1. **Polyglot** systems from chapter 8 in book

**Redis**

* it assists in populating data to CouchDB,
* it acts as a cache for recent Neo4j changes, and
* it enables quick lookup for partial value searches.

**CouchDB** as SoR (authorative data source)

* CouchDB’s document structure is an easy way to store band data because it allows for nested artist and role information
* we will take advantage of the changes API in CouchDB to keep our third data source in sync (Returns a sorted list of changes made to documents in the database, in time order of application, can be obtained from the database’s **\_changes** resource)
* We’ll store relationships between bands, band members, and the roles the members play.

**Neo4j**

* + - * Neo4j is our relationship store. Although querying the CouchDB SOR directly is perfectly reasonable, a graph database affords us a simplicity and speed in walking node relationships
      * We’ll store relationships between bands, band members, and the roles the members play.

Each database has a specific role to play in our system, but they don’t natively communicate with another, which means that we need to build a translation layer between them. We’ll use Node.js to populate the databases, communicate between them, and act as a simple front-end server. Since gluing multiple databases together requires a bit of code!

Population

The first item of business is to populate our databases with the necessary data. We take a two-phased approach here, by first populating Redis and then populating our CouchDB SOR.

We’ll be using a large data set holding information about bands, including the names of those bands, band members, which role each band member played, and more.

This data set contains a lot of information, but we’re interested only in extracting the member or artist name, the group or band name, and their roles in that band stored as a comma separated list. For example, *John Cooper* played in the band *Skillet* as the *Lead vocalist*, *Acoustic guitar* player, and *Bassist*.

Ultimately we want to structure John Cooper and the other members of Skillet into a single CouchDB document like the following, stored at the URL http://localhost:5984/bands/Skillet:

{

"\_id": "Skillet",

"name": "Skillet"

"artists": [

{

"name": "John Cooper",

"role": [

"Acoustic guitar",

"Lead vocalist",

"Bass"

]

},

….

{

"name": "Korey Cooper",

"role": [

"backing vocals",

"Synthesizer",

"Guitar",

"Keyboard instrument"

]

}]}

This file contains well over 100,000 band members and more than 30,000 bands.

You may wonder why we bother populating Redis and not just dive right into populating CouchDB. Acting as an intermediary, Redis adds structure to the flat TSV data so that subsequent insertion into another database is fast.

SOR Insertion

CouchDB will play the role of our system of record (SOR). If any data conflicts arise between Redis, CouchDB, or Neo4j, CouchDB wins. A good SOR should contain all of the data necessary to rebuild any other data source in its domain.

Since phase 1 was all about pulling data from a TSV and populating Redis, this phase is all about pulling data from Redis and populating CouchDB

Relationship Store

Next on the docket is our Neo4j service that we’ll use to track relationships between artists and the roles they play. We could certainly query CouchDB outright by creating views, but we are rather limited on complex queries based on relationships.

With our initial data in place, now we need to keep Neo4j in sync with CouchDB should any data ever change on our system of record. So, we’ll kill two birds by crafting a service that populates Neo4j on any changes to CouchDB since the database was created.

We also want to populate Redis with keys for our bands, artists, and role so we can quickly access this data later. Happily, this includes all data that we’ve already populated in CouchDB, thus saving us a separate initial Neo4j and Redis population step.

Whenever a change is detected, we do two things: populate Redis and populate Neo4j.

1. From Neo4j developers news

Conference recommendation engine

<https://technology.amis.nl/2018/11/20/building-a-conference-session-recommendation-engine-using-neo4j-graph-database/?mkt_tok=eyJpIjoiTVdJME5EWTRORGcwTW1ZeiIsInQiOiJcLzV0aEJXTEl3aWxLeTBNM2FsSzE4YkEraXk4azM1WTVJREkrNjVkY2d0MnlYeGQ4Q0Z6RWIwM0dGXC92b21nVXViMExWdkZyTVFPRHFra0tEQ3hDc0ZqKzdLYXQrUXJ2bUFteXI1N0FieE82SXJkY1FHYUdCZDhIenRNWXZORWhRIn0%3D>

APOC

<https://www.youtube.com/watch?v=V1DTBjetIfk>

Graph Algorithms (in parallel) algo library

<https://neo4j.com/docs/graph-algorithms/current/introduction/>

Similarity Algorithms

<https://www.youtube.com/watch?v=-rM8EHLmsjg&mkt_tok=eyJpIjoiTVdJME5EWTRORGcwTW1ZeiIsInQiOiJcLzV0aEJXTEl3aWxLeTBNM2FsSzE4YkEraXk4azM1WTVJREkrNjVkY2d0MnlYeGQ4Q0Z6RWIwM0dGXC92b21nVXViMExWdkZyTVFPRHFra0tEQ3hDc0ZqKzdLYXQrUXJ2bUFteXI1N0FieE82SXJkY1FHYUdCZDhIenRNWXZORWhRIn0%3D&ab_channel=Neo4j>