Sidney La Fontaine and Hari Muralikrishnan
Large Scale Storage and Retrieval
Professor Rachlin

Product Catalog Analysis

Question 1:

Relational Database: Data Warehouse

```
Select *
From Product
Where diameter == 44 and brand == 'Tommy Hilfiger' and dial_color == 'beige';
```

This query is very simple due to our basic data model, where we have just one table with a column attribute existing for every possible property a product could have. However, this strategy has many disadvantages. The primary one is that the table is sparsely populated since most products only have a few key properties, such as this watch we are querying for. As the number of products in the table increases, the number of columns will exponentially increase as well. This can lead to storage issues and open up possibilities of redundancy. To summarize, the data model is basic and the query is simple, but this comes with the tradeoff of an inefficient method to store large amounts of data. Think about how a product catalog like Amazon, and how many millions of columns they would need with this setup!

Relational Database: Category Tables

```
Select *
From watches left join product on watches.product_id = product.product_id
Where diameter == 44 and brand == 'Tommy Hilfiger' and dial_color == 'beige';
```

This query isn't as direct as the previous one, because while we still directly select for specific properties, we have to join with a product table to find more details about the price and availability since all the data isn't stored in one table. Our data model is not as simple as before. Instead of having one big table with all the information for every product in one tuple, it is spread out over a few tables. This data model will reduce the amount of redundancy, which in turn lowers the chance of any inconsistent data. This setup is advantageous because tables are not sparsely populated. The big disadvantage of this style of data storage is that it often requires using joins when querying the table, and joins can be incredibly inefficient as the number of tuples involved increases like in a product catalog for a larger store. Furthermore, another weakness of this model is that the number of category tables will be extremely large as more products are added to the catalog. Since each table has a strict schema in place, slightly different categories (one or two different properties) would need completely separate tables. And it would be problematic when there exist products that don’t fall exactly into one obvious category.

Relational Database: Property Tables

```
Create View all_watches as
Select *
From Property
Where category_id == 1; (assuming the category_id for watches is 1)

//the query below returns all the unique id’s of products that match the constraints, we decided
//not to select all the attributes because that would lead to a result with redundant attributes
//(ex: product_id listed 3 times)
Select DISTINCT(a1.product_id)
From all_watches a1, all_watches a2, all_watches a3,
Where a1.product_id = a2.product_id and a2.product_id = a3.product_id and
        (a1.key == ‘Diameter’ and a1.value == 44) and (a2.key == ‘Brand’ and a2.value ==
        ‘Tommy Hilfiger’) and (a3.key == ‘dial-color’ and a3.value == ‘Beige’)
```

This query is much more complicated than either of the other ones, that is because of the inherent flexibility in the property table format. To complete the query you have to do many joins, which is incredibly inefficient, although we eliminated most of the products within the first view by selecting only the watches. In fact, for every single property you are querying for, it adds an extra join. This makes a very specific product query highly complex, which is a big disadvantage. This data model is moderately flexible because unlike the category tables you do not have a separate table for each category; all the products go in the same table. However, this data model will be very redundant because most products will have many key-values that will each have to be in different tuples, which produces a table where one product will have its information repeated many times, which opens the data up to inconsistencies. This tradeoff comes with the advantage of having one table to represent ALL properties and is NOT sparsely populated.

Key-Value Store: Redis

Our data model inside Redis would contain a set of key-values, Category, where the key is category_id, and the value is a list of all the product_id’s representing products part of that category. Then we would have another set of key-values, Products, where the key is product_id, and the value is all the information for that product stored as a hashmap where the keys and values are the properties and their values. The API query command logic would look like:

Get Category:1 (assuming the category_id for watches is 1)

For each product_id from the first Get, do: *Get Product:_*

Then we will have the hashmap for each watch, at which point we will filter out all the hashmaps in the API where the diameter is not 44, the brand is not Tommy Hilfiger, and the dial color is not beige.

This is a simpler data model, as we only have two collections of data stored in Redis. The bulk of the information for each product is stored as a hashmap, which is efficient for storage because there are keys/values for only those pertaining to the product (aka no NaN's). The problem is that since Redis is a key-value store, it doesn't have the ability to query over the values of items. So this data model allows for a clean storage but doesn't allow for specific querying. We tried to battle this issue by adding in the Category collection so that we could at least filter out products based on the category they belong to, such as Watches, even if we couldn't query for specific properties like exact diameter, brand, and dial values inside Redis itself (the API would have to take care of this once Redis returns the values).

This query isn't too complex in that it is a generic structure that won't change as products grow -- it has the flexibility to scale efficiently. Redis is amazing for speed and simplicity as long as the users correctly store the data in a consistent manner. Assuming this, our Redis query only has to find the products for a specific category, get its hashmaps, and then the API needs to query those for the specific values before returning results on price/availability.

Document Store: MongoDB

Collection = products

key-values:

Category
Price
Is_available
...
(more key-values depending on category, for example the category watch has diameter, brand, and dail_color)

```
db.products.find({category: ‘watches’, diameter: 44, brand: ‘Tommy Hilfiger’, dial_color: ‘beige’})
```

MongoDB is the BEST solution to this product catalog scenario because it combines the advantages of the various relational model scenarios with the Redis scenario. First, the data model is very straightforward. We have one collection called Products, where each product is JSON containing keys for attribute types and values for attribute values, for example each product has the keys category, price, and is_available. Since MongoDB is a document store, it is best suited for storing data this way. The big advantage is that Mongo also allows us to not only store and retrieve these keys/values quickly, but also query over the values! This makes our query a lot simpler as we can directly put our hand into the data and pick out those products matching the exact constraints we are looking for. To summarize, MongoDB is the most natural fit for this scenario. making our data model simple AND our query extremely simple. We also store data in an efficient manner without any NaN's, since documents can have different key/values (no strict schema). This shows we combined the various strengths of the 4 previous setups into 1 here.

Author Contributions:

We both believe in working collaboratively, so we did all of our work together and in person. We used the pair programming technique to work on our project together, including the analysis.