Elasticsearch

Elasticsearch (ES) is an analytics engine that supports querying multiple types of data. It's distributed, free and opensource. It uses *Apache Lucene* internally as a search engine. It's accessible via REST API and is provided with a set of tools branded as Elastic Stack, with most popular of them being Logstash and Kibana. And most importantly, it was designed to reduce costs of performing queries and data-analysis tasks on big sets of data - both the infrastructural cost and the

Limitations of elasticsearch

Despite its adventages in querying performance and convenience, elasticsearch is not a great choice as an only database for real-life, business application. It does not feature transactions nor rollbacks. It's not real-time, that means that the data may be available even after over 1 second. Elasticsearch also may lose its data in some scenarios. Therefore, every team introducing elasticsearch to their stack must be aware of these limitations and address them thoroughly. ES is not an ACID database and should not be treated as such, meaning no business logic requiring consistency and security should be built solely upon it.

The project idea

During this project, we intend to show how to make a relational database (postgresql) work together with ElasticSearch and what are the benefits of doing that. The approach we will use is often described as data redundancy. That means that we will be persisting data both in a database more suited for safe persistence, and in the elasticsearch.
We will be working on the example app in the app source folder. We plan to focus on the process of introducing advanced search and data-analysis features to a very basic reddit-like website. In the process of benchmarking and implementing the features, we used this reddit comments dataset that provided us with 10 million data entries.

The application

The application that we will be working on is a very basic reddit-like website. It has the following features: * viewing top reddit comments * navigating to their subcomments * writing our own comments During this article we will add data analysis features that will allow us to query this massive dataset in a couple of handy ways.

App structure

The app consists of the following parts: - repositories (`source/app/repositories`) - services (`source/app/services`) - routes (`source/app/routes`) - views (`source/app/views`)

Each fo these parts have their own responsibilities. `Repositories` store the database logic and configuration, including querying and inserting new data to PostgreSQL and ElasticSearch. `Services` implement the business logic of the application. `Routes` define how the http api of the application behaves. `Views` consits of `jade` templates defining the views that then get rendered to `html` files.

Data structure

Each reddit comment consists of the following fields: - `subreddit` - where the comment came from - `author` - nickname of the comment's author - `body` - comment body - `parent_id` - what article/comment this comment is refering to - `name` - unique url identifier that will be used by us a primary key - `created_utc` - when the comment was created

Initial application endpoints

Below is the list of the endpoints of the basic application, not including the search endpoints that will be added in the process presented in this project.

GET / Returns view of main page of the website, it can be used to navigate to top comments.

GET /comments/top?after=:n Returns view of list of comments with the most upvotes. It has an option `after` query parameter that is used to paginate the results.

GET /comments/t/:parent_id Returns view of the parent comment with `name=:parent_id` and list of its subcomments

POST /comments/t/:parent_id Adds a new subcomment to the `parent_id` comment.

Usage of PostgreSQL in the app

We used the `pg` and `pg-promise` libraries to operate on the PSQL database. Using it requires creating a connection pool:

```
const pool = new Pool({
  user: 'reddit',
  host: 'localhost',
  database: 'reddit',
  password: '',
  port: 5432,
});
```

That can be then accessed with `pool.query` method to perform queries using this connection pool. The basic queries that the app is performing are:

Fetching top comments

```
async function fetchTopComments(limit, offset) {
  return await (await pool.query(`
    SELECT * FROM reddit_data ORDER BY ups DESC
    LIMIT ${limit} OFFSET ${offset}`)).rows;
}
```

It also features basic pagination with `LIMIT` and `OFFSET` keywords.

Fetching single comment

```
async function fetchComment(threadId) {
  return await (await pool.query(
      `SELECT * FROM reddit_data WHERE name='${threadId}'`
  )).rows;
}
```

Fetching comments for a thread with given id

```
async function fetchCommentsForThread(threadId, limit = 20) {
  return await (await pool.query(
      `SELECT * FROM reddit_data WHERE parent_id='${threadId}'
      ORDER BY ups DESC LIMIT ${limit}`
  )).rows;
}
```

Persisting a new comment

```
async function persistComment(subreddit, author, createdAt, body, ups, parentId) {
  let randomId = uuid.v4().replace('-', '');
  await pool.query(`INSERT INTO reddit_data
    (name, subreddit, subreddit_id, author, edited, controversiality,
    created_utc, body, ups, downs, score, parent_id, archived)
    VALUES ('${randomId}', '${subreddit}', '${subreddit}', '${author}', 'false',
    '0', '${createdAt}', '${body}', '${ups}', 0, '${ups}', '${parentId}', 'false')
  `)
  console.log("inserted")
}
```

It uses a random v4 UUID trimmed of dashes as id for the new entry.

App interface

Interface of the app is very simple, below are a few screenshots:

/

Figure 1: alt text

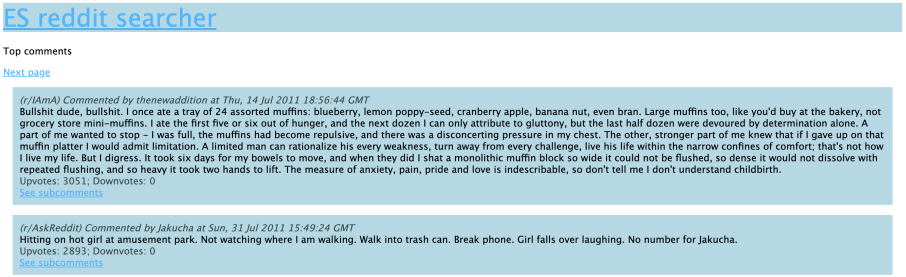


Figure 2: alt text

/comments/top

/comments/t/:thread_id

Setup

As a first step, we need to set up our environment.

Importing the data and setup

Installation (macOS)

The first step is installation of ES stack and postgres databases.

```
> brew install elastic/tap/elasticsearch-full
> brew install elastic/tap/kibana-full
> brew install postgresql
```

Starting the databases

The following commands can be used to run the databases:

Figure 3: alt text

```
> elasticsearch
> kibana
> initdb -D postgres
> postgres -D postgres
```

Feeding the databases with data

Postgresql Now, we have to import all the required data. We created an utility app at the dataimport/postgresql source directory. In order to run it, there has to be a `dataset.json` file in the project root directory. It creates a necessary table to store the data, and then exports the data to csv. Then, the data can be imported with data import utility tool in DataGrip.

Elasticsearch In order to import data to the elasticsearch we created a utility app available at dataimport/elasticsearch source directory. After running it, it will produce files containing fields mapping and prepared dataset for elasticsearch. In order to import them to the running database, the following commands must be run:

```
curl -X PUT "localhost:9200/reddit_data?pretty" -H 'Content-Type:application/json'
    --data-binary @elastic-mapping.json

npx elasticdump --input=elastic-dataset.json --output=http://localhost:9200/reddit_data
    --type=data --transform="doc._source=Object.assign({},doc)" --limit=10000
```

Using ElasticSearch

API

ElasticSearch is accessible via a REST API. Most operations on it are performed via various endpoints that this API provides.

Creating index

`index` in ES is the top-level dataset that is used to store the entries. Creating a new one is as easy as calling: `PUT /reddit-data` However, we needed to provide some basic settings for the new index. We needed to provide datatypes of the fields of reddit entries. In ES it is done through index mappings. In our case the mappings were defined as follows:

```
mappings: {
  properties: {
    'name': { type: 'keyword' },
    'subreddit': { type: 'keyword' },
    'subreddit_id': { type: 'keyword' },
    'author': { type: 'keyword' },
    'edited': { type: 'text' },
    'controversiality': { type: 'integer' },
    'created_utc': { type: 'date' },
    'body': { type: 'text' },
    'ups': { type: 'integer' },
    'downs': { type: 'integer' },
    'score': { type: 'integer' },
    'parent_id': { type: 'keyword' },
    'archived': { type: 'text' },
  },
}
```

For each field we specified a type that was used to index the entries as we entered them into the database.

`keyword` vs `text` type The most important distinction we made in the types is the decision between `keyword` and `text` for the string fields. As ES learning source states:

`The primary difference between the text datatype and the keyword datatype is that text fields are analyzed at the time of indexing, and keyword fields are not. What that means is, text fields are broken down into their individual terms at indexing to allow for partial matching, while keyword fields are indexed as is.`

If we used `keyword` type for `body` of each reddit comment, we would not be able to do efficient full-text search on the comments content and match single words or sets of words of it.

Inserting the entries

The endpoint used to put a single entry into the index is: `PUT /reddit_data/:entry_id` (reddit_data is name of our index) The entry data is passed as a json in the request body, and then the entry is added and indexed in the index. However, we had to insert over 10 million of them, so doing it one by one was not an option.

Bulk API In order to insert a massive amount of entries, we had to use the `elasticdump` tool, that is in fact using the `Bulk API` of elasticsearch. It uses the following endpoint: `POST reddit_data/_bulk` And it passes newline-separated jsons as data. For inserting new entries, each line must be structured as follows: `{ "create": {... entry data as json} }` However, other operations are also allowed, including `update` and `delete`.

Fetching the data

To query the data from elasticsearch we used the `_search` endpoint: `GET reddit_data/_search` We used it via `elasticsearch` npm library that is in fact a simple wrapper over the REST http client. We will be diving more into querying in the future sections. The basic structure of the query requires a `query` field in the json body. This field provides a query description that will be then used to search the dataset. For example, searching for `apple` in the body of reddit comments can be described as a following query:

```
query: {
  match: {
    body: 'apple'
  }
}
```

`match` operation performs a full-text search for the provided string in the `body` fields of the entries. This search is technically a `fuzzy` search, that means that the `body` field does not need to include literally `apple` string, but any string that is similiar. We will describe it in later on. Another useful field is the `size`. It specifies what is the maximal number of entries that should be returned from elasticsearch in this single query. Default value is 20, so for better benchmarking we raised it to `5000` in all of the queries.

Scoring

Scoring is the core concept of querying in elastic search. Each query gets an assigned score that is then used to order the returned entities respectively. ElasticSearch is built upon Lucene and it had a decisive impact on how the scoring mechanism works. Score is assigned to an entity with the Lucene's `Practical Scoring Function`:

```
score(q,d)  =
```

```
  queryNorm(q)
· coord(q,d)
· sum (
      tf(t in d)
    · idf(t)²
    · t.getBoost()
    · norm(t,d)
  ) (t in q)
```

Where: - `score(q,d)` is the scoring function for document (entry) d and query q - `queryNorm` is the quotient used to normalize the query and is calculated as `queryNorm = 1 / sqrt(sumOfSquaredWeights)`. Normalizing the query is the process that allows queries to work together despite different structure and different weights assigned to subqueries. - `coord` is the coordination factor. It denotes how big a part of the queried text was found in the entry fields. - `tf` is the term frequency, meaning how many the queried term occured in the entry fields. - `idf` is the inverse document frequency. Its value says how often the searched term occurs in all the entries in the index. If the term is uncommon, the it should be rewarded. - `norm` is the `field length norm`, and that's basically saying how long the field in the document is. If a field is shorter, then it should be more rewarded that the match occured.

Implementing and benchmarking complex searching

Full-text search

Full-text searching features allow us to search for a specific term (or similiar ones) in the text content. In our case, we want to search for a given term (or terms) in the comments. In our application, we want to create a new endpoint that will allow us a full text search on the full set of entries:

`GET /search/textSearch?string=:searchString` ### Basic PostgresSQL approach

```
async function textSearchPsql(searchString) {
  return await (await pool.query(
    `SELECT * FROM reddit_data
      WHERE body LIKE '%${searchString}%' LIMIT 5000`
  )).rows;
}
```

Above function uses basic `LIKE` syntax. This function was introduced by us in the `searchRepository`, and then accessed from views via services. However, that naive approach can't be used in the production. Firstly, it does not behave as we want it to. It only matches comments that include `searchString` literally inside of them. For example, given search string `i love bananas` and the

comment body `i love all bananas` there would be no match, and that's not the behaviour we want. Secondly, the performance of this solution is very, very bad.

Benchmarks

Search string	Returned entries (5000 limit)	Time
bananas	5000	24581ms
capybara	38	33080ms
i love you	765	24138ms

Elasticsearch approach

In contrast to the naïve PostgreSQL `LIKE` approach, the Elasticsearch gives us everything out of the box. The search is fuzzy, so query `i love bananas` would match `i lve bananas`. It treats the terms separately, so it will work easily for strings like `i love all bananas`. And last, but not the least, it's a lot faster.

```
function extractSearchBodies(hits) {
  return hits.map(extractSearchBody);
}

function extractSearchBody(hit) {
  return hit._source
}

async function textSearchElastic(searchString) {
  const { body } = await esClient.search({
    index: 'reddit_data',
    size: 5000,
    body: {
      query: {
        match: {
          body: searchString
        }
      }
    }
  })
  return extractSearchBodies(body.hits.hits);
}
```

Due to the response structure, we had to extract it using the `extractSearchBodies` function. The query is very simple and is the essence of what ES was designed to do.

Benchmarking

Search string	Returned entries (5000 limit)	Time
bananas	4757	1264ms
capybara	40	57ms
i love you	5000	2722ms

We can see a massive level of performence improvement. In the case of terms that do not appear often (`capybara`) the performance difference is 3 orders of magnitude. We also see difference in number of returned entries. It is due to the algorithm of calculating `score` for the searched terms - entries are returned if the resulting score lands over a given threshold.

PostgreSQL Text Search Vector

However, the approach we used in PostgreSQL is too naive. There is a feature for full text search in PostgreSQL and to make a fair comparasion we will implement it as well. Fast full-text searches in PostgreSQL are based on text search vectors (tsvector) that store the text fields as vectors of terms that is then stored in sorted form allowing fast traversal.

As a first step we have to create a GIN for `body` tsvectors index (in order to have the tsvectors prepared before requests):

```
CREATE INDEX body_tsv_index ON reddit_data
    USING gin(to_tsvector('simple',body));
```

First observation is that this operation was very slow. It took over 20 minutes to complete, but now we can write an efficient full-text search query using postgresql:

```
async function textSearchPsqlTsv(searchString) {
  return await (await pool.query(
    `SELECT * FROM reddit_data
      WHERE to_tsvector('simple', body) @@ to_tsquery('''${searchString}''')
      LIMIT 5000`
    )).rows;
}
```

`@@` operator executes a given query on a tsvector. Above code will search for the `searchString` in the tsvectors created for `body` fields.

Benchmarking

Search string	Returned entries (5000 limit)	Time
bananas	4726	726ms
capybara	40	20ms
i love you	5000	352ms

The PostgreSQL TSV is the fastest approach. However, we think that there is an important reason to it being so fast - it's just much simpler than the ElasticSearch scoring. It still does not fullfill all of our expectations - it just checks if the given terms are included in the text search vectors. ElasticSearch model would still give us results that would help us build much better full-text search function to offer to the app's users. And it's also much simpler in terms of the written code and preparations.

UI screenshots

Figure 4: alt text

Text search input

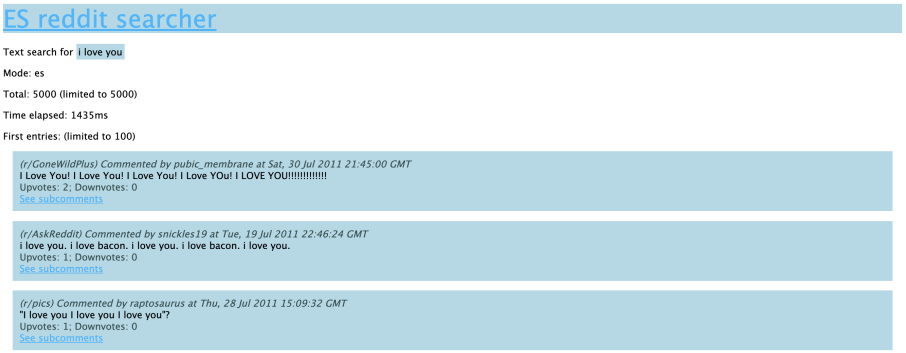


Figure 5: alt text

Text search results

Custom regexp search

Regexp search can be used to find given patterns in data sets. It can be particularly useful in finding emails, phone numbers, addresses and other structured data in text content. However, completing regexp searches in real time can be a demanding task. We created a new endpoint and a view available at: `GET /search/regexSearch?string=:regexpString`

PostgreSQL approach

PostgreSQL offers `~` operator that returns a boolean value that denotes whether a given string matches the regexp. Using this syntax we can define a regexp search function in node:

```
async function regexSearchPsql(regex) {
  return await (await pool.query(
    `SELECT * FROM reddit_data
      WHERE body ~ '${regex}'
      LIMIT 5000`
    )).rows;
}
```

This function will return all comments where the comment's body matches the given regular expression.

Benchmarking Below are the results of testing this method. The last regexp is a simple expression for short emails. | Search regexp | Returned entries (5000 limit) | Time | |———————————|———————————-|———| | `colou?r` | 5000 | 2686ms | | `ni{2,5}ce` | 144 | 29746ms | | `[a-zA-Z0-9]{1,10}@[a-zA-Z0-9]{1,10}\.com` | 2639 | 40808ms |

Elasticsearch approach

Elastic search offers an out of the box feature of regexp matching. It can by accessed by a `regexp` query. ES offers additional options for regex matching: - `flags` allows to enables additional features and syntax for writing regular expressions, for examples `<a-b>` syntax for numeric ranges - `max_determinized_states` limits amount of states in the finite automaton generated in order to match the query. It's particullarly useful for limiting complexity of the queries entered by app's users. - `rewrite` determines how the ES will rewrite the query to make it possible to calculate a score for it.

```
async function regexSearchElastic(regex) {
  const { body } = await esClient.search({
    index: 'reddit_data',
    size: 5000,
    body: {
      query: {
        regexp: {
          body: {
            value: regex,
            flags: 'ALL',
            case_insensitive: true,
            max_determinized_states: 10000,
            rewrite: 'constant_score'
          }
```

```
        }
      }
    }
  })
  return extractSearchBodies(body.hits.hits);
}
```

Benchmarking

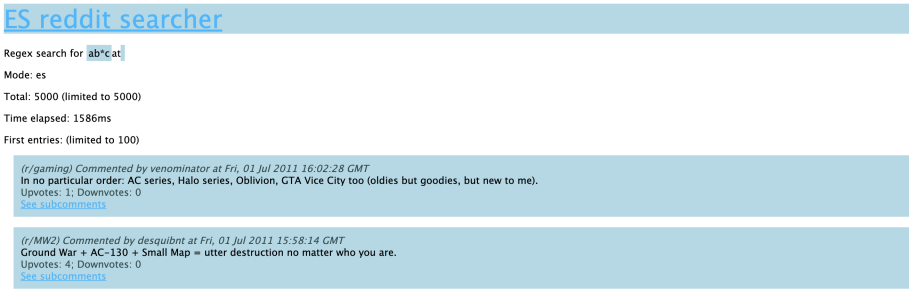
Search regexp	Returned entries (5000 limit)	Time
`colou?r`	5000	2202ms
`ni{2,5}ce`	377	277ms
`[a-zA-Z0-9]{1,10}@[a-zA-Z0-9]{1,10}\.com`	5000	1183ms

Once again, ElasticSearch provided much better and consistent performance. It's noticable that it even returned more entries for some of the expressions. It is due to the same mechanism that was present in the previous part about the full-text search. ES queries are rewritten in a way to be fuzzy, so queries that are similiar to the pattern, but not necessarily matching, are also returned.

UI screenshots

Regexp search input

Regexp search results



Weighted search

Very useful type of searching complex text data is a weighted search. Given a couple of fields, we often want to find if any of these fields contain a given string. What's more, we often want to favor some fields over other. That's when the weighted search is useful. It allows to perform a full-text search in a couple of fields and base scoring on a weighted scores of subqueries performed per field. In our case, we want to search for a given string in: - `author` with weight equal to 10 - `body` with weight equal to 4 - `subreddit` with weight equal to 2

The most important to us is a case when the author name matches the string, then the comment body, and then subreddit name. We created an endpoint that allows querying and presents a view of the weighted search results: `GET /search/weightedSearch?string=:regexpString`

PostgreSQL TSV approach

In PostgreSQL we can once again utilize `text search vectors` and create a new field `tsv` that will contain the `tsvector` values that will be used in this query:

```
ALTER TABLE reddit_data ADD COLUMN tsv tsvector;

UPDATE reddit_data SET tsv =
   setweight(to_tsvector(author), 'A') ||
   setweight(to_tsvector(body), 'B') ||
   setweight(to_tsvector(subreddit), 'C') WHERE 0=0;
```

The `A, B, C` letters denote the weight of the vector elements. Default values of these weights are matching the 10:4:2 ratio, so we do not need to adjust it further. What's important to note, these queries took **over 40 minutes** to execute. What's more, during the execution of above queries, the following error appeared over a 100 of times: `[54000] word is too long to be indexed` This means that not all the comments are searchable, because some of them were too long to create a tsv vectors out of them. Now, we can execute a `SELECT` query utilising the new `tsv` field, just like we did in the `full text search` part. However, this time we need to order the entries by the tsv rank that is calculated by the `ts_rank_cd` method by matching given terms to each term of the created tsv field, and then returning the weighted count of matches.

```
async function weightedSearchPsqlTsv(searchString) {
  return await (await pool.query(
    `SELECT *, ts_rank_cd(tsv, query) AS rank
      FROM reddit_data, to_tsquery('''${searchString}''') query
      WHERE query @@ tsv
      ORDER BY rank DESC
      LIMIT 5000`
    )).rows;
}
```

SELECT *, ts_rank_cd(tsv, query) AS rank FROM reddit_data, to_tsquery("'${searchString]"') query WHERE query @@ tsv ORDER BY rank DESC LIMIT 5000

Benchmarking

Search string	Returned entries (5000 limit)	Time
bananas	5000	28044ms
capybara	48	28608ms
i love you	5000	458ms

Once again, the queries took very long to execute and no user would want to wait that much to see their search results.

Elasticsearch approach

Weighted search is another feature that is provided out of the box in elasticsearch. For a query of type `query_string` we have to specify `fields` that will be matched against. If we provide their names with `^` and a number next to them, they will interpreted as having a weight equal to the given number.

```
async function weightedSearchElastic(searchString) {
  const { body } = await esClient.search({
    index: 'reddit_data',
    size: 5000,
    body: {
      query: {
        query_string: {
          query: searchString,
          fields: [
            "author^10.0",
            "body^4.0",
            "subreddit^2.0"
          ]
        }
      }
    }
  })
  return extractSearchBodies(body.hits.hits);
}
```

The code is very simple and requires no setup and additional indexing at all.

Benchmarking

Search string	Returned entries (5000 limit)	Time
bananas	2551	1323ms
capybara	40	28ms
i love you	5000	2604ms

These results are much better than in the postgresql TSV approach and can be considered real-time.

Additional thoughts

Postgresql TSV approach is not capable of providing dynamic weights for the queries. The queried field must be prepared before querying, and it can take a very long time. For our dataset consisting of 10 million entities, it took over 40 minutes. For more entries, it would probably get linearly worse. However, using ElasticSearch, we were able to execute dynamic queries on arbitrary fields in much shorter time.

UI screenshots

Weighted search input

Weighted search results

Searching in recent comments

Another great example of flexibility of ElasticSearch querying mechanism is its ability to search based on multiple conditions and mix their scores. Let's say we were given a task of implementing a feature that would allow our users to search in the recent comments for a given term. In this part we implemented a ES querying function that performs this operation.

`function_score` query

`Function score query` is another type of query in elasticsearch that makes it possible to build a complex query that integrates a couple of subqueries, while allowing a custom scoring combination algorithm. The most important fields of this query type are: - `score_mode` that specifies how the scoring is combined for all of the queries. - `functions` that allow specifying additional functions used

in the querying (their score is then combined using `score_mode`) - `query`, the query to be combined with queries in `functions`

Decay functions

Decay functions score entries with a function that returns a value depending on a distance of a numeric field from the given origin. We can specify a distribution that will be used to calculate the distance, for example the `gaussian` distribution. That's exactly what we want to use in our feature - we want to take the most recent posts. Below is the gaussian decay function that we will be using to fetch the recent posts.

```
gauss: {
  created_utc: {
    origin: date,
    scale: "10d",
    decay: 0.5,
    offset: "2d"
  }
},
```

The above snippet says that we want to match the `created_utc` field against the provided `date` with a gaussian distribution calculated based on the provided values. The resulting distribution is best described with the following image:

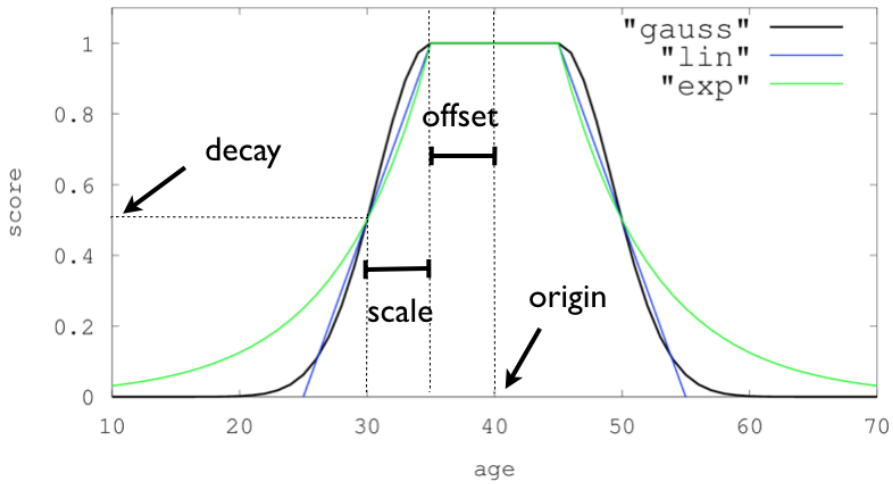


Figure 6: alt text

By combining the gaussian decay function and a simple string matchin with a `multiply` score_mode, we get exactly what we wanted to achieve - a scoring

algorithm returning a product of the string match score and the gaussian decay score:

```
async function recentSearchElastic(date, searchString) {
  const { body } = await esClient.search({
    index: 'reddit_data',
    size: 5000,
    body: {
      query: {
        function_score: {
          functions: [
            {
              gauss: {
                created_utc: {
                  origin: date,
                  scale: "10d",
                  decay: 0.5,
                  offset: "5d"
                }
              },
            }
          ],
          score_mode: "multiply",
          query: {
            match: { body: searchString },
          }
        }
      }
    }
  })
  return extractSearchBodies(body.hits.hits);
}
```

Benchmark

Search string	Date	Returned entries (5000 limit)	Time
bananas	2011-07-14	2551	549ms
capybara	2011-08-21	40	25ms
i love you	2011-05-19	5000	1961ms

Once again, the queries were fast and there would be no problem with providing the users with this feature.

UI screenshots

Recent search input

Recent search results

Summary

In this project we implemented a basic commenting platform based on a massive dataset imported from reddit. It provided basic features like commenting and viewing top comments and subcomments, but most importantly - we provided a couple of complex searching features. We believe that it would be much harder, or even next to impossible, to provide them just with the PostgreSQL database. By using ElasticSearch, we implemented these features in a simple and efficient way.