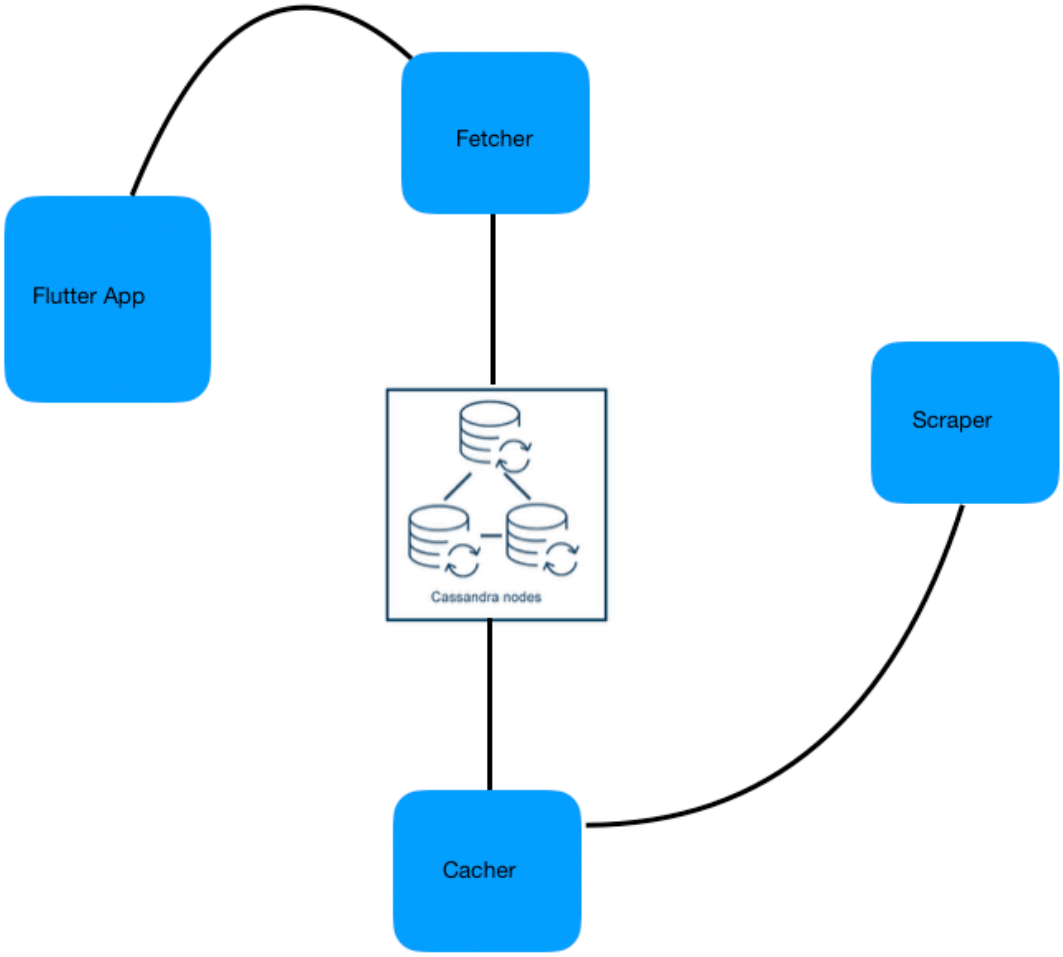
Product Scraping Application Design Overview



Design philosophy:

Web scraping at its core is an application that has to go through frequent design changes because of how fast the webpages change. A product page might not be valid anymore or all its tag names and classes names might have changed making the whole scraping fail. One has to also realize that scraping is a relatively slower process and might be cause of a potential bottleneck in the whole pipeline. In short, the following had to be taken into account for designing the whole system

1. Should be scalable - can have millions of pages and records stored

2. Fetching should be fast - since there is frontend app that constantly polls and renders
3. Data should be highly available
4. Should support frequent code changes and database table changes (since meta info is indefinite)
5. Should easily support additional features such as search, reverse image search, filtering, ranking and so on…

Based on the above points, it is best to use a micro service model where you can split the load between individual services and manage them separately.

Dive deep into the design:

The application constitutes 5 services

1. Flutter application - The frontend for the whole app that renders scraped information
2. Cassandra cluster - A highly available NoSQL db which is the backbone of the whole system
3. Scraper - An easily scalable python service that simply scrapes a web page and returns information
4. Cacher - Runs periodically to call scraper APIs and caches info in Cassandra and updates metrics after every scrape performed
5. Fetcher - communicates with Cassandra and delivers data to frontend.

Cassandra Cluster:

There could potentially be less to no relationship between various columns you want to store in the tables. What is going to be stored is a simply a bunch of dynamic records that constantly keep changing. The length of the record as well as the number of records are going to be growing and the database therefore has to be scalable. It doesn't make sense to go with a monolithic RDBMS like MySQL or PostgreSQL due to the above reasons. You don't want to have the overhead of changing the schema (and therefore indexes)continuously. NoSQL databases can support this dynamic structure. When you have millions of records, data will be partitioned across multiple nodes for reliability and if you can choose the right index, fetching

would be a piece of cake! The best part - you can always add more indexes on the table. The nodes would learn and reorganize.
That being said, one of the main reasons to with Cassandra amongst the sea of options available is because

1. It is Open Source and therefore highly customizable
2. Have prior experience with the libraries and hence development would be faster

Let's talk Schema:

Table **productpageinfo**

Key	Value	Primary Key	Clustering Key	Index
PID	Hash of Generic Product Name	Y	N	
PURLID	Hash of Product URL	Y	N	
PNAME	Generic Product Name	N	Y	
PURL	Product URL	N	N	
IsExpired	Has URL expired	N	N	
Total Tries	Total url hits	N	N	
Total Misses	Scraping failures	N	N	
CreatedOn	Timestamp	N	N	
ModifiedOn	Timestamp	N	N	
ModifiedBy	**User**	N	N	

Table **productinfo**

Key	Value	Primary Key	Clustering Key	Index
PID	Hash of Product Name	N	Y	
PURLID	Hash of Product URL	Y	N	
MetaInfoID	ID for MetaInfo records	N	N	Y
ImageGroup	List of Image record IDS	N	N	

Key	Value	Primary Key	Clustering Key	Index
CreatedOn	Timestamp	N	N	
ModifiedOn	Timestamp	N	N	

Table **productimageinfo**

Key	Value	Primary Key	Clustering Key	Index
PURLID	Hash of Product URL	N	Y	
IURLID	Hash if Image URL	Y	N	
IURL	ImageURL	N	N	
CreatedOn	Timestamp	N	N	
ModifiedOn	Timestamp	N	N	

Table **productmetainfo**

Key	Value	Primary Key	Clustering Key	Index
PID	Hash of Product Name	N	Y	
PURLID	Hash of Product URL	Y	N	
MetaInfoID	ID for MetaInfo records	N	N	
Title	Product Title	N	N	Y
Description	Product Description	N	N	
Retailer	Product retailer	N	N	Y
Seller	Product Seller	N	N	Y
Price		N	N	Y
CreatedOn	Timestamp			
ModifiedOn	Timestamp	N	N	

At first glance, it might look similar to a RDBMS schema with multiple tables and foreign key references. It is designed that way to avoid storing redundant data and to distribute different objects evenly according to different indexes. For instance, you could have multiple Images in a single product page but the meta info is going to be the same. A separate table is therefore

dedicated to Image records where they are distributed across nodes by IURLID. The IURLID is obtained through XXH64 hash on Image URL. Given a URL, you can always retrieve the record by finding the IURLID.

The metainfo table is indexed on PURLID since PURLID is unique for every product page. The table also supports secondary indexing on title, retailer, seller and price to support filtering and searching. There is then the productinfo table which acts as a summary and gateway for image and meta info table. It points to all the image records and meta info record belonging to a page. It is also indexed on PURLID and PID, so that you can retrieve all critical info by giving Product URL and Product Name.

The entire schema is tied together with productpage table which is the input table. It has all records that are added by user along with additional metrics info such as hits, misses and so on. The application currently selects all records from this table without using any keys which is not advisable in a NoSQL environment. In order to fasten the fetch process, you can have the data reside in lesser number of nodes compared to other tables. When the application eventually uses a ranking or filtering also, you can index your searches and then scale the data to be distributed across nodes.

The schema supports/can support following features:

1. Image and MetaInfo retrieval given PNAME, PURL
    - Hash PNAME to find PID
    - Hash PURL to find PURLID
    - Retrieve list of IURLIDS from productinfo using PURLID and PID
    - Retrieve image based on IURLID
    - Retrieve metainfo based on PURLID
2. Search on Retailer
    - Use index to search for matching records on metainfo table
    - Obtain PID, PURLID
    - Use PID, PURLID to get productinfo record
    - Obtain list of IURLIDS
    - Obtain image records through IURLID
3. Reverse Image search given a URL
    - Hash URL to find IURLID
    - Use IURL to retrieve Image records and PURLID
    - Use PURLID to retrieve metainfo from metainfo table

On a traditional RDBMS stack, all the above queries would be a bunch of JOINS across multiple tables. It is slow and poorly scalable. With NoSQL, you can retrieve the the records easily with indexes in O(1) time. The only operation that takes O(p) time (where p is the number of nodes) is the meta info search. But since number p is not going to be abnormally high, we can brush it off.

Scraper

The scraper deserves its own service simply for two reasons

1. It's codebase has to go through frequent changes and therefore deployments
2. Potential bottleneck, can be made faster by spinning multiple nodes
3. Can have separate AWS lambdas for every scrapper class (Amazon, Walmart etc..)

Python was chosen to write the scraper mainly because of the amazing beautifulsoup library. It is the best in the market. While Python is slow, we are anyways prefetching and caching the data. The user will therefore never realize it.

Cacher

The whole point of a cacher service is to reduce the scraping overhead. You do not want the fetch API call to wait until every single page in the db has been scraped. There is a high chance for the API call to timeout given large number of Product URLs. Cacher alleviates this by periodically running all the URLs in the db, calling the scraper to scrape them and storing the retrieve information into the Cassandra tables.

And because there is no dependency between records, you can easily have multiple cachers spun out with each concurrently fetching data from a particular node, scraping URLs and storing them into the db.

The cacher is written in Golang for two reasons

1. Go has efficient and light weight concurrency threads called go routines (they are parallel objects that run on top of threads). You can literally spin out a go routine for every product page record the cacher reads from the database making the whole process a lot more faster. You also don't need to worry about synchronization primitives as the go routines handle them by default.
2. Go is faster and easier to pick up

Fetcher

The fetcher currently has only one purpose - serving the front end with data. While it could essentially be a thread or go routine in the cacher module itself, the reasons to spin out a separate service for this are

1. To support multiple UI features in the future - like authentication, filtering, searching and so on
2. To have information available even when cacher is down or cacher is undergoing changes. Cacher undergoes changes a lot more frequently than fetcher.

The fetcher is written in Golang for the same reasons as cacher - you can have multiple concurrent operations running easily. For instance, if user wants to only display data belonging to Amazon and Walmart, you can make concurrent queries on the the same database for two different search indexes.

Summary

The application can easily be scaled - increase the number of records? Add search features? Add filtering features? Introduce crawling before scraping? You name it. That is the most important factor that went into consideration while designing the models and services. You can add load balancers before any services to make the application a lot more reliable and faster.