TTDS Team 14 Report

Daniel Kirkman, Kyle Cotton, Martin Lewis, Maksymilian Mozolewski, Wassim Jabrane, Annabel Jakob

Contact Email: s1849352@ed.ac.uk

1 Introduction

Wikipedia is the world's foremost online encyclopedia, used by millions of users every day. Our search engine is called `OnlyGraphs`, alluding to our goal of providing a graph-based search, allowing users to find relevant articles on Wikipedia. We have built a search engine with advanced search features for a subset of 1.6m Wikipedia articles, which tailor the search results better to the users expectations and requirements. This subset consists of the most popular articles (the most recently updated articles) on Wikipedia. The more popular articles are in general larger than the less popular articles. This gives us a data set of around 7GB in size which is how much data we store in the Postgres database before indexing. Once the user finds an article of interest, the graph search aids them in finding related articles. The graph representation not only visualises related articles, but also shows the proximity of articles to each other as well as their popularity by scaling the size of an article based on its page rank score.

The system has been implemented using the Rust programming language, giving us blazingly fast search performance and reliability via its type safety system.

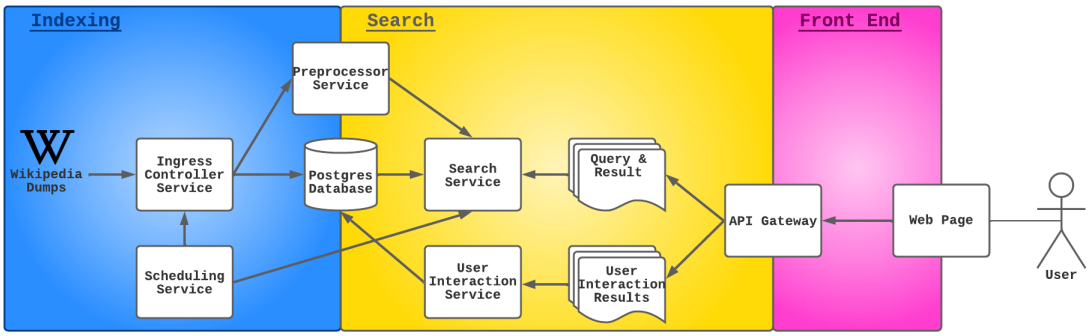

Figure 1: System Architecture

2 Data Ingress

This section outlines the information retrieval and storage of the Wikipedia data, along with the required preprocessing.

The Ingress Controller service is responsible for the retrieval of raw data from Wikipedia and the processing of this data into a format which can easily be indexed by the Search Service module at run-time.

We use a database to keep a persistent storage of the data which allows us to recover the system in the unlikely event that the search service should crash. It is important to note that our database is not used for searching. All data in the database is read into in-memory data-structures (see section 3), which are used to service queries.

Wikipedia kindly provides a mirror of the latest version of the English Wikipedia articles (called a Wikipedia dump) every two weeks [1]. Wikipedia explicitly forbids bulk scraping of their web pages [2], hence it would not be feasible or ethical for us to attempt to scrape the entirety of the English Wikipedia, which is over 6 million articles (not including any redirects).

Our system automatically queries the rss-file indicating if a new Wikipedia dump-file has been released. If a new file has been released, then we automatically download and decompress the dump file. Individual articles inside Wikipedia dumps are written in Wikitext (Wikipedia's internal markup format).

The Ingress Controller extracts the following information for each article from Wikitext format and stores it in our database:

1. **Text:** the text content from the article as seen by the user on a Wikipedia article.
2. **Links:** the Wikipedia articles which this article links to.
3. **Categories:** the categories which this article is tagged as being a part of. Added by users to Wikipedia articles and useful for describing the content of an article. For example the English Wikiepdia article for "Boris Johnson" [3] has the category "Prime Ministers of the United Kingdom".
4. **Abstract:** we generate our own abstract which is used to display a description of the article on our front end.

[1] https://dumps.wikimedia.org/enwiki/
[2] https://en.wikipedia.org/wiki/Wikipedia:Database_download#Why_not_just_retrieve_data_from_wikipedia.org_at_runtime?
[3] https://en.wikipedia.org/wiki/Boris_Johnson

5. **Infoboxes:** tables present on Wikipedia articles used to collect and present a summary of the article information. There are different types of infobox [4] on Wikipedia. The type can be useful in telling us what type of article we are viewing. For example the English Wikipedia article on "Greggs" (the popular bakery chain), [5] contains an infobox with type "company". There are over 4 million infoboxes on wikipedia.
6. **Citations:** we collect and preprocess the details of each distinct citation on an article. There are over 48 million distinct citations on the English Wikipedia.

e have implemented a version control system in the Ingress Controller that allows us to efficiently update our database when a new Wikipedia dump is released. For each article, we keep track of the dump from which the article was retrieved and when the article was last edited on Wikipedia. This allows us to do a differential update of our database, meaning we only have to update articles if they have been edited since the last database update. This drastically reduces the number of articles that need to be updated after each new Wikipedia dump is released.

The Ingress Controller also includes a purge process that ensures that stale content is deleted from the database. This process is executed after a new Wikipedia dump is added and will delete articles which have been deleted since the last update. It also removes any links to articles which no longer exist in our database. This process ensures that the Search Service always has valid data to work with.

2.1 Preprocessing

Preprocessing is a service that is required by many other services. To provide this functionality, a Rust [6] crate has been produced. This is a shared library that can be included as a dependency in any other Rust projects, by adding a single line to the `Cargo.toml` file.

Wikipedia has a large assortment of articles, their domains spanning an enormous space. Given this variety, and the fact the content has no affinity to a particular domain the English ISO Standard list was used.

This introduces a number advantages, namely the *Ingress Controller* discussed in Section 2 and the *Search Service* discussed in Section 3, can have identical preprocessing options (in fact they share the exact same code). As the Wikipedia content and the query are preprocessed in the same manner, this improves the results provided by the system. Additionally there is a significant deduplication code and more efficient use of development time.

3 Search

This section delves into the backbone of the entire search engine system, the backend. There has been continuous refinements to the algorithms and the data structures, in order to allow supporting multiple features and **1.6 million** articles (not including any redirects).

3.1 Search API

The Query engine is accessed via REST API. The API supports two endpoints, normal search (that is, search by using queries) and graph search (also referred to as *relational* search). For each search request, the query is parsed into Abstract Syntax Trees (see 3.2.1). The resulting query is then preprocessed, and finally executed accordingly. Specific to normal search, the query is also passed to a spell correction algorithm for certain types of queries (see 3.7.5), which prompts the user with a new suggested query if there are too few results. The results are then sent to the frontend application. This can be seen in figure 2

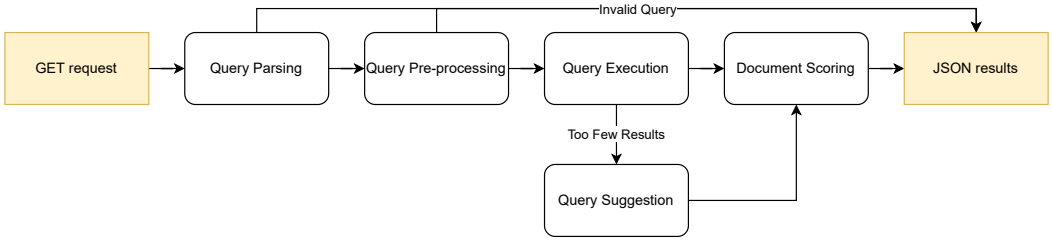

Figure 2: Overall query execution pipeline

The types of search supported are:

- Free text Query. Example: jasmine tea.
- Binary Query - Supports NOT, AND, OR. Example: jasmine AND tea.
- Phrase Query. Example: "jasmine tea".
- Distance Query. Example: #DIST,2,jasmine,tea.

[4] https://en.wikipedia.org/wiki/Wikipedia:List_of_infoboxes
[5] https://en.wikipedia.org/wiki/Greggs

- Advanced search (Filter by different parts of the article, also called *structure* search).
- Wildcard Query. Example: jas*ine.
- Relational (Graph) Search.
- Sorting by update date or by relevance (PageRank + TFIDF).

3.2 Parser

3.2.1 Abstract Syntax Tree

To facilitate searching, we designed a query grammar that specifies legal queries which the search engine must expect. Within Rust, this grammar is translated into an abstract syntax tree (AST). ASTs allow to represent abstract syntactic grammars [2], and are hence useful to represent potentially nested queries. Further, they structure the query and thus simplify the search. Figure 3 visualises the AST corresponding to the query "Hamster AND Pouch OR Cage". The search algorithm now knows to OR the results returned by searches for "Pouch" and "Cage", and then AND these results with those of the search for "Hamster". Within Rust, the AST is implemented as an enumeration, with a sub-type for each type of query. This represents the fact that all types of queries belong to the same super class (that is, the `Query` super class). In addition, this implementation allows queries to contain nested queries of any type.

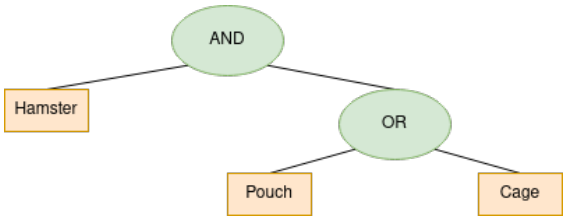

Figure 3: AST corresponding to the query "Hamster AND Pouch OR Cage".

3.2.2 Parsing

The parser uses the nom [6] crate, which is a parser library that facilitates safe parsing [1]. nom provides useful methods, such as consuming a specific number of bytes or recognising alphabetical characters, which simplifies detecting patterns in strings.

We implement separate methods for each type of query. For instance, there exists a method to parse AND-queries and distance queries. If the parser encounters an unexpected pattern within a method, it returns an error. For instance, assume the parser is given the query "hamster pouch" and attempts to parse it as a distance query. For distance queries, the parser expects the first argument to be "#DIST". The aforementioned query does not conform to this and hence an error is returned. When receiving a query, the parser iterates over all query types, attempting to parse the query as that type. If the query is not of a particular type, the parsing method for that query type returns an error and the parser backtracks. If the parser cannot find a method that successfully parses a query, an error is returned to the frontend application. If the parser is successful in parsing a query, an abstract syntax tree representation of the query is returned. The order in which to try query types is important and must be carefully selected as the parser parses the query as the first type which does not throw an error. This order was determined based on the traits of different queries, as well as trial and error. For instance, it is clear that the parser should first attempt to parse a string as a binary query, before attempting to parse it as a free text query, as both query types would be successful. On the other hand, it was somewhat of a surprising discovery that structural queries must be parsed before binary queries.

3.3 Query Pre-processing

Following parsing, the AST is passed through the query pre-processor which applies pre-processing to all tokens in the query. Empty tokens are pruned and single token phrase queries are converted to free-text queries.

3.4 Query Execution Engine

The Query engine accepts the pre-processed query as input, and recursively returns an iterator over postings (according to query type). A posting for the purposes of the engine, is a tuple of document ID and position, each query type needs to operate over postings and output postings as well, allowing for infinite recursive compatibility (within resource limits). We make use of four custom linear merge iterators to compose one final query postings iterator which is then used to generate the results list for a query. The iterators are:

- Distance Merge Iterator (DMI)
- Union Merge Iterator (UMI)

[6] https://docs.rs/nom/latest/nom/

- Intersection Merge Iterator (IMI)

Each such iterator corresponds to an operation over two ordered lists A,B such that for list X the following holds:

$$\begin{aligned} &\forall X[i].\forall j > i. \\ &(X[i].docid \leq X[j].docid) \wedge \\ &((X[i].docid = X[j].docid) \implies (X[i].pos \leq X[j].pos)) \end{aligned} \tag{1}$$

Iterator outputs hold the same condition, and we can describe their operations like so:

$$DMI(A, B, d) = [\exists l \in A. \exists r \in B. r.docid - l.docid \leq d] \tag{2}$$

$$UMI(A, B) = [e.docid \in A \vee e.docid \in B] \tag{3}$$

$$IMI(A, B) = [e.docid \in A \wedge e.docid \in B] \tag{4}$$

we use $F(A, B) = [C(e)]$ to denote the ordered list containing postings which satisfy condition C and are present in A or B. Each iterator is implemented in O(a + b) time, where a and b are the number of postings in the left and right posting list. We execute all our queries using these iterators like so:

- "A AND B": IMI(A,B)
- "A OR B": UMI(A,B)
- "A B C .. Z": UMI(... UMI(UMI(A,B),C),Z)
- "#DIST d , A, B": DMI(A,B,d)
- ""A B C D"":
  DMI(
  DMI(
  DMI(A,B,1),
  DMI(B,C,1),
  1),
  DMI(C,D,1),
  2)
- "#STRUCTURE_ELEM A": A[i] where $A[i].pos \in extent(A[i].docid$,STRUCTURE_ELEM)
- "#LINKSTO r, n, A": A[i] where $A[i].docid \in BFS(r, n)$ where BFS is the breadth first search of the document graph starting at r, and limited to n hops.
- "NOT A": A, but scores for documents containing $terms(A)$ are heavily penalised, i.e. we perform a soft NOT query.

Where A..Z are lists of ordered postings retrieved from the index. Since these iterators are lazy (yield elements only when needed), in many cases we avoid a lot of decompression and computation.

3.5 Query Result Scoring

To return the most relevant articles, we score the articles found by the search engine based on a combination of TF-IDF and page rank for normal searches (that is, not relational searches). For relational searches, we score the articles solely based on the page rank.

The TF-IDF is implemented as explained in the lectures. The TF-IDF considers the occurrence of terms in the entire article, not just the title. Therefore, the articles returned first are those with the most occurrences of the query terms weighted by their importance, not those where the title matches the query. For a thorough explanation of the page rank implementation, please see Section 3.8.

The final document score is the weighted sum of the TF-IDF and page rank score (Equation 5).

$$\text{Document score} = 0.9 * \text{score}_{\text{tfidf}} + 0.1 * \text{score}_{\text{page rank}} \tag{5}$$

This weighting of scores reflects the relevance of the article to a query, as well as the overall importance of an article.

For nested queries, the TF-IDF is computed for all subqueries and then compounded. Assume we retrieve document A for the query "hamster AND pouch OR cage". The term "hamster" has a score of 1 for document A, the term "pouch" has a score of 2 in document A, and the term "cage" has a score of 3 in document A. Then the overall TF-IDF score for the document is 6.

To reduce the computational requirements for NOT-queries, we multiply the TF-IDF of the contents of the NOT-query with a large negative number (-1000). Assume we retrieve document A for the query "hamster AND NOT cage" where "hamster" has a TF-IDF of 1 and "cage" has a TF-IDF of 2 in document A. Then the total score of document A is -1999 (i.e. $1 + (-1000 \cdot 2)$). This way, we are not required to find all the documents which do not contain a particular term. Rather, we can rely on the fact that the documents containing the term would never be shown to the user due to the low score. A drawback of this is that sole "NOT" queries are ineffective (we note Google uses a similar tactic).

3.6 Disk Backing

After pre-processing, the full English Wikipedia contains 60+ million unique tokens. Following the implementation of basic all-in-RAM functionality, RAM has been identified as the bottleneck. We decided to expand the number of articles that the system can handle by introducing disk-based caching. Since most tokens are not likely to be searched by the user, we can reduce the memory consumption by keeping only the most often used tokens in memory (as well as their tf and df). We implemented a LFU (Least Frequently Used) Cache which tracks the number of accesses of each token (both at run time and build time). At build time the cache evicts token lists into disk once the specified limit of tokens is present in memory and retains only the most frequently used tokens (we use 1,000,000 cache size and 600,000 persistent cache size). We found that performing IO in large batches performed much better than mixing IO and CPU time.

During run time the behaviour of the cache is altered slightly. Token lists are only evicted once the record limit is exhausted, one by one as needed, to reduce lag. The cache is composed of an array of encoded posting lists together with a hash map backed priority queue and a ternary tree, both of which point to elements in this array.

Inserting elements into the array is done in O(1) time, and changing the priority or evicting a record requires O(log N) time operations where N is the number of records in memory. The array elements are never removed, since they represent entries which can either be in memory or on disk (implemented as an Enum in Rust), whereas the priority queue only contains elements present in memory (which at worst case is the maximum cache size).

Once a token list is evicted it's written at an offset into a single file, either into a smallest fitting "hole" or at the end. Essentially the file becomes a *first-fit* memory allocator on disk. The internal fragmentation does not increase storage requirements by a large amount.

Overall the cache slows down build time, due to a lot of token lists needing to be written to disk often, but this effect is reduced when the majority of popular tokens can be continuously kept in memory. During runtime the overhead of bringing token lists from disk is not too large, since the tokens which are on disk are likely smaller in size due to Zipf's law. We also note that posting lists are kept encoded whenever in memory until the very moment we execute the query at which point they are decoded iteratively (one posting by one) to avoid unnecessary decoding when performing linear merges.

3.7 Index

Overall, the in-memory index size is 15GB by our approximate calculations, excluding what is stored on disk. We also approximate that the RAM usage per 1 million documents is 9.442 GB (see the Appendix: 7.1).

3.7.1 Structure

The index contains the following data:

- dump_id: Keeps track of the current version of the dump id. This is useful when a new dump is released, allowing us to update our index in real-time.
- posting_nodes: Disk backing structure, contains the ternary tree that maps tokens to cache entries of encoded posting lists, along with information like term frequency and document frequency.
- links: Used in relational search, these are the outgoing links from a particular document to other documents.
- incoming_links: Used in relational search, these are the incoming links from different documents towards a specific document.
- last_updated_docs: Maps a document id to when a Wikipedia article was last updated.
- page_rank: The calculated page rank score for each document.
- extent: a mapping from a structure element string as key (eg. citations) to another map, which contains document ids as keys to ranges of positions as values.

3.7.2 Pre-Index

Before we build the index, we first create a pre-index: an intermediary representation used to store an incomplete index. This is required since links in reverse direction can only be calculated once all documents have been added. The pre-index is built by querying the database in batches of 5000 documents, the postings are then extracted and encoded. We are able

to encode postings in this step one by one, since all documents are queried in ascending ID order, and all posting positions are strictly increasing, so there is no need to sort anything, and we are able to use a designated *EncodedPostingNode* structure to support incremental encoding and compression (and iterative decompression during retrieval).

While adding postings, we keep track of the start and end positions of each structural element (categories, main text, citations etc..) and store them in the extent index.

3.7.3 Final Index

Once we have a pre-index, we calculate all reverse links for each document, as well as the page rank. The complete index is swapped with the previous index under a shared reference counted mutex, providing seamless transitioning between Wikipedia dumps.

3.7.4 Encoding

To compress the large amount of postings in memory and disk throughout index-building, we implemented *Delta Encoding* and *V-byte Encoding* algorithms. For each document, and with the positions sorted in the postings, delta encoding reduces memory size by subtracting two consecutive positions for the same document. Hence, we store the difference in positions instead. While applying delta encoding, we also integrate V-byte encoding, which cuts down the number of bytes used to store integers. This results in a representation of an unsigned integer using 7 bits for each byte, where the eighth bit is used to mark continuation of bytes.

We chose to use V-byte encoding as we favoured the decoding speed compared to other methods like Google's Group Varint and because of its higher compression ratio compared to Elias Gamma code, which enables us to achieve real-time results for search [5, p.54]. Overall, with the original Simple Wikipedia[7] subset on which we tested our implementation before deploying on a larger dataset, we have managed to reduce from 383 MB to 70MB in disk with encoding (and deduplication of consecutive document ID's).

3.7.5 Ternary Tree

Ternary trees are a tree data structure type where each node contains three children at most. Embedded with the Disk Mapping structure, they store key-value pairs. The keys are pre-processed tokens. Thus at each node where we store a character, a particular token is represented as the path from the root to the node containing the last character of the word in the tree. The values are mappings to the corresponding *EncodedPostingList*. The primary motivation of including them is to support wild card search (for a single token) and spelling correction as features. In addition, it was chosen over binary trees as they offer faster insertion than a binary tree, proving to be less costly to build while preserving the space efficiency characteristics of binary trees [8]. The ternary trees were supported thanks to the Rust ternary tree library [9].

The algorithm for spelling correction recommends the first token closest to the incorrectly passed one, depending on Hamming distance (minimal number of substitutions to change one token into the other). These suggestions are only formed if the number of postings found for the passed word is less than a certain threshold. The naive approach is achieved with the help of ternary tree, which helps us traverse the nodes to the closest neighbour. This process is tweaked from the library, to return tokens that are as close or identical to the length of a particular incorrect word as the first result. Spelling correction is currently supported for most queries, except for wildcard and relational. The former was not done due to the ambiguity regarding the correct spelling. The latter due to time constraints. The algorithm looks at each individual token in the query to mitigate any changes if need be. The new formed query, if any, is then displayed as a suggested prompt that the user can click on. Currently, these tokens are stored in stemmed form. It is costly to support ternary search tree on non-stemmed tokens as an addition to storing an inverted index for retrieval. Hence, future work could include using ngrams or neural networks trained and evaluated on a corpus of Wikipedia articles, to find the correct word given context.

3.8 Page Rank

The page rank is computed at indexing time and subsequently stored in a hashmap within the index data structure for easy access when scoring query results. The page rank for a page π_v is computed according to Equation 6[3].

$$\pi_v = (1-\epsilon) \sum_{(w,v)\in E} (\frac{\pi_w}{d_w}) + \frac{\epsilon}{N} \tag{6}$$

Where N is the total number of documents, ϵ is the reset probability [3] (in our case this is set to 0.85). E is the set of pages linking to π_v and d_w is the number of outgoing links of page π_w. The starting value of all page ranks is set to $1/N$ [3] and the page rank of all pages is computed iteratively until convergence.

[7] `https://simple.wikipedia.org/wiki/Main_Page`
[8] `https://www.cs.upc.edu/~ps/downloads/tst/tst.html`
[9] `https://docs.rs/ternary-tree/latest/ternary_tree/`

4 Front-End

The front end is one of the most important sections of the project overall. The back-end can be fantastic but if the front-end is poor then the user is going to have a bad time.

The frontend is built in JavaScript (JS) with the React library [10] (specifically with the Next JS framework [11]). This was chosen as it is a very common method of producing a frontend and one with which the frontend team had experience.

The frontend team also made use of the Material-UI component library [12], this is a commonly used library (providing components which you would likely recognise) that allows for the use of pre-built components (buttons, inputs, icons, etc.) to speed up development and to maintain a consistent design language.

The design of the front-end was based around the stereotypical search engine front end. The visual design is based on Wikipedia (and reminiscent of Googles styling), the use of the Wikipedia puzzle piece motif can be seen in the logo and the general clean white aesthetic continues throughout the front-end.

4.1 Pages

This section will give a brief overview of the pages. Of course the live website is the best way to experience these.

The front-end is made up of 4 main pages. With a simple homepage with an animated logo and input with auto-complete.

The main result page has standard controls (sort order, results per page, change page) and displays the results in the standard list order you'd expect. The only difference from a standard search engine is the link to carry out relational queries based on that result.

The relational results page allows the user to make queries and see a graph representation of the relationships between the Wikipedia pages.

Advanced search is like other search engines, allowing a very simple way for the user to interact with the more advanced features like AND, OR, phrase search and structural searches. These advanced queries will work fine if put into the normal search box but average users will find this tricky.

There is additionally the advanced queries page (that is linked from the advanced search page) that describes how users can write their own advanced queries without the need for the the advanced search page. This was put together by the Search team.

4.2 Major Features

Auto-Complete - Like most search engines the front end offers auto-complete. This helps the user to more quickly find their search term and offer additional ideas to them. The auto-complete pulls from a file that contains the titles of all the articles in Simple Wikipedia. This offers 282,000 options (4.8MB) which should hopefully contain the vast majority of queries. It also works for EN Wikipedia on the assumption that Simple is a subset of EN. The use of the EN article titles list is not possible as it is unreasonably large (1.2GB). However out of the 282,000 items in the file there are some that don't make much sense and we couldn't justify sorting them manually, so it is possible that some nonsensical suggestions may appear.

Graph - The graph in the relational results page is based on the react-d3-graph library [13] which does the basic part of showing the graph. The front-end team has added to this the on click (takes you to that page) and on hover (displays abstract) behaviour. Additions to the graph includes nodes size based on PageRank value and colour based on distance from the root node.

Loading Screens - The method by which the website makes its queries and gets results is asynchronous (React's useEffect hook). This should result in the website never hanging but instead during this async load the loading screen is displayed. It consists of an animation of our logo rotating and in regular operation should only be displayed briefly during page changes. If the load fails then the loading screen will stop and the user will be shown an error page.

User Feedback - Results selected by a user are returned to an endpoint on the back-end along with the query the user made.

5 Deployment

The deployment of the project has been an evolution of many different workflows. As the architecture of the system has improved, so have the the deployment steps. The final architecture detailed in Section 1, requires no manual interaction after provisioning the production environment on Google Cloud Platform and running the deployment script.

[10] https://reactjs.org/
[11] https://nextjs.org/
[12] https://mui.com/
[13] https://www.npmjs.com/package/react-d3-graph

5.1 Docker

Docker is a containerisation technology, that allows an application, or an entire application stack to be isolated from other running applications and even the host system.

Docker is the essential technology that is underpinning the deployment of the system. We have taken a service based approach to the system as detailed in Section 1, meaning that each of the services corresponds to a separate *Docker Image*, and *Docker Container*. The inter-container communication is provided by the default *Docker Network* using the internal DNS configuration of the containers.

This allows the code of one application running in one container, to reference another network service provided by a different application in a different container, by simply referencing it by its service name, as defined in the `docker-compose.yml` file.

One of the driving factors for using Docker is the isolation from the context in which it is being executed in. The systems performs the same when being tested locally on our machines, as it does once it has been deployed to the production system discussed in Section 5.4. The isolation also provides enhanced security to the host system, bridges into or out of the sandbox have to be explicitly stated in the `docker-compose.yml` file along with the other definitions of the system.

5.2 CI/CD Pipeline

With any large software system it is essential to leverage tooling and automation to reduce complexity, and the developers cognitive load. OnlyGraphs is supported by a CI/CD Pipeline that when a new *release* it published, will trigger the building of the image using the service specific `Dockerfile`, after completion this is pushed to our image repository on the Docker Hub, so that the image can be utilised on both our production and development systems.

This allows for milestones in the project to be marked with tagged releases of the software, and supporting roll back in the situation where an error has been inadvertently introduced.

5.3 Protocol Buffer & GRPC

As stated in Section 5.1, OnlyGraphs is a system comprised of a number of inter-communicating containers, although Docker solves the issues of these containers addressing each other, it does not solves the issue of what protocol/mechanism should be employed to enable/support the communication.

Protocol buffers can be seen as a language-neutral, platform-neutral, specification of serializable data structures [4]. The inputs to and outputs from each of the services are then encapsulated into Protocol Buffer `Message`s to then be validated and compiled into language specific data structures.

The separation of the specification of the API away from the services themselves introduces a number of benefits, the most notable are mentioned below. In the early states of the development of the system, there were a number of programming languages being used before the final transition to Rust [6]. Protocol Buffers enabled rapid prototyping of the system, in a language each of the team members are most comfortable with, ultimately helping the iterative evolution of the system as a whole. Having the API of each of the services clearly defined in distinct `.proto` files, it is very easy to see the interface each service provides and how they interact when used in conjunction with the Architectural System Diagram seen in Section 1, enabling for more succinct developer conversations regarding the implementation of the system.

Protocol Buffers define the `Message`s that are exchanged between services, *gRPC* (Google Remote Procedure Call) then builds on top of this to construct the attachable interface for the caller and callee and the hooks into the application logic.

gRPC provides a high performance Remote Procedure Call (RPC) framework generating both the client stubs and server implementation via a Rust `trait` implementation. This allows us to write our services as well-tested and designed Rust library crates, then have a smaller wrapped binary crate with the server implementation using the library crate as a dependency.

5.4 Google Cloud Platform: Compute Engine

The final part of the deployment is the provisioning of our cloud infrastructure. OnlyGraphs is powered by *Google Cloud Platfom Compute Engine*, which is essentially a Linux virtual private server hosted by Google that we can access.

Our use of Docker discussed in Section 5.1 dictates that *Docker* and *Docker Compose* must be installed - that is all, no other additional software is required. This process is also automated via a script in our *infrastructure* repository on the OnlyGraphs GitHub page, allowing the machine to be setup the same every time, simplifying the entire deployment workflow.

6 Individual Contributions

To promote collaboration and allow group members to work on areas they were most interested in, we split our group into 3 sub-teams. Each sub-team was responsible for a functional section of the system and are shown in the table below:

Team	**Members**
Project Manager	Daniel
Indexing & Deployment	Kyle*, Daniel
Search	Maks*, Annabel, Wassim
Front End	Martin*, Annabel

Table 1: Sub-teams (note * denotes Team Lead)

6.1 Daniel Kirkman

As *Project Manager* of the group I was responsible for setting milestones, progress tracking and helping to create an environment where all members could have as much input into the project as possible. When required, I also had to make the final decision on any difficult design aspects of our system. During meetings I was responsible for creating an agenda, chairing the meeting and then producing minutes. I was grateful to be able to lead a group of people who were as hard working as our team.

I also worked alongside Kyle in the Indexing team where my role was to design and implement the Ingress Controller Service. Prior to starting this coursework I had no experience with Rust but throughout I have developed my Rust knowledge to a level where I now feel confident to write large applications using it. I also had to learn the Wikitext format used to store Wikipedia articles and design an efficient way to parse an enormous amount of text. I was responsible for analysing how and what information could be retrieved from Wikipedia and worked closely with the Search team to create a specification of how data would be stored in the database.

6.2 Kyle Cotton

Working effectively within a team is essential for the success of any project, to ensure we created a cohesive team structure, we divided the group into teams predicated on our particular specialisms.

Having significant experience with backend web development, and Rust I was elected as the *Tech Lead* for the team, responsible for the overall system Architecture discussed in Section 1, integrating and assisting in the development of the different software artefacts discussed in Sections 2 & 3, and the deployment of the system to our cloud infrastructure discussed in Section 5, in additional to providing Rust support and advise for the rest of the team.

6.3 Martin Lewis

My individual contribution is the front end. I was the lead of the front end team due to having some experience developing for the web and building front ends. I was responsible for all of the coding on the front end. I was also responsible for the production of the logo and all the animations (Logo and Loading Screen) in blender. Basically the buck stops with me on the front end, which is great if you like it but obviously front-end is a rather more subjective area.

I liaised closely with the search team to design the API that the front-end makes queries to and was responsible for triggering a number of changes to that API over the course of the project. Close cooperation was also required to ensure the front-end was kept up to date with the evolving feature set the search team provided.

6.4 Maksymilian Mozolewski

As *Search lead* my responsibilities where:

- Working closely with all members of the search team as well as other teams to ensure smooth inter-module, and inter-system integration
- Writing rust modules for the search team
- Designing REST and GRPC API's exposed by search, in collaboration with other teams
- Splitting the workload and distributing it to search team members

My major contributions were:

- The Query Execution Engine and Pre-processor
- The Hard-Drive-Backed Posting Map and Disk-Memory-Allocator
- The Pre-index and SQL index builder code (the parts which interfaced with SQL)

- Parts of the scheduler

I was also responsible for writing any scaffolding code required for the rest of the team to start individual work in parallel, as well as architecting the overall system and resolving any issues the team members encountered.

In total I wrote around 7K lines of code which made it to the latest release in the search repository (according to *git fame*).

6.5 Wassim Jabrane

As part of the search team, my main contributions were developing parts of the Search API and Index Building. These include creating the basic index structure and functions to populate it, encoding, and finding other ways to reduce our index's size. In addition, to support wild card search and spell correction, I integrated ternary trees with the previously made Hard-Drive-Backed Posting structure. While ensuring the correctness of my algorithms through vigorous tests, I also modularised my code to support compressing other types of data in the future into the Disk-backed index structure, such as extent fields and the documents' last updated dates to support higher amount of Wiki articles.

It was a great pleasure of mine working with a fantastic team towards building a search engine on over a million articles, cooperating with search, frontend, and indexing team to ensure this feat.

6.6 Annabel Jakob

Having had some experience and interest in the visual design of software products, I created mockups showcasing a user-friendly and intuitive design for our product. To create these mockups, i used the Bubble.io [14] online tools. This allowed me to make the mockups interactive and thus mimic a real user interaction. In addition, I provided content for the website, such as the guide for creating advanced queries.

As part of the search team I wrote the query grammar and implemented a query parser corresponding to this grammar. Moreover, I implemented the TFIDF-scoring metric as well as the page rank algorithm. I ensured that my code was thoroughly tested to facilitate easy integration with the team's codebase. At the beginning of this project, we decided on Rust as main programming language. At first, I was intimidated by this choice but eager and happy to learn. I now have a good understanding of Rust and a new-found appreciation for low-level programming languages.

References

[1] Crate nom - Rust Documentation. `https://docs.rs/nom/latest/nom/index.html`. [Online; Accessed 23-March-2022].

[2] Abstract syntax tree — Wikipedia, the free encyclopedia, 2002. [Online; Accessed 23-March-2022].

[3] Goel, Ashish, Goodman, and Rio. Lecture 10: Pagerank, May 2009.

[4] Google. Protocol buffers. [Online; Accessed 2022-03-11].

[5] Lars Martin, S Pedersen, and Anne Cathrine Elster. Postings list compression and decompression on mobile devices. 2013.

[6] Rust Team. Rust language website. [Online; Accessed 2021-09-28].

[14] https://bubble.io/

7 Appendix

7.1 Benchmarks

```
    Dump ID=1
    PostingLists=22385594
    PostingLists(RAM)=688120
    PostingRAMCapacity=1000000
    Docs=1620374
    RAM=15300.357MB
    RAM/Docs=9.442GB/1Million
    {
            postings(in cache):14773.271Mb
            links:450.712Mb
            extent:50.448Mb
            metadata:25.926Mb
    }
})
```

Figure 4: Approximate Index Benchmarks (In memory).

7.2 Relational Search Examples

Figure 5: Relational Search Example. Node sizes correspond to the page rank scores.

7.3 Spell Correction Examples

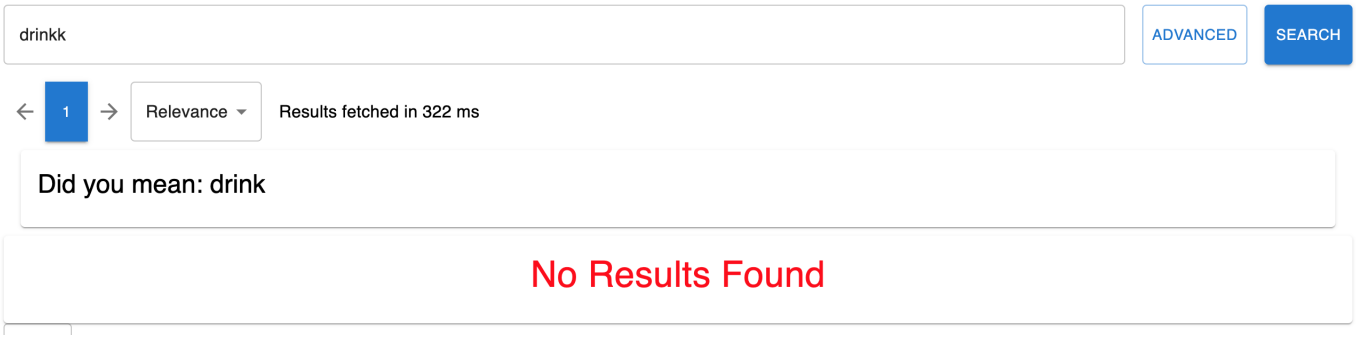

Figure 6: Spelling Correction Example.