**Datamanagement in the Cloud**

1. **Introduction**

***Recap ACID:***

* **Atomicity:** “all or nothing” the whole transaction or no transaction are committed
* **Consistency:** transactions never observe or result in inconsistent data
* **Isolation:** transactions are not aware of concurrent transactions
* **Durability:** once committed, the state of a transaction is permanent

***Cloud Computing is:***

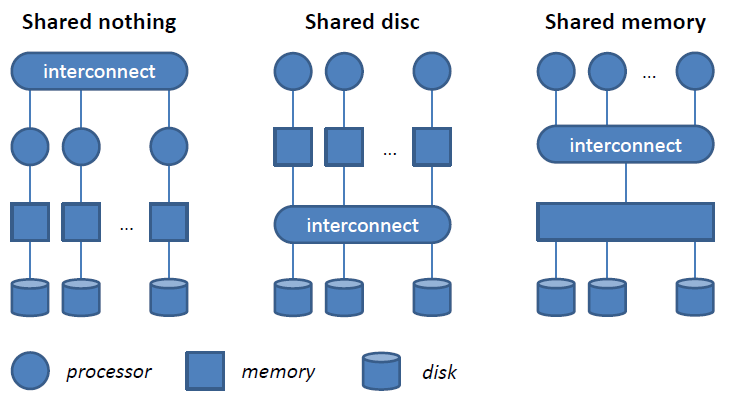
* “Software as a Service (SaaS)”: applications delivered over the Internet as services
* a pay-as-you-go-model
* not an internal data center
* utility computing
  + illusion of infinite computing resources
  + no up-front cost or commitment by users
  + pay for use on short-term basis in need
* virtualization:
  + virtual resources abstract from physical resources
  + centralize and ease administrative tasks
  + improve scalability and work loads
  + increase stability and fault-tolerance
  + provide standardized, homogenous computing platform through hardware virtualization

***Six key features:***

1. ability to **horizontally scale simple operation** throughput over many servers

Simple operation:

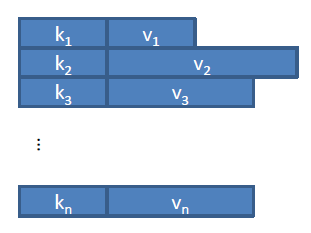
* only key lookups
* read and writes of one or a small number of records
* no complex queries or joins

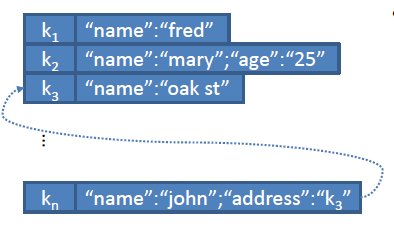
1. ability to **replicate and distribute** (partition) data over many servers
2. simple **call level interface** or protocol
3. **weaker concurrency model** than ACID transactions of most relational DBS
4. efficient use of **distributed indexes and RAM**  for data storage
5. ability to **dynamically add new attributes** to data records
   1. **Scaling Databases**

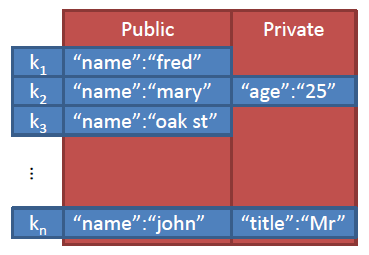
* ***Horizontal:*** Add more nodes
* ***Vertical:*** up-size existing node by adding CPUs, memory
* Move data to where it is needed
* Manage replication for availability and reliability
  1. **Data partitioning**
* ***Horizontal:*** distribute groups of tuples of a relation onto different nodes
* ***Vertical:*** distribute groups of columns of a relation onto different nodes
* ***“sharding” -> horizontal***
  1. **Data Models**

***Terminology***

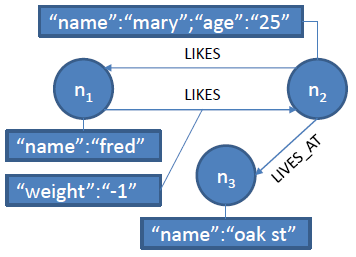
* **Tuple:** row in a relational table, where attribute names and types are defined by a schema, and values must be scalar
* **Document:** supports both scalar values and nested documents, and the attributes are dynamically defined for each document
* **Column family:** groups key/value pairs (columns) into families to partition and replicate them; one column family is similar to a document as new (nested, list-valued) attributes can be added
  + 1. ***Key/Value Data Model***



* Interface:
  + put(key, value)
  + get(key): value
* Data Storage:
  + Values (data) are stored based on programmer-defined keys
  + System does not (need to) know about the structure (semantics) of the value
* Queries are expressed in terms of keys
* Indexes are defined over keys:
  + Some systems support secondary indexes over (part of) the value
    1. ***Document Data Model***
* Interface:
  + set(key, document)
  + get(key): document
  + set(key, name, value)
  + get(key, name): value
* Data storage:
  + Documents (data) is stored based on programmer-defined keys
  + System is aware of the (arbitrary) document structure
  + Support for lists, pointers and nested documents
* Queries expressed in terms of key (or attribute, if index exists)
* Support for key-based indexes and secondary indexes



* + 1. ***Column Family Data Model***
* Interface:
  + define(family)
  + insert(family, key, columns)
  + get(family, key): columns
* Data storage:
  + <name, value, timestamp> triples (so-called columns) are stored based on a column family and key; a column family is similar to a document
  + System is aware of (arbitrary) structure of column family
  + System uses column family information to replicate and distribute data
* Queries are expressed based on key and columns family
* Secondary indexes per column family are typically supported



* + 1. ***Graph Data Model***
* Interface:
  + create: id
  + get(id)
  + connect(id1, id2): id
  + addAttribute(id, name, value)
  + getAttribute(id, name): value
* Data Storage:
  + Data is stored in terms of nodes and (typed) edges
  + Both nodes and edges can have (arbitrary) attributes
* Queries are expressed based on system ids (if no indexes exists)
* Secondary indexes for nodes and edges are supported
  + Retrieve nodes by attributes and edges by type, start and/or end node, and/or attributes
  1. **Consistency Models**
     1. ***CAP Theorem***
* Three properties are desirable and expected from a real-world shared-data systems
  + **C:** data consistency
  + **A:** availability
  + **P:** tolerance of network partition
* Only two of these properties can be satisfied by a system at any given time

***Data Consistency***

* Database systems typically implement ACID transactions
* There are aplications that can deal with looser consistency guarantees and periods of inconsistency

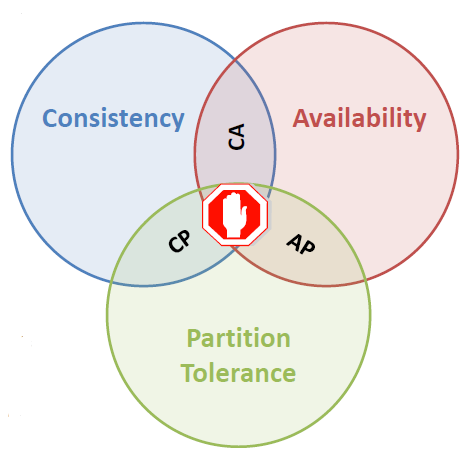
***Availability***

* Services are expected to be highly available
  + Every request should receive a response
  + It can create real-world problems when a service goes down
* Realistic goal
  + Service should be as available as the network it run on
  + If any service on the network is available, the service should be available

***Partition-Tolerance***

* A service should continue to perform as expected if some nodes crash or some communication links fail
* One desirable fault tolerance property is resilience to a network partitioning into multiple components

***Classification of Systems***

* Available-Partition-Tolerant:
  + Dynamo, Riak, Voldemort, TokioCabinet
  + SimpleDB, CouchDB
  + Cassandra
* Consistent-Available
  + RDBMS
  + GreenPlum
* Consistent-Partition-Tolerant
  + MomCacheDB. Redis, Scalaris
  + MongoDB
  + BigTable, HBase, HyperTable
  + VoltDB

***Criticism***

* Asymmetry of CAP properties:
  + Consistency is a property of the system in general
  + Availability is a property of the system only when there is a partition
* There are not three different choices
  + In practice, CA and CP are indistinguishable, since A is only sacrificed when there is a partition
  + Used as an excuse to not bother with consistency
* Other costs to consistency:
  + Overhead of synchronization schemes
  + Latency
    1. ***Strong vs. Weak Consistency***

***Strong consistency***

* After an update is committed, each subsequent access will return the update value

***Weak consistency***

* A number of conditions might need to be met before the updated value is returned
* **Inconsistency window:** period between update and the point in time when every access is guaranteed to return the updated value
  + 1. ***Eventual Consistency***
* Specific form of weak consistency
* “If no new updates are made, eventually all accesses will return the last updated values”
* In the absence of failures, the maximum size of the **inconsistency window** can be determined based on:
  + Communication delays
  + System load
  + Number of replicas
  + …

***Models of Eventual Consistency***

* Causal consistency
  + If A communicated to B that it has updates a value, as subsequent access by B will return the updated value, and a write is guaranteed to supersede the earlier write
  + Access by C that has no causal relationship to A is subject to normal eventual consistency rules
* Read-your-writes consistency
  + After updating a value, a process will always read the updated value and never see an older value
* Session consistency
  + Data is accessed in a session where read-your-writes is guaranteed
  + Guarantees do not span over sessions
* Monotonic read consistency
  + If a process has seen a particular value, any subsequent access will never return any pervious value
* Monotonic Write consistency
  + System guarantees to serialize the writes of one process
* Properties can be combined
  + E.g. monotonic read + session-level consistency
  + E.g. monotonic reads + read-your-own-writes

***Configurations***

* Definitions:
  + **N:** number of nodes that store a replica
  + **W:** number of replicas that need to acknowledge a write operation
  + **R:** number of replicas that are accessed for a read operation
* W+R> N
  + E.g. **synchronous replication**  (N=2, W=2, R =1)
  + Write set and read set always overlap
  + Strong consistency can be guaranteed through **quorum protocols**
  + Risk of reduced availability: in basic quorum protocols, operations fail if fewer than the required number of nodes respond, due to node failure
* W+R=N
  + E.g. **asynchronous replication** (N=2, W=1, R =1)
  + Strong consistency cannot be guaranteed
* R=1, W=N
  + Optimized for **read access:** single read will return a value
  + Write operation involves all nodes and risks not to succeed
* R=N, W=1
  + Optimized for **write access:** write operation involves only one node and relies on lazy (epidemic) technique to update other replicas
  + Read operation involves all nodes and returns “latest” value
  + Durability is not guaranteed in presence of failures
* W < (N+1)/2
  + Risk of conflicting writes
* W+R <= N
  + **Weak/eventual consistency**

***BASE =* B**asically **A**vailable, **S**oft state, **E**ventual Consistency