NOSQL DATABASE SYSTEMS: TECHNIQUES FOR SCALABILITY, AVAILABILITY AND CONSISTENCY
NoSQL Database Systems: Techniques

- Consistency
- Sharding
- Replication
- Storage Management
- Query Processing
- Classification by Techniques
• **CAP Theorem** (Eric Brewer, 2000): in a distributed database system, you can **only have at most two of the following three characteristics:**

  – **Consistency**: all clients have the same view, even in case of updates

  – **Availability**: every request received by a non-failing node in the system must result in a response (i.e., even when severe network failures occur, every request must terminate)

  – **Partition tolerance**: system properties hold even when the network (system) is partitioned (i.e., nodes can still function when communication with other groups of nodes is lost)
• **Asymmetry of CAP properties**
  – Some are properties of the system in general
  – Some are properties of the system only when there is a partition
Critism of CAP Theorem

- [Aba2012]: Abadi, J. *Consistency Tradeoffs in Modern Distributed Database System Design*. In IEEE Computer 45(2)
  
  “... CAP has become increasingly misunderstood and misapplied, potentially causing significant harm. In particular, many designers incorrectly conclude that the theorem imposes certain restrictions on a DDBS during normal system operation, and therefore implement an unnecessarily limited system. In reality, CAP only posits limitations in the face of certain types of failures, and does not constrain any system capabilities during normal operation. ...”

⇒ The theorem simply states that a network partition causes the system to have to decide between reducing availability or consistency

⇒ General: Fundamental tradeoff between consistency, availability and latency ≡ tradeoff between consistency and latency
- [Aba2012]: Rewriting CAP as **PACELC**: 
  - if there is a partition (P), how does the system trade off availability and consistency (A and C); 
  - else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?

Source: F. Gessert, Baqend
• **Strong consistency**
  – After the update completes, any subsequent access will return the updated value, i.e., any subsequent access from A, B, C will return D₁

• **Weak consistency**
  – The system does not guarantee that subsequent accesses will return the updated value D₁ (a number of conditions need to be met before D₁ is returned)
Eventual Consistency

- **Eventual consistency** (Vogel, 2008)
  - Specific form of weak consistency
  - Guarantees that if no new updates are made, eventually all accesses will return $D_1$
  - If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.

Source: Sattler:2010
• Alternative **consistency** model: **BASE** (Eric Brewer, 2000)
  
  – **Basically available**
    • Availability of the system even in case of failures
  
  – **Soft-state**
    • The state of the system may change over time, even without input (clients must accept stale state under certain circumstances.)
  
  – **Eventually consistent**
    • The system will become consistent over time, given that the system doesn't receive input during that time
Variants of Eventual Consistency

- **Causal consistency:**
  - If A notifies B about the update, B will read $D_1$ (but not C!)

- **Read-your-writes:**
  - A will always read $D_1$ after its own update

- **Session consistency:**
  - Read-your-writes inside a session

- **Monotonic reads:**
  - If a process has seen $D_k$, any subsequent access will never return any $D_i$ with $i < k$

- **Monotonic writes:**
  - Guarantees to serialize the writes of the same process

Source: Sattler:2010
Transactions

• **ACID vs. BASE?**
• What about *isolation* on the same node? ⇒ MVCC
• BASE focus mainly on C and I – however, what bout A and D in ACID?

**Atomicity**

• Most NoSQL systems:
  – No concept of transactional features
  – Atomicity only for a single object, row, or document respectively

• Few NoSQL systems only:
  – Concept of transactional features over multiple objects
    • Redis, Google Datastore, ...
    • Restriction: objects have to be located on the same instance
Transactions

Durability

• Durability in relational database systems: write-ahead logging (WAL)

Concepts of Persistence and **Durability in NoSQL systems**

• In-memory only
  – Durability by replication only

• In-memory and write-ahead logging by configuration
  – Redis, CouchBase...

• Write-ahead logging by default
  – Riak, HBase, MongoDB, ...
  – Some systems (e.g. Riak) use the write-ahead log as place of storage
Consistency

- Sharding
- Replication
- Storage Management
- Query Processing
- Classification by Techniques
Sharding

• Hash-based Sharding
  – Hash of data values (e.g. key) determines partition (shard)
  – **Pro**: Even distribution
  – **Contra**: No data locality

• Range-based Sharding
  – Assign ranges defined over fields/attributes (shard keys) to partitions
  – **Pro**: Enables *Range Scans* and *Sorting*
  – **Contra**: Repartitioning/balancing required

• Entity-Group Sharding
  – Explicit data co-location for single-node-transactions
  – **Pro**: Enables ACID Transactions
  – **Contra**: Partitioning not easily changeable

**Implemented in:**
MongoDB, Riak, Redis, Cassandra, Azure Table, Dynamo, ...

**Implemented in:**
MongoDB, BigTable, HBase, DocumentDB, Hypertable, ...

**Implemented in:**
G-Store, Megastore, Relation Cloud, Cloud SQL Server, ...
Sharding: Consistent Hashing

- **Consistent Hashing**
  - Keys and nodes (identified by id or server name) are mapped onto a “circle”
  - Keys are assigned to the node that is next to them in a clockwise direction

Source: http://weblogs.java.net/blog/2007/11/27/consistent-hashing
Sharding: Consistent Hashing (Cont.)

- Consistent Hashing: Example
  - Removal of server C
  - Addition of server D

Source: http://weblogs.java.net/blog/2007/11/27/consistent-hashing
• Consistent Hashing with virtual nodes

Source: http://docs.basho.com/riak/1.0.0/tutorials/fast-track/What-is-Riak/
NoSQL Database Systems: Techniques

✓ Consistency

✓ Sharding

• Replication

• Storage Management

• Query Processing

• Classification by Techniques
Replication

• **Motivation / Benefit**
  – Performance enhancement
  – Availability enhancement (fault tolerance)

⇒ **Tradeoff** between benefits of replication and work required to keep replicas consistent

Two basic (and orthogonal) parameters:

• **When** are the updates propagated?
  – synchronous vs. asynchronous

• **Where** is the update performed?
  – update everywhere vs. selected copy
Replication: When

Asynchronous (lazy)
- Writes are acknowledged immediately
- **Pro**: Fast writes, no coordination needed
- **Contra**: Replica data potentially stale (inconsistent / weak consistency)

Synchronous (eager)
- The node accepting writes synchronously propagates updates/transactions before acknowledging
- **Pro**: Consistent
- **Contra**: needs a commit protocol (more roundtrips), unavailable under certain network partitions

Implemented in:
- MongoDB, Dynamo, Riak, Redis, CouchDB, Cassandra, Voldemort, RethinkDB ...

Implemented in:
- BigTable, HBase, CouchBase, MongoDB, RethinkDB, ...
Replication: When

• How to ensure consistency without synchronous update propagation?
  – Read only from master node (replicas for failover purposes only)
    • e.g. CouchBase
  
  – Quorum Consensus Protocols
Terminology

- **N**: the number of nodes that store replicas of the data
- **W**: the number of replicas that need to acknowledge the receipt of the update before the update completes
- **R**: the number of replicas that are contacted when a data item is accessed through a read operation

\[ W + R > N \]

⇒ **Quorum Assembly**
⇒ **Strong Consistency**

\[ W + R \leq N \]

⇒ **Weak Consistency**
Quorum Consensus: Server-Side

- \( W+R > N \Rightarrow \text{strong consistency} \) through quorum assembly
  - \( W=N \) and \( R = 1 \) ⇒ read optimized strong consistency (ROWA)
  - \( W=1 \) and \( R = N \) ⇒ write optimized strong consistency
  - \( R = W = \frac{N}{2} + 1 \) ⇒ Majority Consensus

- A common choice in NoSQL database systems is \( N=3, R=2, W=2 \)

Source: http://docs.basho.com/riak/1.0.0/tutorials/fast-track/What-is-Riak/
Quorum Consensus: Variations

Unweighted vs. weighted votes
• Weighted votes
  – Each node is assigned a weight, e.g. for “better” replicas
  – Instead of sum of nodes N use sum of weights of nodes

Static vs. dynamic quorum
• Dynamic Quorum
  – Quorums can be chosen separately for each item, e.g. in case of unavailable nodes
Cassandra

• **Write Consistency Level**
  - **Zero**: A write must be written to at least one node. (If all replica nodes for the given row key are down ⇒ hinted handoff)
  - **One/Two/Three**: A write must be written to at least one/two/three replica node(s).
  - **Quorum**: A write must be written on a quorum of replica nodes (N/2 +1).
  - **All**: A write must be written on all replica nodes.

• **Read Consistency Level**
  - **One/Two/Three**: Returns a response from the closest / two / three of the closest replica (may be inconsistent).
  - **Quorum**: Returns the record with the most recent timestamp once a quorum of replicas (N/2 + 1) has responded.
  - **All**: Returns the record with the most recent timestamp once all replicas have responded. The read operation will fail if a replica does not respond.

*LOCAL_QUORUM / EACH_QUORUM: quorum of replica nodes in the same data center / in all data centers*
Asynchronous Write: Write-Related Strategies

• How to propagate a write operation to an unavailable node?

• **Hinted Handoff Algorithms**
  – Do a “hinted” write to an alive node (e.g., nearest live replica)
  – When the failed node returns to the cluster, the updates received by the neighboring nodes are handed off to it
  ⇒ System can continue to handle requests as if the node were still there
  – Implemented in Cassandra, Riak, etc.
  – But: How does a node learn when a node is available?
    • E.g., gossip protocols
      – each node periodically sends its current view of the ring state to a randomly-selected peer (or other protocols to choose the peers)
Asynchronous Write: Read-Related Strategies

• **Read-Related Strategies**
  - A write has not propagated to all replicas
    • Repair outdated replicas after read ⇒ Read Repair
    • Repair outdated replicas that have not been read ⇒ Anti-Entropy

• **Read Repair Algorithm**
  ⇒ A system may detect that several nodes are out of sync with older versions of the data requested in a read operation (*how?* ⇒ *later in this chapter*)
  ⇒ Mark the nodes with the stale data with a Read Repair flag
  ⇒ Synchronizing the stale nodes with newest version of the data requested
  – Implemented in Cassandra, Riak, etc.
Replication: When

**Master-Slave** (*Primary Copy*)

- Only a dedicated master is allowed to accept writes
- Slaves are read-replicas
- Different data items can have different primary nodes
- **Pro**: reads from the master are consistent
- **Contra**: master is a bottleneck and SPOF

**Multi-Master** (*Update anywhere*)

- The server node accepting the writes synchronously propagates the update or transaction before acknowledging
- **Pro**: fast and highly-available
- **Contra**: either needs complicated coordination protocols (e.g. Paxos) or is inconsistent

*Implemented in:*
Redis, HBase, MongoDB, CouchBase, ...

*Implemented in:*
Riak, CouchDB, ...

h_da Prof. Dr. Uta Störl
Big Data Technologies: NoSQL DBMS (Techniques) - SoSe 2017
Conflict Detection and Resolution

• How to detect older versions of data?
• How to detect concurrent writes?
  – Timestamps!
  – Timestamps in a distributed environment?!
    • global unique timestamp := <node identifier, unique local timestamp>
    • Define within each node \( N_i \) a logical clock \((LC_i)^*\), which generates the unique local timestamp
    • If \( N_i \) received a request from a transaction \( T \) with
      timestamp \(< N_j, LC_j>\) and \( LC_i < LC_j \) \( \Rightarrow \) set \( LC_i = LC_j + 1 \)

  – Common approach in NoSQL database systems: Vector Clocks

*Lamport timestamp
Conflict Detection: Vector Clocks

- Vector Clocks
  - Vector clock = list of (node, counter)
  - On receive: element-wise maximum
Conflict Detection: Vector Clocks (Cont.)

- **Vector Clocks** allows to determine whether one object is a direct descendant of the other / direct descendant of a common parent / are unrelated in recent heritage.
Consistency and Replication: Example

• Facebook’s strategy:
  – The master copy is always in one location, a remote user typically has a closer but potentially stale copy.
  – However, when users update their pages, the update goes to the master copy directly as do all the user’s reads for a short time, despite higher latency.
  – After 20 seconds, the user’s traffic reverts to the closer copy, which by that time should reflect the update.

• Source: Eric A. Brewer: *Pushing the CAP: Strategies for Consistency and Availability*. In *IEEE Computer* 45(2)
NoSQL Database Systems: Techniques

✓ Consistency
✓ Sharding
✓ Replication

• Storage Management
• Query Processing
• Classification by Techniques
NoSQL Storage Management in a Nutshell

Typical Uses in DBMSs:
- Caching
- Primary Storage
- Data Structures

RAM:
- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

SSD:
- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

HDD:
- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

Source: F. Gessert, Baqend
NoSQL Storage Management in a Nutshell

Promotes durability of write operations.
Improves latency.
Is good for read latency.
Increases write throughput.

Source: F. Gessert, Bagend
NoSQL Database Systems: Techniques

✓ Consistency

✓ Sharding

✓ Replication

✓ Storage Management

• Query Processing

• Classification by Techniques
Local Secondary Indexing

Partitioning By Document

### Partition I

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Red</td>
</tr>
<tr>
<td>56</td>
<td>Blue</td>
</tr>
<tr>
<td>77</td>
<td>Red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>[12,77]</td>
</tr>
<tr>
<td>Blue</td>
<td>[56]</td>
</tr>
</tbody>
</table>

### Partition II

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>Yellow</td>
</tr>
<tr>
<td>188</td>
<td>Blue</td>
</tr>
<tr>
<td>192</td>
<td>Blue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>[104]</td>
</tr>
<tr>
<td>Blue</td>
<td>[188,192]</td>
</tr>
</tbody>
</table>

Source: F. Gessert, Bagend / M. Kleppmann: "Designing data-intensive applications"
Local Secondary Indexing (Cont.)

Partitioning By Document

Partition I

<table>
<thead>
<tr>
<th>Data</th>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Red</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Blue</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>Red</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Red</td>
<td>[12,77]</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>[56]</td>
</tr>
</tbody>
</table>

Partition II

<table>
<thead>
<tr>
<th>Data</th>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>Yellow</td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>Blue</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>Blue</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yellow</td>
<td>[104]</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>[188,192]</td>
</tr>
</tbody>
</table>

Indexing is always local to a partition.

WHERE color=blue

Source: F. Gessert, Baqend / M. Kleppmann: "Designing data-intensive applications"
Local Secondary Indexing (Cont.)

Partitioning By Document

Partition I

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Red</td>
</tr>
<tr>
<td>56</td>
<td>Blue</td>
</tr>
<tr>
<td>77</td>
<td>Blue</td>
</tr>
<tr>
<td>Term</td>
<td>Match</td>
</tr>
<tr>
<td>Red</td>
<td>[104]</td>
</tr>
<tr>
<td>Blue</td>
<td>[188, 192]</td>
</tr>
</tbody>
</table>

Implemented in

- MongoDB
- Riak
- Cassandra
- Elasticsearch
- SolrCloud
- VoltDB

WHERE color=blue

Scatter-gather query pattern.

Source: F. Gessert, Baqend / M. Kleppmann: “Designing data-intensive applications"
### Global Secondary Indexing

#### Partitioning By Term

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Red</td>
</tr>
<tr>
<td>56</td>
<td>Blue</td>
</tr>
<tr>
<td>77</td>
<td>Red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>[104]</td>
</tr>
<tr>
<td>Blue</td>
<td>[56, 188, 192]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>Yellow</td>
</tr>
<tr>
<td>188</td>
<td>Blue</td>
</tr>
<tr>
<td>192</td>
<td>Blue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>[12, 77]</td>
</tr>
</tbody>
</table>

Source: F. Gessert, Bagend / M. Kleppmann: “Designing data-intensive applications”
Global Secondary Indexing (Cont.)

Partitioning By Term

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>104</td>
</tr>
<tr>
<td>Blue</td>
<td>56, 188, 192</td>
</tr>
</tbody>
</table>

Consistent index maintenance requires distributed transaction.

Data

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>Yellow</td>
</tr>
<tr>
<td>188</td>
<td>Blue</td>
</tr>
<tr>
<td>192</td>
<td>Blue</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>12, 77</td>
</tr>
</tbody>
</table>

Targeted Query

WHERE color = blue

Source: F. Gessert, Baqend / M. Kleppmann: "Designing data-intensive applications"
### Global Secondary Indexing (Cont.)

#### Partitioning By Term

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
<th>Partition I</th>
<th>Partition II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td></td>
<td>Key</td>
<td>Color</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>56</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td></td>
<td>Yellow</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blue</td>
<td></td>
</tr>
</tbody>
</table>

**Implemented in**
- DynamoDB
- Oracle Datawarehouse
- Riak (Search)
- Cassandra (Search)

**Targeted Query**

```sql
WHERE color=blue
```
Query Processing

- **Local Secondary Indexing**
  - Fast writes, scatter-gather queries

- **Global Secondary Indexing**
  - Slow or inconsistent writes, fast queries

- **(Distributed) Query Planning**
  - Scarce in NoSQL systems but increasing (e.g. left-outer equi-joins in MongoDB)

- **Analytics Frameworks**
  - Fallback for missing query capabilities

- **Materialized Views**
  - Similar to global indexing
NoSQL Database Systems: Techniques

- Consistency
- Sharding
- Replication
- Storage Management
- Query Processing

• Classification by Techniques
Redis

Sharding
- Range-Sharding
- Hash-Sharding
- Entity-Group Sharding
- Consistent Hashing
- Shared Disk

Replication
- Transaction Protocol
- Sync. Replication
- Async. Replication
- Primary Copy
- Update Anywhere

Storage Management
- Logging
- Update-in-Place
- Caching
- In-Memory
- Append-Only Storage

Query Processing
- Global Index
- Local Index
- Query Planning
- Analytics
- Materialized Views

Source: F. Gessert, Baqend, 2017
Dynamo and Riak

Sharding
- Range-Sharding
- Hash-Sharding
- Entity-Group Sharding
- Consistent Hashing
- Shared Disk

Replication
- Transaction Protocol
- Sync. Replication
- Async. Replication
- Primary Copy
- Update Anywhere

Storage Management
- Logging
- Update-in-Place
- Caching
- In-Memory
- Append-Only Storage

Query Processing
- Global Index
- Local Index
- Query Planning
- Analytics
- Materialized Views

Source: F. Gessert, Baqend, 2017
HBase

- **Sharding**
  - Range-Sharding
  - Hash-Sharding
  - Entity-Group Sharding
  - Consistent Hashing
  - Shared Disk

- **Replication**
  - Transaction Replication
  - Sync. Replication
  - Async. Replication
  - Primary Copy
  - Update Anywhere

- **Storage Management**
  - Logging
  - Update-in-Place
  - Caching
  - In-Memory
  - Append-Only Storage

- **Query Processing**
  - Global Index
  - Local Index
  - Query Planning
  - Analytics
  - Materialized Views

**Source:** F. Gessert, Baqend, 2017
Cassandra

Sharding
- Range-Sharding
- Hash-Sharding
- Entity-Group Sharding
- Consistent Hashing
- Shared Disk

Replication
- Transaction Protocol
- Sync. Replication
- Async. Replication
- Primary Copy
- Update Anywhere

Storage Management
- Logging
- Update-in-Place
- Caching
- In-Memory
- Append-Only Storage

Query Processing
- Global Index
- Local Index
- Query Planning
- Analytics
- Materialized Views

Source: F. Gessert, Baqend, 2017
NoSQL Database Systems: Techniques

- Consistency
- Sharding
- Replication
- Storage Management
- Query Processing
- Classification by Techniques
NoSQL Database Systems

✓ Foundations

✓ Data Modeling

✓ Application Development

✓ Techniques for Scalability, Availability and Consistency

• Decision Guidance: Select the Right DBMS