Graph-Powered Search: Neo4j & Elasticsearch

BY DR. ALESSANDRO NEGRO, MICHAEL HUNGER, AND CHRISTOPHE WILLEMSEN

CONTENTS

INTRODUCTION

WHY SEARCH AND WHY GRAPH

Search is a key feature of most applications. Users search for products, places, other users, documents, and more. And while structured search was common for early applications, today, driven by the ubiquity of internet search engines, full-text search dominates the usage.

The world around us consists of connected information. Being able to store and query those rich networks of data allows us to support decisions, make recommendations, and predict impacts. **Graph databases** enable both transactional as well as analytical uses on top of our highly connected domains.

Bringing both together allows us to enhance search results with graph-based capabilities like recommendation features or concept search, and also to use advanced search results as entry points to graph traversals.

OUR USE CASE: MULTI-FACETED SEARCH WITH RECOMMENDATIONS

Each domain has specific expectations in terms of search relevance and a different set of issues, constraints, and requirements.

Our use case example is **product search**, used by any retailer (Amazon, eBay, Target, etc.).

Text search and catalog navigation are not only the entry points for users but they are also the main ***"salespeople"***. Compared to other search engines, the set of "items" to be searched is more controlled and regulated.

For the search infrastructure, these aspects have to be taken into account:

- ***Multiple data sources***: Products and related information come from various heterogeneous sources like product suppliers, information providers, and sellers.
- ***Marketing strategy***: New promotions, offers, and marketing campaigns are created to promote the site or specific products. All of them should affect results boosting.
- ***Personalization***: In order to provide a better and more customized user experience, clicks, purchases, search queries, and other user signals must be captured, processed, and used to personalize search results.
- ***Provider information***: Product suppliers are the most important. They provide information like quantity, availability, delivery options, timing, and changes in the product's details.

All these requirements and data sources affect search results in several ways. Designing a search infrastructure for e-commerce vendors requires an entire ecosystem of data and related data flows together with platforms to manage them.

THE VALUE OF SEARCH

Search is a conversation between a user and a search engine.

Search is ubiquitous in modern applications. It's the fastest way to find relevant information in vast amounts of data. The search engine needs the ability to provide relevant results to the user's search terms, as well as to further refine and filter the search.

FACETING

Initial search results are often too broad and need to be filtered or refined, e.g., by using facets. *Facets* are categories derived from the search results that are useful for narrowing a search. Each facet represents an attribute of the structured information like category, price, color, location, etc., and comes with a count of contained results.

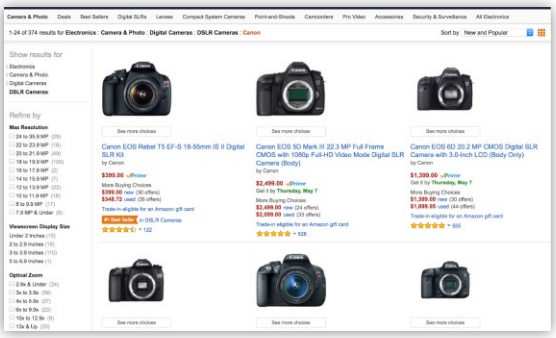
Neo4j is the #1 database for connected data, with native graph storage and processing.

Its property graph model and Cypher query language make it easy to use.

Fast. Natural. Fun.

TRY NEO4J NOW

neo4j



SEARCH ENGINE INTERNALS

1. **Indexing**: Documents are processed to make them searchable.
2. **User input**: Users specify the search request through some form of user interface or API.
3. **Ranking**: The search engine compares the input to the index and ranks documents according to how closely they match the query.
4. **Results display**: The final results are displayed via a user interface.

An index is a **collection of documents** that share some characteristics. For example, you can have different indexes for customer data, product catalog, order data, etc.

It is identified by an (all lowercase) name to perform indexing, search, update, and delete operations against the contained documents.

A document is a *basic unit of information* that can be indexed. Any number of *JSON documents* of different types are stored in those indexes.

INDEXING

Documents that are added to a search index are analyzed and prepared in order to create the *inverted index* data-structure in Lucene and other related structures to allow fast result retrieval during search. These are the steps of the document analysis:

- **Tokenization**
  - Breaking up a string into tokens to be indexed
  - Consistent handling of punctuation, numbers, and other symbols
  - Handle multiple tokenizations of compound words to match possible inputs
- **Downcasing**
  - All words are converted to lowercase for case-insensitive search
- **Stemming/stopword removal**
  - Strips words of suffixes, plurals, and conjugations
- **Synonym expansion**
  - Remove commonly occurring words
  - Newer search engines keep them for better results
  - Resolve synonyms via thesauri and add to index
  - Alternatively, synonym resolution can be done on search terms instead

EXAMPLE FOR INDEX CREATION

In Elasticsearch, during the index creation, it is possible to specify:

- All **settings** for the index
  - Number of shards and replicas
  - Custom analyzers
- The mapping that defines how a document, and the fields it contains, are stored and indexed

To create a simple index with one shard and two replicas for type `customer` with `name` and `description` (using a pre-defined analyzer for English text) fields, this call is used:

```
PUT customers
{
    "settings" : {
        "number_of_shards" : 1,
        "number_of_replicas": 2
    },
    "mappings" : {
        "customer" : {
            "properties" : {
                "name" : { "type" : "keyword" },
                "description" : { "type" : "text",
                                  "analyzer": "english" }
            }
        }
    }
}
```

For more detail on Elasticsearch index creation, refer to the documentation here.

SEARCH QUERY LANGUAGE

Elasticsearch provides a search API for executing queries and a JSON-based Query DSL to define queries.

The DSL has two types of clauses:

- *Leaf clauses*: check for particular value in a field (e.g., match, term, or range queries). These can be used on their own.
- *Compound clauses*: Wrap other leaf or compound clauses to combine them in a logical fashion (e.g., `bool` or `dis_max`), or to alter their behavior (e.g., `constant_score`).

The behavior of a *query clause* depends on the context:

- *Query* context: search for documents matching the query and calculate a relevance score
- *Filter* context: just check if a document matches; no scores are calculated.

```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title":   "Search"        }},
        { "match": { "content": "Elasticsearch" }}
      ],
      "filter": [
        { "term":  { "status": "published" }},
        { "range": { "publish_date":
                     { "gte": "2015-01-01" }}}
      ]
    }
  }
}
```

More detail can be found here.

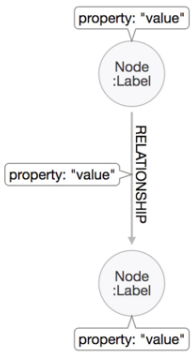
THE POWER OF THE GRAPH

Connected data is all around us and enables new types of applications. Everything from history and politics to economics, markets, science, and information processing is based on interrelated information.

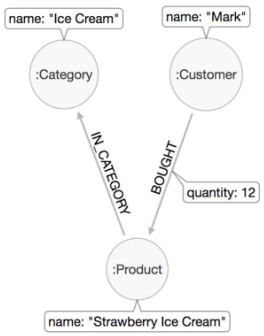
Use cases like social networks, routing, shopping, entertainment recommendations, and online data management (e.g., health tracking) become more and more important.

Graph Databases were developed to efficiently store, manage, and query that highly variable and richly connected data.

GRAPH DATA MODEL



The underlying **property graph model** consists of labeled nodes connected by directed, named relationships, both of which can hold arbitrary properties. It allows for expressive representation of any domain or use case.



The property graph allows you to consistently use the same model throughout all phases of a software project, making it also usable for non-technical stakeholders.

With the schema optional model, you can evolve your domain model as quickly as your requirements change, as no costly schema changes and migrations are necessary.

GRAPH QUERY LANGUAGE

To keep the richness of a graph model drawn on a whiteboard, we can use the declarative **Cypher query language**. It uses **ASCII-Art** to represent graph patterns of nodes and relationships.

```
(:Customer)-[:BOUGHT]->(:Product)
-[:IN_CATEGORY]->(:Category)
```

The query language is centered around these graph patterns. They can be used in clauses, expressions, and conditions to find and create data. Most of the non-pattern clauses were borrowed from other query languages like SQL.

Import (CSV) data, creating products:

```
LOAD CSV WITH HEADERS FROM "url" AS row
MERGE (c:Category {id: row.catId})
 ON CREATE SET c.name = row.catName;
CREATE (p:Product {id: row.prodId})
 SET p.name = row.prodName
CREATE (p)-[:IN_CATEGORY {added:row.date}]->(c)
```

Query data - products for category:

```
MATCH (p:Product)-[rel:IN_CATEGORY]->(c:Category)
WHERE c.name CONTAINS "Ice"
RETURN p.name, c.name
ORDER BY rel.added DESC;
```

BUILT-IN LUCENE INDEXES

For a long time, Neo4j has been using **Apache Lucene** to provide a lookup mechanism for starting points of graph traversals. Especially with explicit index procedures, you can use the full power of the Lucene query syntax to select the nodes your graph traversal would start with. Explicit Lucene indexes can also be configured with custom Analyzers and Stemmers as well as case sensitivity.

```
CALL db.index.explicit.searchNodes('products',
  $searchQuery )
YIELD node as product, weight
MATCH (product)<-[:BOUGHT]-(user)-[:BOUGHT]->(reco)
RETURN reco, count(*) as freq, weight
ORDER BY freq * weight DESC LIMIT 10
```

DATABASE EXTENSIONS

As an open-source database system, Neo4j can be extended in a variety of ways. You can subscribe to database updates via a `TransactionEventHandler`, which gives you access to added, updated, and removed Nodes and Relationships and their labels and attributes. This can be used, for example, to send data to other, eventual consistent systems like a search engine or auditing system.

```
public String beforeCommit(TransactionData data) {
  data.createdNodes.forEach(node -> {
    replicateToElastic(node);
  });
}
```

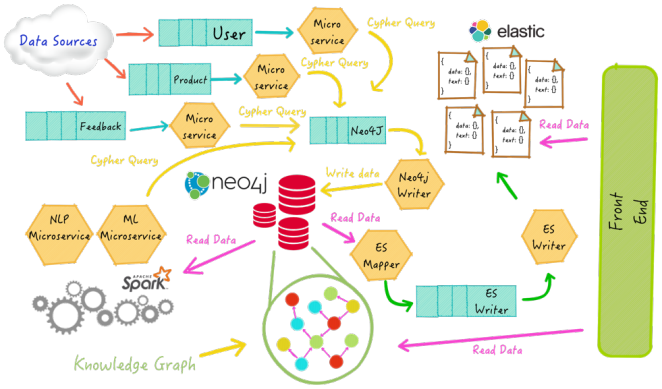
Furthermore, we can add user-defined procedures and functions that expose new capabilities to the Cypher query

language, e.g., direct access to other data systems (like Elasticsearch) or even the underlying built-in Lucene indexes.

A GRAPH-CENTRIC ARCHITECTURE FOR SEARCH PLATFORMS

The following high-level architecture contains:

- The knowledge graph as central element
- The type and amount of data to be processed
- The related data flows
- Requirements in terms of textual search capabilities



The data flow is composed of several data sources like product information sources, product offers, sellers, click streams, feedback, and more. All of these data items flow from outside to inside the data architecture using *Apache Kafka* as the queue and streaming platform for building real-time data pipelines. Information goes through a multi-step process where it is transformed before being stored in *Neo4j*, the main database of the infrastructure.

The raw sources are then enriched and processed and new relationships between objects are created. In this way, the knowledge graph is created and maintained. Many machine learning tools and data mining algorithms as well as Natural Language Processing operations are applied to the graph and new relationships are inferred and stored. In order to process this huge amount of data, an *Apache Spark* cluster is seamlessly integrated into the architecture through the *Neo4j Spark Connector*.

At this point, data is transformed into several document types and sent to an *Elasticsearch* cluster where it is stored as documents. In Elasticsearch, these documents are analyzed and indexed for providing text search. The front end interacts with Neo4j to provide advanced features that require graph queries that cannot be expressed using documents or simple text searches.

Neo4j is at the core of the architecture – it is the main database, the **"single source of truth"** of the product catalog, since it stores the entire knowledge graph on which all the searches and navigations are performed. It is a viable tool in a relevant search ecosystem, offering a suitable model not only for representing complex data (text, user models, business goals, and context information), but also for providing efficient ways of navigating this data in real time.

SINGLE KNOWLEDGE GRAPH, MULTIPLE VIEWS

A **knowledge graph** is composed of entities and relationships that describe facts in the real world. The use of graphs for representing complex knowledge and storing them in an easy-to-query model has become prominent for information management. Sometimes defined as "*encyclopaedias for machines*," knowledge graphs have become a crucial resource for advanced search, machine learning, and data mining applications.

MULTIPLE DATA SOURCES, A SINGLE GRAPH

Graphs — and in particular, knowledge graphs — provide the right information structure to merge data coming from multiple data sources in a *single source of truth*.

The search architecture must be able to handle highly heterogeneous data in terms of sources, schema, volume, and frequency of generation.

Moreover, these data sources have to be accessed as a **single data source**, so they must be normalized and stored using a unified schema structure that satisfies all the informational and navigational requirements of a search.

A simplified example of such a knowledge graph is represented in the following image:

Once the plethora of data is organized in an organic and homogeneous structure, it goes through a **process of enrichment** that comprises three main categories: cleansing, existing data augmentation, and data merging.

1. **Cleansing.** It's usually well worth the time to parse through elements, look for mistakes such as misspellings and duplications, and correct them. Otherwise, users might not find a document because it contains a misspelling of the query term.

2. The existing data is **post-processed** to augment the features already there. For instance, machine learning techniques can be used to classify or cluster documents. The possibilities are endless. After this new metadata is connected to the main elements, it can serve as a valuable feature for users to search through.

3. **New information is gathered** from external sources like ConceptNet 5. The goal is to provide users with every possible opportunity to find the document they are looking for, even though users express the same meaning using different but related words. This means more and richer search features.

The result of that process is a highly connected knowledge graph that represents the single source of truth for each entity. This valuable source of knowledge needs to be accessed and navigated to be effective.

PROVIDE MULTIPLE VIEWS IN ELASTICSEARCH FOR FAST ACCESS

Graph queries allow you to access data exploiting connections between elements, but they don't enable efficient textual searches. On the other side, search engines like Elasticsearch provide fast, reliable, and easy-to-tune textual searches.

In order to leverage such functionalities and provide efficient search to end users, data can be projected from the graph in a document format and stored in indexes in Elasticsearch. There, such documents can be analyzed according to the mapping defined for the index, and then become available for access using textual searches.

DEFINING MULTIPLE VIEW INDEXES

Since the knowledge graph represents multiple sets of information, the same data source can be used to serve multiple scopes, such as navigation and search, faceting, item details, and so on. Multiple views are, therefore, extracted and stored in several indexes in Elasticsearch, each of which could provide a different perspective on different types of search features.

Each view has to be designed, and this process requires you to:

1. Define the query to extract salient data from the graph;
2. Identify the list of fields in the document;
3. Choose the analyzers;
4. Define the mapping to the document.

This is an example of a query used to export the list of features for a specific product, taking different languages into consideration:

```
MATCH (p:Product)-[:HAS_DATA]->(pd:ProductData)
                -[:HAS_ATTRIBUTE]->(f:Attribute)
WHERE id(p) = {id}
MATCH (f)-[:HAS_KEY]->(k:Key)
MATCH (f)-[:HAS_VALUE]->(v:Value)
RETURN k.field_name as key,
    collect(v.locale + ":" + v.data) as values
```

SYNCING UPDATES TO ELASTICSEARCH

The updates in the graph database should be reflected in the Elasticsearch indexes. Ideally, the users should be able to define the graph entities to be replicated as well as the semantics of the documents that will be sent to Elasticsearch.

We call this configuration "replication mapping," and as shown in the sections below, the GraphAware Neo4j to Elasticsearch plugin makes this configuration easy and flexible — it uses a JSON format and the Spring Expression Language.

THE NEO4J ELASTICSEARCH PLUGIN

The GraphAware Neo4j to Elasticsearch plugin is an extension that takes care of replicating updates to the Graph as JSON documents to Elasticsearch.

Based on *replication mapping*, it does the following:

1. Inspect transactional changes asynchronously;
2. Determine if an update is relevant to be replicated;
3. Transform changes into JSON and send to Elasticsearch.

```
"node_mappings": [{
  "condition": "hasLabel('Person')",
  "type": "persons",
  "properties": {
    "name": "getProperty('firstName') + ' '
           + getProperty('lastName')"
  }
}]
```

This simple replication mapping dictates that updates (creations, updates, or deletes) of `Person` nodes should be replicated as an ES document with a `name` entry being a concatenation of the `firstName` and `lastName` properties.

EXAMPLE SETUP AND UPDATE WALKTHROUGH

Install and configure the plugin:

Download and install the following jars in the plugins directory of your Neo4j database:

- graphaware-server-community-all-3.3.x.jar
- graphaware-neo4j-to-elasticsearch-3.3.x.jar

Enable the plugin and provide your first replication mapping:

Add to `conf/neo4j.conf` configuration

```
com.graphaware.runtime.enabled=true
com.graphaware.module.ES.1=\
  com.graphaware.module.es.ElasticSearchModuleBootstrapper
com.graphaware.module.ES.uri=localhost
com.graphaware.module.ES.port=9201
com.graphaware.module.ES.relationship=(true)
com.graphaware.module.ES.mapping=\
  com.graphaware.module.es.mapping.JsonFileMapping
com.graphaware.module.ES.file=mapping.json
```

Configuration in `conf/mapping.json`

```
{ "defaults": {
   "key_property": "uuid",
   "nodes_index": "node-index",
   "relationships_index": "relationship-index",
   "include_remaining_properties": true,
   "blacklisted_node_properties": ["password"],
   "blacklisted_relationship_properties": ["uuid"],
   "exclude_empty_properties": false},

  "node_mappings": [{
     "condition": "hasLabel('Person')",
     "type": "persons",
     "properties": {
       "name": "getProperty('firstName') + ' '
              + getProperty('lastName')",
       "labels": "getLabels()"
     },{
     "condition": "getLabels().length == 0",
     "type": "nodes-without-labels"
   }],

  "relationship_mappings": [{
     "condition": "isType('WORKS_AT')",
     "type": "workers"
}]}
```

- *defaults* map that will be applied to all definitions
- *uuid* setting: which property of a node or relationship will be used as the unique identifier as well as the ES document ID
- *include_remaining_properties* has to include the properties of the object that are not specified in the definition

Now, restart the Neo4j server and the extension is ready to use.

CUSTOMIZING RESULTS USING RECOMMENDATIONS

Users' behavior and preferences can become a new source of information.

PROFILE-BASED PERSONALIZATION

Tracking knowledge of individual users happens with profiles. At query time, the user profile and its information is used to boost documents accordingly.

User profiles can be gathered differently:

- *explicitly* users provide information (ages, location, gender, and so on);
- *implicitly* determine user interests, demographics, and other attributes by observing user behavior.

User profiles can be used during search, e.g., by respecting brand affinity or interest in reviews or detailed product information.

BEHAVIORAL-BASED PERSONALIZATION

The notion described above can be improved with *collaborative filtering*. This technique uses historical information about *user-item interactions* ("users that bought this also bought these") to find items that naturally clump together by predicting peer-group behavior.

The output is a model that determines which items are most closely associated to a given user or item.

TYING USER BEHAVIOR AND PROFILE INFORMATION BACK TO THE SEARCH INDEX

Let's consider a standard, text-only search approach by incorporating collaborative filtering or profile as a multiplicative boost.

Our starting query is:

```
{
  "query": {
    "multi_match": {
      "query": "summer dress",
      "fields": ["title^3", "description"]
    }}}
```

To incorporate collaborative filtering, apply a multiplicative boost using a `function_score` query:

```
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query": "summer dress",
          "fields": ["title^3", "description"]
        }
      },
      "functions": [{
        "filter": { COLLAB_FILTER },
        "weight": 1.1}]
}}}
```

The contents of your `COLLAB_FILTER` filter determines the boosted documents; their score is multiplied with the `weight: 1.1` (plus 10%).

The different scores affect the ordering of the search results so that users will see results first that are more relevant based on their previous behavior.

With this approach the strategies for incorporating collaborative filtering can be implemented, each of which corresponds to a different `COLLAB_FILTER`- and indexing strategy. The strategies can also be combined.

Boost at Query Time. If the output of the collaborative filtering process is a set of user-to-item affinities, a suitable `COLLAB_FILTER` function can be specified:

```
COLLAB_FILTER = {
  "terms": {
    "id": ["item4816", "item3326", "item9432"]
  }
}
```

The system knows that the user performing the query could be interested in these named items. The search result is then modified to boost the score of these product by 10%.

Boost at Index Time. Add a new field to the documents being indexed named, for example, *`users_who_might_like`* with a list of all users who might like a given item. At query time, the `COLLAB_FILTER` will contain the user performing the query so that this will be used for boosting the results.

```
COLLAB_FILTER = {
  "term": {
    "users_who_might_like": "user121212"
  }
}
```

SEARCHING WITH GRAPH BOOST

There are two different implementations that allow you to seamlessly extend searches provided by Elasticsearch with features such as customization and concept search.

GRAPH-AIDED SEARCH APPROACH

The Graph-Aided Search Plugin improves search results by using data stored in Neo4j. After the search, and before returning the results to the user, this plugin requests additional information from the graph to achieve its goal.

Two main features are exposed by the plugin:

- ***Result Boosting***: Change the scores of results. The new score can be computed by combining *search score* with the *graph score* with different weights or custom formulas.

  Usage examples include boosting:

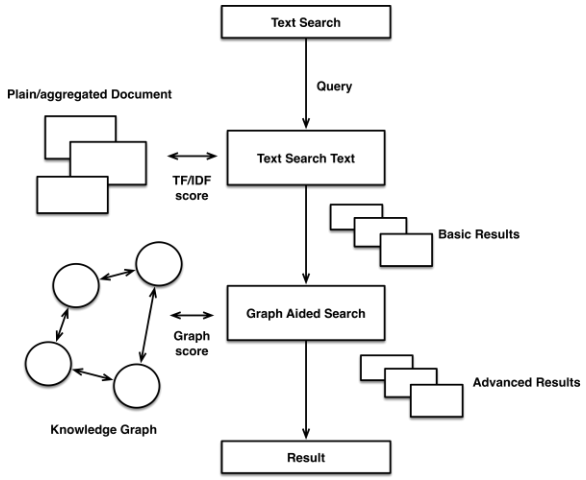
  1. based on interest prediction (recommendations),
  2. based on friends' interests/likes,
  3. content-based scoring, and
  4. other graph-based scoring methods.

- ***Result Filtering***: Removing result documents with filters using Cypher queries that match document IDs.

Example query with boost:

```
{ "query" : {
      "match_all" : {}},
  "gas-booster" :{
    "name": "SearchResultCypherBooster",
    "query":
      "MATCH (input:User) WHERE id(input) = 2
       MATCH p=(input)-[r:RATED]->
       (product)<-[r2:RATED]-(other)
       WITH other,
       collect(reduce(i=0, r in rels(p) | i+r.rating))
       as ratings
       WITH other,
            reduce(x=0, rating in ratings | x+rating)
            as score
       WITH other, score
       ORDER BY score DESC
       MATCH (other)-[:RATED]->(reco)
       RETURN reco.objectId as id, score
       LIMIT 500",
    "maxResultSize": 1000,
    "scoreName": "score",
    "identifier": "id"
     }
}
```

This query boosts the products in the result that are of interest to the customer, based on previous ratings.



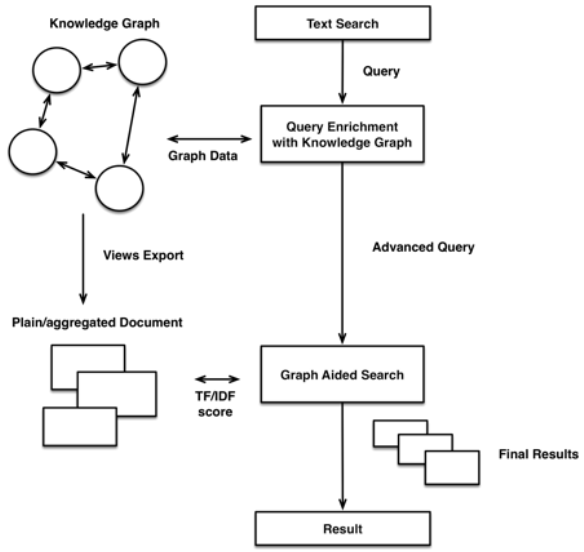
The query processing steps are:

1. Intercept and parse any "Search query" and try to find the `GraphAidedSearch` extension parameter;
2. Process the query extension identifying the type of the extension (*boosting* or a *filter*) and instantiate the related class;
3. Perform the operation required to boost or filter by calling the Neo4j API (or a Neo4j extension like GraphAware Recommendation Engine), passing all necessary information, e.g., Cypher query, target user, etc.;

4. Return the filtered/boosted result set back to the user.

KNOWLEDGE GRAPH APPROACH

The previously described approach has the drawback of having to post-process a potentially large set of search results, which can be compute-intensive. We can flip that around by pre-processing the query that gets sent to the search engine.



The knowledge graph is the source of truth. Its data is inserted into Elasticsearch as several documents, with potentially different structures or additional fields to existing documents.

During this phase, concept search, personalizations and any other advanced features described earlier are applied. So for instance, fields like `users_who_might_also_like` or `concepts` are added to documents. Token vectors are enriched with *synonym hierarchies*.

Once the documents are created and stored, we can use the following technique:

1. Intercept the query and enrich it using data in the knowledge graph (user preferences, user profile, other concepts), as of filters or queries;
2. The advanced query is then submitted to Elasticsearch;
3. The results of the query are returned to the user.

This workflow is easier and more performant since the enrichment phase happens at the beginning and is performed on a smaller dataset. Only that enriched query runs on the search engine without post-processing, ands so it will be very fast.

REFERENCES

- [1] D. Turnbull, J. Berryman – Relevant Search, Manning
- [2] A. L. Farris, G. S. Ingersoll, and T. S. Morton – Taming Text, Manning
- Neo4j
- Elasticsearch
- GraphAidedSearch Extension

ABOUT THE AUTHORS

DR. ALESSANDRO NEGRO is the Chief Scientist at GraphAware. He has been a long-time member of the graph community and he is the main author of the first-ever recommendation engine based on Neo4j. At GraphAware, he specialises in natural language processing, recommendation engines and graph-aided search. Before joining the team, Alessandro gained over 10 years of experience in software development and spoke at many prominent conferences, such as JavaOne. Alessandro holds a Ph.D. in Computer Science from University of Salento. He is based in Southern Italy but travels to clients around the world.

MICHAEL HUNGER has been developing software for more than 30 years in all kinds of domains. For the last few years he has been working with Neo4j, filling many roles. As caretaker of the Neo4j community and ecosystem he especially loves to work with graph-related projects, users and contributors. He is currently project lead for spring-data-neo4j, neo4j-graph-algorithms and the neo4j-apoc-procedures libraries. As a developer Michael enjoys many aspects of programming languages, (human and machine) learning, participating in exciting and ambitious open source projects and contributing and writing software-related books and articles. He's also a frequent speaker at and organizer for software development conferences.

CHRISTOPHE WILLEMSEN is a Principal Consultant at GraphAware. Before joining the team, Christophe worked as communication systems engineer for the Belgian Navy for almost 15 years and taught courses about maritime distress and emergency communication systems. He is a skilled software engineer who has led many big projects in HR and Operations Departments. Christophe is the author of Graphgen, an online graph generation engine for Neo4j, which received a Graphie Award during GraphConnect San Francisco 2014, and an active member of the Neo4j community.

DZone BROUGHT TO YOU IN PARTNERSHIP WITH neo4j

Refcardz Feedback Welcome: refcardz@dzone.com

Sponsorship Opportunities: sales@dzone.com

DZone, Inc. 150 Preston Executive Dr. Cary, NC 27513

888.678.0399 - 919.678.0300