Non-Relational Databases & Cloud Datastores

Pelle Jakovits

Tartu, 9 March 2020
Outline

• Relational databases
• NoSQL Movement
  – Relational model
  – Non-relational data models
  – Non-relational databases
  – Relational vs Non-Relational
• Cloud managed datastores
The Relational Model

• Data is stored in tables
  – Different entity types in different tables

• Strict relationships between tables
  – Foreign key references between columns

• The expected format of the data is specified with a restrictive schema
  – Fixed data types
  – Allow/deny null values

• Data is typically accessed using Structured Query Language (SQL)
Database Scaling

• Vertical scaling – scale the server
• Horizontal scaling – across a cluster of server
• Relational model does not scale well horizontally
• There are too many dependencies in relational model
• Database sharding is one approach
Sharding in Relational databases

https://www.mysql.com/products/cluster/scalability.html
Dependencies in relational databases
The NoSQL Movement

• Driven by the rise of „Big Data“ and Cloud Computing
• NoSQL is a broad term with no clear boundaries
  – The term NoSQL itself is very misleading
  – NoSQL -> NonSQL -> Non Relational
• Emergence of persistence solutions using non-relational data models
• Non-relational data models are based on Key - Value structure
• Simpler schema-less key-value based data models scale better than the relational model
• Tradeoff between data consistency and high availability
CAP Theorem

- It is impossible for a distributed computer system to simultaneously guarantee all three of the following:
  - **Consistency** - every read receives the most recent write or an error
  - **Availability** - every request receives a proper response
  - **Partition/Fault tolerance** - the system continues to operate despite arbitrary partitioning (network failures, dropped packets)

- Choice is usually between consistency or availability when failures happen.

- NoSQL solutions which are more focused on availability try to achieve eventual consistency
CAP Theorem

Availability

MySQL, Postgre, Oracle SQL
Dynamo, Cassandra, CouchDB, Riak
BigTable, HyperTable, HBase, MongoDB, BerkeleyDB, Redis

Consistency

Partition Tolerance
Eventual consistency

• In the context of distributed databases:
  – Consistency means: read request to any database node should return the same data
• Eventual consistency: data of each node of the database will become consistent eventually!
• Eventual consistency allows to provide low latency with the greater risk of returning stale data.
• When trying to aim for both consistency or availability
  – Will have high latency instead (wait until consistent)
Eventual consistency

Node 1

write request (X)

Node 2

write request (X) forwarded to Node 2 eventually

All nodes get updated and consistent eventually

read request (X)

stale data (X) returned as X hasn't been updated on Node 2
Non-relational Data Models
The Key-value Model

- Data stored as key-value pairs
- The value is an opaque blob to the database
- Examples: Dynamo, Riak, Apache Ignite, ArangoDB, Berkeley DB, Couchbase, FoundationDB, InfinityDB, MemcacheDB, MUMPS, Oracle NoSQL, OrientDB, Redis, SciDB, ZooKeeper
Benefits of the Key-value Model

- Horizontal scalability
  - Data with the same Key stored close to each other
  - Suitable for cloud computing
- Flexible schema-less models suitable for unstructured data
- Queries: Get, Put and Delete.
- Fetching data by key can be very fast
Key-Value Model Key design

• AWS S3 "keys":
  – https://s3.Region.amazonaws.com/bucket-name/KeyName

• Keys can be complex:

  1. employee:1:firstName = "Martin"
  2. employee:2:firstName = "John"
  3. payment:1:1:amount = "10000"
  4. payment:1:1:date = "01/12/2019"
  5. payment:2:1:amount = "5000"
  6. payment:2:1:date = "01/12/2019"
The Document-oriented Model

• Data is also stored as key-value pairs
  – Value is a "document" and has further structure
• No strict schema
  – Expectation is that documents contain schema
  – Not enforced in any way
• Query data based on document structure
• Examples: CouchDB, MongoDB, ArangoDB, BaseX, Clusterpoint, Couchbase, Cosmos DB, IBM Domino, MarkLogic, OrientDB, RethinkDB
Example JSON document

```json
{
  "name":"Asheville Veedub Volkswagon Repair & Restoration",
  "address":"851 Charlotte Hwy",
  "attributes":{
    "BusinessAcceptsCreditCards":"true",
    "ByAppointmentOnly":"false",
    "GoodForKids":"true"
  },
  "business_id":"0KwutFa520HgPLWtFv02EQ",
  "categories":"Auto Repair, Automotive",
  "city":"Fairview",
  "is_open":1,
  "latitude":35.5431561,
  "longitude":-82.4419299,
  "neighborhood":"
  "postal_code":"28730",
  "review_count":7,
  "stars":4.5,
  "state":"NC"
}
```

- Aggregates are described in JSON using map and array data structures
CouchDB

- Document oriented (JSON)
- RESTful query interface
- Built in web server
- Web based query front-end Futon
CouchDB

• MapReduce is the primary query method
  – JavaScript and Erlang

• Materialized views as results of incremental MapReduce jobs

• CouchApps – JavaScript-heavy web applications
  – Built entirely inside CouchDB
  – Without a separate web server for the logic layer
Map function

• Map function is applied to every document pair in the input list
• Input to the user defined map functions is a single JSON document
• Output is a key and value pair.

\[ \text{map} \ (\text{doc}) \rightarrow (k', v') \]
Reduce function

- All Key&Value pairs produced in the Map step are grouped by the key and values are combined into a list
  - This happens between Map and Reduce stages
- Input to a Reduce function is a unique key and a list of values: \((\text{Key}, [\text{Value}])\)
- Reduce (aggregation) function is applied on the list of values
- Output is another key and value pair

\[
\text{reduce} \ (k', [v']) \rightarrow (k'', v'')
\]

\[
v'' = F_{\text{Reduce}}([v'])
\]
Query example

Documents:

```
{ "id": 2, "position": "programmer", "first_name": "James", "salary": 100 }
{ "id": 7, "position": "support", "first_name": "John", "salary": 23 }
```

Let's compute average salary for each unique position

Map:
```
function(doc, meta) {
    if (doc.position && doc.salary) { emit(doc.position, doc.salary); }
}
```

Reduce:
```
function(key, values, rereduce) {
    return sum(values)/values.length;
}
```

Some Reduce functions have been pre-defined: _sum(), _count(), _stats()
The Column Family Model

• Data stored in large sparse tabular structures
• Columns are grouped into column families
  – Column family is a meaningful group of columns
  – Similar concept as a table in relational database
• A record can be thought of as a two-level map
  – Column Family -> Column -> Value
• New columns can be added at any time
• Examples: BigTable, Cassandra, HBase, Accumulo
Cassandra

• High scalability and high availability
• Column family model
• Uses SQL-Like Cassandra Query Language (CQL)

• Powered By Cassandra:
  – Apple's: 75,000 nodes storing over 10 PB of data
  – Netflix: 2,500 nodes, 420 TB, 1 trillion requests per day
Cassandra

• Static column family (Classical SQL "table"-like)
  – Fixed number and types of columns
  – Rows are not required to have values

• Dynamic column family (Horizontally growing "table")
  – No fixed column names or types
  – Each row can contain different number of columns
  – Columns can be added on the fly

• Super-column families
  – contains super columns that contains column families
## Column Family Example

### Static column families

<table>
<thead>
<tr>
<th>_id</th>
<th>names</th>
<th>contacts</th>
<th>messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_id</td>
<td>username</td>
<td>firstname</td>
</tr>
<tr>
<td>a001</td>
<td></td>
<td>jsmith01</td>
<td>John</td>
</tr>
<tr>
<td>b014</td>
<td></td>
<td>pauljones</td>
<td></td>
</tr>
</tbody>
</table>

### Dynamic column family

<table>
<thead>
<tr>
<th>name</th>
<th>username</th>
<th>firstname</th>
<th>lastname</th>
<th>phone</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsmith</td>
<td>jsmith</td>
<td>John</td>
<td>Smith</td>
<td>555 0001</td>
<td><a href="mailto:jsmith@example.com">jsmith@example.com</a></td>
</tr>
<tr>
<td>pauljones</td>
<td>pauljones</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Dynamic column family**

- **Names**
  - username: jsmith
  - firstname: John
  - lastname: Smith

- **Contacts**
  - phone: 555 0001
  - email: jsmith@example.com

- **Messages**
  - item1: Message1
  - item2: Message2
  - new message

---

**Static column families**

- **Names**
  - username: jsmith
  - firstname: John
  - lastname: Smith

- **Contacts**
  - phone: 555 0001
  - email: jsmith@example.com

- **Messages**
  - item1: Message1
  - item2: Message2
  - ... | ... |
  - itemN: MessageN

---
Cassandra query example

```sql
SELECT race_name, point_id, lat_long AS CITY_LATITUDE_LONGITUDE
FROM cycling.route;
```

<table>
<thead>
<tr>
<th>race_name</th>
<th>point_id</th>
<th>city_latitude_longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>47th Tour du Pays de Vaud</td>
<td>1</td>
<td>('Onnens', (46.8444, 6.6667))</td>
</tr>
<tr>
<td>47th Tour du Pays de Vaud</td>
<td>2</td>
<td>('Champagne', (46.833, 6.65))</td>
</tr>
<tr>
<td>47th Tour du Pays de Vaud</td>
<td>3</td>
<td>('Novalle', (46.833, 6.6))</td>
</tr>
<tr>
<td>47th Tour du Pays de Vaud</td>
<td>4</td>
<td>('Vuiteboeuf', (46.8, 6.55))</td>
</tr>
<tr>
<td>47th Tour du Pays de Vaud</td>
<td>5</td>
<td>('Baulmes', (46.7833, 6.5333))</td>
</tr>
<tr>
<td>47th Tour du Pays de Vaud</td>
<td>6</td>
<td>('Les Clées', (46.7222, 6.5222))</td>
</tr>
</tbody>
</table>

Graph Databases

- Data stored as nodes and edges of a graph
- The nodes and edges can have fields and values
- Aimed at graph traversal queries in connected data
- Examples: Neo4J, FlockDB, AllegroGraph, ArangoDB, InfiniteGraph, Apache Giraph

Neo4J

- Open source NoSQL Graph Database
- Uses the Cypher Query Language for querying graph data
- Graph consists of Nodes, Edges and Attributes
You like Neo4j, let's find you a closest Neo4j expert through your network of friends.

MATCH (you {name:"You"})
MATCH (expert)-[:WORKED_WITH]-> (db:Database {name:"Neo4j"})
MATCH path = shortestPath( (you)-[:FRIEND*..5]-(expert) )

RETURN db, expert, path

https://neo4j.com/developer/cypher-query-language/
NoSQL characteristics: 
 Aggregate-oriented

• Non-relational data models aim to store data that needs to be aggregated together

• Aggregate is a collection of data that is treated as a unit
  – E.g. a customer and all his orders

• In normalized relational databases aggregates are computed using GroupBy operations

• Goal is to **de-normalize** data so minimal (or no) grouping and joining is needed!

• Keyed aggregates make for a natural unit of data sharding
NoSQL characteristics: Partitioning

- In Key-value model, Key acts as an index
- Secondary indexes can be created in some solutions
  - Implementations vary from partitioning to local distributed indexes
- Data is partitioned between different machines in the cluster
  - Similar to sharding
- Usually data is partitioned by rows or/and column families
- Users can often specify dynamic partitioning parameters
  - Gives control over how data is distributed
  - Very important for optimizing query speed
  - Use Case: Monthly reporting? Partition data by Months or Weeks.
  - Must be taken into account from the very start!
Why use Relational databases?

• Highly available and highly consistent
  – Old, tried and tested technologies
  – Well optimized for performance

• Strict schema means less mistakes
  – Easier to control the quality of data
  – Easier to avoid missing data, wrong data types, etc.

• Very good performance as long as data fits into a single machine
When use Non-Relational?

• When volume of data grows too large
  – Easier and cheaper to scale

• When the schema is not fixed
  – Store a query unstructured data
  – Prototyping: Build applications without pre-defining schema
  – Updating schemas is very simple

• When using structured documents (JSON, XML)
  – Similar schemainside documents, but can vary greatly
  – Nested sub-structures

• Many open source options
Cloud Data stores
Cloud Data storage services

• The goal of Cloud data storage services is to provide a Managed storage services
  – STorage as a Service (STaaS)
• Provider takes care of installing, deploying, configuring, scaling, sharding, backups, etc.
• Often provide different pricing options depending on object/file access frequency, expected latency, length of storage
• Overall fit into the same SQL, NoSQL database models
Types of Cloud Data stores

1. Bucket/Blob Storage
2. Managed SQL storage services
3. Managed NoSQL storage services
4. Managed File System services
Object / Bucket / Blob Storage

- Follow the Key-value non-relational model
- Storing large amount of unstructured data
  - No schemas
  - Images, Videos, Log files, backup files, etc.
- Amazon S3, Azure Blob storage, Google Cloud Storage, IBM Cloud Object Storage
- May have different modes for storage:
  - Real-time vs Low-frequency vs Archived
AWS S3

- Amazon Simple Storage Service (Amazon S3)
- Divides data into buckets
  - A logical unit of storage - unique web folder
  - Bucket can contain virtually unlimited folders and files
  - 100 buckets per user
- Storage classes
  - S3 Standard
  - Intelligent-Tiering (Automatic storage class selection, min 30 days)
  - Standard-Infrequent Access (Less frequent, but rapid access, min 30 days)
  - One Zone-Infrequent Access (Single availability Zone, min 30 days)
  - S3 Glacier (data archiving, access in minutes to hours, min 90 days)
  - S3 Glacier Deep Archive (access in 12 hours, min 180 days)
# AWS S3 prices

<table>
<thead>
<tr>
<th>Mode</th>
<th>Storage TB/Month</th>
<th>1M INFO requests</th>
<th>Data Retrieval per TB</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3 Standard</td>
<td>$23.55</td>
<td>$5.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>S3 Intelligent, standard</td>
<td>$23.55</td>
<td>$5.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>S3 Intelligent, Infrequent Access</td>
<td>$12.80</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>S3 Standard - Infrequent Access</td>
<td>$12.80</td>
<td>$10.00</td>
<td>$10.00</td>
</tr>
<tr>
<td>S3 One Zone - Infrequent Access</td>
<td>$10.24</td>
<td>$10.00</td>
<td>$10.00</td>
</tr>
<tr>
<td>S3 Glacier</td>
<td>$4.10</td>
<td>$50.00</td>
<td>$30.00</td>
</tr>
<tr>
<td>S3 Glacier Deep Archive</td>
<td>$1.01</td>
<td>$50.00</td>
<td>$20.00</td>
</tr>
</tbody>
</table>
Managed Cloud SQL storage

• Two main types:
  – Simple SQL server on demand
  – Fully managed, sharded SQL cluster on demand

• **Amazon (RDS)** - service for dynamically deploying and scaling SQL servers (Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, and SQL Server)

• **IBM Db2** – IBM own engine from 83. SQL, BigSQL (Hadoop), Data Warehouse, Analytics

• **Google Cloud SQL** - Managed MySQL, PostgreSQL, or SQL Server
Managed Cloud NoSQL storage

- **Key-Value**: AWS DynamoDB, Google Cloud Datastore
- **Document DB**: AWS DocumentDB, IBM Cloudant (CouchDB), Google Cloud Firestore
- **Column oriented**: AWS Managed Apache Cassandra Service, Google BigQuery
- **Graph DB**: AWS Neptune, IBM Graph
Cloud managed File Systems

• Amazon Elastic File System
  – Scalable, elastic, cloud-native NFS file system
  – Designed for providing parallel shared file system to thousands of EC2 instances.
  – Can be used with on-premise servers (NFS v4.0)
  – Fully managed (redundant, elastic and scalable)
  – **Standard vs Infrequent Access** storage classes.
    • Age-off policy (14, 30 days) for moving less frequently used files to slower/cheaper storage
Conclusions

• There has been a rise in non-relational (NoSQL) data stores
• This is related to the rise of cloud computing
  – Horizontal scaling is cheap
  – key-value models offer better scalability
• Cloud data storage services provide managed relational or non-relational databases as service
• The NoSQL landscape is extremely varied
  – key-value, document-oriented, column family, Graph database models
  – In reality, classification of databases into different models is not so straight forward
• Non-relational databases are often aggregation oriented
That is All

• This week's practice session
  – Using NoSQL databases: CouchDB

• Next Lecture
  – Private and Hybrid clouds