Technische Universität München

Ludwig-Maximillians-Univesität München

Faculty of Informatics

Bachelor's Thesis in Bioinformatics

PubSeq: Amino Acid-based Search Engine for MEDLINE Abstracts

PubSeq: Aminosäuresequenz basierte Suchmachine für MEDLINE Abstrakten

---

Supervisor:
Prof. Dr. Burkhard Rost
Advisors:
Dr. Guy Yachdav
Juan Miguel Cajuela

Author:
Pandu Raharja

August 2015

Declaration of Authorship

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Signed: ______________________________

Date: ______________________________

“People usually think that progress consists in the increase of knowledge, in the improvement of life, but that isn’t so. Progress consists only in the greater clarification of answers to the basic questions of life. The truth is always accessible to a man. It can’t be otherwise, because a man’s soul is a divine spark, the truth itself. It’s only a matter of removing from this divine spark (the truth) everything that obscures it. Progress consists, not in the increase of truth, but in freeing it from its wrappings. The truth is obtained like gold, not by letting it grow bigger, but by washing off from it everything that isn’t gold.”

L. N. Tolstoy

Abstract

Background

In genetic research, it is imperative for biomedical researcher to stay updated on the current state of identified proteins. It was hard – and is getting harder, especially after widespread use of Next-Generation Sequencing (NGS) – for researcher to keep updated on the research into protein he/she is currently investigating. This is furthermore exacerbated by the fact that existing search engines only allow querying abstracts using protein names.

Methods

In this project, I present the first search engine that allows user to find all publications mentioning proteins that are similar or identical to the one he/she's interested in. To achieve this, I created a Solr Index that lists down all gene names that were mentioned in each of MEDLINE abstracts and titles. I then populated the index by scanning the whole MEDLINE corpus, tagging protein names found in title and abstract, normalizing those names into UniProt IDs and pushing the ID mentions onto Solr index. Given user's sequence query, the program runs a BLAST on the sequence and normalizes blast results to UniProt IDs. The program then retrieves articles mentioning this ID and return these to user. For the good usability I offer the whole service in a web interface available in following address.

Zusammenfassung

Hintergrund

In genetischer Forschung ist es erzwingend, dass der/die biomedische ForscherIn mit der aktuellen Landschaft von identifizierten Protein sich ständig informiert. Es war schwierig – und wird immer schwieriger sein, vor allem nach dem verbreiteten Ansatz von Next Generation Sequencing (NGS) Technoligien, um der/die Forscherin mit dem Protein von der Interesse in aktuellem Zustand zu halten. Die Tatsache, dass die aktuelle Suchmaschine von den Artikeln nur Namenbasierte Suche unterstützt, hilft leider nicht weiter.

Methoden

In diesem Projekt stellen wir eine Suchmaschine vor, die erlaubt den Benützer, basiert auf Aminosäuresequenz nach den Artikeln suchen, die das Protein oder die Ähnliche erwähnen. Um dies zu erreichen hatten wir einen Solr Index erstellt, der alle erwähnte Proteine innerhalb jedes MEDLINE Artikels auflistet. Wir füllen sich diesen Index in dem wir den gesamten MEDLINE Corpus durchscannen und alle Proteinname mithilfe eines NLP-Programs detektieren. Wir wurden dann diese Namen in UniProt IDs normalisieren. Diese normalisierte Namen wurden schlielich in unserem Solr Index hinzufügen. Um die Benutzbarkeit dieser Dienstleistung zu maximieren hatten wir auch eine Webschnittstelle entwurfen, die in folgender Addresse verfügbar ist.

Acknowledgements

First and mostly, I would like to thank Prof. Burkhard Rost for the holistic supports provided, be it through the lab infrastructures or himself personally. I would also to thank two of my advisors, Dr. Guy Yachdav and Juan Miguel Cajuela, who have in spite of their busy schedules and great distances (and time differences) patiently advised me through the project. Knowing that both are about to finish their PhD programs, I wish them all the best of luck in their future endeavors. I would also like to thank Andre Ofner for helping with the evaluation of the systems.

Also, I would like to thank Tatyana Goldberg for administrative and all-around support during my stay at the lab. Also my gratitude for Tim Karl, our awesome system administrator, who has helped us tremendously in incorporating each of the cogs in our pipeline into one coherent system. While not involved in our project personally, I would like to thank Prof. Lars Juhl Jensen of University of Copenhagen for giving us access to his tagger program.

I would also to thank Andre Ofner for the help in validating the systems. I would also like to express my gratitude towards Robert Leaman and Zhiyong Lu from National Institue of Health. While we ended up using different implementation of normalizer in our system, their contributions during our earlier attempts in the project are not to understate.

Research wouldn't happen without grants and patrons. Therefore I would like to thank grant organizations that have contributed financial supports to the lab and its extension. I am full aware that without sufficient infrastructure and human capital support endowed by several grants, this project would be impossible to kick start and finish.

Finally I would also thank all Rostlab members and its extensions, without whom this work would all but possible.

Contents

Declaration of Authorship ii

Abstract iv

Zusammenfassung v

Acknowledgements vi

Contents vii

List of Figures xi

List of Tables xiii

Abbreviations xv

1 Introduction 1
- 1.1 An Easier Biomedical Research 1
- 1.2 Overview of This Thesis 2

2 Background 3
- 2.1 FASTA and BLAST 3
- 2.2 UniProt 5
- 2.3 MEDLINE 6
- 2.4 Natural Language Processing 7
  - 2.4.1 Named Entity Recognition 7
- 2.5 Previous Works 10

3 Organizations and Components 13
- 3.1 Introduction 13
- 3.2 PubSeq Persistent Components 14
- 3.3 Web Interaction 15
- 3.4 Tagging Pipeline 16

4 PubSeq Tagging Pipeline **19**
- 4.1 Introduction . . . 19
- 4.2 MEDLINE Abstracts . . . 20
  - 4.2.1 Specifications . . . 22
- 4.3 XMLAbstractFormatter.java . . . 23
  - 4.3.1 Parameters . . . 23
  - 4.3.2 Specifications . . . 24
- 4.4 tagcorpus.cxx . . . 24
  - 4.4.1 MEDLINE Input Table . . . 25
  - 4.4.2 entities.tsv . . . 26
  - 4.4.3 names.tsv . . . 26
  - 4.4.4 STRING Tagger Annotation . . . 27
- 4.5 AnnotationBackmapper.java . . . 27
  - 4.5.1 Parameters . . . 28
  - 4.5.2 Specifications . . . 28
- 4.6 Annotater.java . . . 28
  - 4.6.1 Parameters . . . 30
  - 4.6.2 Specifications . . . 30
- 4.7 StatisticsUtils.java . . . 31
  - 4.7.1 Specifications . . . 31
- 4.8 IndexerNew.java . . . 31
  - 4.8.1 Parameters . . . 33
  - 4.8.2 Specifications . . . 33
- 4.9 IndexerNew.java . . . 34
  - 4.9.1 Parameters . . . 34
  - 4.9.2 Specifications . . . 34

5 PubSeq Solr Index **35**
- 5.1 Solr: Fast and Scalable Indexing . . . 35
- 5.2 Index Definitions . . . 39
- 5.3 Information Retrieval . . . 40

6 PubSeq Web Service **43**
- 6.1 PubSeq Virtual Machine . . . 43
  - 6.1.1 VM Specifications . . . 44
- 6.2 PubSeq Web Server . . . 45
  - 6.2.1 Components . . . 45
  - 6.2.2 index.html . . . 47
- 6.3 Usage . . . 47

7 Results and Validations **51**

8 Conclusion and Outlook **53**

A PubSeq Paths and Source URLs **55**
- A.1 Source Codes 55
  - A.1.1 Solr's `schema.xml` 55
  - A.1.2 Node.js's `app.js` 55
- A.2 Project Location 56

B Figures and Tables **59**

Bibliography **61**

List of Figures

2.1 Overview of DNorm pipeline . . . 9
2.2 Schematic interaction between BLAST, UniProt, MEDLINE and NER Tagger . . . 11

3.1 An Overview of how the 'Persistent Components' of PubSeq environment interacts. . . . 14
3.2 A Sequence Diagram modeling of PubSeq web interaction. . . . 16
3.3 Schematic representation of PubSeq Tagging Pipeline. . . . 17

4.1 (Resized) Overview of PubSeq Tagging Pipeline with all essential programs showed as nodes and important input/output files shown . . . 21

5.1 The key data structure supporting information retrieval is the inverted index within Solr system. . . . 37
5.2 Diagram of the main components of Solr. . . . 38

6.1 Schematic representation of PubSeq systems with the Virtual Machine environment shown. . . . 44
6.2 Schema of Node.js's Processing Model. . . . 46
6.3 PubSeq Web Interface. . . . 49
6.4 Sample sequence button in PubSeq web interface. . . . 49
6.5 Blocked screen in PubSeq web interface. . . . 49
6.6 The first result of PubSeq query. . . . 50
6.7 PubSeq web query state diagram. . . . 50

B.1 Overview of PubSeq Tagging Pipeline with all essential programs showed as nodes and important input/output files shown . . . 60

List of Tables

4.1 Specifications table for PubSeq MEDLINE Abstracts 22
4.2 Specifications table for XMLAbstractsFormatter.java 24
4.3 Specifications table for AnnotationBackmapper.java 29
4.4 Specifications table for Annotater.java 30
4.5 Specifications table for StatisticsUtils.java 31
4.6 Specifications table for IndexerNew.java 33
4.7 Specifications table for SolrUploader.java 34

5.1 ACID vs. BASE database property models. ACID vs. BASE database property models. Some NoSQL databases implement ACID, others do not. Taken from (Wachinger, 2013) [1], adopted from (Brewer, 2000) [2]. 36
5.2 Fields definition table for `pubseq` index within the Solr system. 39

Abbreviations

BLAST	**B**asic **A**lignment **S**earch **T**ools
FTP	**F**ile **T**ransfer **P**rotocol
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
LAB	**L**ocal **A**rea **N**etwork
MEDLINE	**Med**ical **Li**terature Analysis Retrieval Onli**ne**
MeSH	**Me**dical **S**ubject **H**eading
NER	**N**amed **E**ntity **R**ecognition
NLM	U.S. **N**ational **L**ibrary of **M**edicine
NLP	**N**atural **L**anguage **P**rocessng
OLN	**O**rdered **L**ocus **N**ame
VM	**V**irtual **M**achine
UniProt	**Uni**versal **Prot**ein Resources

Chapter 1

Introduction

1.1 An Easier Biomedical Research

We'll try to present the main idea of this project in following story. Imagine you in the position as a biomedical researcher, are currently investigating some unknown enzymes that somehow were over-expressed in a patient with medical conditions. Upon some more investigating, you managed to get the sequence of several proteins. Without prior knowledge of the proteins, you would naturally BLAST the sequences and wait a little while while the BLAST is searching the sequence against your local database or some online service. Upon the results were coming, you would naturally want to check the resulting proteins one by one, at least the best matching ones. For each protein, you would want to search for articles that have dealt with this protein before.

Imagine that, instead of having to go through blasting the sequence manually and searching for articles one by one, you could just put in a sequence in a website, wait for a while and get the site returns a list of articles that mention the proteins with similar or exact sequence to the one you have. Not only you would save time and resource during the parts that were handled by website itself, you as a researcher could focus more on the substantial part of the research – that is, finding as much essential information about the unknown protein in as little overhead as possible. Therefore, we created a web service that realizes this. In the service, user would only have to put in the sequence of unknown protein, press the search button and receive at the end a list of articles that mention proteins with identical or similar sequence to queried proteins.

With this small contribution, we hope not only to bridge the gap between sequence and knowledge discovery in biomedical research, but also give researcher more flexibility and insights in their literature research. With also ongoing feature extensions and updates, we would also hope that the service would serve more researchers with more conveniences both in medium and long run.

1.2 Overview of This Thesis

In this thesis, we will describe how we came with the idea of creating PubSeq, how we did that and what we planned in the future regarding our implementation.

In **Chapter 2**, we will discuss how bridging the knowledge gap has been attempted in the past and how our contribution would fit in the bigger picture. We also discuss some of the methods that are relevant in our project. Also, we would look into how our project builds upon existing knowledge and technology.

Chapter 3 introduces the system as a whole. How we organize the sub-components together. We would also skim through the technological side of the projects here, while keeping the reader aware of the bigger picture. We would discuss our rationale behind selecting some of technology stacks that we used. All the while,we would also show some the visual examples from our component here. By the end of the chapter it is hoped that the reader would understand how each single component interacts with others within our system.

Chapter 4 explains in detail our Tagging Pipeline. Here the reader would see how the MEDLINE corpus would be processed, annotated and pushed onto Solr Index.

Chapter 5 explains our storage technology, the Solr index. Besides the technology itself, we would explain how we configure and structure our index that would fit the resulting data from Tagging Pipeline.

Chapter 6 investigates our web server component. We would first present the technological stacks that were used in this component and other specifications. We would then present the program from the perspective of end user. We would show how convenient would that be for a researcher to use our application, which would make our case for value proposition of PubSeq search engine.

Chapter 7 covers quantitative measurement of the quality of our website. We would focus mostly on how our system performs, especially with regards to the sensitivity and specificity. We would focus mostly on the the quality of protein tagging within our data (see Chapter 5 for detail). We would also muse on how our system would have an edge over similar UniProt ID-based abstract search service provided by UniProt [3] [4] [5].

Chapter 8 covers our conclusion of the system so far. There we presented our own ideas on how the website could and would be improved. we would again reiterate the the merits of using the PubSeq as the search engine for abstracts based on protein sequence.

Chapter 2

Background

This chapter introduces the concepts and techniques that are relevant throughout this thesis. First, the concept of similarity search, especially the two software suite FASTA and BLAST would open our chapter. And then, we would introduce various contemporary concepts in bioinformatics and bioinformatics-related infrastructure such as UniProt and MEDLINE. Additionally, we would introduce the concept of named entity recognition (NER) within the field of Natural Language Processing and how it would be relevant for us. Finally we would see how our project relates to previous works in similar topics and how it would improve, provide alternative or give additional insight to them.

2.1 FASTA and BLAST

As the title of this thesis already conveyed, the main idea of this project is to bridge the accessibility and knowledge gap between sequence and the main source of knowledge and reference of previous discoveries – a vast corpora of publications in natural sciences – through a modern search engine. Given a sequence of amino acids, it would be impossible for a human to directly identify directly the protein, let alone the characteristics and the functions and the characteristics of the protein.

Several attempts on bridging one component of the gap, specifically between sequence and other known sequences, was done in eighties and earlier nineties. In 1981, Smith and Walterman published the algorithm computing complete local sequence alignment, which was further improved by Gotoh in 1982 [6] and Altschul (Altschul and Erickson, 1986 [7]). This was however deemed too slow, especially if used for the purpose of one-against-all search, which was heavily (and still is) used for sequence-based knowledge discovery in biomedical research.

In 1985, Lipman and Pearson published the first paper mentioning the DNA and protein sequence alignment program FASTA [8]. During the first publication, FASTA was designed and intended to search for similar protein sequences. It takes a sequence of amino acids and searches against entries within a corresponding database by using local sequence alignment to find similar sequences. In general, FASTA takes four steps in computing three scores that characterize sequence similarity [9]:

1. Finding identify regions with high density of sequence identities and pair identities between two sequences. FASTA achieved a fast computation in this step by using a look up table, a map that describes for each character where it appears within sequence. In conjunction with the lookup table, FASTA also uses the diagonal method to find all regions of similarity between the two sequences, counting matches and penalizing for intervening mismatches. This diagonal could be visually seen in two sequence alignment as series of matches ('dots') in match matrix between two sequences.

2. Rescanning of the 10 regions with highest sequence identities using PAM250 matrix. PAM250 matrix refers to assumed point accepted mutation (PAM) matrix after 250 mutations, which is basically the 250-th power of initial PAM matrix. The probability of each entry within PAM matrix was acquired from analysis of phylogenetic trees (Dayhoff, 1978 [10]).

3. Annealing of both ends of alignment and calculating similarity score is the sum of the joined initial regions minus a penalty (usually 20) for each gap [9].

4. Construction of optimal alignment using Needleman-Wunsch Algorthm [11] on the best matching region. The program would then return the similarity score of this alignment along with the best score from step 2 and 3.

In 1988, Pearson and Lipman improved the software by adding support and improvement, among others, for nucleic acid similarity search, translated nucleic acid search [12]. This allowed researchers to do trans-domain search between nucleic and amino acids.

Further down the road, in 1990, Altschul et al. published the Basic Alignment Research Too [13], better known in its acronym as BLAST. The algorithm, like FASTA, is based on heuristics search and is structured in similar manner to BLAST. BLAST takes a sequence to search for and a sequence or a set of sequences to search against. In modern usage, the set of sequences is provided by some database. The algorithm would then run in following main steps [14]:

1. Removal of low complexity regions or sequence repeats from query sequence. Low complexity refers to sequence with few elements.

2. Creation of k-gram sequences from query sequence.

3. For each word from step 2, listing of possible matching words and selection of high scoring words. Matching words are the all possible combinations of words with same length as the k-gram word. For each possible word a score is calculated, which is based on substitution matrix. The best scoring words are then passed onto next step. This differs from FASTA, which focuses more on common words in database.

4. Organization of remaining high scoring words into efficient search three. Both step 3 and 4 would be repeated for each word from step 2.
5. Scanning of database for exact matches with remaining high-scoring words.
6. Extension of database match to high-scoring-segment pair (HSP). This is done by annealing both ends of match until the matching score begins to decrease.
7. Listing of all HSPs that are significant enough.
8. Evaluation of statistical significance of the HSPs. BLAST models statistical significance using Gumbel extreme value distribution [15], in which the probability of observing score S higher than equal to x is defined as

   $$P(S \leq x) = 1 - exp(-e^{-\lambda(x-\mu)})$$

   with

   $$\mu = log(Km'n')/t$$

   The parameters μ and K are fitted from the distribution of results from high scoring pairs. m' and n' are effective length of the query and database sequences.
9. Make two or more HSP regions into one alignment. In a given hit sequence from database, the algorithm would attempt merging the regions into one had the score of combined region is larger than individual score.
10. Computation of sequence alignments using Smith-Walterman Algorithm [16].

Nowadays, both FASTA and BLAST were distributed not only locally but also online by various providers such as National Center of Biotechnology Information (NCBI) [1] and European Bioinformatics Institute (EBI) [2] [3].

2.2 UniProt

UniProt (*Universal Protein Resource* [3]) is platform containing various high quality databases that are essential in bioinformatics research. A joint venture between European Bioinformatics Institute (EBI), Swiss Bioinformatics Institute (SBI) and Protein Information Resources (PIR), it provide four main sets of database:

- **UniProt KnowledgeBase (UniProtKB)**, containing protein database. Our database of interest in this system, there are two main constituents of UniProtKB: the manually annotated, reviewed UniProtKB/SwissProt and automatically entried, unreviewed UniProtKB/TrEMBL. Currently, there are 549,008 reviewed and 50,011,027 unreviewed protein sequences within the UniProtkB environment[4]. We consider the current ever widening over-representation of unreviewed proteins within the KnowledgeBase be a potential challange in our project going forward.

---

[1] `http://blast.ncbi.nlm.nih.gov/Blast.cgi`, accessed 8/19/2015

[2] `http://www.ebi.ac.uk/Tools/sss/wublast/`, accessed 8/19/2015

[3] `http://www.ebi.ac.uk/Tools/sss/fasta/`, accessed 8/19/2015

- **UniProt Archieve (UniParc)**, a comprehensive and non-redundant database containing all protein sequences from publicly available protein sequence database[17]. It achieved redundancy by searching for each sequence only once. Multiple matching sequences from various databases would then be merged into one UniParc entry.
- **UniProt Reference Cluster (UniRef)**, a set of three databases of clustered sets of proteins. The databases house clusters sets of proteins with 50% (UniRef50), 80% (UniRef80) and 90% (UniRef90) sequence similarity [18].
- **UniProt Metagenomics and Environmental Data (UniMES)**, a repository for metagenomics and metagenomics sequences. UniMES was initially developed to address issues arising from sequences that are obtained from non-cultured or currently unknown organisms, which are sometimes used in metagenomics studies [3]. The data from UniMES is included in UniParc but not UniProtKB.

Each entry within UniProt main databases possesses a unique identifier. For UniProtKB, every entry within the database is associated with both UniProtKB accession number (UniProtKB-AC) and UniProtKB identifier (UniProtKB-ID). In this project, we use not only UniProtKB-ID as the identifier of choice when it comes to UniProt, but also as our main project identifier protein names. That is, a protein is always identified with UniProtKB-ID within our search engine, each annotation of mentions in text would be represented as UniPritKB-ID and our BLAST search will also return UniProtKB-ID. Consequently, this limits our search and indexing space within the scope that is already defined by UniProtKB. However, considering proteins that are of interest to the scientist are almost always already indexed by UniProtKB[17], we don't really see this issue as potential problem.

2.3 MEDLINE

MEDLINE contains journal citations and abstracts for biomedical literature around the world [19]. It is maintained by U.S. National Library of Medicine and currently houses over 24 millions of journal abstracts and the numbers keep growing along the new publications, which currently clocks at 7 per cent growth per year, which corresponds to over one million new articles in the last years [20]. In MEDLINE environment alone, it adds between 2,000 - 4,000 references each day [19]. Last year alone, it added 750,000 new references into its repository [19]. The subject of journals that are indexed by MEDLINE ranges from life sciences, behavioral sciences, chemical science and bioengineering. While the overwhelming majority of the references indexed are journal articles, there are several newspaper and magazine entries that are deemed useful as reference in biomedical research. Ninety five percent of the articles are written in English and 84% have English abstracts written by the authors of the articles.

---

[4] `http://www.uniprot.org/`, accessed 8/19/2015

For most part, a researcher could access MEDLINE through PubMed [5] [21]. This already covers about about 90% of its indexed abstracts. For complete access, it offers leasing mechanism. It distributes daily the new references in XML format via FTP protocol. We use this leasing mechanism in our project for intial references collection and daily updates.

2.4 Natural Language Processing

The rapid development in sequence similarity search coupled with explosion of genome-wide sequencing, which was even more augmented by the advent of post-Sanger and – recently – New Generation Sequencing (NGS), means that the problem of identifying sequence is more or less explained. There is however one part that is missing from our picture: how to get the information on how the sequence was mentioned in previous publications?

Come Natural Language Processing (NLP). Natural Language Processing is a interdisciplinary field that deals with the interaction between computer and human languages (hence the natural language). The aspects of natural languages such as named entities recognition (NER) [22], morphological segmentation [23], speech recognition and analysis [24] fall into the auspice of natural language processing. The methods used in natural language processing is mostly statistical-based [25] and some of the methods have been known to be used in other fields such as Conditional Random Fields [26] and its special case Hidden Markov Chain, which is one of the more commonly used methods in bioinformatics (e.g. Salzberg, et al. [27] and Burge, et al. [28]).

In this thesis we would make use of one aspect of natural language processing: named entity recognition (NER).

2.4.1 Named Entity Recognition

The term named entity recognition was first coined at the Sixth Message Understanding Conference (MUC-6) in 1996 [22] [29]. The problem statement of named entity recognition roughly goes as follow:

Given an input text, locate and classify elements of text according to defined categories of entity (a Named Entity)

The pre-defined set of categories range from unique identifier such as name and location to expression of times and numeric values such as date and percent expression [22]. To take an example, consider the first paragraph of following recent article from the Wall Street Journal [6]:

---

[5] `http://www.ncbi.nlm.nih.gov/pubmed`, accessed 19/08/2015

> Uber Valued at More Than $50 Billion
>
> **Ride-sharing app, which just closed a funding round, reaches mark faster than Facebook**
>
> Uber Technologies Inc. has completed a new round of funding that values the five-year-old ride-hailing company at close to $51 billion, according to people familiar with the matter, equaling Facebook Inc.'s record for a private, venture-backed startup.

A named entity recognition program trained for company names and monetary values would identify and tag following annotations from the text:

> **Uber** `(company)` Valued at More Than **$50 Billion** `(monetary_value)`
>
> **Ride-sharing app, which just closed a funding round, reaches mark faster than Facebook** `(company)`
>
> **Uber Technologies Inc.** has completed a new round of funding that values the five-year-old ride-hailing company at close to **$51 billion** `(monetary_value)`, according to people familiar with the matter, equaling **Facebook Inc.** `(company)`'s record for a private, venture-backed startup.

Here we see how various formats of company names could be tagged in this hypothetical case. Indeed, a good named-entity recognition tagger should be able to perform exactly this kind of task.

There are various ways of implementing Named Entity Recognition (NER), all ranging from supervised, semi-supervised to unsupervised learning [22]. Nowadays, most NER relies on supervised learning methods [22] such as Hidden Markov Model (HMM) [30], Decission Trees [31], Maximum Entropy Model [32], Support Vector Machines (SVM) [33] and Conditional Random Fields (CRF)[34], which is a discriminative generalization of HMM[35].

In this project, we attempted on two distinct NER implementations: **DNorm** in the earlier phase and non-publicly available NER tagger given to us by Lars Juhl Jensen at University of Copenhagen (which would for our convenience would be called **STRING Tagger** or **(PubSeq) NER Tagger** from now on) later on. DNorm (Leaman and Lu, 2013 [36]) is a NER designed to tag and normalize disease names within PubMed abstract. Given a PubMed abstract text, it tags the disease entities within text and normalize said entities into standardized form (in this case, MEDIC concepts [37]). The tagging component of DNorm is based on BANNER, which in turn is a CRF-based

---

[6] `http://www.wsj.com/articles/uber-valued-at-more-than-50-billion-1438367457`, accessed 8/19/2015

NER tagger [38]. To normalize the entities, the program utilizes pairwise approach of Learning to Rank [39] (see, for example, Joachims 2002 [40], which is used to rank the results of web search). The idea of ranking as a method of normalization could indeed be explained analogously in light of search engine itself, given a tagged string s, a set of standardized entity names N and a scoring function $M(s, n) \in R$, we would like to find out to which standardized name this string maps best to.

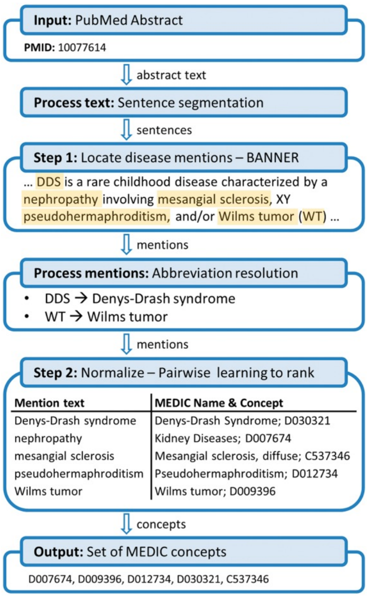

Input: PubMed Abstract
PMID: 10077614
abstract text
Process text: Sentence segmentation
sentences
Step 1: Locate disease mentions – BANNER
… DDS is a rare childhood disease characterized by a nephropathy involving mesangial sclerosis, XY pseudohermaphroditism, and/or Wilms tumor (WT) …
mentions
Process mentions: Abbreviation resolution
• DDS → Denys-Drash syndrome
• WT → Wilms tumor
mentions
Step 2: Normalize – Pairwise learning to rank
Mention text
MEDIC Name & Concept
Denys-Drash syndrome
Denys-Drash Syndrome; D030321
nephropathy
Kidney Diseases; D007674
mesangial sclerosis
Mesangial sclerosis, diffuse; C537346
pseudohermaphroditism
Pseudohermaphroditism; D012734
Wilms tumor
Wilms tumor; D009396
concepts
Output: Set of MEDIC concepts
D007674, D009396, D012734, D030321, C537346

Figure 2.1: Overview of DNorm pipeline. The figure was taken from Leaman, Dogan and Lu, 2013 [36].

The reason why we initially resorted to DNorm is because of its good performance with regard to precision, recall and F-measure [36]. Also, the underlying tagging mechanism BANNER appeared to perform well in BioCreative benchmarks for tagging various biological entities, most notably gene names [41] [38]. Trying to extend DNorm to support gene names recognition *and normalization*, however, proved to be very difficult. We figured out that there was no publicly available compatible data set that we could use to train our model for gene normalization. Secondly, since the DNorm uses matrix to represents both the normalization and similarity model [36], creating matrix models for the whole set of proteins would have the program create a matrix with millions of row – which wouldn't scale in normal cluster environment [7]. Also, the fact that DNorm was optimized for disease names means that there is no guarantee that the resulting benchmark for protein names would be as good as disease tagging and normalization benchmark.

Later on, we switched to STRING Tagger. Unlike NER methods that are already mentioned in this chapter, our current NER Tagger relies on mapping between normalized entity names and their corresponding alternative names to annotate and normalize named entities within in the text. It checks whether a part of text closely resembles one or more elements in the mapping. It would then calculate the probability of the part of text being the instance of one of the entities (see more on Chapter 4). While there is no publication specifically dedicated for gene annotations, the underlying engine was used to tag and annotate taxonomic names in PubMed text (Pacifilis, et al., 2013 [42]). Moreover, we used part of STRING database (Szklarczyk, et al., 2011 [43]) as the dictionary between normalized entity names and its variation of names that would be used to identify named entities in text and at the same time normalize them.

---

[7] The largest machine in our cluster provides 64 GB of memory in normal situation. A transition matrix with 100,000 starting and 100,000 target states with each transition represented as 64 bytes double precision entry will take *at least* 8 * 100000 * 100000 ≈ 80 GB memory to initiate, *conservatively*. We know that the number of the reviewed proteins alone is already beyond that at the time of writing of this thesis.

2.5 Previous Works

There are several publications and tools that attempt on similar goals as we do that have not been reviewed in previous section. While we are not aware of other project that realized end-to-end article search based on sequence, there are several (very successful) attempts on doing parts of it.

In the realm of **gene/protein names tagging and/or normalization**, there are various programs that are known to tag and normalize well such as GNAT (Hackenberg, et al. 2011 [44]), ABNER (Burr, 2015 [45]) and LINNEAUS (Gerner, et al. 2010 [46]). The latter was originally designed to detect and annotate taxonomy names within text but is now "intended for general-purpose dictionary matching software"[8].

For searching for protein mentions within PubMed corpus we are for example aware of. Given a UniProtKB-ID, a user can search for the list of articles mentioning the protein. User could, for example, search articles that mention human tumor protein P53 (`P53_HUMAN`) [9]. We are unfortunately not aware how it was done (manually annotated or done using NLP methods) or whether it covers only parts of article that are visible within PubMed corpus or the whole article nor there exists any formal documentation/publication regarding the system.

---

[8] `http://linnaeus.sourceforge.net/`, accessed 8/19/2015

[9] `http://www.uniprot.org/citationmapping/?query=uniprot:P04637`, accessed 24/08/2015

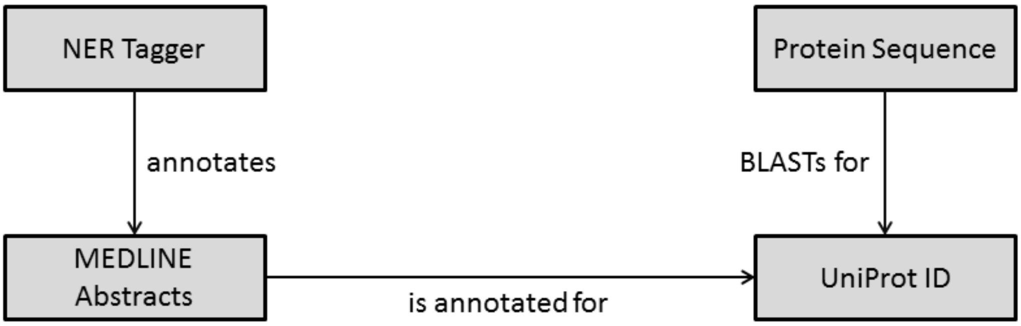

NER Tagger
annotates
MEDLINE Abstracts
is annotated for
Protein Sequence
BLASTs for
UniProt ID

Figure 2.2: Schematic interpretation on how the concepts of BLAST, UniProt, MEDLINE and NER are interrelated.

Chapter 3

Organizations and Components

This chapter enumerates the components constituting the PubSeq system and explains how each in principle functions. There would also be technicalities of the programs and rationales on how each of the component is used and implemented.

3.1 Introduction

In the most general term, there are three main components that build up the PubSeq environment. The three components could be represented as questions:

- How to *create* the data?
- How the data is going to be *stored*?
- How the user is going to *retreive* the data?

We *create* the data in which we process the raw data from our source into indexable entry data. We then *store* the data in scalable manner for the user to use. Finally, we will facilitate how a user could *retreive* our data.

We formalize this concept further by creating three main components with each represents the answer of the question before:

- **PubSeq Tagging Pipeline** (**Tagging Pipeline** for short) addresses the first question. In this pipeline we would process the data from its main source, the MEDLINE corpus which is updated daily as XML file, onto indexable input file containing list of UniProt annotations in the abstracts, among others.
- **PubSeq Solr Index** (**Solr Index** for short) addresses storage issue. All processed data would then in an open source enterprise search platform, Solr [47].

- **PubSeq Web Server** (**Solr Server**) addresses the issue of content delivery to end user. Here we would explain the program in more technical manner. We would also convey how the program would see in the perspective of end user – that is how the program runs as user proceeds on using the search engine, later in Chapter 4.

3.2 PubSeq Persistent Components

To understand how the components were structured, I think it is better for the reader to first get to know how the persistent components, that is, the components that are running around-the-clock interact:

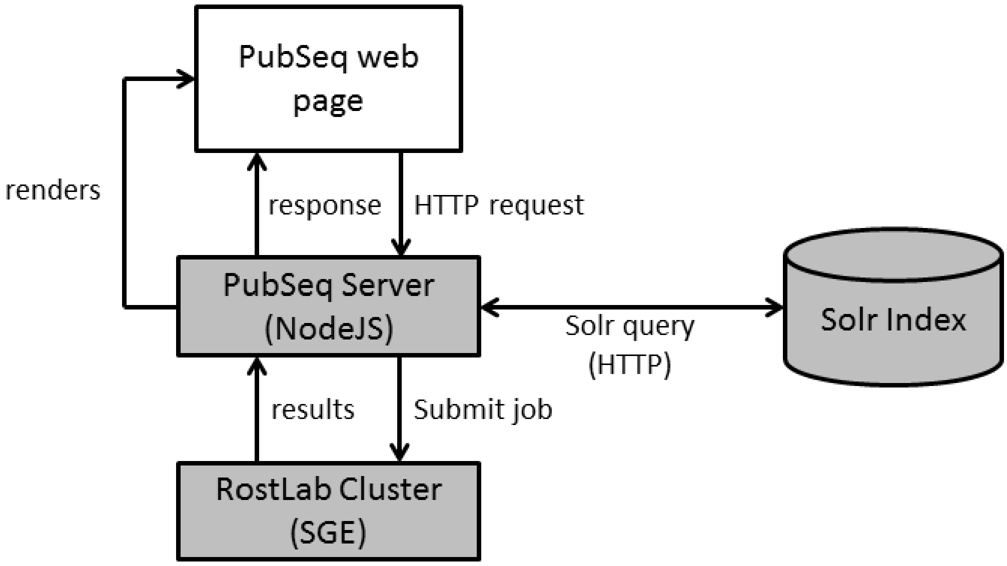


Figure 3.1: An overview of how the Persistent Components – the components that are running around the clock – interact with each other. Here we can see two of three main components explained before: Solr Index, Solr Server (and its rendered web app instance, PubSeq Web Page) which is also connected with RostLab Cluster via SunGrid Engine (SGE). Compare this diagram with spatially differentiated diagram of the system with Virtual Machine shown (Figure 6.1 in Chapter 6)

Here we can see three main components within the interaction environment: Solr Index, PubSeq Server, PubSeq web page and RostLab Cluster (SunGrid Engine).

- **PubSeq Web Page** is rendered by PubSeq server upon GET request on PubSeq home path and communicates through HTTP protocols to PubSeq server. There are currently POST and GET methods that are launched by this page onto the server. Beside the server, the PubSeq web server, the web page also presents some links to outside world, most notably to PubMed web interface [21]. The linkage to PubMed web interface was embeded in each of the results of the query within PubSeq (see Chapter 4 for details)

- **PubSeq Web Server** is server component of our system. It handles HTTP requests addressed to the web path and renders PubSeq web page. Internally it processes both queries and results from both web page and server reply. The server also communicates to RostLab's clusters for submitting BLAST query for a given sequence.
- **PubSeq Solr Index** contains the whole indexing of MEDLINE and proteins mentions within article. While the main mechanism of updating the index will be explained thoroughly in later chapter, it is important to know for reader that the index only communicates via HTTP [47]. As thus, user can check on the machine that the server runs on, the sample content of the Solr server by doing curl followed by the query. For more details see dedicated chapters bellow.
- We utilize **RostLab Cluster** for BLASTing input sequence given by the user.

3.3 Web Interaction

We model our web interaction in following time-dependent sequence diagram [48] on Figure 3.2.

Here we see how four main persistent components of PubSeq interact. First, the user opens the PubSeq page. This will spawn an instance of PubSeq web page. Upon inserting some sequence and pressing query button, the web page would then submit the initial HTTP request [49] onto the server (first **POST sequence**). Our Node.js server [50] would then handle the query and create a script and input file containing aforementioned sequence. It would then submit the query onto RostLab cluster through qsub (**Submit job**) [51] and return first response containing message informing the web client that the job has been submitted (**RESPONSE submitted**). The web client would then re-check the server once in a while (ca. 10 seconds) to find out whether the result of the query has been created (**POST check** and **RESPONSE running**). Once the job has been completed the results would be saved in a predefined location using pre-defined names (see later chapter for more details). During the next iteration of checking routine from web app, if the output is already there, the request handler would then parse the output file and prepare a Solr query that would be used for the sequence. This query would then be submitted onto Solr client (**Query Solr**). The query response would then be forwarded to web page in the RESPONSE massage (**RESPONSE results**) and the results would be shown to the user. Had the user have to update the results (mostly by moving between result pages), an update POST would be initiated (**POST sequence**). This update POST contains prepared statement that doesn't require another BLAST, and thus would be directed toward

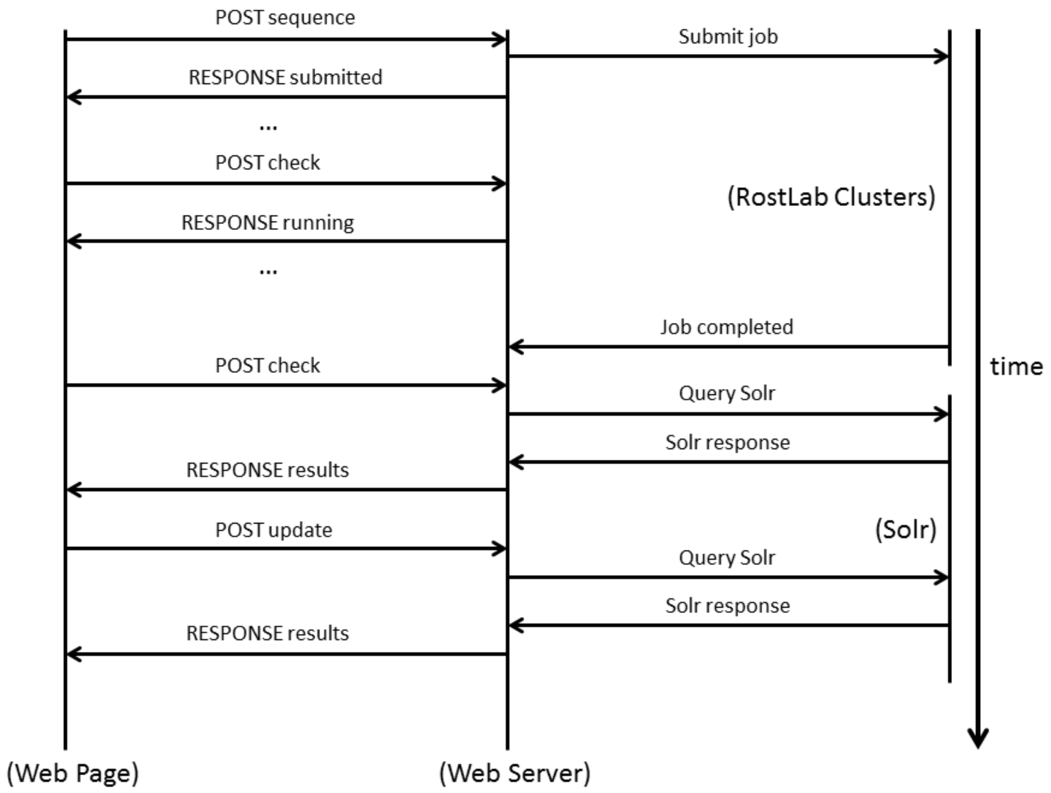

POST sequence
Submit job
RESPONSE submitted
...
POST check
(RostLab Clusters)
RESPONSE running
...
Job completed
POST check
time
Query Solr
Solr response
RESPONSE results
POST update
(Solr)
Query Solr
Solr response
RESPONSE results
(Web Page)
(Web Server)

Figure 3.2: Time-dependent sequence diagram of PubSeq web interaction. The y-axis represents the time (from top to bottom) and each of the columns in on the x-axis represents components that interact with each others. Note that the in the third column, there are two entities that interact with Web Server in distinct time spans.

Solr index directly (second **Query Solr**). Just like initial Solr response, the resulting query would be returned to user. As long as user doesn't leave the page or query new sequence, this iteration could be done as many times as possible.

3.4 Tagging Pipeline

Tagging pipeline refers to the process of creating and updating the Solr index that would be used for the server to search for articles containing protein mentions. In this pipeline, a set of documents from MEDLINE database would sequentially processed and finally be pushed into our index. The whole process is done in periodic/one-off basis. The schematic representation with each single step obscured can be seen in Figure 3.3.

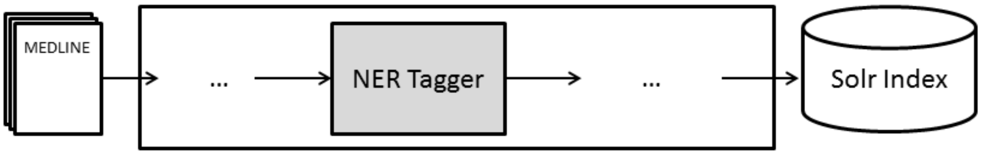


Figure 3.3: Schematic diagram representing the broad process of Solr Tagging Pipeline. On the left side the input files, MEDLINE abstracts in XML formats, would be processed sequentially until the data is ready to be pushed onto the Solr index. We deliberately highlighted NER Tagging from within the pipeline to emphasize its importance in our pipeline.

Here we see that the process consists of smaller processes. As already said, there could be two way of running this process: one-off and periodic update. While one-off Tagging Pipeline could be done on RostLab's **jobtest/jobtest2** or similar physical servers within RostLab infrastructure, the periodic update process should be only done via SunGrid Engine call scheduled in crontab [52] (more details later).

We deliberately showed the NER Tagger process within this Pipeline to reader to emphasize its importance. Lars Juhl Jensen was kind enough to give us his program that we use to run on our pipeline. There would be more details on both our NER Tagger and Tagging Pipeline in later chapter(s) but for now we would focus on big picture detail.

Chapter 4

PubSeq Tagging Pipeline

This chapters technically describes the whole routines of PubSeq Tagging Pipeline. Each single process will be presented and explained along with its program definitions and call arguments.

4.1 Introduction

In this chapter, we would discuss various steps belonging to Tagging Pipeline. Following main steps belong to the Tagging Pipeline:

1. **Formatting** downloaded MEDLINE abstract into input files that are compliant with STRING Tagger. (1)
2. **Named Entity tagging** done by STRING Tagger. (2)
3. **Post-processing** of the results and **preparation** for the entry into Solr index. (3)
4. **Updating** of results onto Solr Index. (4)

The processes are then further divided into several single programs:

- `XMLAbstractsFormatter.java`
- `tagcorpus.cxx`
- `AnnotationBackmapper.java`
- `Annotater.java`
- `StatisticsUtils.java`

- `IndexerNew.java`
- `SolrUpdater.java`

The division of the whole pipeline into smaller tasks are reasoned through following arguments:

- The NER Tagger 2 was developed in C++ while we would implement the rest of pipeline (1, 3 and 4) in Java. This means that pre- and post-tagging procedures would have to be implemented separately. Also since the NER Tagger wasn't written by us and therefore support would be very lacking, we would rather leave the NER Tagger as it is.
- Related to previous argument: Solr is implemented in Java and its most comprehensive API (written by the developers of the Solr themselves) is written in Java [53]. Therefore using Java in 4 would be almost necessary for various convenience reasons.
- The pipeline generally is very memory intensive. During the process, several maps that each would gulp easily teens of gigabyte of memory are utilized. Therefore each step that utilizes such huge maps are all separated into one single routine.
- Dividing the pipeline into smaller components would make it easy to debug, since each routine easily takes several minutes if not hours to run. However this also makes expanding features within the Tagging Pipeline more difficult.

Besides processes mentioned above, there is one process that doesn't exactly belong to Tagging Pipeline but is closely coupled and synchronized with it: downloading and storing of MEDLINE abstracts. Both download and maintenance of MEDLINE corpus and the Tagging pipeline would be covered in following sub-chapters.

As mentioned several times before, the Tagging Pipeline would be run every day in the morning in two steps: updating the MEDLINE corpus and the Tagging Pipeline itself. As the Tagging Pipeline concludes, all updates for the day that were fetched from MEDLINE leasing scheme server [19] would be indexed within Solr.

4.2 MEDLINE Abstracts

We used NLM's leasing scheme [19] to retrieve the MEDLINE corpus. Each day, an updated from MEDLINE server is downloaded via FTP protocol onto our server. The update will happen daily at around 8 A.M. server time (CET/CEST). Each update is formatted as an XML file, each with an incrementing numerical identifier: the files are named `meldine15nXXXX.xml` with `XXXX` the integer identifier – that is, the update file `meldine15n1057.xml` comes right after `meldine15n1056.xml` and before

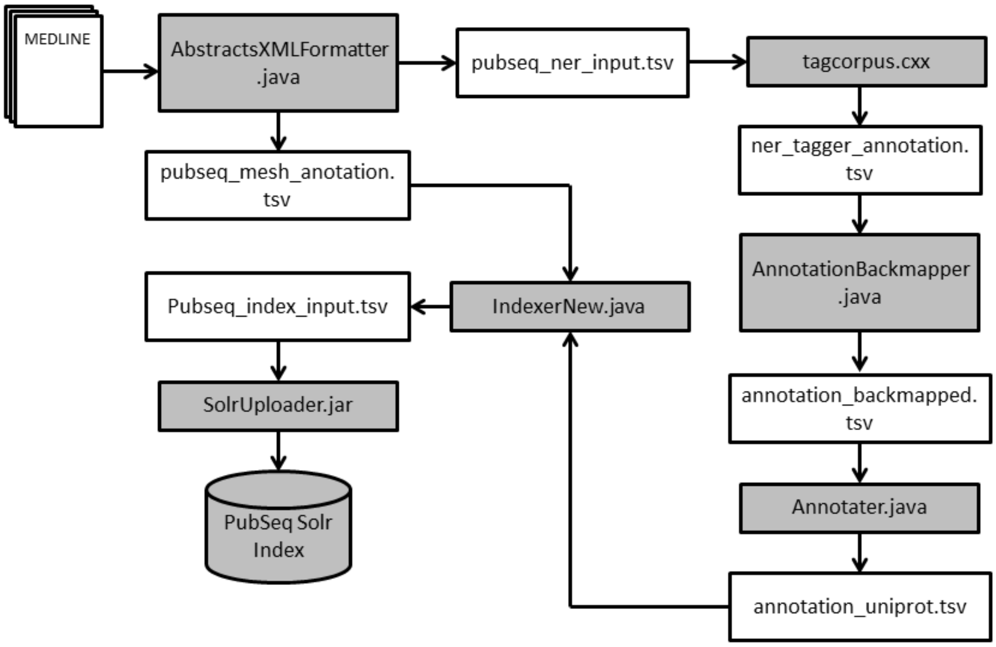

MEDLINE
AbstractsXMLFormatter .java
pubseq_ner_input.tsv
tagcorpus.cxx
pubseq_mesh_anotation. tsv
ner_tagger_annotation. tsv
Pubseq_index_input.tsv
IndexerNew.java
AnnotationBackmapper .java
SolrUploader.jar
annotation_backmapped. tsv
PubSeq Solr Index
Annotater.java
annotation_uniprot.tsv

Figure 4.1: Overview of PubSeq Tagging Pipeline. Here we see how the raw data, MEDLINE XML file, is going through layers of program and ends up as input data for Solr Index. Here we also noticed that the workflow is structured as directed acyclic graph, where some data from non-immediate previous step would be used in later steps. The larger, original size of the workflow could be found in Appendix B.1

`meldine15n1058.xml`. The definition of MEDLINE XML file currently follows National Library of Medicine's (NLM) MEDLINE/PubMed Document Type Definition (DTD) [54], which at the time of thesis writing, is currently at the version dated 01/01/2015 [1].

Following is a slightly redacted example of a MEDLINE reference, which is taken for the paper by Karapakis-Liaskos and Ferrero, Nat. Rev. Immunol., 2015[2]:

```
<MedlineCitation Owner="NLM" Status="In-Data-Review">
<PMID Version="1">25976515</PMID>
<DateCreated>
<Year>2015</Year>
<Month>05</Month>
<Day>26</Day>
</DateCreated>
<Article PubModel="Print-Electronic">
<Journal>
<ISSN IssnType="Electronic">1474-1741</ISSN>
<JournalIssue CitedMedium="Internet">
```

---

[1] accessed 8/20/2015

[2] doi:10.1038/nri3837

```
<Volume>15</Volume>
<Issue>6</Issue>
<PubDate>
<Year>2015</Year>
<Month>Jun</Month>
</PubDate>
</JournalIssue>
<Title>Nature reviews. Immunology</Title>
<ISOAbbreviation>Nat. Rev. Immunol.</ISOAbbreviation>
</Journal>
<ArticleTitle>Immune modulation ... membrane vesicles.</ArticleTitle>
<Pagination>
<MedlinePgn>375-87</MedlinePgn>
</Pagination>
<ELocationID EIdType="doi" ValidYN="Y">10.1038/nri3837</ELocationID>
<Abstract>
<AbstractText>Gram-negative ... nanotechnologies.</AbstractText>
</Abstract>
<AuthorList CompleteYN="Y">
<Author ValidYN="Y">
<LastName>Kaparakis-Liaskos</LastName>
<ForeName>Maria</ForeName>
<Initials>M</Initials>
<AffiliationInfo>
<Affiliation>MIMR-PHI Institute of ..., Australia.</Affiliation>
</AffiliationInfo>
</Author>
<Author ValidYN="Y">
<LastName>Ferrero</LastName>
<ForeName>Richard L</ForeName>
<Initials>RL</Initials>
<AffiliationInfo>
<Affiliation>MIMR-PHI Institute of ..., Australia.</Affiliation>
</AffiliationInfo>
</Author>
</AuthorList>
<Language>eng</Language>
<PublicationTypeList>
<PublicationType UI="D016428">Journal Article</PublicationType>
</PublicationTypeList>
<ArticleDate DateType="Electronic">
<Year>2015</Year>
<Month>05</Month>
<Day>15</Day>
</ArticleDate>
</Article>
<MedlineJournalInfo>
<Country>England</Country>
<MedlineTA>Nat Rev Immunol</MedlineTA>
<NlmUniqueID>101124169</NlmUniqueID>
<ISSNLinking>1474-1733</ISSNLinking>
</MedlineJournalInfo>
<CitationSubset>IM</CitationSubset>
</MedlineCitation>
```

4.2.1 Specifications

Table 4.1: Specifications table for PubSeq MEDLINE Abstracts

Parameter	Value
URL, repo	none
Path, clone	none
Path, running program	`/mnt/project/rost_db/medline`

4.3 XMLAbstractFormatter.java

This class represents a formatter that parses the MEDLINE abstract files and transforms it into STRING Tagger-compliant format. It follows NLM DTD for MEDLINE [54] as parsing reference. In the directory where the updates were downloaded, it lists down all possible XML files and parses files that had not been processed before. For each XML file, it creates a tree representation of the file through full. Owning to its tree-like structure [55] [54], this implies that each `MedlineCitation` tag is itself is a tree (see Section 4.2) . For each `MedlineCitation` it extracts needed information such as PubMed ID, title, journal name, publication date, abstract, authors and associated MeSH IDs [56] and writes it out.

4.3.1 Parameters

```
path_to_medline ner_output_path mesh_output_path counter_file counter_max
```

- `path_to_medline`: the location of MEDLINE xml abstracts. The program will recursively traverse all sub directories of this path and parse files matching following name pattern: `meldine15nXXXX.xml`.

- `ner_output_path`: path the first output file (with name) should be written onto. The first output file would then be used for Tagging by PubSeq NER Tagger.

- `mesh_output_path` : to path the second output file (with name) should be written onto. The second output file contains associative table between PMID and list of MeSH IDs associated with the publication. Since STRING Tagger input is fixed we write this information in separate file.

- `counter_file`: file containing the location of last parsing. The program parses `XXXX` of the name format and figures whether current integer is larger than the one written in `counter_file` before parsing the content of the update file. Upon the completion of the run, the program will replace the value within `counter_file` with the largest `XXX` it encountered. This would be then the starting point of the next run.

- `counter_max`: file containing the maximum `XXXX` this program should read. If `counter_file` contains 1000 and `counter_max` 1500 it will only parse `meldine15n1001.xml` all the way through `meldine15n1500.xml` in this run. Note that `counter_max` is not updated after run. If `counter_max` doesn't exist in system then the it assumes `Integer.MAX_VALUE` [3] as upper bound of the parsing.

Table 4.2: Specifications table for XMLAbstractsFormatter.java

Parameter	Value
URL, repo	See Appendix A

Path, clone	`/mnt/project/pubseq/dev/pubseq-tagging`
	`/PubSeq/src/org/pubseq/utils/`
	`XMLAbstractsFormatter.java`

Path, running program	`/mnt/project/pubseq/dev/pubseq-tagging`
	`/PubSeq/src/org/pubseq/utils/`
	`XMLAbstractsFormatter.java`

4.3.2 Specifications

4.4 tagcorpus.cxx

This program is the core component of our Tagging Pipeline. It takes a tab-separated file as input and tags various entities within each abstract. Originally developed by Jensen et al at University of Copenhagen, the backbone of the program is based in NER tagging environment that was used in (Pacifilis, et al, 2013 [42]), which in turn utilizes dictionaries curated within the framework of STRING database (Szklarczyk, et al, 2011 [43]).

In very broad explanation, the program uses various kind of mapping for various entities. While parsing a text, the program checks for word combinations that looks similar to the entries that were mapped (i.e. the word has small Levensthein distance [57] to some word within the map). For an NER program, the program is astonishingly fast. A full set of MEDLINE corpus with ca. 26 million articles could be parsed in under 12 hours (the mileage may vary according to server specification and number of threads involved).

Our entry point into the program is `tagcorpus.cxx`. There are several modi of running the program. However, in the modus that we use to tag MEDLINE abstracts, we call the tagger in following manner:

```
input_file | path_to_named_entity_tagger/tagcorpus path_to_pubseq_programdata/entities
    .tsv path_to_named_entity_tagger_data/names.tsv path_to_named_entity_tagger_data/
    global.tsv path_to_named_entity_tagger_data/local.tsv
    path_to_named_entity_tagger_data/groups.tsv > output_file_path
```

- `input_file` is the formatted input from `XMLAbstractsFormatter.java`.
- `path_to_named_entity_tagger/tagcorpus`: the location of compiled binary `tagcorpus.cxx`, our gateway to the program. See Appendix A for the detail of program structure.
- `path_to_pubseq_programdata/entities.tsv`: the location of table `entities.tsv`.

---

[3] http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html, accessed 08/20/2015

- `path_to_pubseq_programdata/names.tsv`: the location of table `names.tsv`.
- `path_to_pubseq_programdata/global.tsv`: the location of table `global.tsv`.
- `path_to_pubseq_programdata/local.tsv`: the location of table `local.tsv`.
- `path_to_pubseq_programdata/groups.tsv`: the location of table `groups.tsv`.

While there are several input files besides the prepared MEDLINE abstracts file itself, there are only four tables that we are mainly concerned with: the MEDLINE input table itself, `entities.tsv`, `names.tsv` and the output table.

4.4.1 MEDLINE Input Table

The NER Tagger takes a tab-separated table with following format as input:

- first column: PMID of the article with format of `PMID:article_pmid`.
- second column: list of authors. Each author is separated with semicolon.
- third column: journal name.
- fourth column: publication date.
- fifth column: title.
- sixth column: abstract.

Following is an example of valid input line, taken for the paper by Gu, et al, The Journal of Cell Biology, 1999 [4]:

```
10579727        M Gu;X Xi;G D Englund;M C Berndt;X Du   The Journal of cell biology
      1999-11-29T00:00:01Z    Analysis of the roles of 14-3-3 in the platelet
    glycoprotein Ib-IX-mediated activation of integrin alpha(IIb)beta(3) using a
    reconstituted mammalian cell expression model.     We have reconstituted the
    platelet glycoprotein (GP) Ib-IX-mediated activation of the integrin alpha(IIb)
    beta(3) in a recombinant DNA expression model, and show that 14-3-3 is important
    in GPIb-IX signaling. CHO cells expressing alpha(IIb)beta(3) adhere poorly to vWF.
     Cells expressing GPIb-IX adhere to vWF in the presence of botrocetin but spread
    poorly. Cells coexpressing integrin alpha(IIb)beta(3) and GPIb-IX adhere and
    spread on vWF, which is inhibited by RGDS peptides and antibodies against alpha(
    IIb)beta(3). vWF binding to GPIb-IX also activates soluble fibrinogen binding to
    alpha(IIb)beta(3) indicating that GPIb-IX mediates a cellular signal leading to
    alpha(IIb)beta(3) activation. Deletion of the 14-3-3-binding site in GPIbalpha
    inhibited GPIb-IX-mediated fibrinogen binding to alpha(IIb)beta(3) and cell
    spreading on vWF. Thus, 14-3-3 binding to GPIb-IX is important in GPIb-IX
    signaling. Expression of a dominant negative 14-3-3 mutant inhibited cell
    spreading on vWF, suggesting an important role for 14-3-3. Deleting both the
    14-3-3 and filamin-binding sites of GPIbalpha induced an endogenous integrin-
    dependent cell spreading on vWF without requiring alpha(IIb)beta(3), but inhibited
     vWF-induced fibrinogen binding to alpha(IIb)beta(3). Thus, while different
    activation mechanisms may be responsible for vWF interaction with different
    integrins, GPIb-IX-mediated activation of alpha(IIb)beta(3) requires 14-3-3
    interaction with GPIbalpha.
```

---

[4] doi: 10.1083/jcb.147.5.1085

4.4.2 entities.tsv

`entities.tsv` is a three-column tsv table listing down all unique entity that is taggable by the program. Entity is the smallest unit from speech that is differential from each other. While being the smallest, an entity comes in many forms, with each lexically identifiable with each other. For example, *United States of America*, *America* and *USA* belong to the same entity and are differentiable from *France*, *Republic of France* and *French Republic*, all of which are identifiable with each other and belong to same entity. In STRING Tagger environment, we used STRING ID [43] as our identifier *within the Tagger* – not to be confused with UniProt ID as unique protein identifier in PubSeq system.

Each of three column is defined as follows:

- first column: the STRING ID of an entity. STRING ID is unique identifier of each entity within STRING Tagger environment. An entity could be understood as one single unique named-entity that is recognizable in text (see explanation bellow).
- second column: the designation of entity within NCBI Taxonomy Browser [5] [58]. Since NER Tagger tags various kind of named-entity from protein to disease names, each kind of tag has to be assigned to an identifier describing what kind of named-entity does one entity belong to. A gene/protein belonging to human, for example, will be assigned value of 9606 [6].
- third column: the representation of this entity in alphanumeric form. The name which appears in this entry would be referred as **standard name** in later parts.

As explained before, a single entity could map to multiple written values. This mapping is described in another table, `names.tsv`.

4.4.3 names.tsv

`names.tsv` is mapping table between STRING ID (see the definition for `entites.tsv` above) and all its written forms. There are two columns in this table:

- first column: the STRING ID.
- second column: the name of the entity.

As expected, the relation of this table is **1 x N**, that is, each of the element on the right is associated with only one element of the right and each of the elements on the left is associate with N elements on the right. Also, for each String ID from table entities.tsv, the entry from the third column of the table is also contained in `names.tsv`.

---

[5] `http://www.ncbi.nlm.nih.gov/taxonomy`, accessed 23/08/2015

[6] `http://www.ncbi.nlm.nih.gov/taxonomy/?term=9606`, accessed 23/08/2015

4.4.4 STRING Tagger Annotation

After finishing the tagging process, the NER Tagger would produce a file that lists down all annotations that it found. The output file is tab-separated, with each row representing a single annotation. There are 8 columns in the table:

- first column defines the PMID on which this annotation is found.
- second column defines the start of internal ordering of annotation within STRING Tagger.
- third column defines the end of internal ordering of annotation within STRING Tagger.
- fourth column defines the start offset of the annotation within the input text.
- fifth column defines the end offset of the annotation within the input text, inclusive.
- sixth column defines tagged string as it appears in the input text.
- seventh column defines designated associated taxonomy of entities the string is annotated with (see second column of `entities.tsv`).
- eighth column defines the STRING ID of the entity the string is annotated with (see first column of `entities.tsv`.

As expected, the eight and ninth columns of the output file are coupled. Note also that a string within input text could be annotated with multiple entities. The reason for this is because since the string found in input text is likely to be ambiguous and has therefore to be tagged with multiple entities. To take as an example, *America* could mean both *United States of America* and the continent. Also given the enormous size of entities within `entities.tsv`, combinatorically the likelihood that two entities posses very similar written forms could not be trivialized.

4.5 AnnotationBackmapper.java

This class modifies the results from STRING Tagger. The program AnnotationBackmapper.java does following task:

- Backmapping of the annotations (see bellow).
- Computing statistics of the number of articles with annotation and the number of annotations. Note that the annotations taken statistics of include all possible kind of annotations that the NER tagging done (i.e. not only protein annotation)

The main task of this program is to map the sixth column of each of the NER Tagger annotations back to the name as it appears in entities.tsv. That is, given a result:

```
12345 1 1 11 13 some annotation 9606 23456789
```

and the entry of `23456789` within `entities.tsv`:

```
23456789 9606 standard name
```

this program replaces `some annotation` with `standard name`:

```
12345 1 1 11 13 standard name 9606 23456789
```

This mapped back would be used in later part of the Tagging Process. Also knowing the standard name would help in development process and sanity check, since standard name is usually (in case of protein) unique within some databases (for example, most proteins have ENSEMBL ID as its standard name), albeit not uniformly – the identifiers hailed from various systems including ENSEBML [59], VectorBase [60] and some even have ordered locus (OLS) name as identifier [7].

4.5.1 Parameters

```
ner_annotations entities_table result_path statistics_path
```

- `ner_annotations`: the result of STRING Tagger.
- `entities_table`: the table as appeared in STRING Tagger.
- `results_path`: the path the backmapped annotations are to be written to.
- `statistics_path`: the file the statistics is to be written to.

4.5.2 Specifications

4.6 Annotater.java

This program ”annotates” the tagging results of the NER tagger. Given a result row from NER Tagger, it checks whether the annotation is protein annotation – if so, it then tries to find all UniProt IDs that this annotation maps to. As described before,

---

[7] http://www.uniprot.org/help/gene_name, accessed 23/08/2015

Table 4.3: Specifications table for AnnotationBackmapper.java

Parameter	Value
URL, repo	See Appendix A
Path, clone	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq /src/org/pubseq/annotation /AnnotationBackmapper.java`
Path, running program	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq /src/org/pubseq/annotation /AnnotationBackmapper.class`

every single annotation from NER tagger is associated with STRING ID and each STRING ID is described in `entities.tsv` with a standard name. This is also the case for protein annotation, it is associated with some identifier as standard name. The identifiers used are, however, not strucrtured and belong to single identifying system such as UniProt ID or ENSEMBL (see Section 4.5). This might seem problematic as the start since we decided to only use UniProt ID as standard and unique identifier for proteins. The NER Tagger however, provides such mapping implicitly. As explained in previous chapters, the table `names.tsv` maps each String ID to list of names it is associated with. Within the context of protein entity, each of the STRING ID maps to various names and other identifiers associated with this STRING ID, among others the UniProt IDs. We therefore had to merged these two tables with regard to the String ID and create a new table that maps standard name (that is, name that appears with STRING ID within table `entities.tsv` to other names that are associated with the same STRING ID. This step fortunately has been done in other step and belongs to the Maintenance Pipeline.

As thus, the program only has to parse the resulting merged table from the Maintenance Pipeline and maps the sixth column (already in standard name, see Section 4.5) to the list of all associated names. For each of associated names, an entry is written to output file in format similar to the output format of NER tagger with the sixth column replaced with associated names. That is, for each backmapped annotation from previous step:

```
12345 1 1 11 13 standard name 9606 23456789
```

we checked whether standard name is a UniProt ID. If so, then we write this to output file. Otherwise we check whether there exists some mapping from standard name to list of names within the merged files. If there is one, write for each values the same entry with sixth column replaced:

```
12345 1 1 11 13 uniprot_id_1 9606 23456789
...
12345 1 1 11 13 uniprot_id_m 9606 23456789
```

The multiple mapping of standard name to multiple UniProt IDs could be explained best by the fact that named-entity tagging is prone with lexical ambiguity [61]. A protein named X might have associated with several entries in UniProt database, each with exact or almost exact assigned name. This is especially true in case of unreviewed proteins. There are for example several cases of UniProt entry named *Brain-derived neurotrophic factor* (BNDF) [8]. All aforementioned entries are both lexically and ideolexically the same – they represent very similar things that have very similar name, which would be very hard to disambiguate [61]. This is moreover exacerbated by the fact that many unreviewed proteins have names that are similar or exactly similar to the to the ones that are already reviewed. Looking at context around the annotation location might help to determine whether a named-entity belong to protein or not or whether a tagged protein name belongs to human or not (for example, by looking at whether the sentence/paragraph/paper delves into human disaese etc), however, exactly named protein belonging to same species is hard to differentiate by looking at text. Therefore the multiple mapping.

4.6.1 Parameters

```
ner_backmapped_annotation merged_table uniprot_kbid output_file
```

- `ner_backmapped_annotation`: the result of previous step
- `merged_table`: the merged-filtered table created from `entities.tsv` and `names.tsv` (see program description above).
- `uniprot_kbid`: the mapper between UniProt Accession ID and its associated UniProtKB ID [9]. We use this as a reference for the full list of existing UniProt ID.

4.6.2 Specifications

Table 4.4: Specifications table for Annotater.java

Parameter	Value
URL, repo	See Appendix A
Path, clone	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq` `/src/org/pubseq/annotation/Annotater.java`
Path, running program	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq` `/src/org/pubseq/annotation/Annotater.class`

---

[8] http://www.uniprot.org/uniprot/?query=Brain-derived+neurotrophic+factor&sort=score, accessed 23/08/2015

[9] ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/idmapping/, accessed 23/08/2015

4.7 StatisticsUtils.java

This step is mostly deprecated but is still connected with other program. It was initially intended as statistics mechanism that produces some descriptive statistics on the landscape of annotation for the whole MEDLINE abstract. Since the Tagging Pipeline is done regularly now, this method is not necessary for the whole pipeline.

4.7.1 Specifications

Table 4.5: Specifications table for StatisticsUtils.java

Parameter	Value
URL, repo	See Appendix A
Path, clone	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq/src/org/pubseq/utils/StatisticsUtils.java`
Path, running program	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq/src/org/pubseq/utils/StatisticsUtils.class`

4.8 IndexerNew.java

`IndexerNew.java` is the last step before pushing the input data onto Solr index. The main purpose of this step is to put all information that were acquired during the Tagging Pipeline together. There are three kinds of information that we are particularly interested in form our pipeline:

- List of proteins that are mentioned in an article
- List of taxonomy association of the proteins in the article
- List of MeSH IDs of the article

The first information was obtained during NER Tagging process, while the latter two were obtained while parsing raw MEDLINE XML input. We will use the initially formatted MEDLINE file (see Section 4.3 bellow) as the template. The three kinds of information will each occupy one new column in the updated file. Hence, there would be then nine columns in the output file:

- first column: PMID of the article. Unlike the output format of `XMLAbstractFormatter.java`, the PMID will appear in plain ID form without the prefix `PMID:`).
- second column: list of authors. Each author is separated with semicolon.
- third column: journal name as appeared in XML input file.

- fourth column: publication date.
- fifth column: article title.
- sixth column: article abstract. Multi-abstracts (i.e. abstracts with sub-organizations such as introduction, methods, results etc) would be representented as one string.
- seventh column: list of UniProt ID mentions, separated by space.
- eight column: list of MeshID associated with the article, separated by space.
- ninth column: unique list of UniProt ID species associations, separated by space.

To achieve, this, we initialized a map for each of the information, with PMIDs as key and the list of UniProt IDs/unique species/MeSH IDs as value. We do this by simply parsing the results from previous steps that contain the information we'd like to index:

- **List of UniProt ID Mentions** are available from post-processed NER tagger annotations. We use the output of Annotater.java. The program implements the index as Hash Map with PMID as key and list of UniProt as values. The file will be parsed sequentially and the UniProt ID annotation (sixth column) for the PMID (first column) would be pushed into the map. At the end, a temporary file will be written with added column created from this index. Upon embedding the merging will done and the result written. Afterwards, the map is deleted to free up memory.
- **List of MeSH IDs** are available from `XMLAbstractFormatter.java`. Since the file is already in form of implicit map (two columns with first column containing PMID and second list of MeSH), the program just parses the file sequentially into the index. The later steps follow roughly the one for UniProt ID mentions.
- **List of Species Associations** are available from post-processed NER tagger annotation. As reader can see in the output format of PubSeq NER Tagger, the seventh column contains the taxonomy association of the annotation. As such, this step follows closely the first procedure (List of UniProt ID mentions). The only different would be that the parser uses seventh instead of sixth column as value.

he reason behind doing this sequentially is, again, memory. In the worst case when the program has to index the whole MEDLINE abstract, memory usage for each of the step easily exceeds 20 GB. As such sequential steps were used instead of all-in-one approach.

4.8.1 Parameters

```
template_path template_index_col_num output_dir_path first_file_path
    first_file_index_col_num first_file_value_col_num second_file_path
    second_file_index_col_num second_file_value_col_num third_file_path
    third_file_index_col_num third_file_value_col_num final_name
```

- `template_path`: path to template file.
- `template_index_col_num`: column number of template file's reference column (PMID). This will be used for embedding template with additional columns.
- `output_dir_path`: the directory where interim and final values are to be stored.

For each of the three files:

- `path`: output file path.
- `index_col_num`: column number of the file's reference column (PMID).
- `value_col_num`: column number of the file's value.

Also,

- `final_name`: final name of the output file.

4.8.2 Specifications

Table 4.6: Specifications table for IndexerNew.java

Parameter	Value
URL, repo	See Appendix A
Path, clone	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq /src/org/pubseq/utils/IndexerNew.java`
Path, running program	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq /src/org/pubseq/utils/IndexerNew.class`

4.9 IndexerNew.java

The class from which SolrUpdater.jar derives from, **SolrUpdater.java**, consists of module that will push new documents (Solr term for an entry) onto the index. Given the input table with updated columns from IndexerNew.java, this programs parses and pushes each row from the input. For each row from the input, the program checks whether such document already exists in the Solr index. If such document exists, it updates the attributes of existing document by adding new values coming from the parsed row. Otherwise, the program creates a document instance containing the values from parsed row as attributes. The document instance would then be pushed onto the index. The whole process utilizes the Solr native API for Java, SolrJ [53] [10] [11].

4.9.1 Parameters

```
abstract_output_path index_url
```

- `abstract_output_path`: path of directory containing input table.
- `index_url`: URL on which the PubSeq core is accessible from see Chapter 5 for the description of core within Solr environment. The URL of a core within an environment is basically a subpage of the index itself. That is, for an index located at `http://localhost:8983/solr/`, the address of core named `pubseq` would be `http://localhost:8983/solr/pubseq/`.

4.9.2 Specifications

TABLE 4.7: Specifications table for SolrUploader.java

Parameter	Value
URL, repo	See Appendix A
Path, clone	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq/src/org/pubseq/utils/SolrUploader.java`
Path, running program	`/mnt/project/pubseq/pandu/pubseq-crawler/PubSeq/SolrUploader.jar`

---

[10] `https://wiki.apache.org/solr/Solrj`, accessed 23/08/2015

[11] `https://cwiki.apache.org/confluence/display/solr/Using+SolrJ`, accessed 23/08/2015

Chapter 5

PubSeq Solr Index

This chapter tries to answer the second question posed in Section 3.1: *how the data is going to be stored?* We would see the reason behind our choice of storage technology. We would then also try to specify the definition of our index.

5.1 Solr: Fast and Scalable Indexing

Apache Solr is an open source enterprise search platform. Initially developed by Yonik Seeley at CNET Networks, it was later published as open source through donation to Apache Software Foundation [1]. The system is based on Lucene internally, which implements the indexing routine and mechanism [62].

Unlike SQL based system which uses tabular data representation as underlying data storage mechanism, Solr utilizes indexing mechanism in its system [47]. This non-SQL characteristics of data storage and maintenance are generally known as NoSQL. (Brewer, 2000) [2] did some nice overview of comparison between NoSQL and conventional SQL, which could be seen on Table 5.1

While most NoSQL implementations don't strictly follow ACID criterions, most don't fall into BASE category either. Also, many NoSQL implementations focus on particular use case rather than aim at general data storage purpose. In this regard, Solr aims to be able to store and index in the order of millions and billions documents while optimizing for – among others – faceted search [63], string search and result paging[2]. Solr also supports implicit definition of text syntax through definition of stopping sign, language rules etc – this malleability is used for example, for Solr to be able to optimize for specific language it indexes [53]. This way, Solr not only stores the data efficiently but in a way understand the logic of the data.

Formally, we define our reason of using Solr as follows:

---

[1] https://issues.apache.org/jira/browse/SOLR-1, accessed 25/08/2015

[2] https://cwiki.apache.org/confluence/display/solr/Pagination+of+Results, accessed 24/05/2015

Table 5.1: ACID vs. BASE database property models. ACID vs. BASE database property models. Some NoSQL databases implement ACID, others do not. Taken from (Wachinger, 2013) [1], adopted from (Brewer, 2000) [2].

ACID (relational model)	BASE (NoSQL)
Atomicity, Consistency, Isolation, Durability	Basically Available, Soft state, Eventually consistent
Isolation	Availability First
Focus on Commit	Best Effort
Nested Transactions	Approximative Answers Acceptable
Non-quaranteed Availability	Aggressive (Optimistic)
Conservative (pessimistic)	Simpler
Difficult Evolution (schema)	Faster, Easier Evolution

- **Reverse indexing**. Possibly the strongest proposition of using Solr, reverse indexing refers to listing down *where in which documents does a string appears in the index*, instead of *which strings appear in which document in the index* 5.1. This reverse paradigm enables Solr to retrieve string related queries in almost constant time [53].
- **Automatic Query Weighting**. In its query syntax, Solr enables weighted string matching query. Take for example following content of Solr query:

  ```
  unirotid:P53_HUMAN^2 AND uniprotid:P53_HORSE
  ```

  In this query, `unirotid:P53_HUMAN` would be given twice as much weight as `uniprotid:P53_HORSE`. For complex analysis involving multiple string queries this proves to be a powerful feature.
- **Easy Deployment**. Each unpacked Solr directory could be run to create an instance of server. The only external requirement for Solr is installed Java Virtual Machine (JVM). This makes easier for us for example to test our index in various environment, therefore reducing iteration overhead.
- **Native Java Implementation**. Combined with its native Java API, SolrJ, it ensures easier interaction and development of the index.
- **Result Pagination**. Solr natively supports results pagination[3]. This means that, for very huge query result the system doesn't have to dump the whole result in memory. For our specific purpose, this proves helpful.

There are also other reasons for deploying Solr such as generally lower schema definition complexity, generally powerful syntax etc., which makes even more case for using Solr over other both SQL and non-SQL systems.

---

[3] `https://cwiki.apache.org/confluence/display/solr/Pagination+of+Results`, accessed 25/08/2015

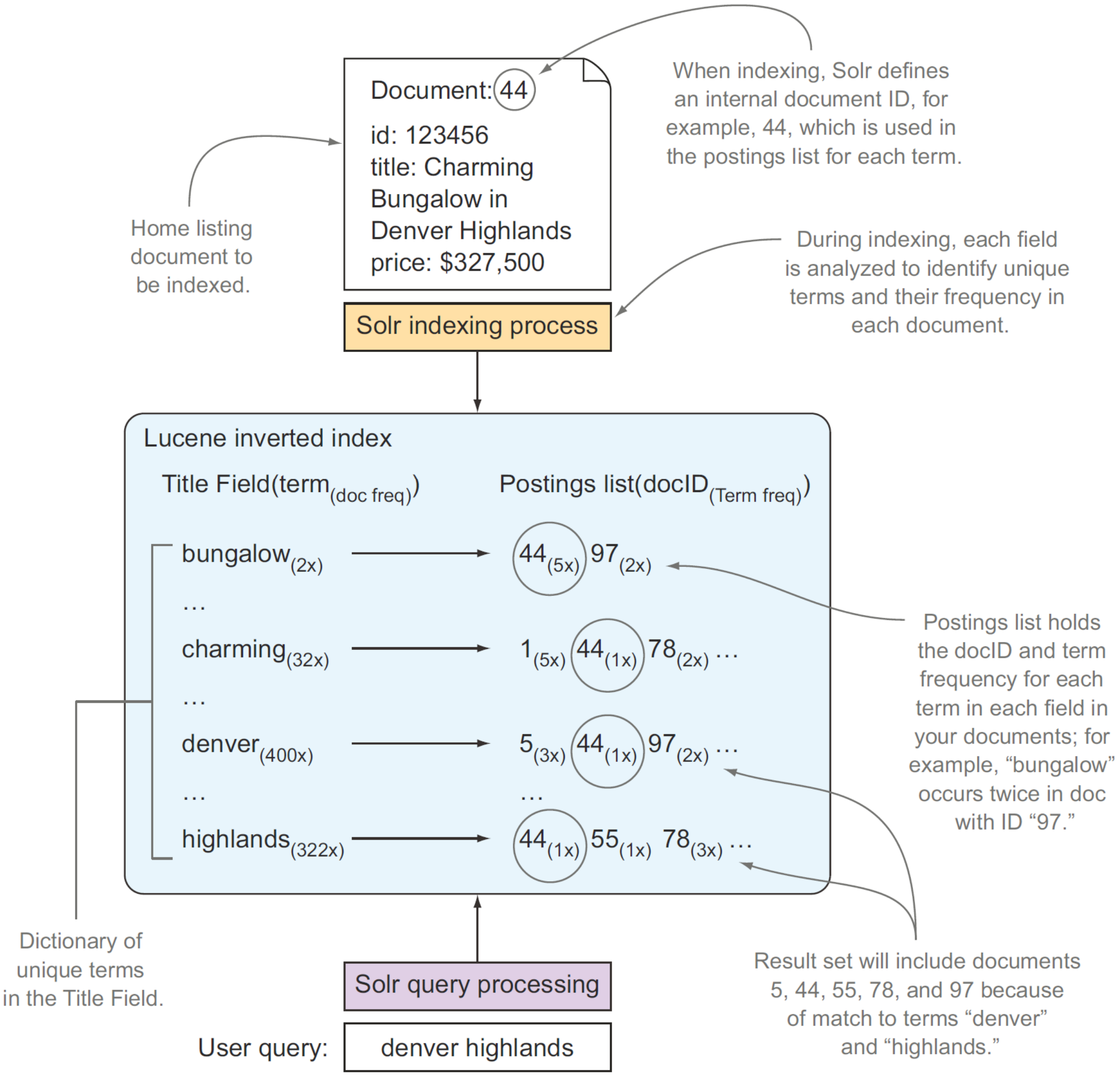

Document: 44
id: 123456
title: Charming Bungalow in Denver Highlands
price: $327,500
When indexing, Solr defines an internal document ID, for example, 44, which is used in the postings list for each term.
Home listing document to be indexed.
Solr indexing process
During indexing, each field is analyzed to identify unique terms and their frequency in each document.
Lucene inverted index
Title Field(term(doc freq))
Postings list(docID(Term freq))
bungalow(2x)
44(5x) 97(2x)
…
charming(32x)
1(5x) 44(1x) 78(2x) …
…
denver(400x)
5(3x) 44(1x) 97(2x) …
…
…
highlands(322x)
44(1x) 55(1x) 78(3x) …
Postings list holds the docID and term frequency for each term in each field in your documents; for example, "bungalow" occurs twice in doc with ID "97."
Dictionary of unique terms in the Title Field.
Solr query processing
User query:
denver highlands
Result set will include documents 5, 44, 55, 78, and 97 because of match to terms "denver" and "highlands."

FIGURE 5.1: Overview of reverse indexing mechanism employed by Solr. Here we see how a document is being parsed and represented internally within the mapping of tokens. Beyond Solr, reverese indexing is powerful tool in the field of Information Retrieval (IR) which enables finding occurrences in much reduced time [64]. The fundamentals of reverse indexing in IR follows roughly the same concepts of Solr's reverse indexing. Figure adopted from (Grainger, Potter and Seeley, 2014 [53])

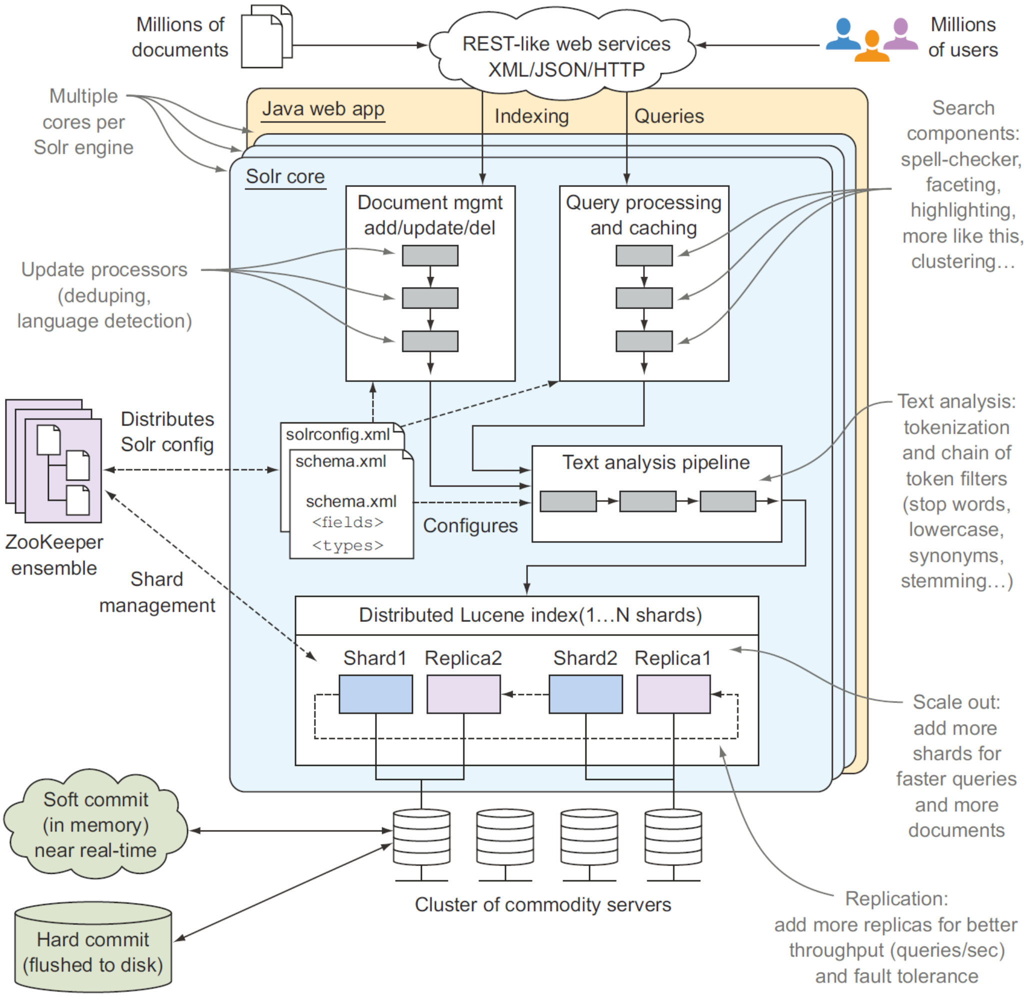

Millions of documents
REST-like web services XML/JSON/HTTP
Millions of users
Multiple cores per Solr engine
Java web app
Indexing
Queries
Search components: spell-checker, faceting, highlighting, more like this, clustering…
Solr core
Document mgmt add/update/del
Query processing and caching
Update processors (deduping, language detection)
Distributes Solr config
ZooKeeper ensemble
solrconfig.xml
schema.xml
schema.xml
<fields>
<types>
Configures
Text analysis pipeline
Text analysis: tokenization and chain of token filters (stop words, lowercase, synonyms, stemming…)
Shard management
Distributed Lucene index(1…N shards)
Shard1
Replica2
Shard2
Replica1
Scale out: add more shards for faster queries and more documents
Soft commit (in memory) near real-time
Hard commit (flushed to disk)
Cluster of commodity servers
Replication: add more replicas for better throughput (queries/sec) and fault tolerance

FIGURE 5.2: Diagram of the main components of Solr. There are two main modes of index access: query and indexing. Both modes are handled by the top level REST-like service (wrapped in Jetty [65], Tomcat [66] or something similar). Each instance consists of multiple core, with each core defined by configuration files, namely `solrconfig.xml` (which applies globally across all cores) and `schema.xml`. Both indexing and query routines would be going through text analysis pipeline for tokenization and other lexical analysis. Solr would then store/fetch several meta-information in/from appropriate Lucene index. Figure adopted from (Grainger, Potter and Seeley, 2014 [53])

5.2 Index Definitions

Each Solr engine consists of several **cores** (see Figure 5.2). Solr wiki page wrote following regarding Solr Core [4]:

“ Solr Core: Also referred to as just a ”Core”. This is a running instance of a Lucene index along with all the Solr configuration (`SolrConfigXml`, `SchemaXml`, etc...) required to use it. A single Solr application can contain 0 or more cores which are run largely in isolation but can communicate with each other if necessary via the `CoreContainer`. From a historical perspective: Solr initially only supported one index, and the `SolrCore` class was a singleton for coordinating the low-level functionality at the ”core” of Solr. When support was added for creating and managing multiple Cores on the fly, the class was refactored to no longer be a Singleton, but the name stuck. “

Needless to say, Solr core is the lowest abstraction that is configurable within Solr environment. Each Solr core is defined by its own schema. In the mode we were running our Solr on[5], we utilize `schema.xml` to define our schema. For someone more familiar in SQL-based lingo, a core could be thought analogously as table and schema its definition. This definition only works so far since communicating between two cores within Solr is not as trivial as it would be the case for tables in SQL environment.

Similar to SQL fashion, one of the most important aspect of schema definition within Solr is fields definition. We first define our fields in following table:

TABLE 5.2: Fields definition table for `pubseq` index within the Solr system.

name	type	indexed	stored	required	multiValued
title	text_general	`True`	`True`	`True`	`False`
abstract	text_general	`True`	`True`	`False`	`False`
authors	text_general	`True`	`True`	`False`	`True`
journal	text_general	`True`	`True`	`True`	`False`
uniprotid	text_general	`True`	`True`	`False`	`True`
speceisid	text_general	`True`	`True`	`False`	`True`
meshid	text_general	`True`	`True`	`False`	`True`

type refers to what kind of Java class this field implements. If **indexed** is true, the value of the field can be used in queries to retrieve matching documents. If **stored** is true, the actual value of the field can be retrieved by queries. if **required** is true, the core instructs Solr to reject any attempts to add a document which does not have a value for this field. If **multiValued** is true, it indicates that a single document might contain multiple values for this field type[6]. Each multi-valued entry is represented as list internally. This could be invoked using any class implementing List interface [7]. As query result, depending on the return format (XML, JSON or CSV), this would be returned as either XML list, JSON list oobject or CSV cell-internal list respectively.

---

[4] `https://wiki.apache.org/solr/SolrTerminology`, accessed 25/08/2015

[5] Depending on which Solr one is running, a schema could be defined either by `schema.xml` file or through Schema API (see here: `https://cwiki.apache.org/confluence/display/solr/Schema+API`, accessed 25/08/2015). Both interfaces trigger approximately the same configuration internally.

[6] `https://cwiki.apache.org/confluence/display/solr/Defining+Fields`, accessed 25/08/2015

Upon creatoin of core, this configuration would be internalized via `schema.xml`. An example of instance of `schema.xml` can be found in Appendix A.1.1.

Following is a slightly abridged example of a document within Solr, which is taken for the paper by Takahashi and Yamanaka, Cell, 2006[8]:

```
{
        "pmid":"16904174",
        "authors":["Kazutoshi Takahashi", "Shinya Yamanaka"],
        "journal":"Cell",
        "pubdate":"2006-08-25T00:00:01Z",
        "title":"Induction of pluripotent stem cells from mouse embryonic and adult
    fibroblast cultures by defined factors.",
        "abstract":"Differentiated cells can be reprogrammed to an embryonic-like
    state by transfer of nuclear contents into oocytes or by fusion with embryonic
    stem (ES) cells. Little is known about factors that induce this reprogramming.
    Here, we demonstrate induction of pluripotent stem cells from mouse embryonic or
    adult fibroblasts by introducing four factors, Oct3/4, Sox2, c-Myc, and Klf4,
    under ES cell culture conditions. Unexpectedly, Nanog was dispensable. These cells
    , which we designated iPS (induced pluripotent stem) cells, exhibit the morphology
     and growth properties of ES cells and express ES cell marker genes. Subcutaneous
    transplantation of iPS cells into nude mice resulted in tumors containing a
    variety of tissues from all three germ layers. Following injection into
    blastocysts, iPS cells contributed to mouse embryonic development. These data
    demonstrate that pluripotent stem cells can be directly generated from fibroblast
    cultures by the addition of only a few defined factors.",
        "uniprotid":["B3CJ88_HUMAN", "B3CJC0_HUMAN", "Q3V2B6_MOUSE", "B3CJ65_HUMAN", "
    A2RS90_MOUSE", "B3CJB3_HUMAN", "Q3UES8_MOUSE", "B7ZCH2_MOUSE", "B3CJD1_HUMAN", "
    B3CJ82_HUMAN", "B3CJ66_HUMAN", "NANOG_MOUSE", ... , "KLF4_MOUSE", "J7H3Z5_HUMAN",
    "J7H416_HUMAN", "H0YBG3_HUMAN", "H0YBT0_HUMAN", "F2VTX5_HUMAN"],
        "meshid":["D000328", "D000818", "D002454", "D017690", "D002478", "D004268", "
    D004622", "D005347", "D020869", "D018398", "D006801", "D051741", "D051379", "
    D008819", "D008822", "D050814", "D020411", "D039904", "D016271", "D055748", "
    D015534"],
        "speciesid":["10090", "9606"],
        "_version_":1504409903648735232
}
```

5.3 Information Retrieval

Solr communicates mainly through HTTP protocol. This makes developer to be able to iterate through development process quickly. Each Solr syntax is structured in following way:

```
http://hostname:port/solr/core_name/select?wt=json&indent=true&q=solr_query
```

The complete syntax of `solr_query` can be seen in (Grainger, Potter and Seeley, 2014 [53]). There are also several online resources such as here[9] and here[10]. The syntax involves mostly a conjunctions/disjunctions of predicate. Each predicate commonly comes in form of `field:value`. That is, to find list of papers by Yamanaka that were published in journal Cell one would write as query:

---

[7] `http://docs.oracle.com/javase/7/docs/api/java/util/List.html`, accessed 25-08-2015

[8] doi: 10.1016/j.cell.2006.07.024

[9] `http://www.solrtutorial.com/solr-query-syntax.html`, accessed 25/08/2015

[10] `https://wiki.apache.org/solr/SolrQuerySyntax`, accessed 25/08/2015

```
...q=authors:yamanaka AND journal:cell
```

Note that URL applies its own encoding rules [67]. The complete url of above query, done on `pubseq` core within PubSeq Solr index, would look like following:

```
http://localhost:8983/solr/pubseq/select?wt=json&indent=true&q=authors%3Ayamanaka%20
    AND%20journal%3Acell
```

The process of data retrieval cold be done both synchronously and asynchronously.

Chapter 6

PubSeq Web Service

In this chapter we address the third question posed in Section 3.1: *How the user is going to retreive the data?*. First, we would define our Virtual Machine environment on which our system runs. We would then delve into the structure of our Node.js server, how it initializes a web page and how it would interact with our service. Finally we would see how a user would generally use PubSeq, from giving in sequence to receiving and navigating over the results.

6.1 PubSeq Virtual Machine

Two main components of PubSeq: PubSeq Solr Index and PubSeq Node.js server lay within a defined Virtual Machine within Rostlab internal network. Once a user opens a path that is directed to our Solr web server, the request would first land on Rostlab.org web server. Rostlab.org web server would then forward this request further to our PubSeq Node.js web server. The service would then render the page that the user will use to communicate with our server. All requests and responses between web page and PubSeq Node.js server would be proxied through Rostlab.org apache server (see Figure 6.1). Our Virtual Machine is accessible internally via an assigned IP address and could also access other components within Rostlab network such as the SunGrid Engine (SGE). This IP address is however, not accessible from outside world. This guarantees that our Virtual Machine, particularly the Solr Index, which communicate via HTTP, couldn't be accessed from outside the world and thus prevents an unwanted disruption such as database injection.

To further ensure consistency of our Virtual Machine, we create a snapshot of the the VM once every day. Out of daily snapshots, we would keep the snapshots form the last two days. This way, we can always roll back our VM in case something would happen on it.

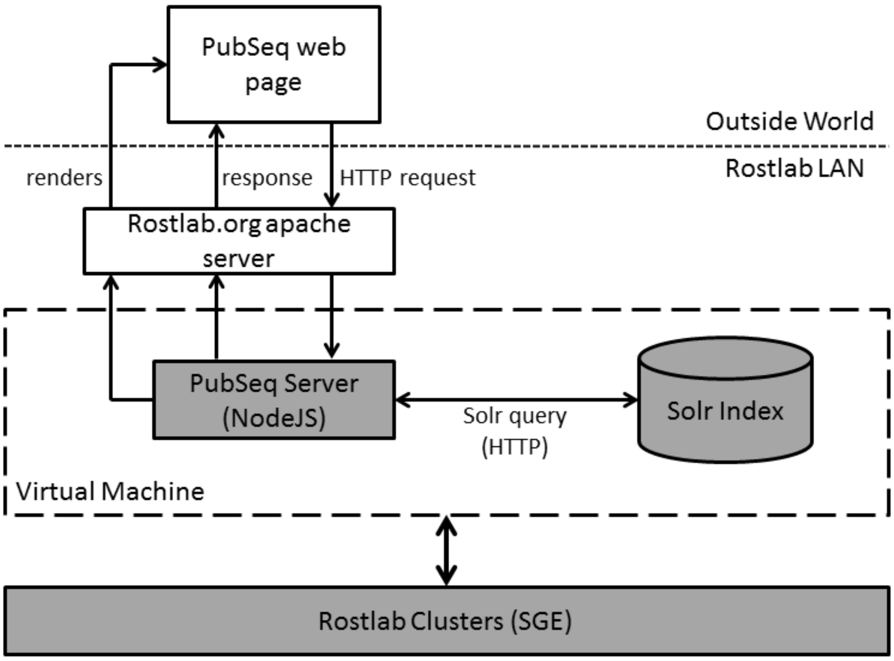

PubSeq web page
Outside World
Rostlab LAN
renders
response
HTTP request
Rostlab.org apache server
PubSeq Server (NodeJS)
Solr query (HTTP)
Solr Index
Virtual Machine
Rostlab Clusters (SGE)

Figure 6.1: Schematic representation of PubSeq systems with the Virtual Machine environment shown. Here we see the relative position and content of our Virtual Machine within Rostlab local area network (LAN). Rostlab SunGrid Engine (SGE) cluster and most other constituents within Rostlab networks are accessible from Virtual Machine *vice versa*.

6.1.1 VM Specifications

Following are the specifications of our Virtual Machine:

Kernel:

```
Linux pubseq-web.rostclust 3.16.0-4-amd64 #1 SMP Debian 3.16.7-ckt11-1 (2015-05-24)
    x86_64 GNU/Linux
```

Distribution:

```
Disribution:            Debian GNU/Linux 8.1 (jessie)
Release:                8.1
Codename:               jessie
```

Processors:

```
Core:                   1
CPU MHz:                2599.998
BogoMIPS:               5199.99
CPU RAM GB:             6
Architecture:           x86_64
```

Here we see that our VM has relatively low RAM and number of core compared to other constituents within the network [1]. The reason for this is because computationally expensive computations within our system, PubSeq Tagging Pipeline and BLAST, are always delegated to Rostlab SunGrid Engine (SGE). The two components that persistently run on our VM, Node.js Server and Solr, don't take a lot of memory and computing power. A lightweight Solr instance usually takes about 500 MB memory to run – in our case, we only limit our memory use use to 1 GB RAM. Node.js is elastically implemented – that is, it only requires memory that it needs. This means that in a given time, when there is no request, memory usage is nothing but negligible.

6.2 PubSeq Web Server

In this project, we used Node.js for server side networking and scripting purpose. Node.js is a runtime environment for applications written in JavaScript. Unlike conventional JavaScript which runs on webpage within web browser engine, Node.js enables JavaScript application to run in command line environment. It provides event-driven and lock-free I/O API [68], which makes it suitable for real-time web application. Node.js achieves this by utilizing the so-called Event Loop (see Figure 6.2).

Node.js is based on V8 Virtual Machine, which unlike other more traditional JavaScript VM, compiles JavaScript to native machine code instead of interpreting the code and then compiling it [69]. All these characteristics make Node.js suitable for running a web application that handles thousands of concurrent connections with minimal overhead possible. We also chose Node.js as our web server environment since it allows compatibility with regard to data communication between web server and client, since native JSON would be used to communicate between the two. Specifically for Node.js, we use Express framework to implement our server, which is the most commonly used web service framework in Node.js [70].

6.2.1 Components

There are two main components in our Node.js server: **app.js** and web page files.

app.js[2] is the main entry in our server. It contains components that are needed to make the server running such as:

- Express.js application definition and initiation [3].
- HTML rendering engine definition (see **Web Page Files** bellow).

---

[1] One typical node in Rostlab's cluster has about 32 GB RAM and 12 cores

[2] See Appendix A.1.2 for the paht to app.js within the repository.

[3] See `http://expressjs.com/4x/api.html` (accessed 26/08/2015) for Express.js API.

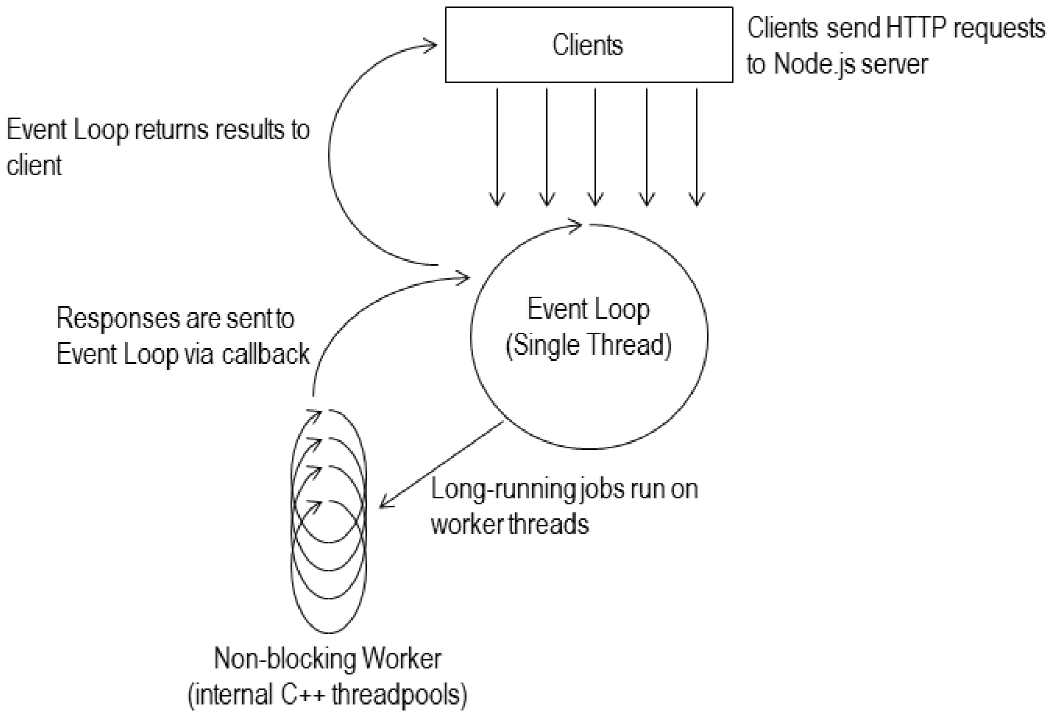

Clients
Clients send HTTP requests to Node.js server
Event Loop returns results to client
Event Loop (Single Thread)
Responses are sent to Event Loop via callback
Long-running jobs run on worker threads
Non-blocking Worker (internal C++ threadpools)

FIGURE 6.2: Node.js internal processing model. Upon the arrival of HTTP request in Node.js server, Event Loop, which is implemented as single thread, passes the request onto worker threads while also adding callback onto its stack. Upon finishing submitted job, callback function will be called to notify Event Loop. It would then return job's results to the requesting client. By avoiding spawning thread for every request, Node.js avoids the overhead that could occur in server that handles thousands of request in a given time. This makes it appropriate for multi-users web application. Figure adopted from (Stannard, 2011 [71]) with modifications.

- Additional prototypical String functions such as hash code creating function and `startsWith()` [4]
- `GET`, `POST` handlers for each of available pages, if such method is defined within the page.
- Cache implementation [5].
- Several utility functions.

Web Page Files consist of pages that are available within the web application environment. Each page is written in Jade syntax, which makes it easier for user to write complex HTML file [6]. Upon `GET` request for a page, **app.js** would render the page file using the HTML rendering engine for jade. This engine would parse the `.jade` file and create proper HTML file, which would then be returned to client.

---

[4] Note that `startsWith()` will be implemented in upcoming ECMAScript 2015 (ES6) Standard, see `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith` (accessed 26/08/2015).

[5] `https://www.npmjs.com/package/node-cache`, accessed 27/08/2015

6.2.2 index.html

Of most importance is the **index.html** (available as **index.jade** prior rendering process) among web pages that are available within the web interface environment. The web page contains the query interface that would communicate with the web server. It uses Twitter's front-end framework Bootstrap to create simple mobile-first web interface [72]. The page consists of:

- **Navigation Bar**, which contains links to other pages.
- **Jumbotron**[7], which contains:
  - Search box
  - Query Button

The current snapshot of PubSeq web interface can bee sin in Figure 6.3.

6.3 Usage

To use the webpage, user first has to put in amino acid sequence he/she is interested in. Alternatively, a user could also ask for sample sequence to be given by PubSeq web page (see Figure 6.4). There are currently five sequences that would be given to user at random[8]. Upon hitting the **Query** button, the web page would be blocked during the entire query process (see Figure 6.5). This would prevent user from multiple query on the same page at once. During this process, the client proceeds by submitting a `POST` method once every 10 seconds. There are several modes of `POST` method:

- `new`. The initial `POST` method. This method will tell the server that the client is starting a query with a given sequence. The server would check the cache whether the sequence has not been queried before. If not, it will submit a BLAST job onto Rostlab SunGrid Engine (SGE) and return the client with JSON containing `status:'running'`. Otherwise it would use cached list of UniProt IDs to query for articles mentioning the IDs in Solr Index. The list of articles would then be returned to client.

---

[6] `http://naltatis.github.io/jade-syntax-docs/`, accessed 26/06/2015.

[7] `http://getbootstrap.com/components/`, accessed 26/08/2015

[8] See **index.jade** to see the list of sequences.

- `check`. The mode **check** is done after the client receive JSON from server containing `status:'running'`. The `POST` will be done approximately 10 seconds after last message was receive. It will check whether the BLAST run was already done. If not, it will return again JSON containing `status:'running'`. Otherwise it will proceed to Solr and return mentioning articles to client.
- `update`. The mode **update** is done when the user move between result pages. In this modus, the server would basically query the Solr by continuing from last query[9].

The interrelations between states could be presented as state diagram as seen in Figure 6.7.

After the query/page transition `POST` method is finished, the results would be presented to the user. As of the writing of this thesis, four kind of information are shown to user:

- Number of matching documents.
- The Solr query used for the sequence.
- List of articles. Each article conveys following information:
  - Article Title
  - PubMed ID
  - Authors
  - Publication date
  - The first 400 characters of abstract
- Page number with optional left and right move page button.

Figure 6.6 shows the first article that was shown in results page.

---

[9] `https://cwiki.apache.org/confluence/display/solr/Pagination+of+Results`, accessed 27/08/2015

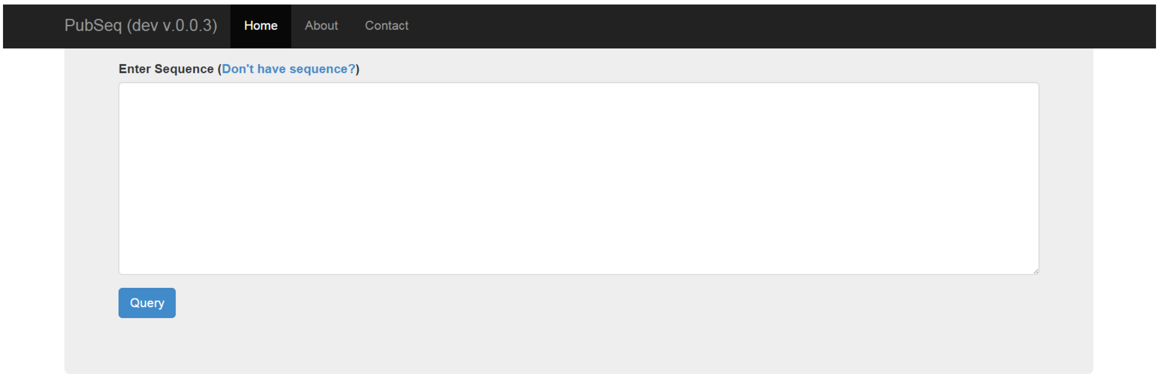

PubSeq (dev v.0.0.3)
Home
About
Contact
Enter Sequence (Don't have sequence?)
Query

FIGURE 6.3: PubSeq web interface.

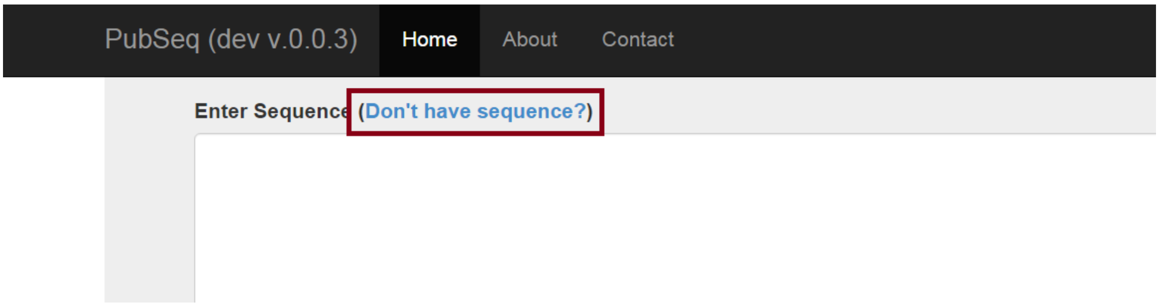

PubSeq (dev v.0.0.3)
Home
About
Contact
Enter Sequence (Don't have sequence?)

FIGURE 6.4: Sample sequence button in PubSeq web interface.

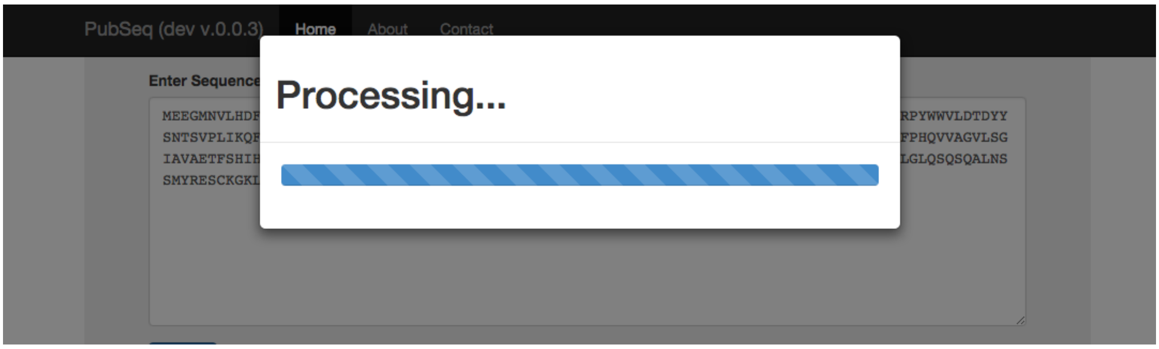
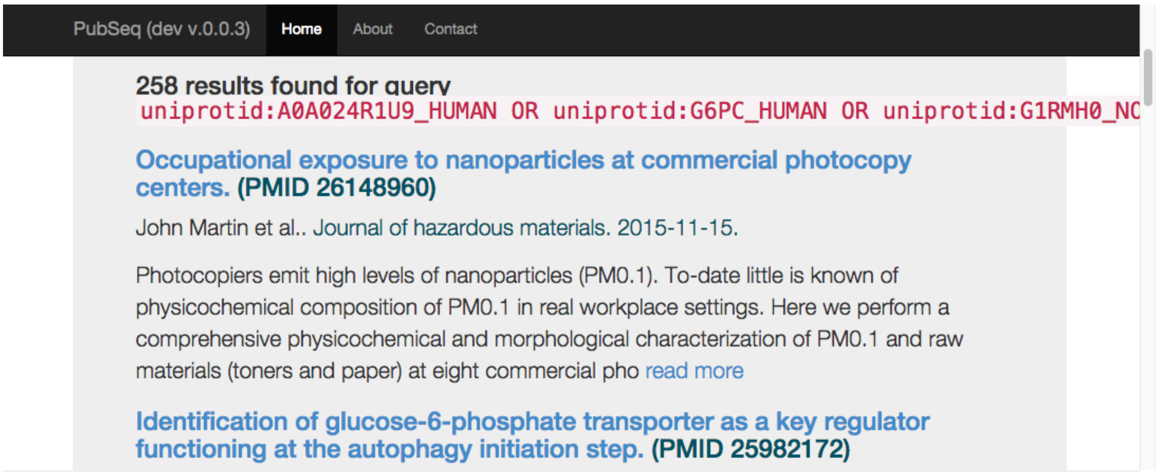

PubSeq (dev v.0.0.3)
Home
About
Contact
Enter Sequence
Processing...
MEEGMNVLHDF
SNTSVPLIKQF
IAVAETFSHIH
SMYRESCKGKL
RPYWWVLDTDYY
FPHQVVAGVLSG
LGLQSQSQALNS

FIGURE 6.5: Blocked screen in PubSeq web interface.



258 results found for query
uniprotid:A0A024R1U9_HUMAN OR uniprotid:G6PC_HUMAN OR uniprotid:G1RMH0_NO

Occupational exposure to nanoparticles at commercial photocopy centers. (PMID 26148960)

John Martin et al.. Journal of hazardous materials. 2015-11-15.

Photocopiers emit high levels of nanoparticles (PM0.1). To-date little is known of physicochemical composition of PM0.1 in real workplace settings. Here we perform a comprehensive physicochemical and morphological characterization of PM0.1 and raw materials (toners and paper) at eight commercial pho read more

Identification of glucose-6-phosphate transporter as a key regulator functioning at the autophagy initiation step. (PMID 25982172)

FIGURE 6.6: The first result of PubSeq query.

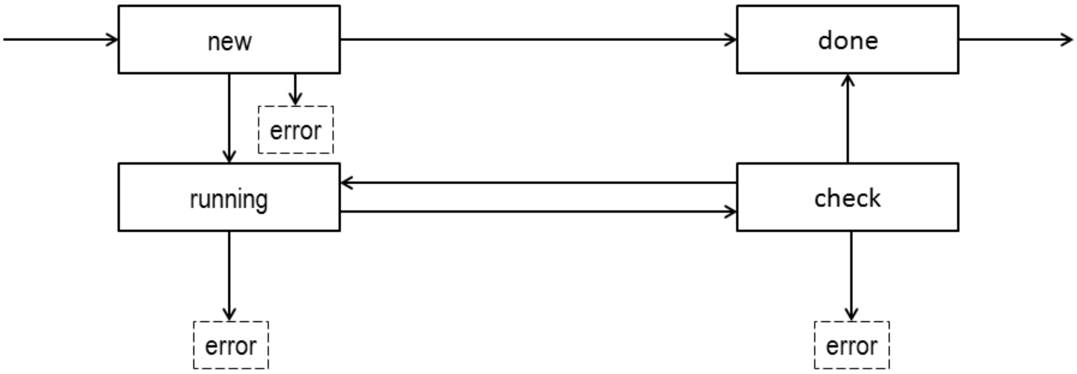


FIGURE 6.7: Diagram depicting the state transitions during PubSeq web query process.

Chapter 7

Results and Validations

Chapter 8

Conclusion and Outlook

Appendix A

PubSeq Paths and Source URLs

A.1 Source Codes

There are several source codes that are relevant to our project:

- `https://rostlab.informatik.tu-muenchen.de/gitlab/gyachdav/pubseq-crawler` contains the tagging ('crawler') pipeline of our project. It also contains several scripts that are used in this project (most notably `annotate_new.sh` and `maintenance.sh`.
- `https://rostlab.informatik.tu-muenchen.de/gitlab/gyachdav/pubseq-frontend` contains the node.js implementation of our project.

A.1.1 Solr's `schema.xml`

The mirror of `schema.xml` that is used in our project can be found at:

`HOME/solr_config/schema.xml`

within 'crawler' repository of the PubSeq project above.

A.1.2 Node.js's `app.js`

Our server's app.js can be found at:

`HOME/pubseq_webapp/app.js`

within 'frontent' repository of the PubSeq project above.

A.2 Project Location

Generally the project directory is located at `/mnt/project/pubseq/` within Rostlab server. The directory is further divided into following categories:

```
- /mnt/project/pubseq
|- index_input_docs
|- log
|- named_entity_tagger
|- programdata
|- rundata
|- solr_index
|- scripts
|- stats
```

The overview of each of directories is as follow:

- `index_input_docs` contains the files that would be indexed onto Solr. In other words, this directory contains all files that were created during Tagging Pipeline. During the last step of Tagging Pipelines, the program `SolrUpdater.jar` would point at this directory and index files that match certain pattern of file name.
- `log` contains logs from Tagging Pipeline.
- `named_entity_tagger` contains the **Lars Tagger component** of the Tagging Pipeline.
- `programdata` contains several program-related tab-separated files that would be used and/or produced during the Tagging Pipeline. Note that program-related data is different from run-related data (which are located at `rundata` in which program-related data remain mostly constant across all runs while run-related data were created during one of the steps within the Tagging Pipeline.
- `rundata` contains interim results from Tagging Pipeline and other Tagging-related data. Since Tagging Pipelines consist of multiple programs with each taking input and writing output, it is inevitable that there would be several interim values. The Tagging Pipeline is designed to write standardized in-between values. While the files would not be removed upon completion of one Tagging Pipeline run, they would be overwritten during the next run. The interim values wouldn't be backed up in some consistent environment.
- `solr_index` contains the Solr directory for the project. For readers who are not familiar with Solr, assuming it as "NoSQL Database" would be sufficient. Solr index is self-contained in the way that the only thing that is needed for it to run properly is its own directory (and JDK).

- `scripts` contain scripts that would be run for various purposes. The most essentials of all scripts are the `annotate_new.sh` and `maintenance.sh`. `annotate_new.sh` wraps the whole Tagging Pipeline process and calls sequentially each component within the pipeline. `maintenance.sh` contains the Maintenance Pipeline and like `annotate_new.sh` calls each component within Maintenance Pipeline sequentially. Both Tagging and Maintenance Pipelines would be further described in later chapters.
- `stats` is deprecated or might not be necessary for the whole process. It contains some descriptive statistics from the last run of Tagging Pipeline. For each Tagging Pipeline, some set of statistic files were created that more or less describe the nature of the run.

Appendix B

Figures and Tables

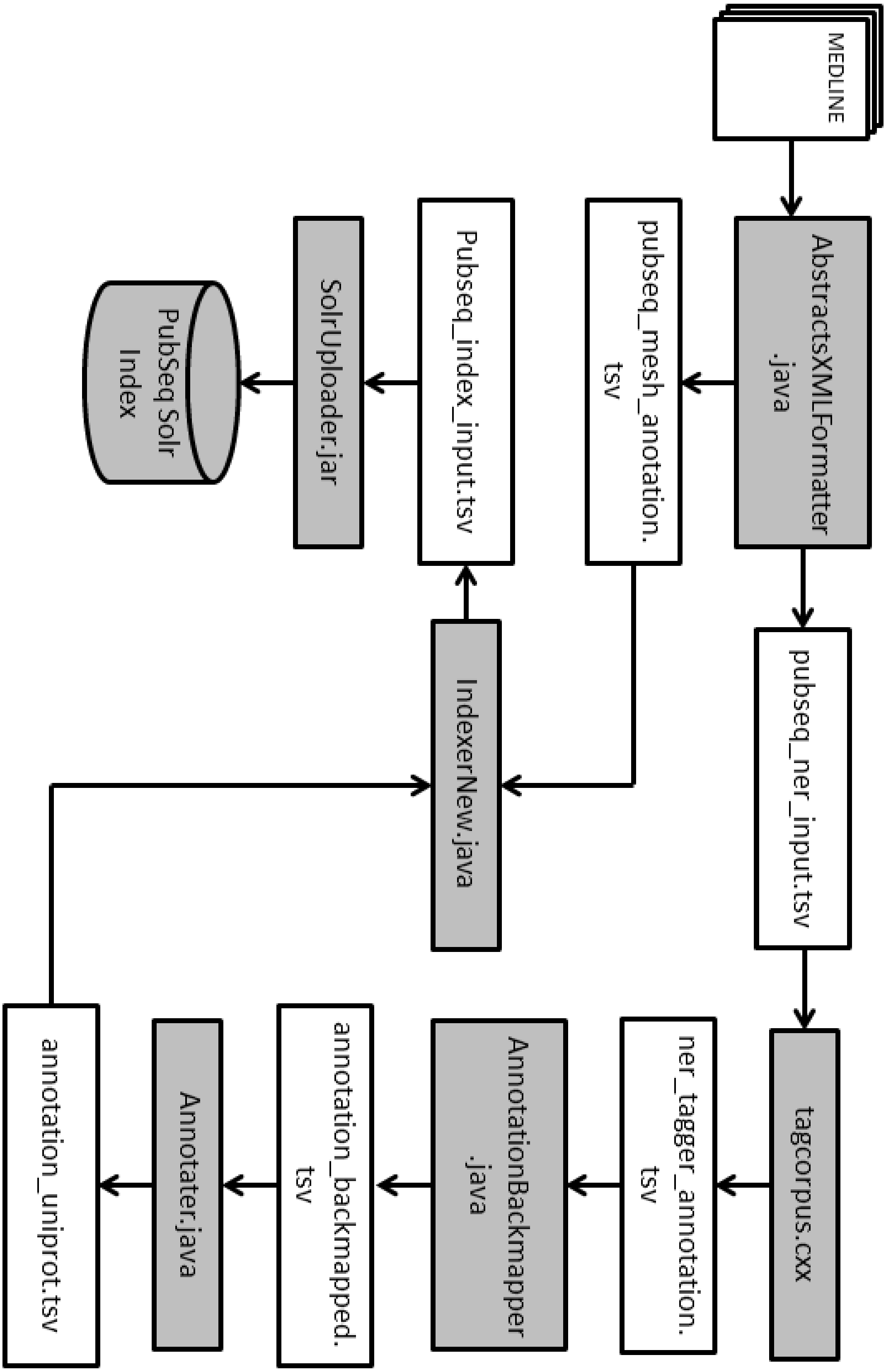

MEDLINE
AbstractsXMLFormatter .java
pubseq_mesh_anotation. tsv
pubseq_ner_input.tsv
tagcorpus.cxx
ner_tagger_annotation. tsv
AnnotationBackmapper .java
annotation_backmapped. tsv
Annotater.java
annotation_uniprot.tsv
IndexerNew.java
Pubseq_index_input.tsv
SolrUploader.jar
PubSeq Solr Index

Figure B.1: Full-sized Overview of PubSeq Tagging Pipeline.

Bibliography

[1] Benedikt Wachinger. *Next Generation Knowledge Extraction from Biomedical Literature with Semantic Big Data Approaches.* PhD thesis, Technische Universität München, 01 2013.

[2] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[3] UniProt Consortium et al. The universal protein resource (uniprot). *Nucleic acids research*, 36(suppl 1):D190–D195, 2008.

[4] UniProt Consortium et al. Ongoing and future developments at the universal protein resource. *Nucleic acids research*, 39(suppl 1):D214–D219, 2011.

[5] The UniProt Consortium. Uniprot citation mapping, 2015.

[6] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[7] Stephen F Altschul and Bruce W Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of mathematical biology*, 48(5-6):603–616, 1986.

[8] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[9] William R Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in enzymology*, 183:63–98, 1990.

[10] Margaret O Dayhoff and Robert M Schwartz. A model of evolutionary change in proteins. In *In Atlas of protein sequence and structure.* Citeseer, 1978.

[11] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[12] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[13] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[14] David W Mount and David W Mount. *Bioinformatics: sequence and genome analysis*, volume 2. Cold spring harbor laboratory press New York:, 2001.

[15] Emil Julius Gumbel and Julius Lieblein. *Statistical theory of extreme values and some practical applications: a series of lectures*, volume 33. US Government Printing Office Washington, 1954.

[16] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[17] Rasko Leinonen, Federico Garcia Diez, David Binns, Wolfgang Fleischmann, Rodrigo Lopez, and Rolf Apweiler. Uniprot archive. *Bioinformatics*, 2004.

[18] Baris E Suzek, Hongzhan Huang, Peter McGarvey, Raja Mazumder, and Cathy H Wu. Uniref: comprehensive and non-redundant uniprot reference clusters. *Bioinformatics*, 23(10):1282–1288, 2007.

[19] U.S. National Library of Medicine. Medline, 2015.

[20] Peder Larsen and Markus Von Ins. The rate of growth in scientific publication and the decline in coverage provided by science citation index. *Scientometrics*, 84(3):575–603, 2010.

[21] U.S. National Library of Medicine. Pubmed web interface, 2015.

[22] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[23] Fernand Meyer and Serge Beucher. Morphological segmentation. *Journal of visual communication and image representation*, 1(1):21–46, 1990.

[24] Lawrence Rabiner and Biing-Hwang Juang. Fundamentals of speech recognition. 1993.

[25] Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing.* MIT press, 1999.

[26] Charles Sutton and Andrew McCallum. An introduction to conditional random fields for relational learning. *Introduction to statistical relational learning*, pages 93–128, 2006.

[27] Steven L Salzberg, Arthur L Delcher, Simon Kasif, and Owen White. Microbial gene identification using interpolated markov models. *Nucleic acids research*, 26(2):544–548, 1998.

[28] ChristopherB Burge. Modeling dependencies in pre-mrna splicing signals. *New Comprehensive Biochemistry*, 32:129–164, 1998.

[29] Ralph Grishman and Beth Sundheim. Message understanding conference-6: A brief history. In *COLING*, volume 96, pages 466–471, 1996.

[30] Daniel M Bikel, Scott Miller, Richard Schwartz, and Ralph Weischedel. Nymble: a high-performance learning name-finder. In *Proceedings of the fifth conference on Applied natural language processing*, pages 194–201. Association for Computational Linguistics, 1997.

[31] Satoshi Sekine, Ralph Grishman, and Hiroyuki Shinnou. A decision tree method for finding and classifying names in japanese texts. In *Proceedings of the Sixth Workshop on Very Large Corpora*, 1998.

[32] Andrew Borthwick, John Sterling, Eugene Agichtein, and Ralph Grishman. Exploiting diverse knowledge sources via maximum entropy in named entity recognition. In *Proc. of the Sixth Workshop on Very Large Corpora*, volume 182, 1998.

[33] Masayuki Asahara and Yuji Matsumoto. Japanese named entity extraction with redundant morphological analysis. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 8–15. Association for Computational Linguistics, 2003.

[34] Andrew McCallum and Wei Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 188–191. Association for Computational Linguistics, 2003.

[35] John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[36] Robert Leaman, Rezarta Islamaj Doğan, and Zhiyong Lu. Dnorm: disease name normalization with pairwise learning to rank. *Bioinformatics*, page btt474, 2013.

[37] Allan Peter Davis, Thomas C Wiegers, Michael C Rosenstein, and Carolyn J Mattingly. Medic: a practical disease vocabulary used at the comparative toxicogenomics database. *Database*, 2012:bar065, 2012.

[38] Robert Leaman, Graciela Gonzalez, et al. Banner: an executable survey of advances in biomedical named entity recognition. In *Pacific Symposium on Biocomputing*, volume 13, pages 652–663. World Scientific, 2008.

[39] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[40] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142. ACM, 2002.

[41] Larry Smith, Lorraine K Tanabe, Rie J Ando, Cheng-Ju Kuo, I-Fang Chung, Chun-Nan Hsu, Yu-Shi Lin, Roman Klinger, Christoph M Friedrich, Kuzman Ganchev, et al. Overview of biocreative ii gene mention recognition. *Genome biology*, 9(Suppl 2):S2, 2008.

[42] Evangelos Pafilis, Sune P Frankild, Lucia Fanini, Sarah Faulwetter, Christina Pavloudi, Aikaterini Vasileiadou, Christos Arvanitidis, and Lars Juhl Jensen. The species and organisms resources for fast and accurate identification of taxonomic names in text. *PloS one*, 8(6):e65390, 2013.

[43] Damian Szklarczyk, Andrea Franceschini, Michael Kuhn, Milan Simonovic, Alexander Roth, Pablo Minguez, Tobias Doerks, Manuel Stark, Jean Muller, Peer Bork, et al. The string database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic acids research*, 39(suppl 1):D561–D568, 2011.

[44] Jörg Hakenberg, Martin Gerner, Maximilian Haeussler, Illés Solt, Conrad Plake, Michael Schroeder, Graciela Gonzalez, Goran Nenadic, and Casey M Bergman. The gnat library for local and remote gene mention normalization. *Bioinformatics*, 27(19):2769–2771, 2011.

[45] Burr Settles. Abner: an open source tool for automatically tagging genes, proteins and other entity names in text. *Bioinformatics*, 21(14):3191–3192, 2005.

[46] Martin Gerner, Goran Nenadic, and Casey M Bergman. Linnaeus: a species name identification system for biomedical literature. *BMC bioinformatics*, 11(1):85, 2010.

[47] David Smiley, Eric Pugh, Kranti Parisa, and Matt Mitchell. *Apache Solr Enterprise Search Server.* Packt Publishing Ltd, 2015.

[48] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The.* Pearson Higher Education, 2004.

[49] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[50] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, (6):80–83, 2010.

[51] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

[52] Michael S Keller. Take command: cron: Job scheduler. *Linux Journal*, 1999(65es):15, 1999.

[53] Trey Grainger, Timothy Potter, and Yonik Seeley. *Solr in action.* Manning, 2014.

[54] U.S. National Library of Medicine. Medline data type definition (dtd), 2015.

[55] Tim Bray, Jean Paoli, CM Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0, 2011.

[56] Henry J Lowe and G Octo Barnett. Understanding and using the medical subject headings (mesh) vocabulary to perform literature searches. *Jama*, 271(14):1103–1108, 1994.

[57] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[58] Scott Federhen. The ncbi taxonomy database. *Nucleic acids research*, 40(D1):D136–D143, 2012.

[59] T Hubbard, Daniel Barker, Ewan Birney, Graham Cameron, Yuan Chen, L Clark, Tony Cox, J Cuff, Val Curwen, Thomas Down, et al. The ensembl genome database project. *Nucleic acids research*, 30(1):38–41, 2002.

[60] Daniel Lawson, Peter Arensburger, Peter Atkinson, Nora J Besansky, Robert V Bruggner, Ryan Butler, Kathryn S Campbell, George K Christophides, Scott Christley, Emmanuel Dialynas, et al. Vectorbase: a data resource for invertebrate vector genomics. *Nucleic acids research*, 37(suppl 1):D583–D587, 2009.

[61] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 147–155. Association for Computational Linguistics, 2009.

[62] Erik Hatcher, Otis Gospodnetic, and Michael McCandless. Lucene in action, 2004.

[63] Daniel Tunkelang. Faceted search. *Synthesis lectures on information concepts, retrieval, and services*, 1(1):1–80, 2009.

[64] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[65] Eclipse Foundation. Jetty - servlet engine and http server - eclipse, 2015.

[66] The Apache Software Foundation. Apache tomcat, 2015.

[67] Samuel Weiler, Dave Ward, and Russ Housley. The rsync uri scheme. *RFC5781, February*, 2010.

[68] Ryan Dahl and Joyent. Node.js, 2015.

[69] Google Codes. V8 javascript engine, 2015.

[70] Douglas Christopher Wilson. Express – node.js web application framework, 2015.

[71] Aaron Stannard. Intro to node.js for .net developers, 2011.

[72] Mark Otto and Jacob Thornton. Bootstrap, 2015.