReduce)

Cloud Storage - Node Local Storage

There are many different kinds of storage systems employed in clouds:

- Node local storage
  - Local hard disks
  - Often used for temporary data and binaries
  - High performance but quite often storage needs to be over-provisioned, as unused storage can not be easily shared to other nodes
  - Even if RAID is used for data redundancy, the server itself is a single point of failure

Cloud Storage - Network Block Device

- Network block devices
  - Examples: Amazon EBS, Ceph RBD (Rados block device), Sheepdog project
  - Virtual block devices accessed over the network, often with RAID 1-like durability (mirroring)
  - Many systems only allow mounting each block device by one client. Can be made very scalable by serving different clients by different storage servers
  - More traditional alternatives with less automatic management: Linux DRBD (Distributed Replicated Block Device), SAN storage over iSCSI

Cloud Storage - POSIX Filesystems

- POSIX filesystems
  - Examples: NFS, Lustre
  - Shared POSIX filesystems accessed by several clients
  - Often of quite limited scalability due to sharing the metadata of a distributed namespace between clients. Metadata updates are handled by a single server
  - High end storage solutions are not cheap - this approach is using scaling up strategy instead of scaling out

Cloud Storage (cnt.)

- Distributed non-POSIX filesystems
  - Examples: GFS, HDFS
  - Tailored for big files and write-once-read-many (WORM) workloads. Are very scalable in these usage scenarios
  - Both GFS and HDFS are moving towards having more than one metadata server (GFS v2, Hadoop 0.23), as currently this it still is the bottleneck in large (1000+) node clusters

Cloud Storage (cnt.)

- Scalable object stores
  - Example: Amazon S3, Openstack Swift (developed by Rackspace)
  - S3 is a geographically replicated storage system, each data item is stored in at least two geographically remote datacenters by default
  - No nested filesystem directory hierarchy available: Objects are stored in buckets. Each stored item is accessed by a unique identifier
  - Amazon S3 has been designed for 99.999999999% durability and 99.99% availability of objects over a given year. One needs geographic replication to achieve the durability goal

Cloud Datastores (Databases)

- Quite often the term 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

Consistency & Availability in Distributed Databases

We define the three general properties of distributed databases popularized by Eric Brewer:

- Consistency (as defined by Brewer): All nodes have a consistent view of the contents of the (distributed) database
- Availability: A guarantee that every database request eventually receives a response about whether it was successful or whether it failed
- Partition Tolerance: The system continues to operate despite arbitrary message loss

Brewer's CAP Theorem

- In a PODC 2000 conference invited talk Eric Brewer made a conjecture that it is impossible to create a distributed asynchronous system that is at the same time satisfies all three CAP properties:
  - Consistency
  - Availability
  - Partition tolerance
- This conjecture was proved to be a Theorem in the paper: "Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News, 33(2):51-59, 2002."

Brewer's CAP Theorem

Because it is impossible to have all three, you have to choose between two of CAP:

- CA: Consistent & Available but not Partition tolerant
  - A non-distributed (centralized) database system
- CP: Consistent & Partition Tolerant but not Available
  - A distributed database that can not be modified when network splits to partitions
- AP: Available & Partition Tolerant but not Consistent
  - A distributed database that can become inconsistent when network splits into partitions

Example CA Systems

- Single-site (non-distributed) databases
- LDAP - Lightweight Directory Access Protocol (for user authentication)
- NFS - Network File System
- Centralized version control - Svn
- HDFS Namenode

These systems are often based on two-phase commit algorithms, or cache invalidation algorithms.

Example CP Systems

- Distributed databases - Example: Google Bigtable, Apache HBase
- Distributed coordination systems - Example: Google Chubby, Apache Zookeeper

The systems often use a master copy location for data modifications combined with pessimistic locking or majority (aka quorum) algorithms such as Paxos by Leslie Lamport.

Example AP Systems

- Filesystems allowing updates while disconnected from the main server such as AFS and Coda.
- Web caching
- DNS - Domain Name System
- Distributed version control system - Git
- “Eventually Consistent” Datastores - Amazon Dynamo, Apache Cassandra

The employed mechanisms include cache expiration times and leases. Sometimes intelligent merge mechanisms exists for automatically merging independent updates.

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)
- 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)

A Grouping of Data Stores

- Key-value stores
  - Examples: Project Voldemort, Redis, Scalaris, Apache Cassandra (partially)
  - 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
  - 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

A Grouping of Data 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

A Grouping of Data Stores - 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)

A 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
- Example systems: Amazon Dynamo, Apache Cassandra

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. Counterexample: Google Percolator
- 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
- 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

BigTable

- Described in the paper: “Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber: Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26(2): (2008)”
  - A highly scalable consistent and partition tolerant datastore
  - Implemented on top of the Google Filesystem (GFS)
  - GFS provides data persistence by replication but only supports sequential writes
  - BigTable design does no random writes, only sequential writes are used
- Open source clone: Apache HBase

BigTable Data Model

- Bigtable paper describes itself as: ”…a sparse, distributed, persistent multi-dimensional sorted map”
- Namely, BigTable stores data as strings, which can be accessed by three coordinates:
  - `row` - an arbitrary string row key (10-100 bytes typical, max 64 KB)
  - `column` - a column key, consisting of a column family and column qualifier (name) in syntax `family:qualifier`
  - `timestamp` - a 64 bit integer that can be used to store a timestamp
  - Thus we have a map with three coordinates: `(row:string, column:string, time:int64) -> string`

BigTable Conclusions

- BigTable is a scalable write optimized database design
- It uses only sequential writes for improved write performance
- For read intensive workloads BigTable can be a good fit if all of the working set fits into DRAM
- For read intensive working sets much larger than DRAM, traditional RDBMS systems are still a better match

Apache HBase

- Apache HBase is an open source Google BigTable clone
- It very closely follows the BigTable design but has the following differences
  - Instead of GFS, HBase runs on top of HDFS
  - Instead of Chubby, HBase uses Apache Zookeeper
  - SSTable of BigTable are called in HBase HFile (and HFile V2)
  - HBase only partially supports fully memory mapped data

Distributed Coordination Services

- Maintaining a global database image under node failures is a very subtle problem, see FLP Theorem in: “Michael J. Fischer, Nancy A. Lynch, Mike Paterson: Impossibility of Distributed Consensus with One Faulty Process J. ACM 32(2): 374-382 (1985)”
- The Paxos algorithm is a very tricky algorithm that will be able to replicate a distributed database if enough servers are up and running

Distributed Coordination Services (cnt.)

- From an applications point of view the distributed coordination services should be used to store global shared state: Global configuration data, global locks, live master server locations, live slave server locations, etc.
- One should use centralized infrastructure such as Google Chubby or Apache Zookeeper to only implement these tricky fault tolerance algorithms only once