Multi-model and relational databases — a comparison

Percona Live Europe, 22 Sep 2015

Max Neunhöffer
About me

- Max Neunhöffer (@neunhoef) working for ArangoDB
- Mathematician turned database engineer
About

- about me
  - Max Neunhöffer (@neunhoef) working for ArangoDB
  - Mathematician turned database engineer

- about the talk
  - relational databases and NoSQL
  - data modelling and polyglot persistence
  - multi-model databases
  - compare relational and multi-model
  - ArangoDB
In the last millenium ...
In the last millenium ...

Databases were relational.
RDBMS — the good parts

- very mature technology
- sound theoretical foundations (relational calculus)
- decades of research
- SQL, a standard query language
- strong and reliable consistency guarantees (ACID)
- many well-educated database engineers
- quite memory efficient
- solid schema enforcement
RDBMS — the pain parts: scalability

Scalability
Scalability
RDBMS — the pain parts: scalability

Scalability

(horizontal on commodity hardware)

Icon made by Freepik from http://www.flaticon.com
Other reasons for distribution

- geographical availability
RDBMS — the pain parts: distribution

Other reasons for distribution

- geographical availability
- reliability and fault tolerance
**RDBMS — the pain parts: distribution**

<table>
<thead>
<tr>
<th>Other reasons for <strong>distribution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- geographical availability</td>
</tr>
<tr>
<td>- reliability and <strong>fault tolerance</strong></td>
</tr>
<tr>
<td>- high availability</td>
</tr>
</tbody>
</table>
RDBMS — the pain parts: distribution

Other reasons for distribution

- geographical availability
- reliability and fault tolerance
- high availability
- security
RDBMS — the pain parts: different data models

| Fact | Relational systems store ... |
**Fact**

Relational systems store ... **TABLES**
RDBMS — the pain parts: different data models

Fact

Relational systems store ... TABLES

Observation

Often, data arises in different formats: hash maps, links/graphs, inhomogeneous data, objects with relations, recursive data.
RDBMS — the pain parts: different data models

**Fact**
Relational systems store ... **TABLES**

**Observation**
Often, data arises *in different formats*: hash maps, links/graphs, inhomogeneous data, objects with relations, recursive data.

Sometimes, it is better to store the data *as it is!*
RDBMS — the pain parts: different data models

Fact
Relational systems store ... TABLES

Observation
Often, data arises in different formats: hash maps, links/graphs, inhomogeneous data, objects with relations, recursive data.

Sometimes, it is better to store and query the data as it is!
NoSQL — the 1st generation

Developed to solve specific problems, mostly internally in big, rich corporations:

- **BigTable** (Google, 2006)
  (scalability, no transactions, tables)

- **Dynamo** (Amazon, 2009)
  (scalability, high availability, key/value store, client conflict resolution)

- etc. . . .
NoSQL — the 2nd generation

Products (often open source), using the new ideas, mostly with a single data model:

- **Apache CouchDB** (Apache project, 2005) *(document store, scalable, eventually consistent, map/reduce engine for views)*
- **Neo4j** (Neo technology Inc., 2007) *(graph database, Cypher path pattern matching language)*
- **Apache Cassandra** (Apache project, 2008) *(column-oriented, large tables, high write performance)*
- **MongoDB** (MongoDB Inc., 2009) *(document store, very fast, scalable, has secondary indexes)*
- **redis/Redis Cluster** (redislabs, 2009) *(key/value store, high performance, single server, cluster extension)*
- **Apache HBase** (Apache project, 2009) *(large tables, free variant of BigTable, based on Hadoop)*
- etc. . . .
Polyglot Persistence

Idea

Use the right data model for each part of a system.
Polyglot Persistence

Idea
Use the right data model for each part of a system.

Take scalability needs into account!
A typical Use Case — an Online Shop

We need to hold
  - **customer** data: usually homogeneous, but still variations
A typical Use Case — an Online Shop

We need to hold

- **customer** data: usually homogeneous, but still variations

- **product** data: even for a specialised business quite inhomogeneous
A typical Use Case — an Online Shop

We need to hold

- **customer** data: usually homogeneous, but still variations

- **product** data: even for a specialised business quite inhomogeneous

- **shopping carts**: need very fast lookup by session key
We need to hold

- **customer** data: usually homogeneous, but still variations
- **product** data: even for a specialised business quite inhomogeneous
- **shopping carts**: need very fast lookup by session key
- **order and sales** data: relate customers and products
We need to hold

- **customer** data: usually homogeneous, but still variations
- **product** data: even for a specialised business quite inhomogeneous
- **shopping carts**: need very fast lookup by session key
- **order and sales** data: relate customers and products
- **recommendation engine** data: links between different entities
We need to hold

- **customer** data: usually homogeneous, but still variations
  MySQL

- **product** data: even for a specialised business quite inhomogeneous

- **shopping carts**: need very fast lookup by session key

- **order and sales** data: relate customers and products

- **recommendation engine** data: links between different entities
A typical Use Case — an Online Shop

We need to hold

- **customer** data: usually homogeneous, but still variations
  MySQL

- **product** data: even for a specialised business quite inhomogeneous

- **shopping carts**: need very fast lookup by session key

- **order** and **sales** data: relate customers and products

- **recommendation engine** data: links between different entities
A typical Use Case — an Online Shop

We need to hold

- **customer** data: usually homogeneous, but still variations
  MySQL

- **product** data: even for a specialised business quite inhomogeneous


- **shopping carts**: need very fast lookup by session key

- **order and sales** data: relate customers and products

- **recommendation engine** data: links between different entities
A typical Use Case — an Online Shop

We need to hold

- **customer** data: usually homogeneous, but still variations
  - MySQL

- **product** data: even for a specialised business quite inhomogeneous
  - **mongoDB**

- **shopping carts**: need very fast lookup by session key
  - **redis**

- **order** and **sales** data: relate customers and products
  - **mongoDB**

- **recommendation engine** data: links between different entities
A typical Use Case — an Online Shop

We need to hold

- **customer** data: usually homogeneous, but still variations
  - MySQL

- **product** data: even for a specialised business quite inhomogeneous
  - mongoDB

- **shopping carts**: need very fast lookup by session key
  - redis

- **order and sales** data: relate customers and products
  - mongoDB

- **recommendation engine** data: links between different entities
  - Neo4j
Polyglot Persistence is nice, but ...

**Disadvantages**

**Consequence:** One needs *multiple database systems* in the persistence layer of a *single* project!
Polyglot Persistence is nice, but ...

Disadvantages

Consequence: One needs multiple database systems in the persistence layer of a single project!

Wouldn’t it be nice, ...

...to enjoy the benefits without the disadvantages?
## Multi-model database

A multi-model database combines a document store with a graph database and is at the same time a key/value store.
<table>
<thead>
<tr>
<th>Multi-model database</th>
</tr>
</thead>
<tbody>
<tr>
<td>A multi-model database combines a document store with a graph database and is at the same time a key/value store, with a common query language for all three data models.</td>
</tr>
</tbody>
</table>
The Multi-Model Approach

Multi-model database

A multi-model database combines a document store with a graph database and is at the same time a key/value store, with a common query language for all three data models.

Important:
- is able to compete with specialised products on their turf
The Multi-Model Approach

Multi-model database

A multi-model database combines a document store with a graph database and is at the same time a key/value store, with a common query language for all three data models.

Important:
- is able to compete with specialised products on their turf
- allows for polyglot persistence using a single database
## Multi-model database

A **multi-model database** combines a document store with a graph database and is at the same time a key/value store, with a common query language for all three data models.

### Important:
- is able to compete with *specialised products* on their turf
- allows for polyglot persistence using a single database

**This is the 3rd generation of NoSQL databases.**
We need to hold

- **customer** data: usually homogeneous, but still variations
  - MySQL

- **product** data: even for a specialised business quite inhomogeneous
  - MongoDB

- **shopping carts**: need very fast lookup by session key
  - Redis

- **order and sales** data: relate customers and products
  - MongoDB

- **recommendation engine** data: links between different entities
Back to our Use Case — an Online Shop

We need to hold

- **customer** data: usually homogeneous, but still variations
  - MySQL
  - [ArangoDB](https://arangodb.com)
- **product** data: even for a specialised business quite inhomogeneous
  - [mongoDB](https://www.mongodb.com)
  - [ArangoDB](https://arangodb.com)
- **shopping carts**: need very fast lookup by session key
  - Redis
  - [ArangoDB](https://arangodb.com)
- **order and sales** data: relate customers and products
  - [mongoDB](https://www.mongodb.com)
  - [ArangoDB](https://arangodb.com)
- **recommendation engine** data: links between different entities
  - [Neuromat](https://www.neuromat.com)
  - [ArangoDB](https://arangodb.com)
Why is this possible at all?

Document stores and key/value stores

Document stores: have primary key, are key/value stores.
Document stores and key/value stores

Document stores: have primary key, are key/value stores. Without using secondary indexes, performance is nearly as good as with opaque data instead of JSON.
Why is this possible at all?

Document stores and key/value stores

Document stores: have primary key, are key/value stores. Without using secondary indexes, performance is nearly as good as with opaque data instead of JSON. Good horizontal scalability can be achieved for key lookups.
Why is this possible at all?

Document stores and graph databases

Graph database: would like to associate arbitrary data with vertices and edges, so JSON documents are a good choice.
Document stores and graph databases

Graph database: would like to associate arbitrary data with vertices and edges, so JSON documents are a good choice.

- A good edge index, giving fast access to neighbours. This can be a secondary index.
Why is this possible at all?

Document stores and graph databases

Graph database: would like to associate arbitrary data with vertices and edges, so JSON documents are a good choice.

- A good edge index, giving fast access to neighbours. This can be a secondary index.
- Graph support in the query language.
Why is this possible at all?

**Document stores and graph databases**

Graph database: would like to associate *arbitrary data* with vertices and edges, so JSON documents are a good choice.

- A good edge index, giving fast access to neighbours. This can be a secondary index.
- Graph support in the query language.
- Implementations of *graph algorithms* in the DB engine.
NoSQL Performance Test
ArangoDB, MongoDB, Neo4j and OrientDB

- Shortest path: ArangoDB 1311%, Neo4j 844%, MongoDB 1664%, OrientDB 1735%
- Neighbors*: ArangoDB 750%, Neo4j 234%, MongoDB 186%
- Single read: ArangoDB 67%, Neo4j 916%, MongoDB 88%
- Single write: ArangoDB 976%, Neo4j 1718%, MongoDB 196%
- Single write sync: ArangoDB 49%, Neo4j 44%, MongoDB 97%

(*) distance 1 and the distance 2 neighbors, each of them once.
Use case: Aircraft fleet management
We store all data as documents.
A multi-model data modeling approach

<table>
<thead>
<tr>
<th>Idea (see this blog post for details)</th>
</tr>
</thead>
<tbody>
<tr>
<td>We store all data as documents.</td>
</tr>
<tr>
<td>Since vertices and edges of graphs are documents, this allows to mix all three data models.</td>
</tr>
</tbody>
</table>
A multi-model data modeling approach

Idea (see this blog post for details)

We store all data as documents. Since vertices and edges of graphs are documents, this allows to mix all three data models.

- One document (a vertex) for
  - the fleet,
  - each aircraft, (in different vertex collections).
  - each component, and
  - each part
A multi-model data modeling approach

Idea (see this blog post for details)

We store all data as documents. Since vertices and edges of graphs are documents, this allows to mix all three data models.

- **One document** (a vertex) for
  - the fleet,
  - each aircraft, (in different vertex collections).
  - each component, and
  - each part
- **Containment** is via **edges** (an item points to those contained).
A multi-model data modeling approach

Idea (see this blog post for details)

We store all data as documents. Since vertices and edges of graphs are documents, this allows to mix all three data models.

- One document (a vertex) for
  - the fleet,
  - each aircraft, (in different vertex collections).
  - each component, and
  - each part
- Containment is via edges (an item points to those contained).
- Use document queries where the graph structure is irrelevant.
A multi-model data modeling approach

Idea (see this blog post for details)

We store all data as documents. Since vertices and edges of graphs are documents, this allows to mix all three data models.

- One document (a vertex) for
  - the fleet,
  - each aircraft, (in different vertex collections).
  - each component, and
  - each part
- Containment is via edges (an item points to those contained).
- Use document queries where the graph structure is irrelevant.
- Use graphy queries when containment of items matters.
A multi-model data modeling approach

**Idea (see this blog post for details)**

We store **all data as documents**. Since vertices and edges of graphs are documents, this allows to **mix all three data models**.

- **One document** (a vertex) for
  - the fleet,
  - each aircraft,
  - each component, and
  - each part
  (in different vertex collections).

- **Containment** is via **edges** (an item points to those contained).

- Use **document queries** where the graph structure is irrelevant.

- Use **graphy queries** when containment of items matters.

- Can **mix the two** within a single query.
A mix of them all

FOR p IN parts
    FILTER p.nextMaintenance <= "2015-05-15"
    LET path = GRAPH_SHORTEST_PATH("FleetGraph", p._id, 
        {isMaintainable: true},
        {direction: "inbound",
            stopAtFirstMatch: true})

    LET c = DOCUMENT(path[0].vertex)
    FOR person IN contacts
        FILTER person._key == c.contact
    RETURN {part: p._id, component: c, contact: person}

Find parts, their corresponding maintenance component and join a contact person.
Other use cases

- Enterprise hierarchies and rights management
- Social networks
- Version management
- Complex user-created data
- Workflow management
- Organisation systems
- Knowledge graphs
Other use cases

- Enterprise hierarchies and rights management
- Social networks
- Version management
- Complex user-created data
- Workflow management
- Organisation systems
- Knowledge graphs

Observation

Use cases that benefit from *multi-model* are actually prevalent!
## Comparison of relational and multi-model

<table>
<thead>
<tr>
<th>feature</th>
<th>relational</th>
<th>multi-model</th>
</tr>
</thead>
<tbody>
<tr>
<td>data model</td>
<td>tables</td>
<td>JSON, graphs, key/value</td>
</tr>
<tr>
<td>joins</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>transactions</td>
<td>yes</td>
<td>optional</td>
</tr>
<tr>
<td>relations</td>
<td>via foreign keys, link tables, joins</td>
<td>via foreign keys, link collections, joins or graphs</td>
</tr>
<tr>
<td>schema</td>
<td>enforced</td>
<td>optional</td>
</tr>
<tr>
<td>scalability</td>
<td>only vertical</td>
<td>vertical and horizontal</td>
</tr>
</tbody>
</table>
is a multi-model database (document store & graph database),
ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- including joins between different collections,
ArangoDB

- is a **multi-model database** (document store & graph database),
- is **open source and free** (Apache 2 license),
- offers convenient queries (via HTTP/REST and **AQL**),
- including **joins** between different collections,
- **configurable** consistency guarantees using **transactions**
- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- including joins between different collections,
- configurable consistency guarantees using transactions
- API extensible by JS code in the Foxx Microservice Framework,
ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- including joins between different collections,
- configurable consistency guarantees using transactions
- API extensible by JS code in the Foxx Microservice Framework,
- and enjoys good community as well as professional support.
Extensible through JavaScript

The Foxx Microservice Framework

Allows you to extend the HTTP/REST API by your own routes, which you implement in JavaScript running on the database server, with direct access to the C++ DB engine.
Extensible through JavaScript

The Foxx Microservice Framework

Allows you to extend the HTTP/REST API by your own routes, which you implement in JavaScript running on the database server, with direct access to the C++ DB engine.

Unprecedented possibilities for data centric services:
- custom-made complex queries or authorizations
Extensible through JavaScript

The Foxx Microservice Framework

Allows you to extend the HTTP/REST API by your own routes, which you implement in JavaScript running on the database server, with direct access to the C++ DB engine.

Unprecedented possibilities for data centric services:
- custom-made complex queries or authorizations
- schema-validation
Extensible through JavaScript

The Foxx Microservice Framework

Allows you to extend the HTTP/REST API by your own routes, which you implement in JavaScript running on the database server, with direct access to the C++ DB engine.

Unprecedented possibilities for data centric services:
- custom-made complex queries or authorizations
- schema-validation
- push feeds, etc.
Replication and Sharding — horizontal scalability

ArangoDB provides

- easy setup of (asynchronous) replication,
- sharding with automatic data distribution,
- asynchronous replication in the cluster,
- full integration with Apache Mesos and Mesosphere,
- fault tolerance by automatic failover,
- a self-repairing cluster architecture and easy upscaling.
Replication and Sharding — horizontal scalability

ArangoDB provides

- easy setup of (asynchronous) replication,
- sharding with automatic data distribution,
- asynchronous replication in the cluster,
- full integration with Apache Mesos and Mesosphere,
- fault tolerance by automatic failover,
- a self-repairing cluster architecture and easy upscaling.

Work in progress (nearly there!):

- synchronous replication in cluster mode,
Replication and Sharding — horizontal scalability

ArangoDB provides

- easy setup of (asynchronous) replication,
- sharding with automatic data distribution,
- asynchronous replication in the cluster,
- full integration with Apache Mesos and Mesosphere,
- fault tolerance by automatic failover,
- a self-repairing cluster architecture and easy upscaling.

Work in progress (nearly there!):

- synchronous replication in cluster mode,
- zero administration,
Replication and Sharding — horizontal scalability

ArangoDB provides

- easy setup of (asynchronous) replication,
- sharding with automatic data distribution,
- asynchronous replication in the cluster,
- full integration with Apache Mesos and Mesosphere,
- fault tolerance by automatic failover,
- a self-repairing cluster architecture and easy upscaling.

Work in progress (nearly there!):

- synchronous replication in cluster mode,
- zero administration,
- convenient downscaling and self-balancing.
Distributed applications run well together on DCOSes like Mesosphere, Docker Swarm.
Data Center Operating System Integration

- Distributed applications **run well together** on DCOSes like Mesosphere, Docker Swarm
- **DCOS**: helps to build distributed apps (automatic failover, scaling)
- Distributed applications **run well together** on DCOSes like Mesosphere, Docker Swarm
- **DCOS**: helps to build distributed apps (automatic failover, scaling)
- **ArangoDB**’s design lends itself well for Apache Mesos integration.
Data Center Operating System Integration

- Distributed applications run well together on DCOSes like Mesosphere, Docker Swarm

- DCOS: helps to build distributed apps (automatic failover, scaling)

- ArangoDB’s design lends itself well for Apache Mesos integration.

- It is a win-win-cooperation.
Links

https://www.arangodb.com

https://www.arangodb.com/foxx/

http://mesos.apache.org/

https://mesosphere.com/