Query Languages for Document Stores

NoSQL matters conference
2013-04-26

Jan Steemann

me

- I'm a software developer
- working at triAGENS GmbH
- on and with  ArangoDB

Documents

Documents

- documents are self-contained, aggregate data structures
- consisting of named and typed attributes, which can be nested / hierarchical
- can model complex business objects with documents

Example document (order)

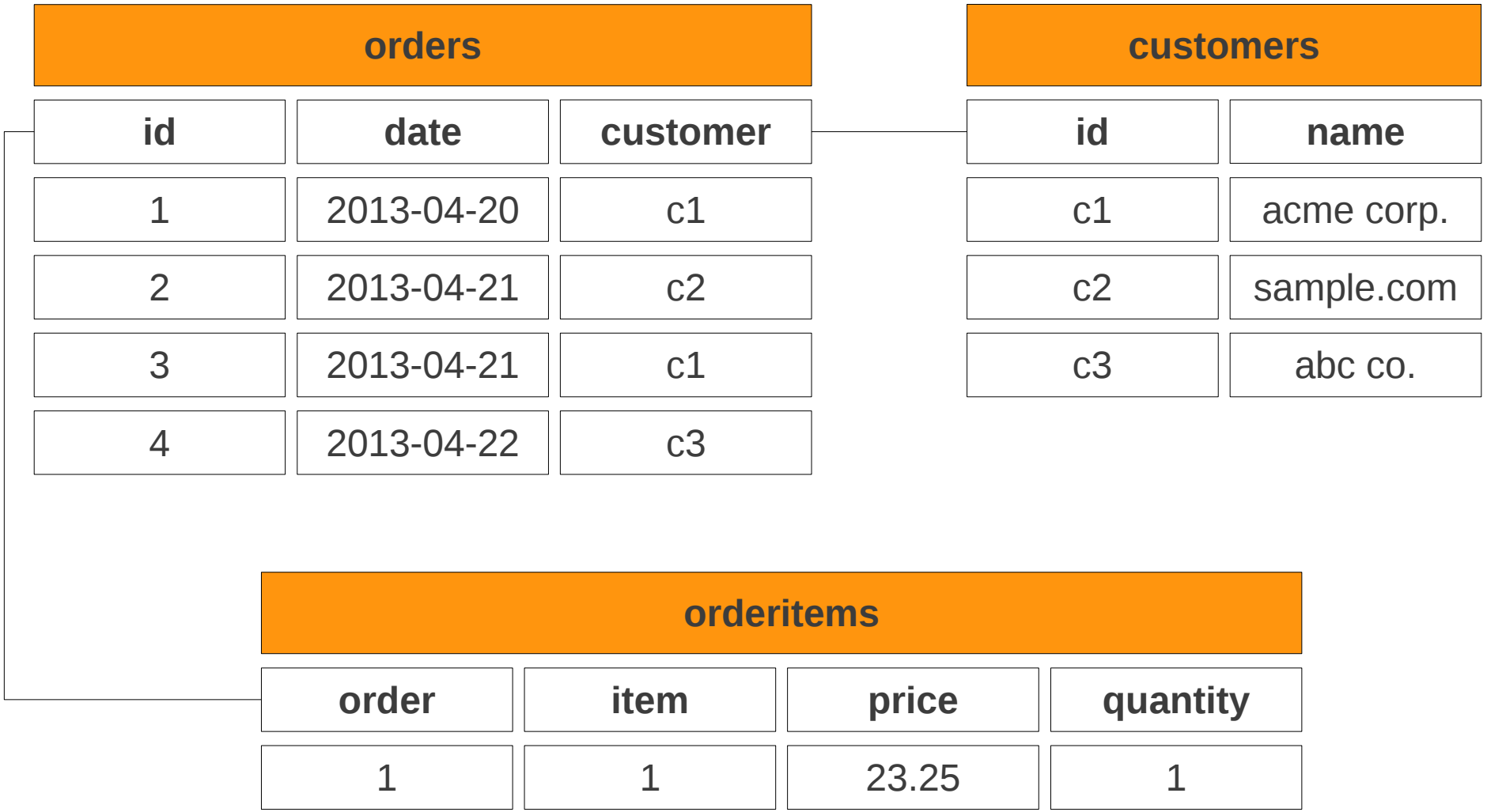
```
{
  "id": "abc-100-22",
  "date": "2013-04-26T09:15:32"
  "customer": {
    "id": "c-199-023",
    "name": "acme corp."
  },
  "items": [
    {
      "id": "p-123",
      "quantity": 1,
      "price": 25.13
    },
    ...
  ]
}
```

Document stores

- document stores are databases specialised for handling documents
- they've been around for a while
- got really popular with the NoSQL hype
- the ones with most adoption at the moment: MongoDB, CouchDB, RavenDB
- some key / value stores and graph databases can process documents, too

Why Document Stores?

Persistence the relational way



orders

id	date	customer
1	2013-04-20	c1
2	2013-04-21	c2
3	2013-04-21	c1
4	2013-04-22	c3

customers

id	name
c1	acme corp.
c2	sample.com
c3	abc co.

orderitems

order	item	price	quantity
1	1	23.25	1

Why document stores?

- document stores allow saving a programming language object as a whole: your object becomes the document
- no impedance mismatch,
  no complex object-relational mapping
- querying documents often easier and faster than querying highly normalised relational data

Schema-less

- there is no pre-defined structure or schema for documents
- each document can have different attributes
- the schema is an implicit part of each document
- this is ideal for storing variable objects,
- less ideal in terms of storage space and when exploring data

Heterogenity of document stores

- different datatype systems are in use: XML, BSON, JSON etc.
- mechanisms to query documents are wildly different

Query languages

What for?

- you don't run a database just for the sake of technology
- you eventually want to do something with its data:
  - transformations
  - aggregations
  - joining

Why?

- a query language allows writing both simple and complex queries, without having to switch the methodology
- well suited for ad-hoc data exploration and querying
- don't need a query language for simple-only access patterns (e.g. query by primary key)

SQL

- in the relational world, there is one accepted general-purpose query language: SQL
- it is quite well-known and mature:
  - 30+ years of experience
  - many developers "speak" it
  - established tools and ecosystem around it
  - standardised (but mind the "dialects" !)

SQL in document stores?

- SQL is good at handling relational data
- not good at handling multi-valued or hierchical attributes, which are common in documents
- provides features many document stores intentionally lack, e.g. joins, transactions
- has not been adopted by document stores

UnQL

- UnQL (Unstructed Query Language) is an attempt to marry SQL, JSON and document processing
- language uses the SQL keywords (SELECT, INSERT, UPDATE, DELETE etc.) to make it easy to use
- initiated by Richard Hipp (of SQLite fame) and Damien Katz (of CouchDB)

UnQL example

```
INSERT INTO orders VALUE {
   id: "123-02-01",
   customer: {
      "id": "c1",
      "name": "acme corp."
   },
   items: [
      ...
   ]
}
```

UnQL status

- project hasn't gone beyond prototype status
- no real activity in the last 2 years
- it seems dead

XQuery

- XQuery is a query and (somewhat) programming language
- targeted mainly at processing XML data
- can process hierarchical data
- very powerful
- extensible
- W3C recommendation

XQuery

- has found most adoption in the area of XML processing
- not available in most popular document stores

What's there?

CouchDB

- querying by something else than document id requires writing a view
- views are populated by incremental map-reduce
- map and reduce functions are JavaScript code

CouchDB map-reduce example

```
map = function (doc) {
  var i, n = doc.tags.length;
  for (i = 0; i < n; ++i) {
    emit(doc.tags[i], 1);
  }
};

reduce = function (keys, values, rereduce) {
  if (rereduce) {
    return sum(values);
  }
  return values.length;
};
```

CouchDB map-reduce

- map-reduce in CouchDB is quite powerful
- provides you with a full programming language
- need to pre-define everything you want to query
- querying across the boundaries of a "database" is not possible
- need different views for different queries

MongoDB

- can apply filters on collections
- this allows finding specific documents:

```
mongo> db.orders.find({
   "customer": {
     "id": "c1",
     "name": "acme corp."
   }
});
```

MongoDB

- can filter on any document attribute or sub-attribute
- can also put together more complex filters, projections, and aggregations
- quite flexible and powerful
- tends to be hard to use for ad-hoc queries

MongoDB complex filtering

```
mongo> db.users.find({
   "$and": [
      {
         "active": true
      },
      {
         "age": {
            "$gte": 40
         }
      }
   ]
});
```

MongoDB

- can also use JavaScript functions for filtering
- can also use map-reduce to query a collection
- querying across the boundaries of a "collection" is not possible

ArangoDB

- ArangoDB provides its own query language, AQL
- it allows executing complex queries on documents, including joins and aggregation
- language syntax was inspired by XQuery
- provides similar concepts such as FOR, LET, RETURN, ...

ArangoDB AQL example

```
FOR user IN users
  FILTER user.active == true

  LET userProjects = (
    FOR project IN projects
      FILTER user.name IN project.members
      RETURN project.name
  )

  RETURN {
    name: user.name,
    whichProjects: userProjects
    countProjects: LENGTH(userProjects)
  }
```

ArangoDB AQL

- AQL queries can combine data from multiple collections
- this allows joining on any document attributes

Graph databases

- you may store documents in graph databases, too
- in graphs, you're not only interested in the actual document but also in how they are interconnected
- the access patterns are quite different than in regular document stores

Graph database traversals

- a common way of querying a graph database is to write custom traversal code in the programming language the database supports
- "Gremlin" has come up as a domain-specific scripting language for graph traversals
- can write the same "Gremlin" traversal query for multiple database back ends
- Neo4j provides Cypher, a query language dedicated to graph traversals

Summary

Summary

- different document stores provide different, often proprietary mechanisms for querying
- some allow ad-hoc querying on any attribute, some don't
- there is no standard query mechanism as there is in the relational world (SQL)
- currently map-reduce is the most cross-platform way for querying

JSONiq

JSONiq

- JSONiq is a data processing and query language targeted at handling JSON data
- it is based on XQuery, but with most of the XML handling removed
- provides FLWOR expressions as in XQuery: FOR, LET, WHERE, ORDER, ...
- JSON is integrated "naturally"

JSONiq example

```
for $char in collection("characters")
where $char.name eq "spongebob"
return {
  friends: $char.friends,
  livesIn: $char.livesIn
}
```

JSONiq

- the language itself is database-agnostic
- also provides mechanisms for joins, aggregation etc.

JSONiq join example

```
for $post in collection("posts")
let $postId := $post.id

for $comment in collection("comments")
where $comment.postId eq $postId

group by $postId
order by count($comment) descending

return {
  id: $postId,
  comments: count($comment)

}
```

JSONiq

- JSONiq looks really powerful and general, at least for querying document databases
- it might not help for querying graph databases
- it's not yet available in any of the major stores, but there are ways to use it anyway
- can use it with the Zorba query processor (client-side) and as an online service for querying MongoDB instances