Cloud Storage

**Definition of Big Data:**

Capture, Process, analyse and visualize a large Dataset in a normal Timeframe. Platform, Tools and Software used are Big Data Technologies.

4 important Vs:

* Volume: Increase Amount of generated Data
* Velocity: Analyse more Data in shorter times
* Variety: More different Data Formats
* Value: Extracting value from data

There are more but the 4 are the most iconic. You want to create Value out of your Data

**The Big Data Technology Stack**

1. Big Data Management
   1. Scalable Data Storage
   2. What Big Data Centers Do
   3. Utilizes Virtualization, IaaS, DBMS, NoSQL
2. Big Data Platform
   1. Hadoop Ecosystems
   2. Data Ingestion and processing
   3. Utilizes Efficiency, Workload, Tools, Parallel
3. Big Data Analytics
   1. Data Science
   2. Transform question to algorithm
   3. Utilizes Machine Learning, Query’s, Transforms, Warehouses
4. Big Data Utilization
   1. Domain Expertise
   2. Asking the right questions
   3. Utilizes Reporting, Dashboards, Alerts, Analytics and Search

Public Cloud Infrastructure build in Datacenters. Range from “Edge” to megascale. Cooling as most expensive factor (50% of power) so they are placed in cold regions (Iceland). Utilizes Economie of Scale: The bigger the datacentre the cheaper the costs.

**Foundation of Cloud Computing**

Based on ideas and research of parallel computing and Distributed systems. Client-Server Paradigm with Thin-client and the computation done with many computers in the cloud in parallel.

Parallel Computing allow solve problems that need ressources beyond a single system. It also reduces the time required to obtain a solution.

Speedup is Calculated by Sequential Runtime / Parallel Runtime with N Parallel Computations.

Amdahls Law:

Speedup when problem/dataset is fixed. You can split the execution in a% sequential part and (1-a)% parallel Part. The lower a, the higher the Speedup. Its only limited by the sequential part. (If the parallel processors doesn’t need to communicate, if they communicate they have an Overhead that can influence the Speedup). Ideal: Double N -> Double Speedup (1/N)

Gustafsons law:

Computation Time is fixed but dataset/problem is scalable. A big enough problem can be efficient run in parallel. Each processor has the same amount of Work. If you add more processors, you can work on double the data. Ideal: Double Processors -> Double Work.

**Parallelism:**

* Data Parallelism: Data is split into chunks and worked on with the same Program in parallel
* Divide the Data Principle

**Distributed Systems:**

Different possibilities to build:

* Single CPU, Single Disk
* Single CPU, Many Disks (Parallel Read)
* Many CPU, Many Disk, Network Connected (Distributed Reads, Extendible set of Servers)

Distributed Systems are ideal for cata-centric applications.

* *Disk transfer rate*. Bottleneck if you have a single Disk, Distribute and parallelize on many machines to eliminate that bottleneck
* *Write once, Read Many*: Write big Files once and the read chunk by chunk
* *Data Locality,* Push the Data closest to the process that handles the data.

**Replication and Consistency**

Distribution can reinforce security of Data with replications. You copy the the Data to another machine. If you add Consistency, the data is the same on all machines.

Consistency is the ability of a system to behave as if the transaction of each user always run in isolation from other transactions and never fail.

Consistency is hard due to multi-user, Concurrency and Replicas

You either have a Primary Copy where every write happens with Synchronous or Asynchronous Replication (Master->Slave Pattern) or Distributed Replication where you can write on every replica with Synchronous and Asynchronous Replication (Master->Master)

**Consistency Management**

Strong Consistency:

Slow with Synchronous Replicas. Favours Consistency in cost of Availability. Blocks User Interference until its consistent again. SQL uses strong consistency (ACID properites). NoSQL breaks the Strong Consistency

Weak Consistency: User can use Data without perfect Consistency. It allows to work with outdated data. If enough time passes and no transactions happen, the data is consistent again (Eventually Consistency). If the update doesn’t reach a Node in Time, the node is removed from service.

**Availability**

* Limit Latency in System
* Fast recovery of Nodes
* Synchronous or Asynchrounus
* Replication of Nodes to ensure Availability

Usually you go for the five 9 (99,999% Availability). The more nodes you have the higher the risk of network/node failures, availability goes down.

**Partition Tolerance**

Scale UP: Take existence Storage system and add more capacity

Scale Out: Add more Nodes but still represent as single system

**CAP Theorem:**

You have 3 Features: Consistency, Availability, Partition Tolerance

You can only have at maximum of 2 at the same time

If you want to have consistency and Availability, you need to work on a single node. Data doesn’t need to be updated somewhere else so its consistency and Available all the time. But you loose the Partition Tolerance

If you want to be consistency and Partition Tolerance, you need to block User interference to ensure the data is updated on each Node. But then your System is not available when you updating the Data

If you want to be Available and Partition Tolerant, you need to allow eventually consistency. User can work on outdated data which invalids consistency

**Relational Databases**

RDBMS (Relational Data Base Management System)

* Stores Data as collection of Tables
* Relational Operates to manipulate Data
* SQL as Query Language since all commercial relational databases use SQL

RDBMS challenged by scalability

* System is Master -> Slave. Writes to Master, Reads from replicated Slaves
* Critical Reads can be wrong if change didn’t propagateded
* Large Data can slow down since Master solely needs to duplicate Data

Solution? Partitioning or Sharding perhaps`?

* Not transparent, app needs to be aware of partitioning
* No relations/joins across partitions
* Loss of referential integrity (Cascade Delete for example)

Traditional Scale Out approach

* More Slave Databases
* In-memory cache for reads (on many reads) -> Hurts Consistency
* Beef up Master (More Power to Master, Vertically Scale)
* Make Slaves more Powerfull
* Denormalize Data -> Loose Joins, no Stored procedues which are costly
* Store Data optimized for your use Case

Why not design a new System to handle this kind of prerequisites? 🡪 NoSQL!

**Not Only SQL Databases (NoSQL)**

Some Characteristics include:

* Class of non-relational datastorage systems
* No Joins
* Not a fixated table schema
* Relax on one or more ACID Properties
  + Atomicity (All changes happen or none)
  + Consistency (Data is the same on each Node)
  + Isolation (Intermediate Transactions are invisible to other Transactions)
  + Durability (Changes to Data persist and stays so even on system failure)
* **Distributed Fault Tolerant Architecture 🡪 Scalable**

How to access Data and store? -> Indexing

Comes in 2 flavors:

* Hashing based techniques like Consistent Hashing
* Tree based techniques like Distributed B-Tree

Definition of Indexing

Very large collection of key-value pairs visualized as row data. An Index associates the key with the physical address of the value.

Indexing should support the 4 dictionary operations

* Insertion *insert(k, v)*
* Deletion *delete(k)*
* Key Search *search(k)*
* Range Search *range(k1, k2)*
* In Distributed Index there should also a node leave and node join operator. To rehash the stored vales

**Distributed Hashing**

Problem of Centralized Hashing. Everyone uses the same hash and the buckets the data goes in are replaced with servers. This does work until a Server leaves the system. All data is now mapped (hashed) incorrectly. You need to rehash everything to make the system work again. We cannot make the hash function dependent on Server Count h(k) mod N (Server Count)

Solution:

Also map the Address of the Server with the same hash function and place it in a Ring. Each server is now, like the key-value, hashed/mapped to the ring. If a new value comes into the ring, it hashes first and looks for the closest previous server (also hashed) to save it there. *So, if S and S’ are adjected server all keys in range [h(S], h(S’)] are mapped to S*

If a server fails, you can use replicas to serve a copy of the data to the ring aswell

You balance the load with multiple Server on the Ring and by using a hash function that guarantees an even distribution on the Address Space.

If a Server Fails and there is no replication, only the data of the failed server needs to be rehashed and not the whole system.

Hash Directory

Someone needs to keep track what is on the Ring

* 1 Master Node which keeps track on every movement of the ring and acts accordingly. This raises some scalability problems
* Each Node has information about its successor. Could require O(N) messages to get to the right Node
* Each Node records log N choosen Nodes. Reduces to O(logN) messages to route queries.
* Each Node knows the full ring
  + O(1) as routing query
  + Heavy to update. (Via Gossiping)

Amazon DB

Uses Consistent Hashing. Features include:

* Each Node holds the Hash Directoy and is updated via gossiping
* Hosts N Replicas per Node that come online after Server fail
* Updates are asynchrony. Update conflicts are solved by application at read
* Each node can detect failure in other nodes when they try to communicate

**B-Tree Hashing**

How to distribute a B-tree. Its not scalable cause all operations follow top-down path

* Cache the Tree structure on client nodes
* Replicate the upper levels of the trees
* Routing Tables, stored at each node so you can navigate horizontally and vertically

Approaches to solve this are BigTable and HBase

* Can be seen as distributed map structure
* One Master and Many Slaves
  + Master do the Administrative tasks and maintains the root node of the tree
  + Slaves (Servers) contain a Cache of stored Data. They can be outdated
  + Ajustments to Cache require at most (height) Tree rounds

HBase

* Data is stored in Column Families rather than rows.
* Each Family is stored in a single File
* You can version the Columns
* Based on Distributed File Systems
* IN-memory Cache
* Region Servers for Data Distribution
* Uses Hadoop Distributed File System

You can Query with the following types

* Range Scan (Start and end row Key)
* Get (row key)
* Full Scan