CouchDB

NDBI040

Jakub Sýkora

Couch DB

Introduction

- CouchDB: A Distributed, NoSQL Document Database
- Scalable, Fault-Tolerant, and Highly Available
- Built on Web Standards: JSON, HTTP, and JavaScript
- Master-Master Replication & Eventual Consistency

CouchDB

Installation

- Requirements: Docker, Python >= 3
- Installation steps and checkpoints explained in README.md
- The following steps:
  - Run docker-compose up -d
  - Run init-cluster.sh (MacOS/Linux) or init-cluster.ps1 (Windows)
  - Run pip install -r requirements.txt
  - Run python init.py

CouchDB
Installation

- Init-cluster script initialises CouchDB in cluster of 3 nodes
- Init.py script inserts data into CouchDB cluster
- For simplicity in all requests we use URL embedded username, password
- To re-execute installation run first ‘docker-compose down —volumes’

CouchDB

Domain for the Project

- Libraries with books, authors, lease offers
- Using real data harvested from SPARQL WikiData query
- datagen.py generates json data files

CouchDB
Data Model

- Data stored as JSON documents
  - Flexible schema, fields can vary between documents
- Documents organised in databases
- Unique identifiers (_id) and revision tracking (_rev)
  - Each document has unique _id
  - _rev tracks changes to the document for conflict resolution

CouchDB
Inserting Data

- We want to insert the following documents into database ‘library’

```
{
    "_id": "author_3",
    "type": "author",
    "authorLabel": "Emil Cioran",
    "dob": "1911-04-08T00:00:00Z",
    "dod": "1995-06-20T00:00:00Z"
}
```

- Best practise is to give each document ‘type’ - since there is no schema in CouchDB database can be used to store multiple types of docs (books, library, offers, authors)

CouchDB
Inserting Data

- Done via HTTP POST requests as any other database operation
- To see detailed example visit data_insert.py script
- In init.py we use `couchdb` Python lib facilitating database operations

CouchDB
Replication and CAP Theorem

- Multi-master replication: each node can accept writes
- Data propagation among nodes is asynchronous
- Network partitioning or concurrent updates can cause inconsistency
- CouchDB aligns with the AP properties of the CAP theorem
  - CouchDB employs eventual consistency
- Not suitable for applications requiring real-time data accuracy
  - e.g., financial transactions

CouchDB

Viewing Data

- Built-in view called _all_docs
- Custom MapReduce views (complex queries, indexed)
- Mango queries (JSON query definition, ad-hoc, possible indexing)

```
"selector": {
   "type": "lease_offer",
   "libraryId": "library_2"
 },
 "fields": ["_id", "_rev", "bookId", "libraryId", "availability", "leaseStart", "leaseEnd"]
```

- Query times (find_query.py)
  - Unindexed Mango query: 0.039917 seconds
  - Indexed Mango query: 0.007773 seconds
  - Map views query: 0.018620 seconds

CouchDB

Introduction to MapReduce Views

- Customizable data querying and aggregation
- JavaScript Map and Reduce functions
  - Map: Process each document and emit key-value pairs
  - Reduce: Aggregate and condense the emitted key-value pairs
- Stored in design documents that exist for each database
- Queried using HTTP, using keys created by MapReduce

CouchDB

MapReduce View Example

- Create a view that will represent secondary index over library's country
- PUT into "<couchdb_url>/<database_name>/_design/views":

```
{
    "_id": "_design/views",
    "views": {
        "libraries": {
            "map": "function(doc) { if (doc.type === 'library') { emit(doc.country, doc); } }"
    }
}
```

- Query libraries by country (HTTP GET):
    - <couchdb_url>/<database_name>/_design/views/_view/libraries?key="Germany"

CouchDB

MapReduce Views Implementation Details

- Views are stored as B-trees
  - On first query of the view, the B-tree index is built
  - B-tree keys are emitted keys from Map function
  - Instead of “key” you can use “startKey” and “endKey” to impl. range q.
- Scalable and optimized
  - Incremental updates to avoid reprocessing all documents
  - Parallel processing across distributed nodes

CouchDB

MapReduce View Limitations

- Limited Query Types (key, startKey, endKey)
- MapReduce views can be resource-intensive
- Learning Curve
- No JOINs or Aggregations
- Index Storage Overhead
- Compaction Requirements

CouchDB

Complex Queries: View Collation and include_docs=true

- SQL Joins can be partly implemented using view collation or include_docs
- View collation JOIN to get books written by author
  - Book doc: iterate authors array, emit ([author_id,], book)
  - Authors doc: emit([author_id], doc)
  - Query: range ?startkey=[“<author_id>"]&endkey=["<author_id>", {}]’
- Data model of domain restricts allow usage of collation, not all JOINs can be achieved

CouchDB

Complex Queries: View Collation and include_docs=true

- Script authors_books_join.py measures three implementations of a query
  - MapReduce view using view collation
  - MapReduce view with include_docs=true
  - MapReduce views to fetch books and authors, join them externally
- Measured on Apple M2 Max
  - View collation (0.008s) is fastest since it uses index directly
  - Include_docs=true (0.01s) needs additional time to fetch the documents
  - External join (0.01s) only fast because of local network and query size

CouchDB

Comparison with SQL constructs

- Aggregates (e.g. COUNT) can be achieved with MapReduce Views
  - authors_count_by_dob.py (shows also GROUP_PY option on [MM,DD,YYYY] keys by using group_level param in query)
- JOINs using MapReduce in simple cases or external JOINs
- WHERE: MapReduce function implementation decides what to emit()
- Sorting: Keys ordering in B+ tree (MapReduce View index)