University of Basel

Chapter 3: Big Data & NoSQL

cs341 Distributed Information Systems

Heiko Schuldt, Spring Semester 2018

Overview of Chapter 3

What is Big Data?

- Currently: characterized by the "3 V's"
  - **V**olume (very large quantities of data)
  - **V**ariety (heterogeneity of data, data integration, semantic interoperability)
  - **V**elocity (dynamics, "fast" data, continuous data streams, time-series, complex event detection and processing)
- In discussion: extension to additional V's **V**eracity, **V**ariability, **V**olatility, **V**alidity, **V**isualization, **V**alue
- To come: all nouns from the Encyclopædia Britannica starting with "V"

And how big is Big Data?

- Well, it depends …

https://www.andertoons.com/math/cartoon/6517/does-this-count-as-big-data

- Facebook
  - 140⁺ billion photos in total
  - approx. 250⁺ million of new photos per day (numbers from 2013, source: Getty Images, 2013 http://www.gettyimagessites.com/iStock-infographics/Trends_in_Mobile_Photography_Infographic_2013.pdf)
- Instagram (numbers from 2015, source http://instagram.com/press/)
  - 40⁺ billion photos in total
  - approx. 80⁺ million new photos per day
- Youtube (numbers from 2014, source: http://www.marketingpilgrim.com/2014/12/in-the-next-60-seconds-300-hours-of-video-will-be-uploaded-to-youtube.html)
  - approx. 300 hrs of new videos uploaded per minute

Big Data is often only associated with Data Analytics

http://tinasgroove.com/

Overview of Chapter 3

Distributed File Systems

- In most cases, Cloud infrastructures feature dedicated distributed file systems
  - They take the particularities of Cloud infrastructures into account: commodity hardware with a large number of servers
  - Consequently, the file system needs to be robust against hardware failures
- Cloud File Systems consider the characteristics of ‘typical’ Cloud applications
  - Workload: mainly two types of reads and a special write operation
    - Large streaming reads ($\gg$ 100 KB, rather several MB)
    - Small random reads
    - Mutation: append data to files rather than updating them. Once written, files are usually kept unchanged
  - Consequences:
    - Multiple clients concurrently appending the same file → atomicity with minimal synchronization overhead
    - Modest number of very large files rather than millions of small files
    - High sustained bandwidth more important than low latency

Google File System

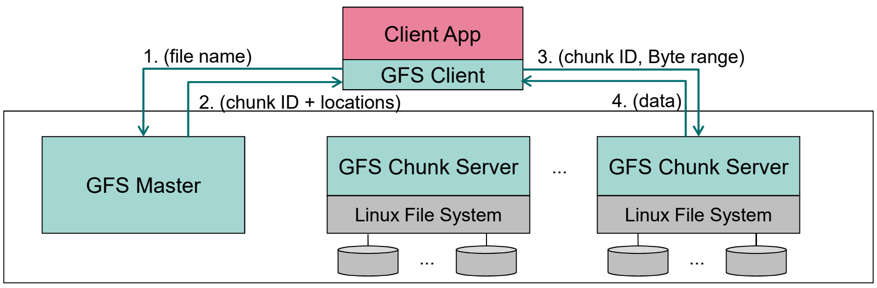
- The Google File System (GFS) is an example of such a Distributed File System tailored to Cloud data management
  - Other implementations:
    - HDFS (Hadoop Distributed File System), open source implementation according to GFS
- In addition to the standard operations (Create, Delete, Open, Close, Read, Write), GFS offers two additional operations on files
  - Snapshot (create a replica of a file or a complete directory tree)
  - Record append (concurrent append operations to the same file while guaranteeing consistency)

GFS Architecture

- a GFS Cluster consists of
  - a single master server
  - several chunk servers
- Files are subdivided into chunks of fixed size (each with a unique ID)
  - Chunks are stored locally as Linux files
  - Size: 64 MB
- Triplication: chunks are stored at least three times (→ reliability)
  - Replication controlled by master
  - Master periodically sends a heartbeat to the chunk servers to collect information on their states (→ re-locate replica if necessary, or create a new one)
- The master server
  - Keeps all metadata, including the location of chunk servers
  - Data is never read or written by client through master (→ scalability)

GFS: Data Access

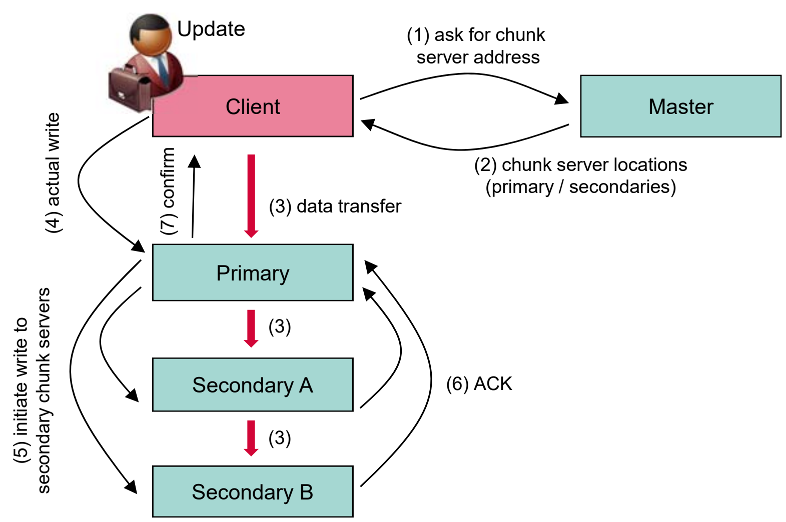
1. Client contacts master by specifying a file name
2. Master returns the addresses of all chunk servers having this file to the client
3. Client picks one of the chunk servers (the closest one) and directly asks for data (by specifying byte range)
4. Data is directly returned to the client by the chunk server
   - Usually, data is not cached at client side (but just metadata, i.e., location of chunk servers)



GFS: Consistency

- Lazy replication, but (mainly) append-only operations
  - Updates first applied at primary, later on propagated to secondary chunk servers (all done by the client)
  - Changes serialized by primary
  - Actual update decoupled from data flow (transfer of changes to primary and secondary, resp.)
  - Locking done at master server
- Distinction of different states after append operations
  - Consistent: all clients see the same data, independently of the chunk server (replica) they access
  - Defined: consistent state and append operation has been atomic
- All updates on replicas applied in the same order
- No stale chunk is returned to the client for update / append operations
  - But: as clients may cache chunk locations, they might nevertheless read premature data

GFS: Update Operations



Overview of Chapter 3

Big Data Processing: MapReduce

- MapReduce: Programming model for processing (and/or generating) very large data sets
  - Inspired by the primitives **map** and **reduce** in Lisp
  - Open source implementation: Hadoop
- Basic Map/Reduce pattern
  - Iterate over a large number of data items / records / …
  - Extract something of interest from each of them (= **map**)
  - Shuffle and sort intermediate results
  - Aggregate intermediate results (= **reduce**)
  - Generate final output
- Users only need to specify two functions:
  - **map()**: processes a key/value pair and generates a set of intermediate key/value pairs
  - **reduce()**: merges all intermediate values associated with the same key

MapReduce

- Main objective: automated parallelization on a large cluster of commodity machines
  - Partitioning the input data
  - Scheduling partitions across a set of machines (→ load balancing)
  - Handling machine failures
  - Handling inter-machine communication

MapReduce: Formal Description

- Calculates a list of key/value-pairs (l, w) as output from a list of key/value-pairs (k, v) as input

  MapReduce: $(K \times V)^* \mapsto (L \times W)^*$
  $[(k_1, v_1), (k_2, v_2), \dots, (k_n, v_n)] \mapsto [(l_1, w_1), (l_2, w_2), \dots, (l_m, w_m)]$

  with: K and L being sets of keys and V, W sets of values

… MapReduce: Formal Description

1. Map: $K \times V \mapsto (L \times W)^*$ with
   `map (k, v) → list (l, w')`
   – produces intermediate results which are of the same type as the final result
   – Map is applied to each key/value-pair of the input list

2. Reduce: $L \times W^* \mapsto W^*$ with
   `reduce (l, list (w'')) → list (w)`
   – is called for each list (identified by the key l) of the intermediate result
   – produces a set of values W* for each of these lists
   – Result is a list of keys l and associated value(s) as produced in the reduce phase

MapReduce: Example

- Sample application: counting the number of occurrences of words in a large collection of documents
  - Key: document name
  - Value: document content

```
map (String key, String value)
       // key:   document name, value: document contents
       for each word w in value
              EmitIntermediate(w, "1");

reduce (String key, Iterator values)
       // key: a word; values: list of counts
       int result = 0;
       for each v in values
              result += ParseInt(v)
       Emit(AsString(result);
```

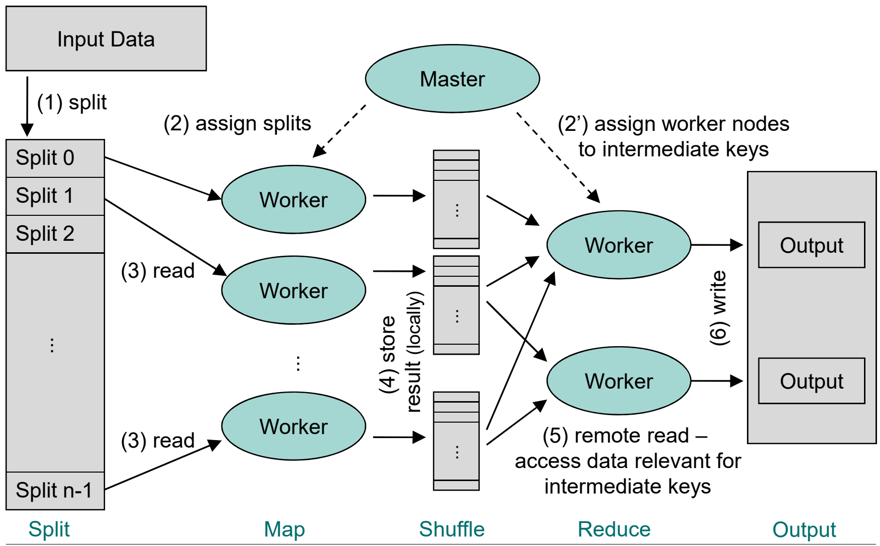
MapReduce Implementation

- Environment: commodity PCs, Linux, inexpensive local disks with GFS on top
- Map invocations:
  - distributed across multiple machines
  - Automatically partitioning input data into a set of M splits (usually between 16 MB and 64 MB per split)
  - Parallel processing of input splits by different machines
- Reduce invocations
  - Partitioning the intermediate key space into R pieces (e.g., using hash function)

MapReduce: Shuffle

- Even though Map/Reduce focuses on the two compute steps, there is an essential step, the Shuffle, in between
  - Shuffle is provided by the MapReduce runtime environment
  - It groups the results of the map phase according to the new keys and redistributes data, if necessary, across nodes
- Before the start of the reduce phase, all intermediate lists **`list (l, w')`** that share the same key **`l`** need to be grouped to **`(l, list (w"))`** and moved to the same node
  - Data distribution is transparent to the user, but it is essential for the overall performance of the MapReduce platform

MapReduce Execution: Overview

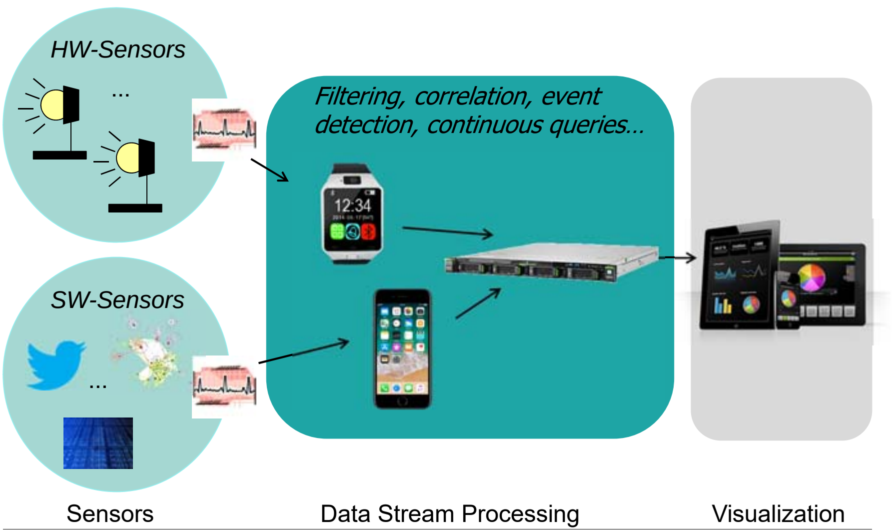


MapReduce: Additional Examples

- Inverted index
  - **`map`**: parses a document and emits **`(word, document ID)`** pairs
  - **`reduce`**: processes all pairs for a given word and emits **`(word, list(document ID))`** pair
  - Inverted index = set of all output pairs
- Reverse Web link graph
  - **`map`**: emits **`(target, source)`**pairs for each link (URL) found to a target site in the source page
  - **`reduce`**: concatenates the list of all source URLs associated with a target URL and emits **`(target, list(source))`** pair
- Count of URL Access Frequency (from web log)
  - Similar to count of word occurrences in document collection
- …

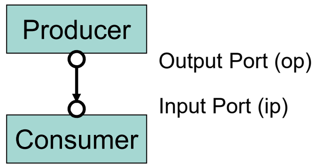
Overview of Chapter 3

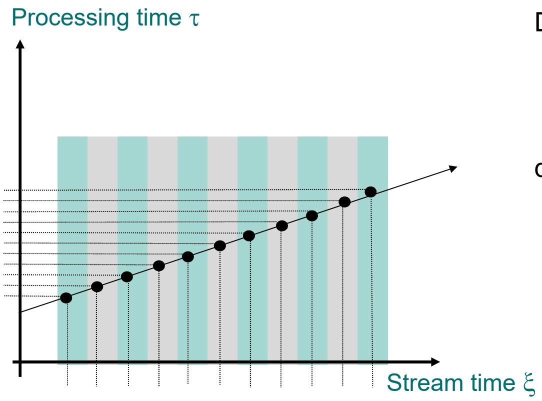
Data Stream Management – Big Picture



A Sample Data Stream Model

- Data Stream DS
  DS = ordered set of data stream elements (de), transmitted between producer and consumer





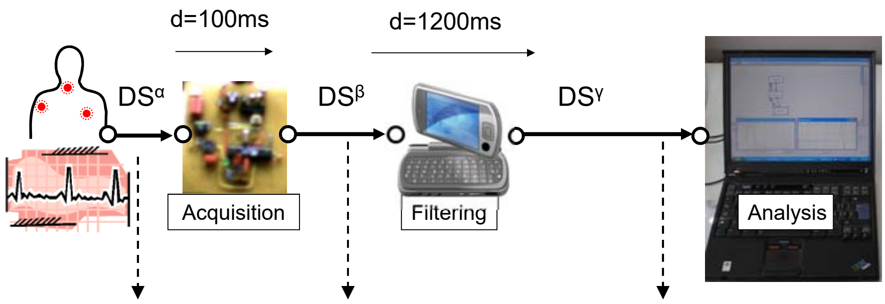
DS = ⟨ DE, <, op, ip, Σ ⟩ with

DE set of stream elements
Σ stream alphabet

de = ⟨ τ, ξ, pd ⟩ with

τ processing (actual) time
ξ stream time, defines logical order
pd ∈ Σ payload data (e.g., sensor reading)

If Time Matters – Different Kinds of Time



de_1^{α} = sensor-time
< 10:00:00.00,
1,
<sample>
<time>10:00:00</time>
<value> 10mV </value>
</sample>
>

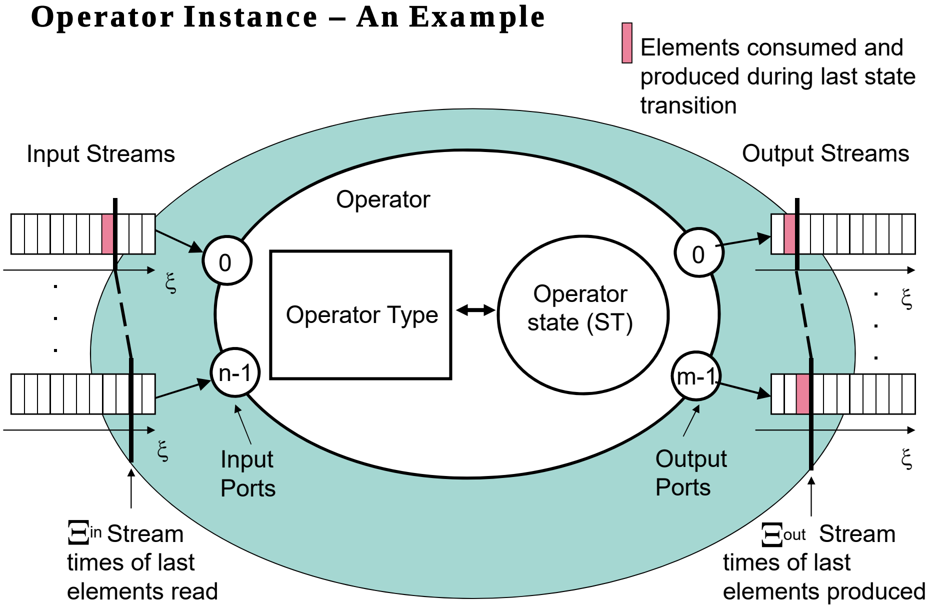
de_1^{β} = stream-time
< 10:00:00.10,
1,
<sample>
<time>10:00:00</time>
<value> 10mV </value>
</sample>
>

de_1^{γ} = processing-time
< 10:00:01.30,
1a,
<fsample>
<time>10:00:00</time>
<fvalue>7mV</fvalue>
</fsample>
>

Data Stream Operators

- Sensors continuously produce data
- Specialized basic operators for processing stream data, e.g.:
  - Signal filtering
  - Joining two or several streams
  - Correlation of streams
  - Time series analysis
  - (Complex) event detection
  - Storage of aggregated values (integration with external systems)
  - …
- Operators
  - are deterministic
  - are usually stateful

Operator Instance – An Example

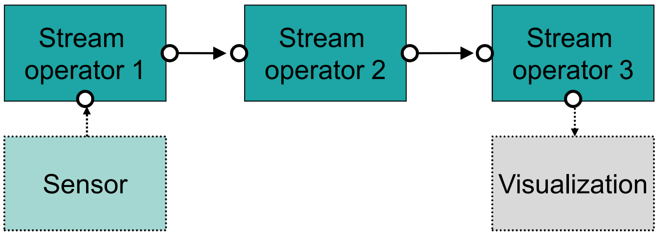


Sliding Windows

- Stream operators usually process data stream elements in sliding windows
  - Collect stream elements within a certain period of time
  - Assume some correlation if data stream elements co-exist in the same window
  - Windows continuously move forwards (i.e., old data stream elements are dropped from the window, new ones will be added)

Stream Applications

- Data stream applications usually encompass several data stream operators
- Deployed on nodes distributed within a network
- Each operator has as input one or several streams and outputs one or several streams
  - Output stream of one operator is used as input stream of a subsequent operator
  - Partial order of operators
  - Dynamic/flexible deployment

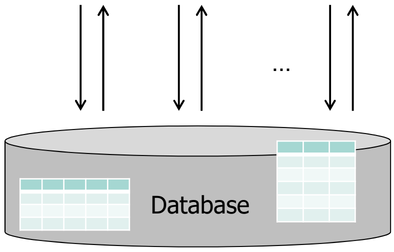


Data Stream Processing vs. Continuous Queries …

- Databases: Traditional model of databases and database queries
  - Data: persistent
  - Queries: transient
    (join the system, will be evaluated on the spot, results are returned)

Queries

```
SELECT *
FROM employees
```



Data

… Data Stream Processing vs. Continuous Queries

- Data Streams: model is put “upside down”
  - Queries: persistent (continuous queries)
  - Data: transient
    (data streams “float” through the system, will be considered as they arrive and will be ignored once they leave the sliding window)

Data

< 10:00:00.00,
1,
<temperature>28.3</temperature>
<humidity> 42 </humidity>
>

…

Queries

```
SELECT * FROM …

SELECT temperature
FROM …
…
```

Data Stream System

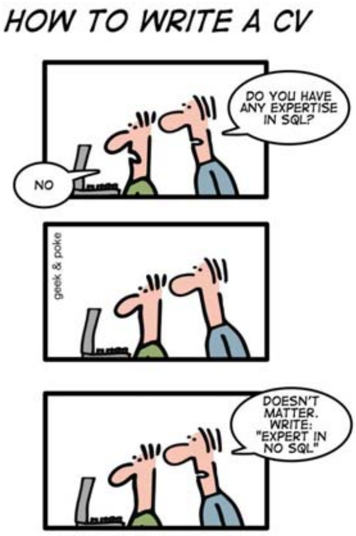
Store query, not data

Overview of Chapter 3

What does *NoSQL* mean?

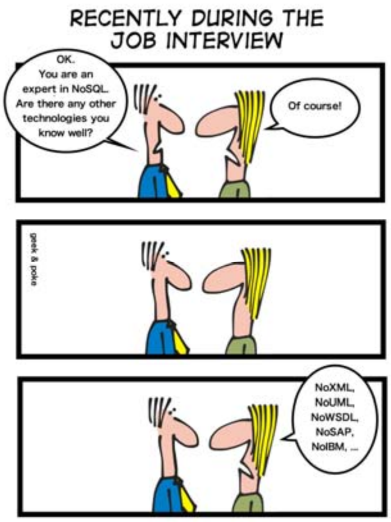
- NoSQL: **N**ot **o**nly **SQL**
  - Non-relational databases
  - No schema
    - Dynamically add new attributes to individual records
  - Inherently distributed
    - Data replication
    - Data partitioning
  - Highly scalable for (very) large volumes of data
    - Horizontal scalability
  - Simple interface
    - No full-fledged declarative query language
    - No joins
  - Relaxed consistency
    - No ACID
    - (BASE)

NoSQL



Leverage the NoSQL boom

Again NoSQL



NoSQL Systems

- Several types of alternatives to relational database systems, most of them especially tailored to 'Big Data':
  - Key-value Store
    - Simple data model ("schema-less schema")
  - Column Store
    - In contrast to traditional row stores, all attributes of the same column are stored on the same database page
  - Document Databases
    - Structure of data is specified via XML or JSON
  - Graph Databases
    - Systems tailored to graph structures (nodes, edges) and graph algorithms

Key-Value Stores …

- A key-value store is based on a simple data model that consists of
  - A unique key
  - An additional value, which is basically a BLOB.
    This BLOB might have an internal structure and different key/value pairs might have differently structured BLOBs.
  - It is the application's task to properly interpret the contents in the BLOB
    → it is a "schemaless database"

Key	Value					
K1	42	2016-02-29	105.71			
K2	'Miller'					
K3	2	'Smith'	'Basel'	42	X'120332828474292'	0.0
K4	7	5	'ID'	5		
K5	2016-02-28	10:28:32	2016-03-01	12:03:56	91.226.202.77	

… Key-Value Stores

- Operations in Key-Value Stores
  - **`get(key)`**returns key-value pair specified by its key
  - **`put (key, value)`** stores key-value pair identified by its key. If it already exists, the value will be overwritten
  - **`delete(key)`** removes key-value pair specified by its key from the database
- Key-value stores are not suitable when …
  - references between data have to be maintained (and referential integrity needs to be enforced)
  - „search by data" instead of „search by key", i.e., when the contents of the value are used to retrieve key-value pairs as the value has to be interpreted by the application.

Key-Value Stores: Meet the Players

- Examples of key-value stores
  - DynamoDB (amazon), **`https://aws.amazon.com/dynamodb`**
  - Berkeley DB, **`http://www.oracle.com/us/products/database/berkeley-db/index.html`**
  - memcached, **`http://www.memcached.org/`**
  - LevelDB (google), **`http://github.com/google/leveldb`**
  - redis, **`http://redis.io/`**
  - riak, **`http://basho.com/products/#riak`**
  - Voldemort (linkedin), **`http://www.project-voldemort.com/`**
  - … and many more …

Row Stores

- Relational databases use “Row Stores”
- Store entire tuples on database pages
- Optimized for OLTP applications (write or read complete tuples)

Customers

ID	Name	City	Balance	Discount
01	Legrand	Geneva	-1'080,00	0.10
02	Marty	Basel	-8'00,00	0.20
03	Frei	Basel	0,00	0,10
04	Janvier	Geneva	0,00	0,10
05	Rossi	Lugano	0,00	0,05
06	Meier	Zurich	-3'800,00	0,05
07	Hürlimann	Lucerne	-100,00	0,05
08	Schmid	Lausanne	-2'235,00	0,10
09	McAllen	Zurich	-550,00	0,00
10	Lacroix	Geneva	-31'000,00	0,20
…	…	…	…	…

database pages

Stored column wise (data)

Column Store Database Systems

- Columns stores are optimized for applications that are characterized by …
- … long, complex read transactions that do not request full tuples (OLAP)
- … and rather few update operations

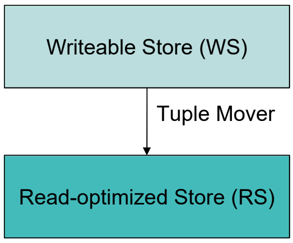
Customers

ID	Name	City	Balance	Discount
01	Legrand	Geneva	-1'080,00	0.10
02	Marty	Basel	-8'00,00	0.20
03	Frei	Basel	0,00	0,10
04	Janvier	Geneva	0,00	0,10
05	Rossi	Lugano	0,00	0,05
06	Meier	Zurich	-3'800,00	0,05
07	Hürlimann	Lucerne	-100,00	0,05
08	Schmid	Lausanne	-2'235,00	0,10
09	McAllen	Zurich	-550,00	0,00
10	Lacroix	Geneva	-31'000,00	0,20
…	…	…	…	…

database pages

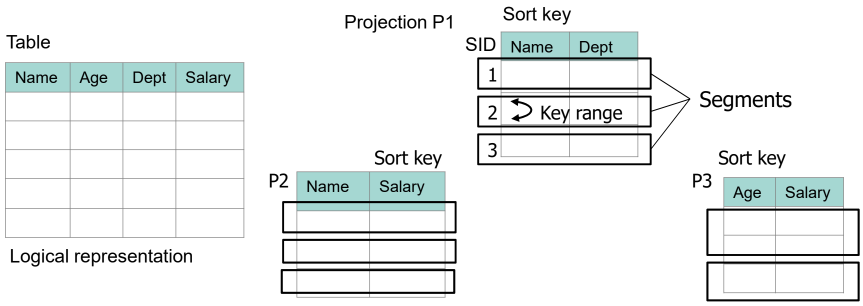
C-Store

- C-Store is an example of a Column Store system
- Writeable Store: allows arbitrary insert and update operations
- Read-Optimized Store: the only write operation supported is a batch update, initiated by the Writeable store (= lazy replication)
  – Tuple Mover: executes batch update
- Relational model as logical data model, SQL at the interface
  – Physical storage: Projections of single (or multiple) attributes



C-Store: Read-Optimized Store ...

- Single attributes can be stored several times in the Read-optimized Store (in different projections)
  – For each projection, there is a Sort Key (the attribute according to which the projection is sorted)
  – Each projection is horizontally partitioned into several segments. Each segment is characterized by a Segment Identifier (SID). This means that each segments encompasses an interval of the Sort Key.

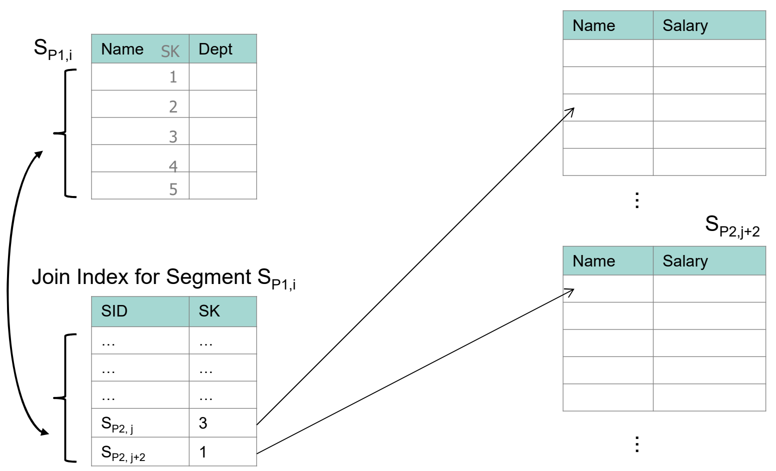


… C-Store: Read-Optimized Store

- The attributes of each segment are characterized by a Storage Key (SK)
- The link to other attributes of the same tuple is done via specialized Join Indexes
- Assumption: P1 (M segments) and P2 (N segments) are projections of the same relation
  - A Join Index then consists of M tables (one per segment of P1)
  - An entry in the Join Index of P1 for segment S_{P1} contains for each entry of S_{P1} a link to the corresponding entries in projection P2
  - This link consists of the segment identifier (SID) and the Storage Key (SK) within the segment
  - Join Indexes thus only establish a unidirectional link between projections

C-Store: Join Index

- Segment $S_{P1,i}$: i^{th} segment of Projection P1



Further Column Store Systems

- Google‘s Bigtable
  - Uses projection and segmentation
  - Distributes segments across several computers
  - Based on GFS (Google File System)
- Hbase: Open Source clone of Bigtable
- Amazon SimpleDB
- Vertica: Commercial implementation of C-Store
- Cassandra (Facebook): combines key/value stores and more sophisticated schemas
- MonetDB: Research prototype, CWI Amsterdam
- Sybase IQ: First commercial column store system
- … and many more systems (both open source and commercial)

Column Stores: BigTable ...

- BigTable is a distributed storage system for managing structured data that is designed to scale to a very large size (up to petabytes) across thousands of commodity servers at Google
  - is used by more than sixty Google products and projects (Web Indexing, Google Earth, Google Finance, Personalized Search)
- These applications place very different demands in terms of:
  - Data size: from URL to web page to satellite images (Billions of URLs, many versions/page – 20KB/page)
  - Latency requirements: from throughput oriented batch-processing jobs to real-time data serving
  - Deployment: from a handful to thousand of servers
  - Very high read/write rates (millions of operations per second)
  - Efficient scans over all or interesting subset of data (crawled data, anchors...)
  - Examined data changes over time, e.g., contents of a web page over multiple crawls

… Column Stores: BigTable …

- Implementation of a column store system
  - distributed multi-dimensional sparse map
- Rows:
  - Each row identified by a row key
- Columns:
  - Column family: column name which might appear multiple times in a row
  - Column key: combination of column family and qualifier
- Multiple versions:
  - Under a column key, several versions of the associated value (enriched with a timestamp that indicates their validity) can be stored
  - Materialize the evolution of the value over time

… Column Stores: BigTable …

Column Family

row key	content	anchor:abc.com	…	anchor:xyz.org		
…	…	…	…	…	…	…
com.cnn.www	<html>.. T1 <html>.. T2 <html>.. T3 <html>.. T4 <html>.. T5 … …	"CNN" T9 …		"CNN.com" T11 …		
…	…	…	…	…	…	…

Timestamps

value

… Column Stores: BigTable

- Large tables broken into tablets at row boundaries
  - Tablet holds contiguous range of rows
  - Aim for 100MB to 200MB of data per tablet
  - Serving machine responsible for ~100 tablets; properties:
    - Fast recovery: 100 machines each pick up 1 tablet from failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load balancing decisions

Document Databases

- From a conceptual point of view, document databases are in between relational DBMS and key/value stores
  - Each record is associated with a unique key
  - In contrast to a key/value-store, the value (document) has an inherent structure which is specified either via JSON or XML
  - JSON or XML documents can be nested

JSON

- JSON: supports two types of elements
  - object: set of key/value pairs; value can be of type string, number, object, or array (which means that nesting is allowed)
  - array: list of values
- Example

```
{
  "firstName": "Ronald",
  "lastName" : "Rump",
  "age"      : 71,
  "address"  :
  {
             "street"  :  "Rump Tower Street",
             "number"  :  1,
             "city"    :  "New York City",
             "zip"     :  "01234"
  },
  "telephone":  [00112345678, 0019876543, 001001001001]
}
```


XML

- XML documents consist of elements which are delimited by tags
  <A> element </A>
  - Tags may be nested, but must not overlap
  - Tags may contain attributes, e.g., <A attribute name="value" ...> </A>
- Example

```
<person>
  <firstName>Ronald</firstName>
  <lastName>Rump</lastName>
  <age>71</age>
  <address>
       <street>Rump Tower Street</street>
       <number>1</number>
       <city>New York City</city>
       <zip>01234</zip>
  </address>
  <phoneWork>00112345678</phoneWork>
  <phoneHome>0019876543</phoneHome>
  <phoneMobile>001001001001</phoneMobile>
</person>
```

Document Databases

- Document databases add a unique ID to the JSON or XML document. Documents can be retrieved via this ID
- Example:

```
{
  "id"        : "person91238323",
  "firstName": "Ronald",
  "lastName" : "Rump",
  "age"      : 71,
  "address"  :
  {
             "street"  :  "Rump Tower Street",
             "number"  :  1,
             "city"    :  "New York City",
             "zip"     :  "01234"
  },
  "telephone":  [00112345678, 0019876543, 001001001001]
}
```


Document Databases vs. Key/Value Stores

- In contrast to key/value stores, document stores also provide an API or query language that allow to retrieve documents based on their content (and structure)
  – Example: select all documents where a particular object has a given value
  – These APIs or query languages are proprietary and depend on the type of documents (and their representation) supported

Document Databases: Meet the Players

- Examples for document databases
  - Couchbase, **`http://www.couchbase.com`**
  - CouchDB, **`http://couchdb.apache.org`**
  - MongoDB, **`http://www.mongodb.com`**
  - OrientDB, **`http://orientdb.com`**
  - … and many more …

Graph Databases

- Graphs are well suited data structures to manage networked information
- Examples:
  - Social networks
  - Semantic Web
  - Spatial information (maps)
  - …
- The applications using such networked information usually rely on very special and sophisticated queries, such as
  - Determine transitive dependencies (e.g., friends-of-friends)
  - Decide whether two elements (nodes) are connected
  - Determine the shortest path between two elements (e.g., satnav systems)
  - …
- Relational schemas provide only limited support for such queries
- Graph databases store graphs natively and provide better query support

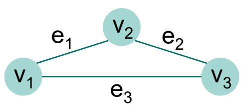
Graph Databases – Recap from cs202 Algorithms & Data Structures ...

- Basics: a graph G = (V, E) consists of
  - A set V of vertices (nodes)
  - A set E of edges connecting two vertices from V with e = $\{v_i, v_k\}$. An edge can be
    - directed (edge can be traversed only in one direction). In this case, one node is the source, the other one the target. In e = $\{v_i, v_k\}$, node v_i is source and node v_k is target; or
    - undirected (edge traversal in any direction)
- A graph is called
  - directed graph if it contains only directed edges
  - undirected graph if it contains only undirected edges
  - multigraph if it may contain several edges between the same two nodes
  - complete if it contains all possible edges
- Two nodes are called adjacent if they are connected via an edge; an edge is incident to a node if it is connected to that node

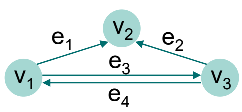
... Graph Databases – Recap from cs202 Algorithms & Data Structures ...

Examples of graphs

- (Simple) Undirected Graph
  - Each edge is represented by a set including two nodes: e = $\{v_i, v_k\} = \{v_k, v_i\}$
  - With |V| = n, a graph may contain up to $\binom{n}{2} = \frac{n\,(n-1)}{2}$ edges



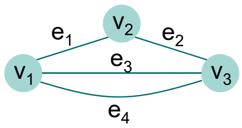
- (Simple) Directed Graph
  - Each edge is represented by an ordered tuple, e = $(v_i, v_k) \neq (v_k, v_i)$
  - With |V| = n, a graph may contain up to $n \cdot (n-1)$ edges (without “self edges”)



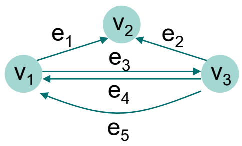
… Graph Databases – Recap from cs202 Algorithms & Data Structures …

Examples of graphs (cont'd)

- Undirected Multigraph
  – each edge is represented by a set including two nodes
  – the edge set E is a multiset of such edges



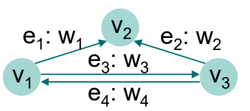
- Directed Multigraph
  – The edge set E is a multiset of tuples



… Graph Databases – Recap from cs202 Algorithms & Data Structures …

Examples of graphs (cont'd)

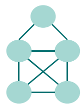
- Weighted Graphs
  – Each edge e_i has an associated weight w_i
    (e.g., cost, distance, capacity, etc.)
  – Edge weights are independent of the type of graph (directed / undirected) and the multiplicity of edges
  – Example: weighted simple directed graph



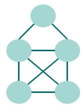
… Graph Databases – Recap from cs202 Algorithms & Data Structures …

Examples of graph algorithms

- Cycle detection (directed graphs)
  – find a path along edges where start and end node is the same
- Eulerian path
  – Traverse a graph by visiting each edge exactly once
  – Start node and end node do not need to be the same



- Eulerian cycle
  – Each edge has to be traversed exactly once
  – Start node and end node are the same
- Hamiltonian path
  – Each node has to be visited exactly once
  (not all edges have to be traversed)

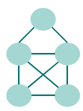


- Hamiltonian cycle
  – Each node has to be visited exactly once
  (start and end node are the same)

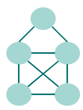
… Graph Databases – Recap from cs202 Algorithms & Data Structures …

Examples of graph algorithms (cont'd)

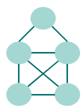
- Spanning Tree
  – Find a subset of the edges of E (from a start node = root) that forms a tree and that visits each node



- Depth-first search (DFS)
  – Traversal of a graph from a given start node
  – Explores as far as possible all nodes' direct neighbors before backtracking



- Breadth-first search (BFS)
  – Traversal of a graph from a given start node
  – Explores all direct neighbors of a node first, before moving to the next level neighbors



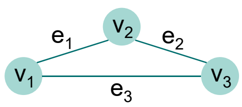
- Shortest path
  – Minimal number of edges between two nodes
  (alternative: minimal weight on path from start to end node)
- … and many more …

Graph Databases: Graph Data Structures …

- Edge List
  - A graph is stored as a set of nodes V and a set of edges E
  - E is
    - Set of sets for undirected graphs
    - Set of tuples for directed graphs
    - Multiset of sets / tuples for multigraphs
  - Advantages:
    - Insertion / deletion of nodes and edges
    - Simple queries asking for all edges or all nodes
  - Disadvantages
    - No or not adequate support for more sophisticated queries (i.e., find particular node or edge, find path, etc.)

… Graph Databases: Graph Data Structures …

- Adjacency Matrix
  - With |V| = n, the adjacency matrix A is an (n × n) matrix where
  - $a_{i,k} = \begin{cases} 0 & \text{if there is no edge} \\ 1 & \text{if there is an edge} \\ n & \text{for multiple edges} \end{cases}$

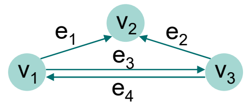


	v_1	v_2	v_3
v_1	0	1	1
v_2	1	0	1
v_3	1	1	0

  - For undirected graphs, the matrix is symmetric
  - Advantages:
    - Lookup for an edge (either by its source or target node)
    - Insertion of new edge (if source and target nodes already exist)
  - Disadvantages
    - Insertion of node requires extension of matrix
    - Search for neighbors requires scan of complete column
    - Large storage overhead, in particular for large, sparse graphs

… Graph Databases: Graph Data Structures …

- Incidence Matrix
  - With |V| = n and |E| = m, the incidence matrix B is an (n × m) matrix where the rows represent the vertices and the columns the edges and
  - $b_{i,k} = \begin{cases} 0 & \text{if node and edge are } \textit{not} \text{ connected} \\ 1 & \text{if node and edge are connected} \end{cases}$

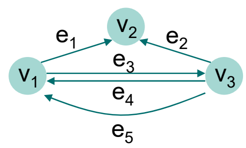


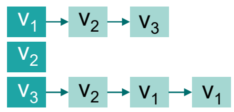
  - For directed graphs, the source node is marked with -1, the target node with +1
  - Advantages:
    - Only existing edges are stored (no empty column)
  - Disadvantages
    - Insertion of nodes / edges costly (extension of matrix)
    - Search for neighbors requires costly scans
    - Large storage overhead

	e_1	e_2	e_3	e_4
v_1	-1	0	-1	1
v_2	1	1	0	0
v_3	0	-1	1	-1

… Graph Databases: Graph Data Structures …

- Adjacency List
  - Stores vertex set V and for each $v_i \in V$, a linked list that contains the neighbors of v_i (adjacent nodes)
    - For directed graphs, the list only contains nodes connected via outgoing edges
    - For multigraphs, nodes may occur several times in a linked list

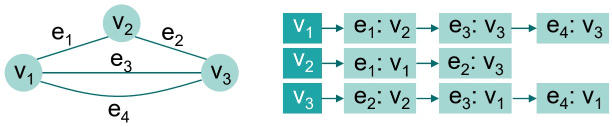




  - Advantages:
    - Insertion of vertices and edges
    - Lookup of neighbors
  - Disadvantages
    - Checking existence of particular edge (especially for a given source node)

… Graph Databases: Graph Data Structures

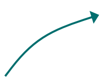
- Incidence List
  - Stores vertex set V and for each $v_i \in V$, a linked list that contains all the incident edges of v_i
    - For directed graphs, only outgoing edges are stored in the incident list



  - Advantages:
    - Insertion of vertices and edges
    - Lookup of neighbors
  - Disadvantages
    - Checking existence of particular edge for a given source node

Graph Databases: Property Graph Model …

- Most graph databases support directed multigraphs
- In addition to the basic graph structure, further information is stored
  - inside the nodes and
  - inside the edges
- Nodes and edges are typed; labels denote the type name
- Type definition specifies a set of attributes. Each attribute consists of
  - a name
  - a value, taken from a given domain
  - *name:value* pairs are also called properties, therefore, these graphs are called property graphs



Type: knows
Source node: Person
Target node: Person
Attributes:
since: Date

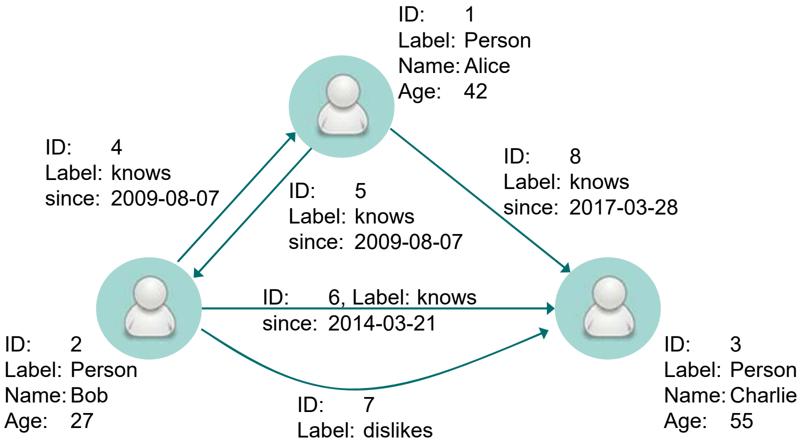
Type: Person
Attributes:
Name: String
Age: Integer

… Graph Databases: Property Graph Model …

- A property graph P is a labeled and attributed multigraph with identifiers where
- P = (V, E, L_V, L_E, ID) with
  - V is a set of nodes
  - E is a set of edges
  - L_V is a set of node labels (type names for nodes). For each $l \in L_V$ there is a type definition t = (l, A) with A being a set of attribute definitions; each $a \in A$ is an attribute definition with a = (attributename, domain)
  - L_E is a set of edge labels (type names for edges). For each $l' \in L_E$ there is a type definition t' = (l', A', sourcetype, targettype) with A' being a set of attribute definitions so that each $a' \in A'$ is an attribute definition with a' = (attributename, domain), sourcetype $\subseteq L_V$ and targettype $\subseteq L_V$
  - ID is a set of unique identifiers (for nodes *and* edges)

… Graph Databases: Property Graph Model

- Example of a property graph (from a highly simplified social network)



Graph Databases: Storing Property Graphs in a Relational Schema

- Nodes: one table (**Nodes**) for all nodes, one table for each node type (i.e., **PersonAttributes**)

Nodes

NodeID	NodeLabel
1	Person
2	Person
3	Person

PersonAttributes

NodeID	Name	Age
1	Alice	42
2	Bob	27
3	Charlie	55

- Edges: one table (**Edges**) for all edges, one table for each edge type (i.e., **knowsAttributes**)

Edges

EdgeID	EdgeLabel	Source	Target
4	knows	2	1
5	knows	1	2
6	knows	2	3
7	dislikes	2	3
8	knows	1	3

knows-Attributes

EdgeID	since
4	2009-08-07
5	2009-08-07
6	2014-03-21
8	2017-03-28

Advanced Graph Models

- Hypergraphs are graphs with hyperedges
  - A “normal” edge corresponds to a two-element subset of V (source and target)
  - An undirected hyperedge $e_h = \{v_i, v_k, \ldots, v_r\} \subseteq V$ consists of a set of nodes
  - A directed hyperedge $e_h = (\{v_i, \ldots, v_r\}, \{v_s, \ldots, v_z\})$ is a tuple consisting of two sets of nodes where $\{v_i, \ldots, v_r\} \subseteq V$ are the source nodes (source set) and $\{v_s, \ldots, v_z\} \subseteq V$ are the target nodes (target set). The cardinalities of both sets may differ.
- Nested Graphs are graphs with hypernodes
  - A “normal” node is atomic
  - A hypernode (recursively) contains an entire graph

Graph Databases: Meet the Players …

- Neo4J (**`http://neo4j.com`**)
  - Based on Property Graph Model
  - Edges are called "relationships"
  - Provides ACID transactions
  - Indexing for node and edge properties based on the Apache Lucene text search engine library
  - Declarative query language Cypher (search for nodes or traverse a graph)
    - **`START`** starting node(s) in the graph
    - **`MATCH`** graph traversal to be considered in the query.
      `->` specifies an edge, **`-[:knows]->`** follows a **`knows`** edge
    - **`WHERE`** additional filters
    - **`RETURN`** specify the return value
    - Example:

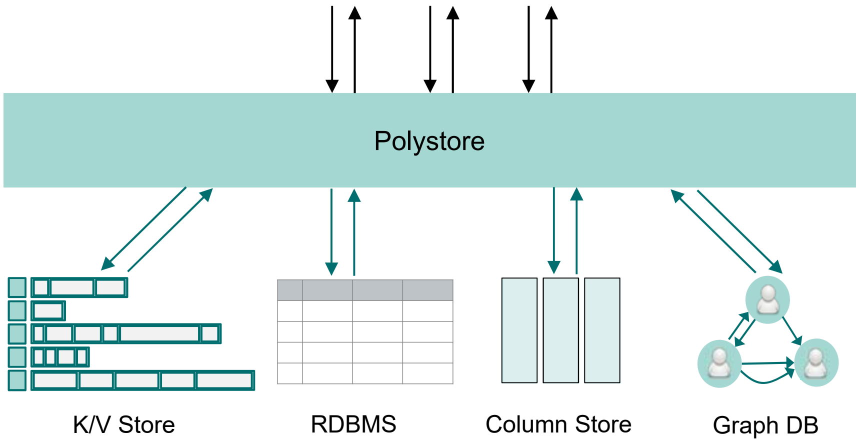
… Graph Databases: Meet the Players

- HyperGraphDB (**`http://hypergraphdb.org`**)
  - Supports hypergraphs
- OrientDB (**`http://orientdb.com`**)
  - Combined document database and graph database
- … and many more …

Polystores …

- In most cases, there is no sharp distinction between
  - different NoSQL systems: e.g., document stores are also Key/Value stores
  - different data formats: e.g., structured and unstructured
  - different application workloads: e.g., may encompass OLTP and OLAP
- No One-size-fits-all:
  - there is not a single system that jointly supports all types of applications
  - Even for concrete applications, it is sometimes impossible to select a database / storage system
- Polystores:
  - combine different databases in one system
  - different storage technologies (main memory, SSD, spinning disk)
  - data replication (in different data formats / models)
  - decision per query where to route a request
  - logical database consisting of several physical databases

… Polystores



Overview of Chapter 3

Data Management in the Cloud

http://gnoted.com/what-is-cloud-computing-simple-terms/

Data Management in the Cloud: Basics

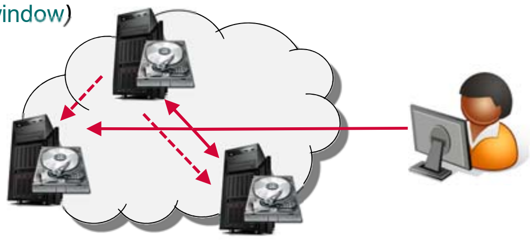
- Cloud Computing Revisited: The Cloud from a Consumer‘s Perspective
- Establish service level agreements with Cloud providers (quality of service, QoS)
  - e.g., Availability
- Elasticity: dynamically request additional resources for peak loads
  - e.g., Scalability

The Cloud from a Provider’s Perspective

- Have enough spare resources to guarantee elastic behavior
  - How to maximize capacity utilization?
- Multi-tenancy
  - Support different tenants with completely different requirements in the same system
- Quality of Service guarantees; for data, this includes
  - Replication (software, data), constrained by CAP Theorem (details later)
  - Performance / latency
- Cloud data management
  - How to provide read operations with different semantics (up-to-date data, stale data)
  - Long-term preservation and archiving
- *Some of these aspects are not convincingly solved yet but are still subject to intensive research*

Different Levels of Consistency

- Strong consistency (eager replication)
  - Each update forces all replicas to be updated
  - Afterwards, all accesses return the new value
  - But: limited availability between 1^{st} and last write operation
- Weak consistency (lazy replication)
  - Update needs to be acknowledged only once
  - Full availability
  - Inconsistent system state during convergence period (inconsistency window)

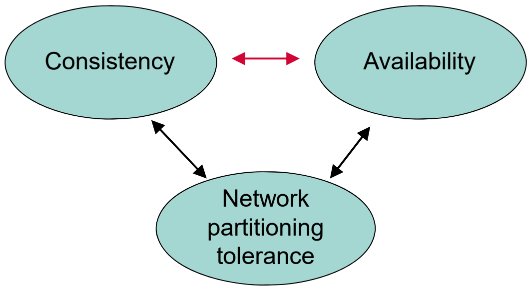


Cloud Data Management – Requirements

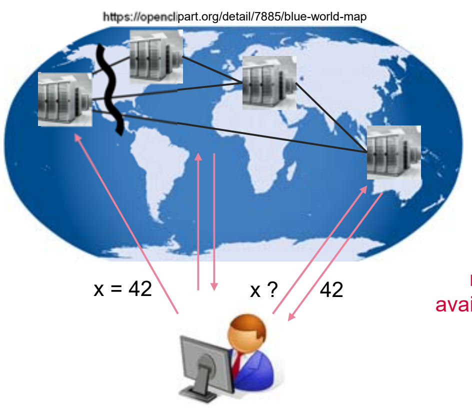
- Tolerance to network partitions
  - If a connection to a computer / rack / datacenter / continent fails, the disconnected partitions must continue to work
  - Clients in the same partition do not even realize the partitioning
- Availability
  - For the client, the system (even though it might be distributed) looks like one physical system and should be usable / accessible anytime
  - Solution: redundancy & replication
- Consistency
  - The more replicas, the more difficult to keep consistency
  - When enforcing consistency, the system is no longer highly available

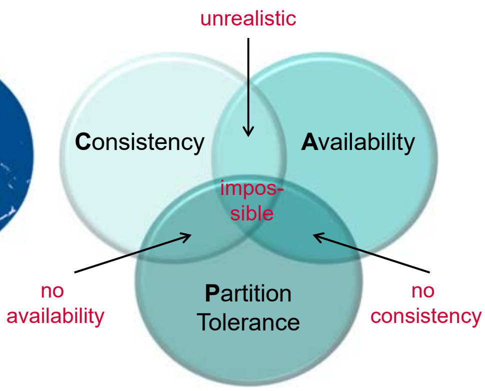
CAP Theorem ...

- Only two of the three requirements **C**onsistency, **A**vailability, tolerance to network **P**artitions can be met at the same time (has been formally proven)
- General assumption: network partitions cannot be influenced by the Cloud providers
  → Trade-off between availability and consistency
- Cloud providers can choose to either weaken
  - Availability, or
  - Consistency



... CAP Theorem



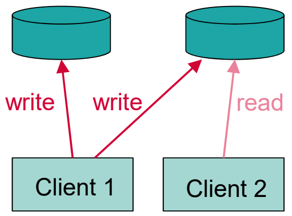


CAP Theorem: General Discussion

- Data Replication (Foundations of Distributed Systems) revisited: Assume we have a distributed system with replicated data and let
  - N be the number of nodes that store a replicas of the data
  - W be the number of replicas that need to acknowledge the receipt of the update before the update completes
  - R be the number of replicas that are contacted when a data object is accessed through a read operation
- Depending on N, W, and R, a system has the following characteristics
  - $W+R > N$ (and $W > N/2$)
    In this case, the write set and the read set always overlap and one can guarantee strong consistency
  - $R+W=N$ (or, even worse, $R+W < N$)
    In this case, consistency cannot be guaranteed as one cannot make sure that the most recent data is read.
    This is also known as weak consistency or eventual consistency

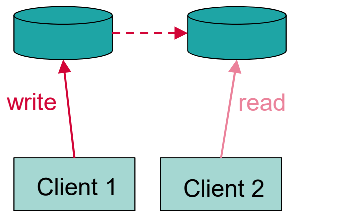
CAP Theorem: Examples ...

- Primary backup DBMS with synchronous replication
- Fault tolerant and optimized for reads
- $N=W=2$, $R=1$



- What if the system cannot write to W nodes?
  → Failure (impacting availability)

- Asynchronous replication
- Fault tolerant and optimized for writes
- $N=2$, $W=R=1$



- Consistency cannot be guaranteed

… CAP Theorem: Examples …

- Usually, high availability in distributed storage systems means $N > 2$.
  In most cases, $N = 3$ (triplication)
  - Systems with an exclusive focus on fault-tolerance:
    $N=3$, with $W=2$ and $R=2$
  - Systems that aim at supporting high read loads:
    $N = 3$ at least, or even higher (in the area of up to hundreds of nodes),
    $R = 1$
  - Systems that aim at providing a high degree of consistency:
    $W=N$, even though this may decrease the probability of the write succeeding
  - Systems that aim at providing a high degree of fault-tolerance, but not consistency:
    $N=3$, $W=1$ (master node), and rely on a lazy technique to update the other replicas

… CAP Theorem: Examples

- Concrete values of N, W and R depend on what property needs to be optimized
  - $R=1$ and $N=W$ optimizes the read case
  - $W=1$ and $R=N$ optimize for a very fast write. But:
    - durability is not guaranteed in the presence of failures
    - if $W \leq N/2$ there is the possibility of conflicting writes because write sets do not overlap (quorum is not reached)

Different Flavors of (Strong) Consistency

- Sequential Consistency
  - This corresponds to the serializability criterion in databases (conflict-preserving serializability, CPSR), i.e., the equivalence to some serial execution
- Causal Consistency
  - Writes that are causally related must be seen in the same order.
  - At the same time, writes (especially from the same transaction) that are not causally related may be executed in different orders on different sites

Different Flavors of (Weak) Consistency …

- Monotonic Read Consistency
  - once a system has returned a particular record to a client, further queries of the same client will only return versions that are at least as fresh as the previously returned one
- Monotonic Write Consistency
  - if a particular client changes some data item two (several) times, then the system has to make sure that the writes happen internally in exactly the same order
- Read-your-Write Consistency
  - a system guarantees that, once a record has been updated, any attempt to read the record will return the updated value

… Different Flavors of (Weak) Consistency

- Writes-follows-reads Consistency
  - A write operation on object x following a read on x by the same transaction is guaranteed to take place on the same or more recent version of x that was read
- Session Consistency
  - Read-your-Write consistency, where the property is only limited to the lifetime of a client session

Eventual Consistency

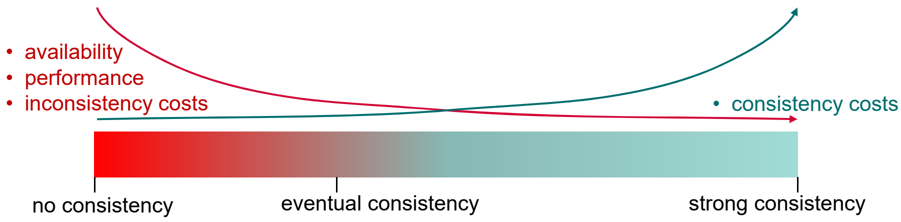
- Eventual consistency is a form of weak consistency
  - Guarantee: if no further updates are made during convergence, all accesses will eventually see the new value
  - Supported (exclusively) by most Cloud providers

  → They might return inconsistent data!
- Weak consistency: Meet the Players
  - Amazon's Read Replicas
    - RDS, the scalable relational database service, has released Read Replicas to meet the performance demands of read-heavy database workloads
    - "You can now create one or more replicas of a given source DB Instance and serve incoming read traffic from multiple copies of your data, elastically scaling out beyond the capacity of a single DB Instance"

ACID vs. BASE

- From database transactions, we know the ACID guarantees
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability
- Most Cloud providers rather provide much softer and blurrier guarantees (because of the CAP theorem): BASE
  - **B**asically **A**vailable: the system is available most of the time, but may occasionally be down
  - **S**oft state: information (state) the user puts into a system will eventually go away if this information is not maintained (i.e., information will expire unless it is refreshed) – *in contrast to the D in ACID*
  - **E**ventual consistency: weak consistency, see before – *in contrast to the I in ACID*

Availability vs. Consistency: Cost-based Data Management



- Current research: model that takes into account
  - Actual and predicted workload
  - Costs of necessary resources: consistency costs
  - Costs for dealing with inconsistencies
  - User requirements: performance, availability
  - Available budget

  → Dynamically select and adapt protocols for distributed data management

CAP in the Non-Failure Case

- CAP essentially deals with failures
- In the non-failure case, there is a trade-off between consistency and latency
  - The higher the consistency level, the higher the latency
  - The lower the consistency level, the better the overall performance (low latency)
- This is also known as PACELC
  - PAC: a permutation of CAP
  - ELC: **E**lse **L**atency vs. **C**onsistency

More on Latency

- Latency matters!
  - Amazon found every 100ms of latency cost them 1% in sales
  - Google found an extra 0.5 seconds in search page generation time dropped traffic by 20%
  - A broker could lose $4 million in revenues per millisecond if their electronic trading platform is 5 milliseconds behind the competition
- Source: **`http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`**

Further Reading

[Aba 12] Daniel Abadi: *Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story*. IEEE Computer 45(2): 37-42, 2012.

[AFG+ 09] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. Report UC Berkeley, 2009.

[Bre 00] E. Brewer: *Towards robust distributed systems*. In: Proc. PODC 2000.

[EFH+ 11] S. Edlich, A. Friedland, J. Hampe, B. Brauer, M. Brückner: *NoSQL – Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser, 2011.

[GL 02] S. Gilbert, N. Lynch: *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News 33(2): 51-59, 2002

[SF13] P. Sadalage, M. Fowler: *NoSQL Distilled*. Addison-Wesley, 2013.

[Wie 15] L. Wiese: *Advanced Data Management – for SQL, NoSQL, Cloud and Distributed Databases*. De Gruyter, 2015