SynBioBLAST

Ethan Ransom, *derethanhausen@gmail.com*

***Abstract*—By adding the ability to search the SynBioHub repository by nucleotide sequence, rather than by title, keyword, or summary, users and administrators gain the ability to investigate the uses of particular sequences and see where uploaders utilized already existing components, without the need for uploaders to have designed their SBOL correctly. Scraping SynBioHub to create a list of unique sequences can be done with a paginated SPARQL query. The source code and executables for NCBI's BLAST are freely available, which allows us to build a BLAST database using these sequences. Finally, we can implement a user interface that allows users to submit queries to BLAST against this database and view the results of these queries. A prototype has shown that this approach does not create availability issues for SynBioHub's backing database, or is unfeasible to run on a single host machine.**

I. Introduction

SynBioHub uses the Virtuoso graph database to store application state, namely, uploaded sequences. While Virtuoso provides operations to perform simple string comparisons, these operations are not suitable for searching for and against large strings. In addition, two components claiming to encode the same structure might have small differences consisting of single base pairs, or small prefixes or postfixes. These differences do not change the charatcter of the DNA sequence, but would cause the sequences to be rejected by simpler string comparision algorithms. This task requires the use of alignment algorithms specifically designed to operate on DNA sequences. The de facto standard in this space is NCBI's BLAST.

The BLAST algorithm was published in 1990 as a significant improvement upon existing search alignment algorithms available at the time. BLAST gains its speed by sacrificing a dynamic programming approach for a heuristic-based one. This approach is less accurate but is up to 50 times faster. BLAST's speed vastly improved the user experience of querying extremely large databases, and has been extremely successful–the original paper has been cited 50,000 times, making it the most cited paper of the 1990s.

II. Methods

The SynBioBLAST prototype is built in a layered, microservice-based fashion. Figure 1 shows the architecture of SynBioBLAST. An alternate architecture is discussed in the Future Work section.

A. Scraping Virtuoso

The Virtuoso Universal Server provides a number of functions, in particular, a graph database with a SPARQL interface. SynBioHub exposes this interface at https://synbiohub.org/sparql, allowing for users to run a subset

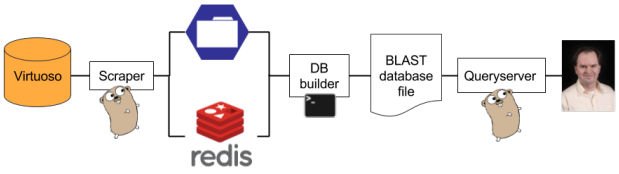

Fig. 1. The architecture of SynBioBLAST.

```
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX sbol: <http://sbols.org/v2#>

SELECT
  ?uri
  ?elements
  ?created
WHERE {
  {
    SELECT
      ?uri
      ?elements
      ?created
    WHERE {
      ?uri a sbol:ComponentDefinition .
      ?uri sbol:sequence ?sequenceUri .
      ?sequenceUri sbol:elements ?elements .
      ?uri dcterms:created ?created .
    } ORDER BY ASC(str(?created))
  }
}
LIMIT {{.Limit}} OFFSET {{.Offset}}
```

Fig. 2. The SPARQL query used to fetch sequences from Virtuoso.

of SPARQL queries. This endpoint can also be configured as an API endpoint, in which case it will return results in a well-known XML format.

1) Flattening the Graph: The query used by the scraper to fetch sequences is shown in figure 2. It retrieves the URIs of all component definitions, and attaches to each URI the elements of the sequence referenced by the component, as well as the creation time of the component.

2) Paginating: Returning in one batch every sequence generated by the query would be a strain on Virtuoso and the server processing the results of the query. The query was converted into a scrollable cursor by sorting sequences based on the sort order of their creation time and wrapping it in a parent query.

Figure 2 shows placeholders for a query limit and offset.

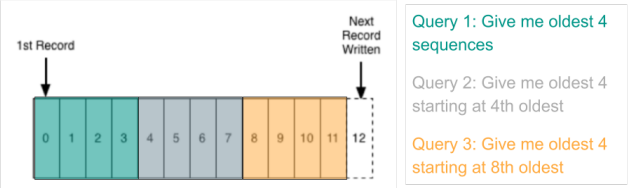

Fig. 3. Sequences are sorted by ascending creation time and fetched in batches.

The limit variable is used to control the batch size of the query. This number should be kept low enough that the load on Virtuoso is minimal. For the first query offset is 0. Additional queries can request a batch of sequences starting after the Nth sequence by passing `OFFSET` as N.

The offset referred to by `OFFSET` is with respect to creation time, ascending. (This is specified by the cursor query.) This sort is stable; sequences that are added while scraping is ongoing will appear at the end of the cursor.

Figure 3 shows three example queries with `LIMIT` set to 4.

3) Deduplication: Because of the nature of flattening a graph into a flat XML file, sequences that are referenced multiple times will appear multiple times in this output. Duplicates need to be detected and handled appropriately. As an added benefit, sequences that were uploaded multiple times through user error will be deduplicated as well.

Deduplication is acheived through the use of the SHA1 hash function. Each sequence is hashed. The hash of the sequence becomes the primary identifier of the sequence from this point until just before BLAST query results are returned to the user. A FASTA file named `<hash>.fasta` containing the sequence, titled with its hash, is written to a configurable directory. For each sequence, the URI associated with it is added to a Redis set keyed using the sequence hash.

In this manner, a sequence that has been seen before will not result in a new FASTA file, but will associate its URI with the existing FASTA file by way of the sequence's Redis set. (Implementations could simply overwrite the FASTA file, as the properities of the hash function virtually guarantee that their contents will be identical. If desired, an implementation could write the file to disk contingent on the file not being present.)

B. Building the BLAST database

NCBI provides a number of programs to create, manipulate, and query BLAST databases. One such program is `makeblastdb`, which assembles an input file of sequences into an indexed BLAST database.

The prototype scraper stores sequences individually, so that new sequences can be scraped incrementally. The builder script uses `cat` to concatenate these files into the `stdin` of `makeblastdb`. The default Linux shell has a maximum argument length of approximately 30,000, which is possibly less than the number of FASTA files. This requires the use of the `find` command is used to enumerate the filenames of the FASTA directory and run the `cat` command for each file.

```
find -name '*.fasta' -exec cat {} \;
  | ./makeblastdb -in -
```

Fig. 4. `find` is used to concatenate the FASTA files and pipe them to `makeblastdb`

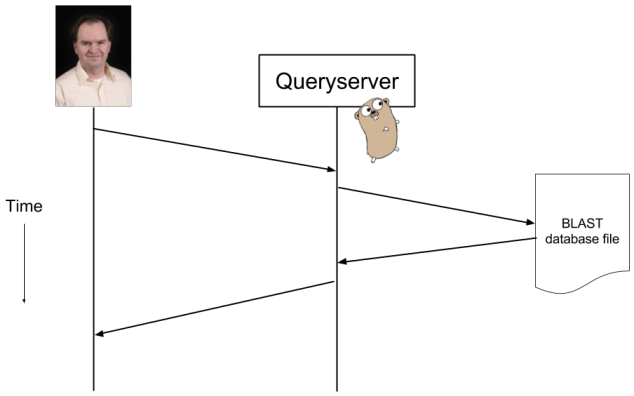

Fig. 5. The query server spawns a child `blastn` process which queries the BLAST database

Figure 4 shows a command similar to the one used to build the database.

C. Running Queries

SynBioBLAST is queried through a webpage user interface, similar to NCBI's reference BLAST implementation. The basic data flow is shown in figure 5. A webserver displays a form to the user. Users input their query sequence and submit the form. NCBI's `blastn` program is used to run the BLAST algorithm on the indexed BLAST database. The output of `blastn` is collected and parsed. The query server then must perform reverse deduplication to convert SHA1 hashes back into lists of uris. A results page is then generated for the user, featuring a link for each matching uri and selected hit information results as reported by the BLAST algorithm.

1) Reverse Deduplication: Sequences are identified in the BLAST database by their SHA1 hash. These hashes are used to assemble a list of keys to retrieve the url sets associated with each sequence. An `SMEMBERS <key>` command is sent to Redis for each key. These commands are pipelined, meaning the latency of deduplication is approximately the round trip time to the Redis server, and is not dependent on the number of results returned by BLAST.

III. Results

A Google Cloud Compute Engine `f1-micro` instance was provisioned to test the prototype. The instance had 8 GB of disk capacity and 592 MB of memory, both of which proved to be more than sufficient.

A. Scraping

Scraping was initially done in smaller batches of 100 or 1000. When Virtuoso handled those queries well, batch size was increased to 5000. Further experimentation could determine the optimal value of this parameter.

At a batch size of 5000, the scraper took roughly 3 minutes to scrape 36,607 sequences from Virtuoso. Of these, 33,325 were unique (3,282 duplicates), resulting in 143 MB of of FASTA files and 20 MB of Redis memory usage.

This resource usage was much less than expected, calling many of the design decisions into question and making new implementations and architectures feasible. See the Future Work section for discussion of these possibilities.

B. Building The Database

As of this writing, building the BLAST database takes 39 seconds of real time and 2 seconds of user CPU time. Clearly, the bottleneck is retrieving the FASTA files from disk. The 37 seconds of disk seek overhead could be virtually eliminated if the slurper appended new sequences to a single FASTA file.

C. Querying

Running a 720 base pair query takes 218 ms. Fetching deduplication information from Redis takes 13 ms, the remainder of the time is spent waiting for the BLAST algorithm. A 5500 base pair query takes 450 ms, with 10 ms taken communicating with Redis. The time spent fetching deduplication information from Redis is considered minimal compared to the time of the query.

D. Implementation

The scraper and query server were both written in Go. Go is a statically-typed complied language designed for web services and distributed systems. The current scraper implementation is 274 lines, while the query implementation is 194. The database builder is a simple shell script, about 15 lines.

IV. Future Work

A. Improvements on the Existing Implementation

- Performance. As discussed above, modifing the scraping code to append new sequences to a large file would speed up the building of the BLAST database by 10x.
- User experience. Figure 6 shows how BLAST displays hit regions for the results of a query. SynBioBLAST's results page is rudimentary at best.

B. Alternative Architectures

Scraping Virtuoso is fast enough that it could be done from scratch each time. This unlocks a number of alternative design paths.

- Batch-based architecture. The deduplication data structures are small enough that they could be built in-memory. (On the order of tens of kilobytes.) The dependency on Redis could be removed, which would simplify deployment and system administration.

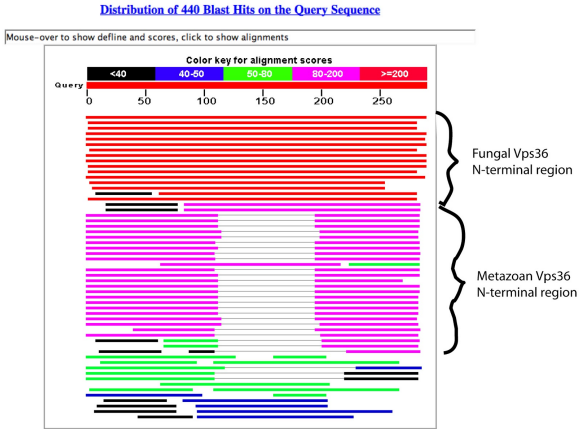

Fig. 6. The official BLAST implementation has graphical hit region displays.

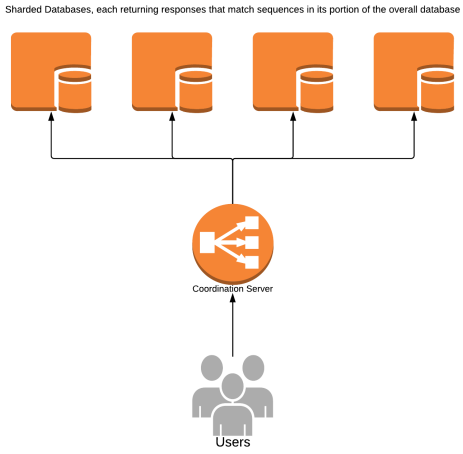

Fig. 7. A sharded architecture, where each server is responsible for a subset of the sequences.

- ComponentDefinition-centric query. If the deduplication of reuploaded sequences is not desired, a different query could be chosen that preserves the many-to-one relationship between Sequences and ComponentDefinitions. This would eliminate the need for deduplication data structures entirely.
- Sharded architecture. The Scraping process could split sequences into any number of sub databases. Queries would be run on all databases in parallel, and the results combined and resorted. The NCBI official BLAST implementation uses this because the databases used are too large to fit on a single machine. This approach could also speed up queries. SynBioHub does not currently contain enough sequences for this to be helpful.

C. Rewriting in JavaScript

The main SynBioHub application is written in JavaScript. Is it possible to integrate the Golang code by running the Scraper and Query Server as child processes, maintainence could be simplified by rewriting in JavaScript. As both components are of low complexity and don't utilize any Golang-specific features, the rewrite would be a fairly direct translation.

V. Conclusion

Due to the generosity of the BLAST authors and the capabilities of Virtuoso, adding heristic-based nucleotide search to applications is straightforward and relatively simple. This functionality allows for easy answers to a variety of research questions and opens up new ways for SynBioHub administrators to curate and index their collections.

Acknowledgment

Many many many thanks to Zach Zundel for providing the SPARQL query, guiding the author through the SynBioHub codebase, and staying up late to shoot down terrible implementation ideas.

References

[1] Altschul, S.F., Gish, W., Miller, W., Myers, E.W. & Lipman, D.J. (1990) *Basic local alignment search tool.* J. Mol. Biol. 215:403-410.
[2] SynBioHub. https://synbiohub.org
[3] Redis. https://redis.io