CS-562 Advanced Topics in Databases

Haridimos Kondylakis

Computer Science Department, University of Crete
NoSQL

• NoSQL:
  • Not Only SQL.

• User case of NoSQL?
  • Massive write performance.
  • Fast key value look ups.
  • Flexible schema and data types.
  • No single point of failure.
  • Fast prototyping and development.
  • Scalability.
  • Easy maintenance.
NoSQL: concept

- NoSQL is a non-relational database management system, different from traditional RDBMS in significant ways.

- Carlo Strozzi used the term NoSQL in 1998 to name his lightweight, open-source relational database that did not expose the standard SQL interface.
NoSQL concept

• In 2009, Eric Evans reused the term to refer databases which are non-relational, distributed, and does not conform to ACID

 ✓ The NoSQL term should be used as in the Not-Only-SQL and not as No to SQL or Never SQL
Motives Behind NoSQL

• Big data.
• Scalability.
• Data format.
• Manageability.

• Scale up, Vertical scalability.
  • Increasing server capacity.
  • Adding more CPU, RAM.
  • Managing is hard.
  • Possible down times
Scalability

• Scale out, Horizontal scalability.
  • Adding servers to existing system with little effort, aka Elastically scalable.
  • Shared nothing.
  • Use of commodity/cheap hardware.
  • Heterogeneous systems.
  • Controlled Concurrency (avoid locks).
  • Service Oriented Architecture. Local states.
    • Decentralized to reduce bottlenecks.
    • Avoid Single point of failures.
  • Asynchrony.
Database Attributes

Databases require 4 properties:

- **Atomicity**
  - When an update happens, it is “all or nothing”

- **Consistency**
  - The state of various tables must be consistent (relations, constraints) at all times.

- **Isolation**
  - Concurrent execution of transactions produces the same result as if they occurred sequentially.

- **Durability**
  - Once committed, the results of a transaction persist against various problems like power failure etc.

- These properties ensure that data is protected even with complex updates and system failures.

- Any data store can achieve Atomicity, Isolation and Durability but do you always need consistency? **No.**

- By giving up ACID properties, one can achieve higher performance and scalability.
CAP theory
CAP Theorem

• Also known as Brewer’s Theorem by Prof. Eric Brewer, published in 2000 at University of Berkeley.

• “Of three properties of a shared data system: data consistency, system availability and tolerance to network partitions, only two can be achieved at any given moment.”

• Proven by Nancy Lynch et al. MIT labs.

Theory of NOSQL: CAP

**GIVEN:**
- Many nodes
- Nodes contain *replicas of partitions* of the data

- **Consistency**
  - All replicas contain the same version of data
  - Client always has the same view of the data (no matter what node)

- **Availability**
  - System remains operational on failing nodes
  - All clients can always read and write

- **Partition tolerance**
  - multiple entry points
  - System remains operational on system split (communication malfunction)
  - System works well across physical network partitions

**CAP Theorem:** satisfying all three at the same time is impossible
CAP theorem for NoSQL

✓ What the CAP theorem really says:
  • If you cannot limit the number of faults and requests can be directed to any server and you insist on serving every request you receive then you cannot possibly be consistent

✓ How it is interpreted:
  • You must always give something up: consistency, availability or tolerance to failure and reconfiguration
A Simple Proof

Consistent and Available
No partition.

![Diagram](image-url)
A Simple Proof

Available and partitioned
Not consistent, we get back old data.
A Simple Proof

Consistent and partitioned
Not available, waiting...

App

New Data

Wait for new data
RDB ACID to NoSQL BASE

Atomicity
Consistency
Isolation
Durability

Basically
Available

Soft-state
(State of system may change over time)

Eventually consistent
(Asynchronous propagation)

Pritchett, D.: BASE: An Acid Alternative (queue.acm.org/detail.cfm?id=1394128)
CAP Theorem: Proof

• A simple proof using two nodes:
CAP Theorem: Proof

• A simple proof using two nodes:

A

\[\text{Not Consistent!}\]

B

Respond to client
CAP Theorem: Proof

• A simple proof using two nodes:

Wait to be updated
CAP Theorem: Proof

• A simple proof using two nodes:

A gets updated from B

Not Partition Tolerant!
A Clash of cultures

✓ ACID:
  • Strong consistency.
  • Less availability.
  • Pessimistic concurrency.
  • Complex.

✓ BASE:
  • Availability is the most important thing. Willing to sacrifice for this (CAP).
  • Weaker consistency (Eventual).
  • Best effort.
  • Simple and fast.
  • Optimistic.
Consistent, Available (CA) Systems have trouble with partitions and typically deal with it with replication.

Consistent, Partition-Tolerant (CP) Systems have trouble with availability while keeping data consistent across partitioned nodes.

Available, Partition-Tolerant (AP) Systems achieve "eventual consistency" through replication and verification.

Consistency: All clients always have the same view of the data.

Availability: Each client can always read and write.

http://blog.nahurst.com/visual-guide-to-nosql-systems
noSQL Data Models

- Key/Value Pairs
- row/tabular
- Documents
- Columns
- Graphs
- and correspondingly...
Categories of NoSQL storages

- Key-Value
  - memcached
  - Redis
  - Dynamo

- Tabular
  - BigTable, HBase

- Column Family
  - Cassandra

- Document-oriented
  - MongoDB

- Graph (beyond noSQL)
  - Neo4j
  - TITAN
Key-Value Stores

• “Dynamo: Amazon’s Highly Available Key-Value Store” (2007)
• Data model:
  • Global key-value mapping
  • Highly fault tolerant (typically)
• Examples:
  • Riak, Redis, Voldemort
Column Family (BigTable)

- Google’s “Bigtable: A Distributed Storage System for Structured Data” (2006)
- Data model:
  - A big table, with column families
  - Map-reduce for querying/processing
- Examples:
  - HBase, HyperTable, Cassandra
Document Databases

• Data model
  • Collections of documents
  • A document is a key-value collection
  • Index-centric, lots of map-reduce

• Examples
  • CouchDB, MongoDB
Graph Databases

• Data model:
  • Nodes with properties
  • Named relationships with properties
  • Hypergraph, sometimes

• Examples:
  • Neo4j, Sones GraphDB, OrientDB, InfiniteGraph, AllegroGraph
Complexity

90% of use cases

still billions of Nodes & relationships
Key-value store
KV-stores and Relational Tables

- KV-stores seem very simple. They can be viewed as two-column (key, value) tables with a single key column.
- But they can be used to implement more complicated relational tables:

<table>
<thead>
<tr>
<th>State</th>
<th>ID</th>
<th>Population</th>
<th>Area</th>
<th>Senator_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>1</td>
<td>4,822,023</td>
<td>52,419</td>
<td>Sessions</td>
</tr>
<tr>
<td>Alaska</td>
<td>2</td>
<td>731,449</td>
<td>663,267</td>
<td>Begich</td>
</tr>
<tr>
<td>Arizona</td>
<td>3</td>
<td>6,553,255</td>
<td>113,998</td>
<td>Boozman</td>
</tr>
<tr>
<td>Arkansas</td>
<td>4</td>
<td>2,949,131</td>
<td>53,178</td>
<td>Flake</td>
</tr>
<tr>
<td>California</td>
<td>5</td>
<td>38,041,430</td>
<td>163,695</td>
<td>Boxer</td>
</tr>
<tr>
<td>Colorado</td>
<td>6</td>
<td>5,187,582</td>
<td>104,094</td>
<td>Bennet</td>
</tr>
</tbody>
</table>

...
The KV-version of the previous table includes one table indexed by the actual key, and others by an ID.

<table>
<thead>
<tr>
<th>State</th>
<th>ID</th>
<th>ID</th>
<th>Population</th>
<th>ID</th>
<th>Area</th>
<th>ID</th>
<th>Senator_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>1</td>
<td>1</td>
<td>4,822,023</td>
<td>1</td>
<td>52,419</td>
<td>1</td>
<td>Sessions</td>
</tr>
<tr>
<td>Alaska</td>
<td>2</td>
<td>2</td>
<td>731,449</td>
<td>2</td>
<td>663,267</td>
<td>2</td>
<td>Begich</td>
</tr>
<tr>
<td>Arizona</td>
<td>3</td>
<td>3</td>
<td>6,553,255</td>
<td>3</td>
<td>113,998</td>
<td>3</td>
<td>Boozman</td>
</tr>
<tr>
<td>Arkansas</td>
<td>4</td>
<td>4</td>
<td>2,949,131</td>
<td>4</td>
<td>53,178</td>
<td>4</td>
<td>Flake</td>
</tr>
<tr>
<td>California</td>
<td>5</td>
<td>5</td>
<td>38,041,430</td>
<td>5</td>
<td>163,695</td>
<td>5</td>
<td>Boxer</td>
</tr>
<tr>
<td>Colorado</td>
<td>6</td>
<td>6</td>
<td>5,187,582</td>
<td>6</td>
<td>104,094</td>
<td>6</td>
<td>Bennet</td>
</tr>
</tbody>
</table>

...
KV-stores and Relational Tables

You can add indices with new KV-tables:

Thus KV-tables are used for **column-based storage**, as opposed to row-based storage typical in older DBMS.

<table>
<thead>
<tr>
<th>State</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>1</td>
</tr>
<tr>
<td>Alaska</td>
<td>2</td>
</tr>
<tr>
<td>Arizona</td>
<td>3</td>
</tr>
<tr>
<td>Arkansas</td>
<td>4</td>
</tr>
<tr>
<td>California</td>
<td>5</td>
</tr>
<tr>
<td>Colorado</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4,822,023</td>
</tr>
<tr>
<td>2</td>
<td>731,449</td>
</tr>
<tr>
<td>3</td>
<td>6,553,255</td>
</tr>
<tr>
<td>4</td>
<td>2,949,131</td>
</tr>
<tr>
<td>5</td>
<td>38,041,430</td>
</tr>
<tr>
<td>6</td>
<td>5,187,582</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Senator_1</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sessions</td>
<td>1</td>
</tr>
<tr>
<td>Begich</td>
<td>2</td>
</tr>
<tr>
<td>Boozman</td>
<td>3</td>
</tr>
<tr>
<td>Flake</td>
<td>4</td>
</tr>
<tr>
<td>Boxer</td>
<td>5</td>
</tr>
<tr>
<td>Bennet</td>
<td>6</td>
</tr>
</tbody>
</table>

**Index**

**Index_2**

OR: the value field can contain complex data (next page)
Key-Values: Examples

• Amazon:
  • Key: customerID
  • Value: customer profile (e.g., buying history, credit card, ..)

• Facebook, Twitter:
  • Key: UserID
  • Value: user profile (e.g., posting history, photos, friends, …)

• iCloud/iTunes:
  • Key: Movie/song name
  • Value: Movie, Song

• Distributed file systems
  • Key: Block ID
  • Value: Block
System Examples

- Amazon
  - Dynamo: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)
- BigTable/HBase/Hypertable: distributed, scalable data storage
- Cassandra: “distributed data management system” (Facebook)
- Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)
- eDonkey/eMule: peer-to-peer sharing system
Key-Value Store

• Also called a Distributed Hash Table (DHT)
• Main idea: partition set of key-values across many machines
Challenges

- **Fault Tolerance**: handle machine failures without losing data and without degradation in performance

- **Scalability**: 
  - Need to scale to thousands of machines 
  - Need to allow easy addition of new machines 

- **Consistency**: maintain data consistency in face of node failures and message losses

- **Heterogeneity** (if deployed as peer-to-peer systems): 
  - Latency: 1ms to 1000ms 
  - Bandwidth: 32Kb/s to several GB/s
Key Operators

• put(key, value): where do you store a new (key, value) tuple?
• get(key): where is the value associated with a given “key” stored?

• And, do the above while providing
  • Fault Tolerance
  • Scalability
  • Consistency
Directory-Based Architecture

- Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys.

```
put(K14, V14)
```

```
<table>
<thead>
<tr>
<th>K5</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>
```

```
N1
```
```
N2
```
```
N3
```
```
N50
```
Directory-Based Architecture

- Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys.

```
get(K14)  V14
```

```
<table>
<thead>
<tr>
<th>K5</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>
```

```
N1

K5  V5

N2

K14 V14

N3

K105 V105

N50

...
Directory-Based Architecture

- Having the master relay the requests → recursive query
- Another method: iterative query (this slide)
  - Return node to requester and let requester contact node

```
put(K14, V14)  N3
put(K14, V14)  Master/Directory
```

```
<table>
<thead>
<tr>
<th>K5</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>
```

```
N1
N2
N3
... 
N50
```
Directory-Based Architecture

- Having the master relay the requests → **recursive query**
- Another method: **iterative query**
  - Return node to requester and let requester contact node

```
get(K14)  N3  V14
```

Master/Directory

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K5</td>
<td>N2</td>
</tr>
<tr>
<td>K14</td>
<td>N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>
```

N1  N2  N3  N50
Iterative vs. Recursive Query

- **Recursive Query**
  - Advantages:
    - Faster (latency), as typically master/directory closer to nodes
    - Easier to maintain consistency, as master/directory can serialize puts()/gets()
  - Disadvantages: scalability bottleneck, as all “Values” go through master/directory

- **Iterative Query**
  - Advantages: more scalable
  - Disadvantages: slower (latency), harder to enforce data consistency
Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures (recursive version)

```
put(K14, V14)
N1, N3

put(K14, V14)
N1

put(K14, V14)
N1, N3

K5  N2
K14  N1, N3
K105  N50
```

Master/Directory

N1  N2  N3  ...  N50

K14  V14
K5   V5
K14  V14
K105 V105
Fault Tolerance

- Again, we can have
  - **Recursive** replication (previous slide)
  - **Iterative** replication (this slide)

```
put(K14, V14)
```

```
K14 V14
N1, N3
```

```
Master/Directory
```

```
<table>
<thead>
<tr>
<th>K5</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>N1,N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>
```

```
N_1
K14 V14
```

```
N_2
K5 V5
```

```
N_3
K14 V14
```

```
N_50
K105 V105
```

```
Scalability

- Storage: use more nodes
- Request Throughput:
  - Can serve requests from all nodes on which a value is stored in parallel
  - Large “values” can be broken into blocks (HDFS files are broken up this way)
  - Master can replicate a popular value on more nodes
- Master/directory scalability:
  - Replicate it
  - Partition it, so different keys are served by different masters/directories
Scalability: Load Balancing

- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available

- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node

- What happens when a node fails?
  - Need to replicate values from failed node to other nodes
Replication Challenges

• Need to make sure that a value is replicated correctly

• How do you know a value has been replicated on every node?
  • Wait for acknowledgements from every node

• What happens if a node fails during replication?
  • Pick another node and try again

• What happens if a node is slow?
  • Slow down the entire put()? Pick another node

• In general, with multiple replicas
  • Slow puts and fast gets
Consistency

• How close does a distributed system emulate a single machine in terms of read and write semantics?

• Q: Assume put(K14, V14’) and put(K14, V14”’ are concurrent, what value ends up being stored?
  
• A: assuming put() is atomic, then either V14’ or V14”, right?

• Q: Assume a client calls put(K14, V14) and then get(K14), what is the result returned by get()?
  
• A: It should be V14, right?

• Above semantics, not trivial to achieve in distributed systems
Concurrent Writes (Updates)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order.

Master/Directory

- $\text{put}(K14, V14')$ and $\text{put}(K14, V14'')$ reach $N1$ and $N3$ in different order.
- What does $\text{get}(K14)$ return?
  - Undefined!
Concurrent Writes (Updates)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order

```
Master/Directory

<table>
<thead>
<tr>
<th>K5</th>
<th>N2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K14</td>
<td>N1,N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>
```

- \(\text{put}(K14, V14')\) and \(\text{put}(K14, V14'')\) reach N1 and N3 in different order
- What does \(\text{get}(K14)\) return?
  - Undefined!

```
N1

<table>
<thead>
<tr>
<th>K14</th>
<th>V14'</th>
</tr>
</thead>
</table>

N2

| K5  | V5   |

N3

| K14 | V14''|

N50

...
Read after Write

- Read not guaranteed to return value of latest write
  - Can happen if Master processes requests in different threads

Master/Directory

<table>
<thead>
<tr>
<th>Key</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>K5</td>
<td>N2</td>
</tr>
<tr>
<td>K14</td>
<td>N1,N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
</tr>
</tbody>
</table>

- get(K14) happens right after put(K14, V14')
- get(K14) reaches N3 before put(K14, V14')!
• Large variety of consistency models:
  • Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
    • Think “one updated at a time”
  • Eventual consistency: given enough time all updates will propagate through the system
    • One of the weakest forms of consistency; used by many systems in practice
  • And many others: causal consistency, sequential consistency, strong consistency, …
Strong Consistency

• Assume Master serializes all operations

• Challenge: master becomes a bottleneck
  • Not addressed here

• Still want to improve performance of reads/writes → quorum consensus
Quorum Consensus

- Improve `put()` and `get()` operation performance

- Define a replica set of size N
  - `put()` waits for acks from at least W replicas
  - `get()` waits for responses from at least R replicas
  - W + R > N

- Why does it work?
  - There is at least one node that contains the update
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N3, N4}
- Assume put() on N3 fails
• Now, issuing get() to any two nodes out of three will return the answer.

```plaintext
get(K14) → V14
get(K14) → nil
```

```
N1  N2  N3  N4
K14 V14 K14 V14
```
References

- **Wenfei Fan**, University of Edinburg
  - Lecture Notes on Research Topics in Distributed Databases

- **David G. Sullivan**, Harvard University
  - Lecture Notes on NoSQL Databases

- **Navneet Potti, Jignesh Patel**, University of Wisconsin
  - DAQ: A New Paradigm for Approximate Query Processing. VLDB 2015
Key-value store case study: Dynamo
System Assumptions and Requirements

• simple read and write operations to a data item that is uniquely identified by a key.

• Most of Amazon’s services can work with this simple query model and do not need any relational schema.

• targeted applications - store objects that are relatively small (usually less than 1 MB)

• Dynamo targets applications that operate with weaker consistency (the “C” in ACID) if this results in high availability.

• there are no security related requirements such as authentication and authorization
System architecture

- Partitioning
- High Availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection
Partition Algorithm

- *Consistent hashing*: the output range of a hash function is treated as a fixed circular space or “ring”.
- *“Virtual Nodes”*: Each node can be responsible for more than one virtual node.
Replication

• Each data item is replicated at N hosts.
• “preference list”: The list of nodes that is responsible for storing a particular key.
Vector Clocks

- Used for conflict **detection** of data.
- Timestamp based resolution of conflicts is not enough.

Time 1:

Time 2:

Time 3:

Time 4:

Time 5:

Conflicts detection

Replicated

Update
Vector Clock

• A vector clock is a list of (node, counter) pairs.

• Every version of every object is associated with one vector clock.

• *If the counters on the first object’s clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.*
# Summary of techniques used in Dynamo and their advantages

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>