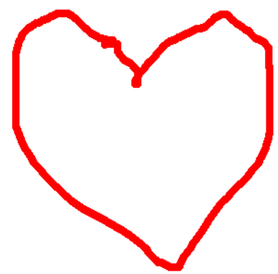
Query mechanisms for NoSQL databases

FrOSCon, 2013-08-24

Jan Steemann

Me

- I'm a software developer,
  working at triAGENS GmbH, CGN
- I work a lot on ArangoDB,
  a NoSQL document database
- I like databases in general



How to save this programming language user object in a database?

```
{
  "id" : 1234,
  "name" : {
    "first" : "foo",
    "last" : "bar"
  },
  "topics": [
    "skating",
    "music"
  ]
}
```

Relational Databases

Relational databases – tables

- data are stored in **tables** with typed **columns**
- all records in a table are **homogenously structured** and have the same columns and data types
- tables are **flat** (no hierchical data in a table)
- columns have primitive data types: **multi-valued data** are **not supported**

Relational databases – schemas

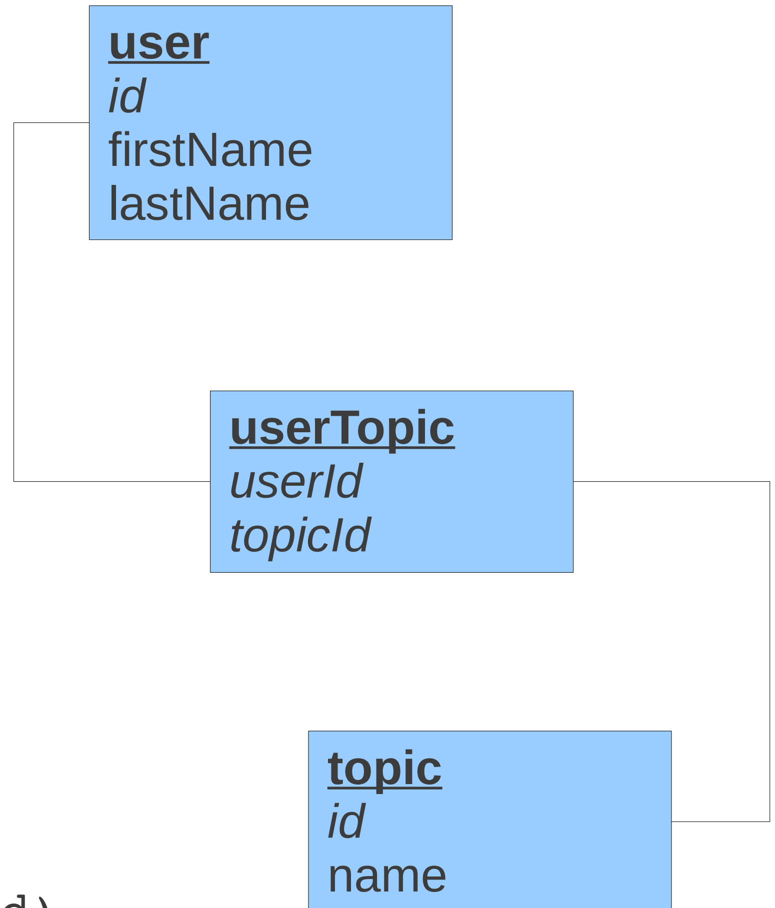
- relational databases have a **schema** that defines which tables, columns etc. there are
- users are **required to define the schema** elements before data can be stored
- inserted **data must match the schema** or the database will reject it

Saving the user object in a relational database

- we cannot store the object as it is in a relational table, we must first **normalise**
- for the example, we end up with **3 database tables** (user, topic, and an n:m mapping table between them)
- note that the object in the programming language now has **a different schema** than we have in the database

Schema we may have come to

```
CREATE TABLE `user` (
  id INTEGER NOT NULL,
  firstName VARCHAR(40) NOT NULL,
  lastName VARCHAR(40) NOT NULL,
  PRIMARY KEY(id)
);
CREATE TABLE `topic` (
  id INTEGER NOT NULL auto_increment,
  name VARCHAR(40) NOT NULL,
  PRIMARY KEY(id),
  UNIQUE KEY(name)
);
CREATE TABLE `userTopic` (
  userId INTEGER NOT NULL,
  topicId INTEGER NOT NULL,
  PRIMARY KEY(userId, topicId),
  FOREIGN KEY(userId) REFERENCES user(id),
  FOREIGN KEY(topicId) REFERENCES topic(id)
);
```



Now we can save the user object

```
BEGIN;

-- insert the user
INSERT INTO `user` (id, firstName, lastName)
VALUES (1234, "foo", "bar");

-- insert topics (must ignore duplicate keys)
INSERT INTO `topic` (name) VALUES ("skating");
INSERT INTO `topic` (name) VALUES ("music");

-- insert user-to-topics mapping
INSERT INTO `userTopic` (userId, topicId)
SELECT 1234, id FROM `topic`
WHERE name IN ("skating", "music");

COMMIT;
```

Joins, ACID, and transactions

- to get our data back, we need to read from **multiple tables**, either with or without **joins**
- to make multi-table (or other multi-record) operations behave predictably in concurrency situations, relational databases provide **transactions** and control over the **ACID** properties (**a**tomicity, **c**onsistency, **i**solation, **d**urability)

The ubiquity of SQL

- note that all we did (schema setup, data manipulation/selection, transactions & concucrrency control) can be **accomplished with SQL queries**
- note: some of the SQL work may be **hidden by object-relational mappers** (ORMs)
- **SQL is the standard means** to query and administer relational databases

NoSQL Databases

Relational databases criticisms (I)

- lots of **new databases have emerged** in the past few years, often because...
  - ...**object-relational mapping can be complex** or costly
  - ...relational databases do not play well with **dynamically structured data** and **often-varying schemas**

Relational databases criticisms (II)

- lots of **new databases have emerged** in the past few years, often because...
  - ...**overhead** of SQL parsing and full-blown query engines **may be significant for simple access patterns** (primary key access, BLOB storage etc.)
  - ...**scaling to many servers** with the **ACID guarantees** provided by relational databases **is hard**

NoSQL and NewSQL databases

- many of the recent databases are labelled
  - **NoSQL (the non-relational ones)** or
  - **NewSQL (the relational ones)**
- because they **provide alternative solutions** for some of the mentioned problems
- especially the NoSQL ones often **sacrifice features** that relational databases have in their DNA

Example NoSQL databases

Apache CouchDB
mongoDB
hadoop
redis
ArangoDB
Couchbase
amazon DynamoDB
Cassandra
riak

NoSQL database characteristics

- NoSQL databases have multiple (but not necessarily all) of these characteristics:
  - **non-relational**
  - **schema-free**
  - **open source**
  - **simple APIs**
- several, but not all of them, are **distributed** and **eventually consistent**

Non-relational

- NoSQL databases are generally **non-relational**, meaning they do not follow the **relational model**
- they **do not provide tables with flat fixed-column records**
- instead, it is common to work with **self-contained aggregates** (which may include **hierarchical** data) or even **BLOBs**

Non-relational

- this **eliminates the need for complex object-relational mapping** and many data **normalisation requirements**
- working on aggregates and BLOBs also **led to sacrificing complex and costly features**, such as query languages, query planners, referential integrity, joins, ACID guarantees for cross-record operations etc. in many of these databases

Schema-free

- most NoSQL databases are **schema-free** (or at least are very relaxed about schemas)
- there is often **no need to define any sort of schema** for the data
- being schema-free allows different records in the same domain (e.g. "user") to have **heterogenous structures**
- this allows a **gentle migration of data**

Simple APIs

- NoSQL databases often **provide simple interfaces** to store and query data
- in many cases, the APIs offer access to **low-level data manipulation and selection** methods
- queries capabilities are often limited so **queries can be expressed in a simple way**
- SQL is not widely used

Simple APIs

- many NoSQL databases have simple text-based protocols or **HTTP REST APIs with JSON inside**
- databases with HTTP APIs are **web-enabled** and can be run as **internet-facing services**
- several vendors provide **database-as-a-service** offers

Distributed

- several NoSQL databases (not all!) can be run in a **distributed** fashion, **providing auto-scalability** and **failover capabilities**
- in a distributed setup, ACID features are often sacrificed for scalability and throughput
- **replication** between distributed nodes is often **lazy**, meaning the database is **eventually consistent**

NoSQL databases variety

- there are **100+ NoSQL databases** around
- they are often categorised based on the **data model** they support, for example:
  - document stores
  - key-value stores
  - wide column/column family stores
  - graph databases
- NoSQL databases are typically **very different from each other**

Document stores

Documents – principle

- documents are **self-contained**, **aggregate data structures**
- they consist of attributes (name-value pairs)
- attribute values have **data types, which can also be nested/hierarchical**

Example document (JSON)

```
{
  "id" : 1234,
  "name" : {
    "first" : "foo",
    "last" : "bar"
  },
  "topics": [
    "skating",
    "music"
  ]
}
```

Objects vs. documents

- **programming language objects** can often be **stored easily** in documents
- lists/arrays, and sub-objects from programming language objects **do not need to be normalised** and **re-assembled** later
- **one programming language object** is often **one document in the database**

Document stores

- document stores have a **type system**, so they can **perform** some **basic validation** on data
- as **each document carries an implicit schema**, document stores can **access all document attributes** and **sub-attributes individually**, offering lots of query power
- today will look at document stores CouchDB, MongoDB, ArangoDB

Document stores – CouchDB

- CouchDB is a **document store** with a **JSON type system**
- similar documents are organised in **databases**
- the server functionality is exposed via an **HTTP REST API**
- to communicate with the CouchDB server, use `curl` or the browser

Saving the user object in CouchDB

- to **create a database** "user" for storing documents, send an HTTP PUT request to the server:

```
> curl -X PUT
  http://couchdb:5984/user
```

- to **save the user object** as a document, send its JSON representation to the server:

```
> curl -X POST
  -d '{"_id":"1234", ...}'
  http://couchdb:5984/user
```

Querying the user object in CouchDB

- to retrieve the object using **its unique document id**, send an HTTP GET request:

```
> curl -X GET
  http://couchdb:5984/user/1234
```

Views in CouchDB

- querying documents by anything else than their id attributes requires **creating a view**
- views are populated with user-defined **JavaScript map-reduce** functions
- views are normally **populated lazily** (when the view is queried) and **incrementally**
- view results are persisted so views are **persistent secondary indexes**

Generic map-reduce algorithm

- **map-reduce** is a **general framework, present in many databases**
- map-reduce requires at least a **map function**
- **map** is **applied on each (changed) document** to **filter** out irrelevant documents, and to **emit data** for all documents of interest
- the emitted data is sorted and passed in groups to **reduce** for **aggregation,** or, if no reduce, is the final result

Filtering with map

```
map = function (doc) {
  for (i = 0;
       i < doc.topics.length; i++) {
    if (doc.topics[i] === 'music') {
      emit(null, doc);
      return; // done
    }
  }
};

[ null, { "_id" : 1234, .... } ]
...
```

Counting with map

```
map = function (doc) {
  for (i = 0; i < doc.topics.length; ++i) {
    // emit [ name, 1 ] for each topic
    emit(doc.topics[i], 1);
  }
};
```

```
[ "skating", 1 ]
[ "skating", 1 ]
[ "music", 1 ]
...
```

Aggregating with reduce

```
reduce = function (keys, values, rereduce) {
  if (rereduce) {
    // reducing a reduce result
    return sum(values);
  }
  // return number of values in group
  return values.length;
};

[ "skating", 2 ]
[ "music", 1 ]
...
```

Map-reduce

- map-reduce functionality is **available in many NoSQL databases**
- it got popular because **map** can be **run fully distributed**, thus allowing the analysis of big datasets
- it is actual programming, not writing queries!

Document stores – MongoDB

- MongoDB is a **document store** with a **BSON** (a binary superset of JSON) **type system**
- similar documents are organised in **databases** with **collections**
- to connect to a MongoDB server, use the `mongo` client (no HTTP)

Saving the user object in MongoDB

- to store the user object, use **save**:

```
mongo> db.user.save({
  "_id" : 1234,
  "name" : {
    "first" : "foo",
    "last" : "bar"
  },
  "topics" : [ "skating", "music" ]
});
```

Querying the user object in MongoDB

- use **find** to **filter on any attribute** or sub-attribute(s):

```
mongo> db.user.find({
  "_id" : 1234
});

mongo> db.user.find({
  "name.first" : "foo"
});
```

Querying using $query $operators

```
mongo> db.user.find({
  "$or" : [
    { "name.first" : "foo"},
    {
      "topics" : {
        "$in" : [ "skating" ]
      }
    }
  ]
});
```

Querying in MongoDB: more options

- find queries can be **combined** with count(), limit(), skip(), sort() etc. functions
- **secondary indexes** can be created on attributes or sub-attributes to speed up searches
- several **aggregation functions** are also provided
- **no joins** or cross-collection queries are possible

Querying in MongoDB: more options

- find queries can be **combined** with count(), limit(), skip(), sort() etc. functions
- **secondary indexes** can be created on attributes or sub-attributes to speed up searches
- several **aggregation functions** are also provided
- **no joins** or cross-collection queries are possible

Document stores – ArangoDB

- ArangoDB is a **document store** that uses a **JSON** type system
- similar documents are organised in **collections**
- server functionality is exposed via **HTTP REST API**
- to connect, use `curl`, the `arangosh` client or the browser

Saving the user object in ArangoDB

```
arangosh> db._create("user");
arangosh> db.user.save({
  "_key" : "1234",
  "name" : {
    "first" : "foo",
    "last" : "bar"
  },
  "topics": [
    "skating",
    "music"
  ]
});
```

Querying the user object in ArangoDB

- to get the object back, query it by its **unique key**:

```
arangosh> db.user.document("1234");
```

- to retrieve document(s) provide **some example values**:

```
arangosh> db.user.byExample({
  "name.first": "foo"
});
```

ArangoDB Query Language (AQL)

- in addition to the low-level access methods, ArangoDB also provides a high-level query language, **AQL**
- the language **integrates JSON naturally**
- AQL allows running **complex queries**, including **aggregation** and **joins**
- indexes on the filter conditions and join attributes will be used if present

Querying with AQL

to query all users with at least 3 topics (including topic "skating") with topic counts:

```
FOR u IN user
  FILTER "skating" IN u.topics &&
         LENGTH(u.topics) >= 3
  RETURN {
    "name" : u.name,
    "topics" : u.topics,
    "count" : LENGTH(u.topics)
  }
```

Aggregation using AQL

to count the frequencies of all topics:

```
FOR u IN user
  FOR t IN u.topics
    COLLECT topicName = t INTO g
    RETURN {
      "name" : topicName,
      "count" : LENGTH(g)
    }
```

Key-value stores

Key-value stores – principle

- in a key-value store, a **value** is mapped to a **unique key**
- to **store** data, supply both key and value:

```
> store.set("user-1234", "...");
```

- to **retrieve a value**, supply its key:

```
> value = store.get("user-1234");
```

- keys are organised in **databases**, **buckets**, **keyspaces** etc.

Key-value stores – values

- key-value stores treat value data as **indivisible BLOBs** by default (some operations will treat values as numeric)
- for the store, the values **do not have a known structure** and will **not be validated**
- as no structure is known, values can only be queried via their keys, not by values or sub-parts of values

Key-value stores – basic operations

- key-value stores **are very efficient** for basic operations on keys, such as **set**, **get**, **del**, **replace**, **incr**, **decr**
- many stores also provide **automatic ttl-based expiration of values** (useful for caches)
- some provide **key enumeration** to retrieve the full or a restricted list of keys

Saving the user object in Redis

- Redis is a (single server) key-value store
- to connect, use `redis-cli` (or `telnet`)
- to **store** the user object in Redis:

```
redis> set user-1234
       <serialized object
        representation>
```

Querying the user object from Redis

- to **retrieve** the user object, supply the key:
  ```
  redis> get user-1234
  <serialized object representation>
  ```
- to query the list of users, we can use **key enumeration** using a prefix:
  ```
  redis> keys user-*
  1) "user-1234"
  ```
- that's about what we can do with BLOB values

Additional querying in Redis

- Redis provides **extra commands to work on data structures** (sets, lists, hashes)
- these commands allow to **Redis to be used for some extra use cases**

Mapping users to topics in Redis

- we can use Redis **sets** to map users to topics
- each topic gets its own set
- and user ids are added to all sets they have topics for:

```
redis> sadd topic-skating 1234
redis> sadd topic-music 1234
redis> sadd topic-skating 2345
redis> sadd topic-running 3456
```

Querying users for topics in Redis

- which users have topic "skating" assigned?

```
redis> smembers topic-skating
1) "1234"
2) "2345"
```

- which users have both topics "skating" and "music" assigned (**intersection**)?

```
redis> sinter topic-skating
              topic-music
1) "1234"
```

Querying distinct values in Redis

- using the **sets** and **key enumeration**, we can also answer the question "what distinct topics are there?":

```
redis> keys topic-*
1) "topic-skating"
2) "topic-music"
3) "topic-running"
```

Data structure commands in Redis

- there is no general-purpose query language so querying is rather limited
- in general, **data must be made to fit the commands**
- the special commands are very useful to implement **counters**, **queues**, and **publish/subscribe**

Other key-value stores

- other key-value stores use the memcache protocol or provide an HTTP API
- some allow users to **maintain secondary indexes**
- these indexes can be used for **equality and range queries** on the index data
- some key-value stores also provide map-reduce for arbitrary queries

Summary

Summary – non-relational

- NoSQL databases are very **different from relational databases** and do **not follow the relational model**
- instead of working on fixed column tables, they work on **aggregates or BLOBs**
- they often **intentionally lack features** that relational databases have
- SQL is not widely used to query and administer

Summary – categories

- there are **different categories of NoSQL databases**, with **different use cases and limitations** each
- **key-value stores** normally focus on high throughput and/or scalability, and often allow limited querying only
- **document stores** try to be more general purpose and often allow more complex queries

Summary – usage

- the **APIs** of NoSQL databases are often **simple**, so it is **easy to get started with them**
- providing database access via **HTTP REST APIs** is quite common in the NoSQL world
- this allows **querying the database directly from any HTTP-enabled clients** (browsers, mobile devices etc.)

Summary – variety

- NoSQL databases are **very different** from each other
- there are yet **no standards** such as SQL is in the relational world
- there is an interesting attempt to establish a cross-database query language (JSONiq)