**Requirement**

Design a Save Search Query Feature Service

Thought process and approach:

Break down the requirements to Functional requirements and Non-functional requirements.  
Keep scope on the flexibility on the area(s) where the scope seems to be extendible

**Functionality Requirement:**

Search Query is a **Query** made by the **User** on a specific **Category** of a **Product**.

Query Comprises of various Filtering **Dimensions** (Dependent upon the Category and Type of Product)

User can ***Save*** the Query (more than one saved search query).

User can ***Load*** the **Saved Query.**

User will ***Receive*** the **Email Alerts** (if the user has ***tagged*** the alert)

Email Alerts will ***List*** the Abstract Data of the **Matched Recommendations** at regular intervals/as soon the listings are added

User may ***tag*** the query for Email Alert or not (optional) **Listing** consists of the *Title*, *Description*, *Call Back Action to the Originating Website*  
Within the Email Alert – Display Max 10 Listings

**Non- Functionality Requirement:**

Ability to re-use the feature for multiple category and types of Products.

Scale to millions of requests daily

**High level Design**

**Database Choice**: The Distributed Search Query Store should be used (stores which enables indexing the document, queries using Lucene engine) like Elastic Search, Azure Cognitive Service, etc. In addition, a NOSQL DB will have to be used as well for storing the transactions.  
  
**Reason**: These are distributed and are designed for searches making them perfectly suited for the Full-Text Searches and are scalable to million(s) of requests. Most of the modern cloud service providers are providing PAAS based managed service solution(s) for these stores as a service. These modern solutions offer indexing un-structured data (including PDF(s), documents, texts, etc. E.g., Elastic Search

If we don’t have the option of using a Cloud Provider, then we will have to go for a customized solution based on any NO SQL provider like MongoDB, Azure Cosmos DB (geo-replicated), which mimic(s) the similar feature. Also, we’ll have to shard using some consistent hashing technique to ensure uniform distribution of data.

Typically, the use-case pattern is that we index the document(s) within these stores and then we can query on these document(s) using the keywords.   
However, for this specific use-case we can look at indexing the saved-queries instead of document(s). If we pre-index the queries, we can use these indexed queries for search-match. Elastic search provides a concept of [*Percolate Query*](https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-percolate-query.html)*.* Percolate query is a specialized type of query that can be used to match queries stored in an index. The percolate query itself contains the documents that will be used as query to match with the stored queries. These types of queries contain the *json object* which can contain the **Multiple Dimension Criteria** used by the end user for their saved queries (e.g., A car with price within range of AED 45000 to AED 90000). It also supports the multiple documents simultaneously with the indexed percolator queries. E.g., multiple listings when introduced will match multiple types of saved search queries per user.

**Architecture Diagram:**

Diagram

Description automatically generated

Note: Please ignore the symbol(s) of the offering(s).

**Explanation of the architecture**

We have different types of databases belonging to different categories of product(s). In above example, *N* types of products.  
  
**Message Queue**- is hooked to the listing DB(s) via the server-less function. One way could be associating the DB based trigger(s) and have a function app listening to those changes. This enables near/real-time listing changes being transferred to the other component(s) of the system. Message queue will be used in the other scenario for sending email(s) as well from the background worker.

**Elastic Search (Saved Query Store)** – This is an elastic search store for persisting the Saved Search Queries (Percolate queries) of the User(s). This data will be pre-processed and will be later used by the percolate worker for search and match.

**Elastic Search (Listing Store)** – This is another collection within the Elastic Search which holds the listing data. This can be used by other full text search queries as well as fulfill the requirement to access the abstract (Title, description, etc.) by the notification service to generate the content of the email (if cache isn’t found).

**Cache Store** – Within the entire solution, there will be a cache store that will be mostly already present, the data like abstract (title, description, etc.) can be pre-cached in the existing cache store for a quick look-up on the runtime (cache invalidation isn’t a big problem as it’s not changing very frequently). We will use a dedicate worker process so that the API(s) always fetch from the cache. The cache worker takes care of ensuring the latest data is synced from the original data source.

**Percolation worker** – This worker does the match making of the percolated queries and the listing document to generate the listings recommended for a given percolated query. We can store the configuration in-memory to cap the number of listings recommended for a category/product.

**Server-less function** will be listening to the event grid and generate the notification on the runtime for the end user. To render the email content, we will need to collect the extra information. This information can be referred from the Azure Redis Cache, NoSQL.

**Save Search Feature API (REST Based)** – This API is behind the load balancer (hosted as an app service, Kubernetes). This will host endpoint for GET and POST API (for loading a Saved Search Query and Saving a Search Query).

**No-SQL Store** – Persistent storage of the user saved query store used to store user’s personalization data along with the saved search query metadata. There will be a page to simply display the saved queries for the user. This can also be used to display such data directly from the Save Search API (LoadSavedQuery method). The store could be Mongo DB or Azure Cosmos. A separate collection within the NO SQL for storing alert related data can help us to decouple the usage and responsibilities.

**Handling Scalability**

All the component(s) chosen as mostly serverless (hence they can scale upto- N requests) and most of them are Pay-As-You-Go so the cost factor is also considered if the traffic volume is less or more. Use of Event Grid and dividing the transactions into async workflows will ensure the runtime queries aren’t acting as a bottleneck to the live user facing queries. Appropriate use of Cache Store will ensure that the master data is quickly fetched wherever applicable (reading saved search queries and referring master data). The Concept of Percolation queries will ensure the match-queries are Realtime, fast and scalable as the load increases on the system. Typically, all the data-stores chosen are Geo-replicated or they have the option to be hosted across multi-geographic location. This will help to serve the request(s) from the nearest available region.

**Separation of Concerns**

Each of the component handle(s) and knows it’s responsibilities.  
Save Search API – It act as an interface for the End Customer to get the preference query and store it in the database for future retrieval.

Event Grid(s) – Ensure all the distributed system(s) are decoupled via the Event(s) and hence asynchronous in nature.  
Elastic Search Percolator- takes the responsibility of indexing the save search feature.  
Elastic Search Listing – takes the responsibility of indexing the document (for reference by other components) and Percolator worker.

Percolator worker – Does the work for match making and send the relevant event further to event grid.

Serverless function- Listens to the listing recommendation and triggers the notification email by collating the data first from the cache store, No-SQL store, as applicable to generate the content of the message and send the email to the End User.

Logical Services: The read (Read Search API) and write (update/create/delete) (Save Seach API) services are separately hosted so that we can have flexibility and control over the scale. The Save Search API can have multiple queries stored in the json format (schema free to accommodate multiple types of products each having different dimension).

**REST API Interface**

Create/Update/Delete API (userID is the ID of the user loggedIn)  
**POST**

[https://savesearchapi.dubizzle.com/{userID}/query/create](https://savesearchapi.dubizzle.com/%7buserID%7d/query/create)

[https://savesearchapi.dubizzle.com/{userID}/query/update](https://savesearchapi.dubizzle.com/%7buserID%7d/query/update)

[https://savesearchapi.dubizzle.com/{userID}/query/delete](https://savesearchapi.dubizzle.com/%7buserID%7d/query/delete)  
  
**Approx. contract of the query**

Graphical user interface

Description automatically generated with medium confidence

**GET**  
[https://savesearchapi.dubizzle.com/{userID}/savedQuery](https://savesearchapi.dubizzle.com/%7buserID%7d/savedQuery)

(make use of cache headers and other out of the box http GET functionality)