Facilitating Clinical Phenotype Development at Scale:

Optimizing the ClarityNLP Platform Using Clinical Trial Data

Christine Herlihy
Georgia Institute of Technology
Atlanta, Georgia
cherlihy@gatech.edu

Charity Hilton
Georgia Institute of Technology
Atlanta, Georgia
cah@gatech.edu

1. Introduction

Our primary objective in this project is to optimize ClarityNLP, an existing open-source platform for computational phenotyping, such that structured queries containing patient-level clinical selection criteria can be run more rapidly against a given database, without sacrificing accuracy [13]. Such optimization(s) will reduce the amount of computational resources required by clinical researchers seeking to select a subset of patients meeting specific criteria from large databases containing a variety of structured and unstructured fields.

This paper proceeds as follows: in Section 2, we outline the challenges facing clinical researchers seeking to use unstructured clinical notes for feature engineering and/or patient selection, and provide a brief discussion of how ClarityNLP addresses many of these problems but requires modifications to do so at scale. In Section 3, we discuss the relevance of computational phenotyping to clinical research, and provide an overview of the domain-specific benefits of reducing runtime and/or improving system throughput.

In Section 4, we provide background information on ClarityNLP and its associated structured query language, NLPQL, as well as on MIMIC-III, the de-identified patient-level dataset we will use. In Section 5, we provide an overview of relevant related works. In Section 6, we outline our methodological approach to generating a set of synthetic NLPQL queries whose component clauses will be parameterized in a way that allows their distribution to roughly approximate the distribution of clinical trial inclusion criteria found in the biomedical literature, albeit with adjustments made to ensure that patients matching the criteria produced are likely to be present in the MIMIC-III dataset with sufficient frequency so as to constitute a non-trivial computational load on the system. In addition, we discuss the set of optimizations we have investigated and implemented in an effort to reduce expected runtime.

In Section 7-9, we outline the set of experiments we conducted against the MIMIC-III dataset to evaluate the impact of our optimizations on expected runtime, holding hardware specs, synthetic query corpus, and database constant [16]. In Section 10, we outline our thoughts on how this project might be expanded in future work.

2. Problem Statement

Phenotypes are measurable criteria for classifying a specific illness or condition. Computational phenotyping refers to the development of algorithms to identify patients with particular observable traits associated with a disease or set of diseases from a broader clinical population [24, 30]. Our efforts in this work are directed at optimizing ClarityNLP, an existing open-source platform for clinical phenotyping, to reduce the expected runtime associated with complex queries and improve system throughput.

The proper baseline against which all computational approaches to phenotype development must be compared is manual chart review, which is the traditional approach to the identification of patients matching specific inclusion criteria. Manual chart review must be conducted by researchers with clinical expertise/training (e.g., physicians, nurses, radiologists, etc.), and is thus a resource-intensive process that can take approximately 35-40 minutes, depending on the complexity of the record(s) and criteria involved [28].

Historically, researchers have relied on rule-based, heuristic approaches and leveraged structured data (e.g., diagnosis codes; medications; quantitative and/or Boolean lab results; etc.) found in electronic medical records to define a set of inclusion and/or exclusion criteria, which they can then use to extract their patient population(s) of interest.

However, in recent years, the marked increase in available computational power, coupled with the rise of electronic medical record (EMR) systems, and the development of increasingly sophisticated neural network architectures, has sparked an interest in semi-supervised and unsupervised approaches to phenotype development, leveraging both structured and unstructured data, including claims data, as well as clinical notes and genomic data.

In addition, researchers are often interested in traits and/or behavioral determinants of health that are unlikely to explicitly appear in structured data (which is typically optimized for billing purposes rather than for clinical research), but may be discoverable and/or extractable when natural language processing (NLP) methods and/or unsupervised learning techniques are applied to unstructured clinical notes.

To this end, our team at Georgia Tech Research Institute has developed ClarityNLP, an open-source platform intended to help researchers apply NLP methods to unstructured clinical notes data and select patients for inclusion in a given

cohort based on membership in phenotypic sets [13]. The primary focus during the initial development phase of ClarityNLP has been on delivering expected functionality and implementing core NLP algorithms; as such, modifications to the system architecture are required if ClarityNLP is to be used to efficiently execute increasingly complex queries over large datasets at scale, and/or in a near real-time environment.

3. Relevance

Phenotype development is an important step in many clinical research work-flows, including: (1) feature engineering for the development of precision-medicine models for a particular disease or set of related diseases; (2) identification of patients who are eligible to participate in clinical trials, which represents a critical step in the research and development life-cycle for pharmaceutical and medical device companies as well as regulatory agencies; and (3) genomic research aimed at identifying and/or understanding the root cause of many different diagnoses, so as to develop appropriate treatment options.

To elaborate on (2), clinical trial studies rely on phenotype development and patient identification not only for the purpose of recruitment, but also to evaluate comparator cohort studies, and/or conduct retrospective research studies based on observational data. These studies have gained in popularity because of their affordability and overall benefit to patient safety [29].

Hospital systems and regulatory agencies also benefit from patient identification methods for the purpose of assessing quality of care, and/or enforcing regulations intended to protect patient safety. Health care systems are incentivized by federal agencies to report and improve their performance on patient-level quality-of-care metrics [19]. Regulatory criteria for quality can be established by a local hospital system, state and/or federal law, a national medical association or a federal agency. Each patient case must be identified and compared against case criteria as defined by these regulatory bodies.

Phenotype development is an inherently interdisciplinary task, where clinical researchers with domain expertise work with computer scientists and biostatisticians to develop and refine selection criteria. One of the challenges is that medical constraints must be translated into machine-interpretable code. This task becomes even more difficult when unstructured text must be mined in an effort to extract socioeconomic and/or behavioral determinants of health, as unstructured datasets typically do not have any labeled ground-truth instances, which means that validation often requires physicians to conduct manual chart review. In such instances, the availability and affordability of physician labor become key bottlenecks with the potential to derail and/or delay otherwise promising research.

Our efforts to optimize ClarityNLP, reduce the computational cost associated with running complex queries and/or against large datasets, and allow the platform architecture to take advantage of multiple cores when possible, are an important step toward reducing the burden on clinical research teams and accelerating the pace at which the research these teams conduct can benefit patients.

4. Background Information

4.1 ClarityNLP

ClarityNLP is an open source, collaborative platform for clinical phenotyping using natural language processing. There are five principal components to ClarityNLP to prepare and analyze clinical documents for phenotype identifications:

1. Document Ingestion
2. Data Mapping
3. NLPQL and Data Processing
4. Results and Validation
5. Sharing and Portability

ClarityNLP combines various NLP algorithms to query and parse data from unstructured clinical text. ClarityNLP is centralized around NLPQL, a custom query language based on CQL (clinical query language)[7]. NLPQL allows users to map documents, term sets, and algorithms, and combine them with logical operations to build complex, shareable clinical phenotypes.

ClarityNLP was built at the Georgia Tech Research Institute, using a Python stack. ClarityNLP incorporates several libraries, including Luigi (for job and task management), spaCy and NLTK (for NLP APIs), and Antlr (for query parsing). ClarityNLP stores text documents in Solr and stores phenotype results in MongoDB. Figure 1 illustrates ClarityNLP's system architecture.

To date, ClarityNLP's software development team's efforts have been focused on: (1) addition of algorithms for extracting specific clinical information (such as extracting medications); (2) addition of machine learning models to extract and classify clinical text; and (3) overall system architecture design and development. Our project focuses on NLPQL and optimization of queries associated with the phenotype development lifecycle. NLPQL is discussed in more detail in the next section.

4.2 NLPQL

ClarityNLP allows users to find data or patients of interest from unstructured clinical text. Users create a description of a desired resulting phenotype or data set with the declarative language NLPQL. An NLPQL "program" is a simple text file containing NLPQL statements. This text file is essentially a definition (called a phenotype) of what the user wants to find, expressed in a portable language.

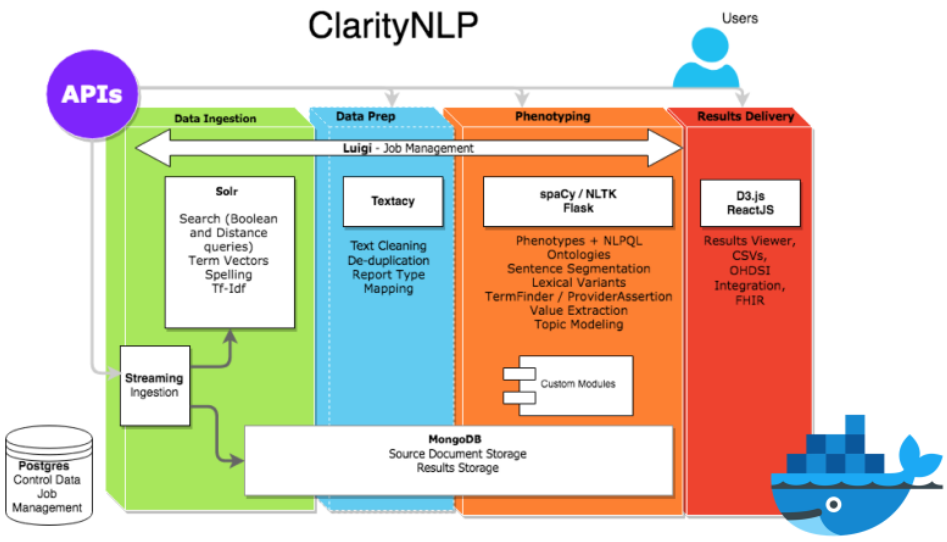

Figure 1. ClarityNLP System Architecture

The purpose of NLPQL is to create a common language around clinical phenotyping using unstructured data. Experts can agree on the definition of a particular phenotype and express the definition in NLPQL. Different institutions can then use ClarityNLP with an identical NLPQL file to extract this phenotype from their own proprietary or access-restricted data sets.

NLPQL has four primary components: (1) documents sets (or document selection); (2) term sets (or a dictionary of term); (3) context; (4) data entities (NLP algorithms, e.g. an algorithm to extract tumor sizes); and (5) operations (logic operations and comparisons across data entities). NLPQL can be executed in the user interface or via RESTful API in ClarityNLP. Execution is distributed across parallel workers using Luigi. More detail on NLPQL is discussed in the ClarityNLP documentation [14]. An example is shown in Appendix A.

4.3 MIMIC-III Critical Care Database

The MIMIC-III Critical Care Database is data set containing information about ICU patients who stayed at the Beth Israel Deaconess Hospital between 2001 and 2012. MIMIC-III has multiple types of clinical observational data, including text. It has been de-identified for research use. Unlike most public clinical data sets, unstructured text is available in this data set, making it suitable for clinical NLP research [11, 16].

5. Related Work

There are many existing general-purpose NLP libraries and platforms, as well as domain-specific clinical decision support tools, that are designed to process raw text and extract information. Examples of general purpose NLP libraries include Gensim, spaCy, the Natural Language Toolkit (NLTK), Apache OpenNLP and Stanford CoreNLP [1, 2, 10, 21, 23]. In addition, there are general NLP libraries for executing NLP pipelines, such as Unstructured Information Management Architecture (UIMA) and General Architecture for Text Engineering (GATE)[6, 8].

Examples from the clinical domain include the Apache Clinical Text Analysis Knowledge Extraction System (cTAKES), Clinical Language Annotation, Modeling, and Processing Toolkit (CLAMP) and NOBLE[5, 25, 26].

cTakes is an open-source clinical NLP platform. cTakes has modules for extracting terms, negation, time expressions and others. cTakes pipelines run all documents and do not support logical expressions. CLAMP has multiple modules for extracting clinical features from text, and provides a layer of customization, using an integrated development environment (IDE) and XML files. CLAMP also has user tools for viewing and annotating results. NOBLE is a general concept extraction algorithm that runs on top of UIMA or GATE.

The primary difference between ClarityNLP and these libraries is NLPQL, which is intended to be a expert-friendly language for building phenotypes, which also provides another layer of customizability. ClarityNLP can target specific documents, concept sets, and algorithms and combine them to form complex phenotypes for patients. ClarityNLP also focuses on technical flexibility in several ways: (1) ClarityNLP allows users to connect their own custom algorithms or external web APIs to the ClarityNLP pipeline and NLPQL queries; (2) ClarityNLP provides an end-to-end setup for storing source documents and results, and as such, provides a

solution for new and less-technically savvy users, including clinical domain experts.

6. Approach

To develop and evaluate a set of system-level optimizations to the ClarityNLP platform, we began by generating a representative set of synthetic NLPQL queries, where the distribution of various inclusion criteria primitives roughly approximates the actual distribution of such criteria within the biomedical literature. To establish a reproducible **baseline**, we used a pre-specified sha from the master branch of ClarityNLP (e.g., the platform without any query optimizations, cascading classifiers, or caching) to run this corpus of synthetic queries against the MIMIC-III dataset, and computed summary statistics on the resulting query-level runtimes.

With this baseline established, we forked the main ClarityNLP repository and implemented a variety of optimizations to the underlying system architecture; these optimizations are discussed in greater detail in subsection 3. We then ran our synthetic queries against MIMIC-III using different, optimized configurations of ClarityNLP, computed summary statistics, and assessed improvements in runtime relative to baseline for each optimized configuration observed.

6.1 Synthetic Query Generation

While historical query term and/or clause co-occurrence patterns would generally represent a logical foundation for developing schema and/or system architecture-based optimizations, ClarityNLP is a relatively new platform, and as such, no representative corpus of historical queries exists. We have adopted a hybrid approach to proxy for such a corpus: we begin by extracting metadata and inclusion criteria from the Aggregate Analysis of ClinicalTrials.gov (AACT) Database for all trials that contain explicitly segmented inclusion criteria ($n = 249,493$) [12].

There is no standard or required format (tabular or otherwise) for enumerating clinical trial inclusion and/or exclusion criteria; as such, the inclusion criteria we extract from the AACT are typically represented as free text, and/or bulleted lists. There is considerable heterogeneity with respect to the cardinality and restrictiveness of the criteria sets encountered: while some trials impose a limited number of high-level restrictions (e.g., gender, wide age range, and a general diagnosis requirement), others contain a lengthy set of specific restrictions, (e.g., multiple diagnoses are required, each of which may be associated with a set of embedded requirements, such as scores within a specified range).

As our primary aim in synthetic query generation is ensuring the benchmarks we develop and co-occurrence patterns we identify are representative of real-world workloads, we consider the development of a robust natural language inclusion criteria-to-NLPQL parser to be a subject warranting future work, and constrain our efforts here to the development and detection of a set of inclusion criteria primitives, ICP, such that $ICP \subsetneq IC$.

We manually reviewed a random subset of the AACT criteria, and developed an initial set, ICP_i : {gender; age; race; location; condition(s); medication(s)}. To ensure that the synthetic queries we generate will, on average, return results rather than $\emptyset$, we computed summary statistics of the MIMIC-III patient population, and used the resulting distribution over sociodemographic categories, as well as clinical conditions and medications, to inform selection and specification of our final inclusion criteria primitives, ICP_f: {gender; race; condition(s); medication(s)}.

We opted to remove the age and location primitives because: (1) age is de-identified in MIMIC-III, yielding birth dates that are set in the future and an offset that is (intentionally) neither derivable nor consistent across patients; and (2) all of the MIMIC-III patient records come from a single location (e.g., the Beth Israel Deaconess Hospital, located in Boston, Massachusetts). The **gender** primitive is composed of two possible criteria: {male; female}; The **race** primitive is composed of five possible criteria: {Asian; Black; Native American; Pacific Islander; White}.

To create the **conditions** and **medications** primitives, we: (1) computed the 10 most commonly occurring conditions and medications within the MIMIC-III patient population; (2) ran each of these terms through ClarityNLP's termset-expander function, so that synonyms for these conditions and medications that are common within the clinical domain will also be detected; and (3) represented each term or group of expanded terms with a common root/ancestor as a bit in the final conditions and medications primitives.

To ensure our experimental runs would be tractable given time and resource constraints, we used the primitives outlined above to generate a synthetic corpus of size 10,000, and then randomly selected 100 NLPQL queries from this corpus.

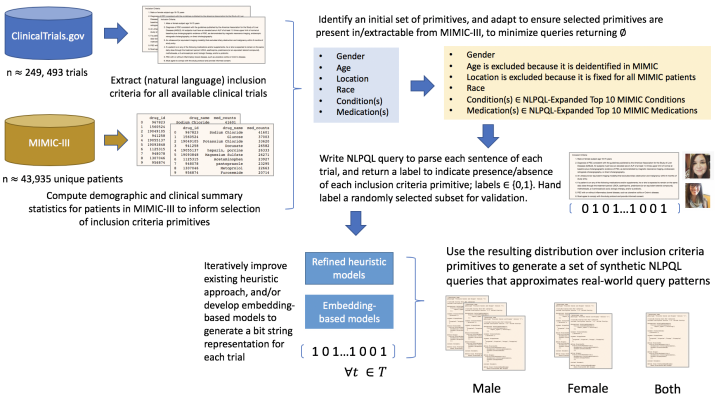

Figure 2. Synthetic Query Generation Process

6.2 Optimizations Implemented

We proposed and implemented two sets of optimizations, segmented into Tier I and Tier II. Tier I optimizations represent foundational improvements that we expected to result in improved runtime, while simultaneously laying the groundwork for more computationally sophisticated, model-based optimizations. Tier II optimizations required the application of supervised and/or unsupervised learning techniques, and the potential payoff associated with these techniques was less clear *prima facie*.

We were able to implement the majority of our proposed optimizations (Tier I: 4 of 6; Tier II: 3 of 4). In some cases, implementation-related results (e.g., accuracy, frequency statistics, etc.) indicated that integrating the optimization into the platform would be unlikely to result in significant speedup relative to baseline, and thus, a subset of the optimizations we implemented were not ultimately integrated. Our proposed implementations are described below, and final implementation/integration status is provided for each.

6.2.1 Tier I Optimizations

As Figure 3 illustrates, we were able to implement 4 of our 6 proposed Tier I optimizations; additional information regarding implementation is below:

	Optimization	Implemented?	Integrated?
1	Allow query chaining	1	1
2	Implement least recently used (LRU) caching	1	1
3	Reorder NLPQL clauses based on downselection potential	1	1
4	Create additional indices	0	0
5	Shift computation closer to data	0	0
6	Precompute and index common NLP tasks in Solr	1	1

Figure 3. Tier I Optimizations

1. **Allow query chaining:**
   a. This optimization was intended to facilitate the chaining of queries, so that the results of query_1 could be used as input to query_2. To implement it, we initially developed a method to limit the NLPQL query space as a filter based on previous executed results.
   b. The filter can be grouped by: (1) patient id (e.g., limit query_2 to patients found in query_1); or (2) by document id (limit query_2 to documents found in query_1). In order to integrate this optimization into ClarityNLP, we created a pipeline that dynamically generates a document filter based on the contents of query_1; this filter is then used as an input to query_2.
2. **Implement least recently used (LRU) caching:**
   a. LRU caches are memory-based caches which keep track of access times for each object inserted in the cache. They have a maximum size parameter, and as new items are added to the cache, the least recently used objects are pushed out. LRU caches have native support in Python [9]. We implemented LRU caches via the *cachetools* library. We cached documents (which have additional metadata and used from job to job), algorithm modules (which are initialized with models, dictionaries and regular expressions), and ClarityNLP results (which are the primary output generated by experiments.
   b. It should be noted that our synthetic query corpus is particularly well-suited to benefit from caching, as our synthetic queries are composed only of our identified set of query primitives. A logical next step for future work would be to add noise to the query generation process (e.g., to generate queries containing both primitives, as well as non-primitive components) to assess the benefits of caching given a more realistic corpus, and/or an engineered change in the distribution of terms used to construct queries.
3. **Reorder NLPQL operations based on down-selection:**
   a. NLPQL queries generally consist of multiple component clauses; some of these clauses represent constraints on the set of documents that will be returned (e.g., they may require a certain set of terms to be present, or require the patient in question to have a certain diagnosis, etc.).
   b. The end user cannot be reasonably expected to write queries in a sequentially optimal way (e.g., with respect to ordering the clauses in such a way that the filter clause with highest down-selection potential is run first, so as to minimize the state space over which all subsequent computations must be run).
   c. Thus, in this optimization, our objective was to: (1) compute the down-selection potential of each of our query primitives, and (2) design a pipeline step that would ingest a synthetic query, detect all primitive(s) present, and re-order the query to ensure optimal sequential execution of each component clause with respect to down-selection.
   d. To determine down-selection potential, we identified the structured data field(s) and/or diagnoses codes that corresponded to each of our query primitives in the structured data component of the MIMIC-III dataset. We then wrote SQL queries to extract all unique patient ids (for demographic information), condition occurrences (for conditions), and drug exposures (for medications) matching each set of structured field(s). Next, we computed the down-selection potential of each primitive by calculating $\frac{|\text{total\_population}|-|\text{matches}|}{|\text{total\_population}|}$, and used the resulting percentages to rank each primitive. We used these ranks

to optimally re-order the sub-clauses within each synthetic query that we parsed; the associated data is available in Appendix C.

4. **Create additional indices for ClarityNLP results:**
   a. ClarityNLP evaluates results from two primary data stores: Solr and MongoDB. In addition, ClarityNLP stores job and control information in a PostgreSQL database. Most Solr queries run in ClarityNLP (observed to-date) return in less than 100 milliseconds, primarily due to Solr's inverted index. By default, ClarityNLP has 8 indexes in the MongoDB data stores. In addition, ClarityNLP has indexes on the PostgreSQL database to guarantee uniqueness and referential integrity.
   b. After reviewing ClarityNLP's reads and writes, we did not observe any missing indexes in ClarityNLP's baseline configuration. We thus chose to focus our efforts on alternate optimizations.
5. **Shift computation closer to data sources instead of loading in-memory on Luigi engine:**
   a. Data locality optimizations have been part of data and systems research since the 1990s. Given that ClarityNLP runs on a virtual machine (VM) architecture, it has less low-level control over data locality. However, ClarityNLP is still able to benefit from higher-level optimizations related to its data stores, specifically MongoDB and Solr.
   b. There is an ongoing effort within the main ClarityNLP project to move logical and mathematical expressions from a Python-based evaluation, using *pandas*, to MongoDB using its built-in *aggregate* functionality. We intend to leverage the experimental framework and synthetic query corpus presented here to evaluate the effectiveness of this modification in future work; however, at present, we elected to focus our efforts on Solr-specific data locality optimizations.
6. **Pre-compute & index common NLP tasks in Solr:**
   a. Our primary optimization in this area is related to data locality as discussed above, but also involves pre-computation of CPU-intensive tasks. Nearly every algorithm implemented in ClarityNLP requires a step that involves document segmentation, either into sentences, and/or into clinically relevant sections. These algorithms have been implemented using a combination of custom libraries and popular open-source NLP libraries, including *spaCy*. The result of this type of segmentation task is static provided that the underlying document(s) parsed do not change. As such, this step in the pipeline is a strong candidate for pre-computation and indexing.
   b. To implement this optimization, we: (1) iterated over each document in our Solr index; (2) ran segmentation by sentence and by section; and (3) stored the resulting arrays in Solr, reducing the amount of computation necessary for each job.

6.2.2 Tier II Optimizations

As Figure 4 illustrates, we were able to explore three and integrate one of our four proposed Tier II optimizations; additional information regarding implementation is below:

	Optimization	Implemented?	Integrated?
1	Cache of commonly occurring query predicates	1	0
2	Train DNNs to compute query primitive results	1	0
3	Train a model to dynamically assign optimal number of Luigi workers and/or Luigi batch size	0	0
4	Use in-memory database for active/recent jobs	1	1

Figure 4. Tier II Optimizations

1. **Cache commonly co-occurring predicates' results:**
   a. We were motivated to consider this optimization because we hoped that by storing the results associated with commonly co-occurring query predicates, we could reduce the computation in all queries that included this each identified subset of terms, and use these cached results as building blocks to efficiently construct more complex queries.
   b. The assumption that certain query predicates would be more likely to co-occur than others is a realistic assumption within the domain of clinical research, as certain predicates are common across the board (e.g. gender, age, etc.), while others are causally linked (e.g., certain medications are prescribed to treat certain conditions; certain conditions are more frequent among specific sub-populations, etc.).
   c. To compute the co-occurrence frequency of each of our query primitives, we constructed a matrix, A, by stacking the Boolean-valued feature vectors associated with each synthetic NLPQL query that we generated. Specifically, row_i in A represents the feature vector associated with query_i, and each element in this row is either 0 or 1 depending on whether the associated primitive is absent or present in query_i. We then computed $A^T A$.
   d. When we examined the resulting co-occurrence matrix, we found that the majority of query primitives co-occurred with very low frequency, and only a small handful of primitives (male, female, glucose) had significant co-occurrence values; Appendix D provides a heat-map representation of this co-occurrence

matrix. Ultimately, we determined that this optimization was unlikely to yield significant speedup, and focused our attention on alternate optimizations.

e. Given that we synthetically generated our query corpus, it would, of course, be possible to synthetically generate an alternate corpus and adjust the probability of inclusion associated with each primitive, and/or introduce dependencies, in an effort to force higher levels of co-occurrence; we intend to pursue this line of inquiry in future work.

2. **Train DNNs to compute primitive-level results**
   a. Our motivation in this optimization was to develop a series of primitive-level deep neural networks capable of ingesting the bag-of-words representation of an NLPQL query and outputting a Boolean label $\in \{0, 1\}$ intended to represent the absence/presence of the primitive in question within the query [17, 20].

   b. To implement this optimization, we ran our single-primitive NLPQL queries over 51,155 notes, to generate primitive-level truth values, with a 1 corresponding to the presence of the primitive within a query, and a 0 corresponding to absence. We then selected a random sample of 20,000 unique notes from our initial set of 51,155 notes, and used scikit-learn's `CountVectorizer` to produce a document-term matrix, in which each row represents a note, each column represents a unigram, and each element in the matrix represents the count (e.g., number of times that the term represented by column j appears in the document represented by row i) [3, 22]. We merged this document-term matrix with our labels matrix on `report_id`.

   c. To train our primitive-level DNNs, we used 3-fold stratified splits for cross-validation, scaled our input document-term vectors, and then used scikit-learn's `imblearn` package, which includes a synthetic minority oversampling function (SMOTE) to attempt to address the class imbalance of our dataset [3, 4, 22]. Specifically, a small set of primitives are over-represented within our synthetic query corpus (as they are more frequently occurring in our empirical data sources); consequently, the chance that a given query will contain a less-commonly-occurring primitive is relatively low, and the distribution of truth labels for each primitive is thus heavily skewed toward 0.

   d. Our network architecture was relatively simple: it included 3 hidden layers of size 128, 64, and 10 with a relu activation function and a sigmoid outer layer to produce binary labels.

   e. We saw significant variation in performance among different primitives, as well as with respect to different performance metrics. As we are mostly concerned with using DNNs for down-selection and/or query optimization purposes, it is important that we are able to catch true positives, even if doing so requires us to accept a certain number of false positives, as this smaller dataset of potential positives can be run through the more expensive NLPQL pipeline.

   f. Time and resource constraints prevented us from exploring a broader range of network architectures. While we did find that we were able to obtain relatively high recall by accepting a larger number of false positives, we were unable to design a network that struck a computationally acceptable balance (e.g., to be useful for down-selection purposes, the DNNs must offer a significant reduction in cardinality relative to the size of the unfiltered dataset).

   g. Results were particularly poor for primitives with very few positive examples in the dataset. In future work, we plan to continue iterating over the network architecture design space and leverage the precision-recall curve to select a more optimal 0-1 cutoff threshold. Since performance of our DNNs was not satisfactory enough to result in useful speedup, we did not integrate these DNNs into the ClarityNLP pipeline; however, average results over the 3 rounds of stratified cross validation are presented in Appendix F.

3. **Use ML to optimize Luigi parameters:**
   a. There are two primary configuration parameters that can be tweaked to modify the way that Luigi executes jobs: batch size, and maximum worker count. ClarityNLP interacts with the Luigi engine by: (1) retrieving the corpus size (e.g., the number of documents); (2) bundling this corpus into tasks, where the number of tasks is found by dividing the size of the corpus by the value of the batch size parameter; and (3) assigning tasks to Luigi workers. Luigi allows jobs up to and including the maximum worker count to run; all other tasks are queued. The maximum worker count is CPU-bound.

   b. In theory, it is possible to evaluate the impact of Luigi's configuration parameters on ClarityNLP's runtime performance by running the same set of NLPQL queries against ClarityNLP instances featuring different combinations of batch size and Luigi workers.

   c. We did explore this relationship by modifying batch size and/or worker count in some of our optimization configurations; however, we did not build a model

for this task. We found that modifying batch size did have an impact on performance as discussed in Section 8.

4. **Use an in-memory database for active/recent jobs:**
   a. Redis is a popular in-memory key-value store. It can be used with greater persistence as a database, or more simply as a cache or message broker. Redis can support a variety of data types, such as strings, hashes, lists, sets, and sorted sets. It also has atomic operations for capturing counts and metrics [18].
   b. We used Redis primarily as cache, caching the same types of ClarityNLP data, as described in our LRU cache implementation. While, it is implemented slightly differently and has differences in performance. Redis and the LRU cache in our experiments are roughly interchangeable. In the case where both are enabled, Redis is preferred.

6.3 Running Experiments

Since ClarityNLP is centered around Docker, we created a Docker environment configuration for each optimization. We configured ClarityNLP to capture the configuration information in the PostgreSQL database during each job execution, for our validation and evaluation. Every optimization we developed was enabled or disabled via a Boolean flag in the environment file. We then created a script to move each environment file from a directory into ClarityNLP, where we ran each NLPQL job based on the active environment configuration.

7. Validation

Our primary validation task in this project was to assess the extent to which the results returned when our synthetic queries are run against our various optimized configurations of ClarityNLP match the baseline results.

	prop_name	accuracy
0	all_except_chained	0.960000
1	cache	0.960000
2	precomp_reorder_redis	0.960000
3	redis_cache	0.960000
4	luigi_workers5	0.959184
5	cache_reorder_nlpql	0.950000
6	reorder_nlpql	0.930000
7	precomp_nlpql	0.927083
8	caching_precomp_reorder	0.920000
9	chained_queries	0.690000
10	batch_size10	0.000000
11	batch_size100	0.000000
12	reorder_redis	0.000000

Figure 5. Validation of Results: Optimized vs. Baseline

Our primary observations from validating our optimizations against the baseline:

a. Most executions of NLPQL queries return similar results. Most of the differences are due to missing job results. This should be studied further.
b. Chaining queries introduced several bugs into ClarityNLP. This optimization should be fully re-implemented before it can be validated against ClarityNLP's baseline.
c. Because batch sizes affect the Solr query by changing the limit, the Solr sort becomes unpredictable. Changing batch sizes returns a similar number of results, but they are not the same exact results.

8. Evaluation

Our primary evaluation criteria that we used to assess the effectiveness of our optimizations (both standalone, and in combination) is expected runtime (e.g., average runtime over all synthetic queries, where $n = 96$) relative to the baseline configuration. We find that our optimizations generally resulted in significant speedup relative to baseline, with a few pointed exceptions.

More specifically, we tested 14 possible combinations (1 which represents the baseline, and 13 which represent configurations which contain either a single optimization or a combination of optimizations); our 13 optimized configurations achieved a minimum speedup of 0.22, a mean speedup of 21.89, a median speedup of 8.89, and a maximum speedup of 66.23 (all relative to the baseline configuration). Appendix E outlines the optimization-level flags associated with each of our 14 configurations, while Figure 6 details the average runtime and speedup relative to baseline achieved by each configuration.

8.1 Discussion

In general, we found that caching-based optimizations resulted in the largest speedup for ClarityNLP. As mentioned above, our sample data is biased toward less noise, and thus this performance boost is expected. However, we would expect some benefit by using caching even in a more noisy data set. We found that reordering queries leads to speedup as well. To achieve optimal benefits of the query reordering optimization, primitives should be pre-calculated to determine optimal down selection potential. In addition, we found that changes to batch size can have strong negative or positive changes to ClarityNLP. This is perhaps the easiest optimization to make when tuning ClarityNLP for performance.

	config_name	average_runtime (h-m-s)	speedup
0	redis	00:00:08.824728	66.230891
1	precomp_reorder_redis	00:00:08.828026	66.206152
2	reorder_redis	00:00:08.830098	66.190617
3	all_except_chaining	00:00:11.372663	51.392503
4	cache_precomp_reorder	00:00:36.075572	16.201258
5	reorder_lru_cache	00:00:36.541302	15.994768
6	lru_cache	00:00:36.672893	15.937375
7	workers_size_5	00:05:17.154091	1.842857
8	reorder_nlpql	00:05:40.415931	1.716928
9	batch_size_10	00:05:50.127623	1.669305
10	precompute_nlp_tasks	00:06:52.598290	1.416559
11	baseline	00:09:44.469653	1.000000
12	batch_size_100	00:20:20.067303	0.479047
13	chained_queries	00:44:42.019651	0.217921

Figure 6. Speedup Relative to Baseline by Configuration

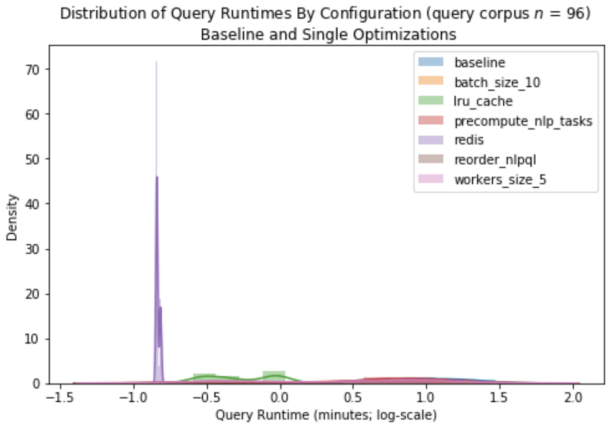

Figure 7. Distribution of Query Runtimes (in log-space) by Configuration, Baseline and Single Optimizations

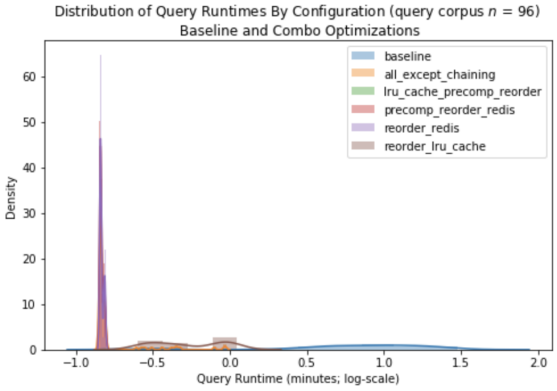

Figure 8. Distribution of Query Runtimes (in log-space) by Configuration, Baseline and Combo Optimizations

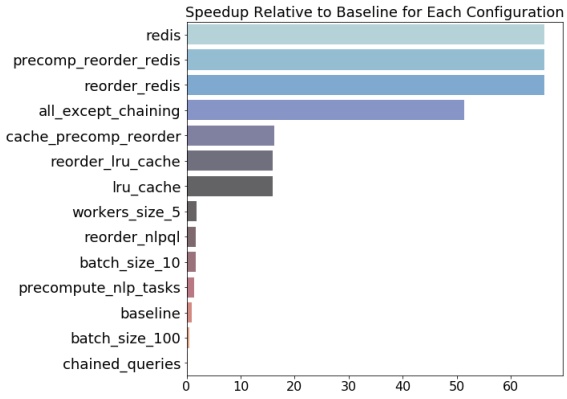

Figure 9. Speedup Relative to Baseline by Configuration

8.2 Cache Hit Ratio

For each experiment using caching (LRU cache and Redis), we evaluated the cache hit ratio (CHR) to better understand performance gains around caches. Cache hit ratio is a measurement of cache hits to cache queries. Generally, a higher cache ratio is correlated with greater performance gains. We found that the LRU cache had a mean CHR of **0.61054** and Redis had a mean CHR of **0.98554**, which is demonstrated in our results, where Redis-based experiments saw the largest speedups. See Figure 9.

9. Resources

To evaluate and execute our optimizations, we ran on an identical environment to the GTRI ClarityNLP research environment. The server instance was a PHI-compliant Amazon Web Services (AWS) server, modeled against the *m5.2x large* specifications. The virtual machine supports 8 vCPUs, 32 GiB memory, with 10 Gbps network bandwidth, and EBS-based SSDs, running Ubuntu 18.04.

To train our primitive-level DNNs in parallel, we leveraged a private cluster available to GTRI researchers, called ICEHAMMER [15]. ICEHAMMER is a cluster environment with 1200 cores, 5TB of memory, 3 IBM POWER8 with Nvidia GPUs, 22 Kepler GPUs, 4 Maxwell GPUs, and 9 Pascal GPUs. In addition, it is equipped with JupyterHub and Slurm.

10. Future Work

10.1 Experiments

To further evaluate the effectiveness of each optimization, further experiments could include more noise within the data set. More realistic primitives could be evaluated from expert-defined NLPQL. To evaluate ClarityNLP for a non-clinical setting, completely synthetic data sets could be utilized. In addition, the experiments would benefit from additional iterations, and additional server environments to evaluate ClarityNLP performance across different system configurations.

10.2 Optimizations

We would like to further explore further optimizations related to:

1. Tuning configuration parameters in ClarityNLP via machine learning models, as described in Van Aken, et al [27]. Candidates for this optimization include: Luigi workers, batch size, LRU cache size, Solr memory settings, etc.

2. Improve the performance of DNNs used to predict note-level labels for query primitives, and, if possible, integrate them into the pipeline as a computationally inexpensive down-selection mechanism.

3. Supporting more complex queries for synthetic query generation and query re-ordering. Our experimental queries were limited to *AND* queries. ClarityNLP is capable of supporting nested queries and mathematical operators. Our future experiment setups and optimizations should support these features in ClarityNLP.

4. Updating models and optimizations in an online manner or nightly schedule. Two of our optimizations, specifically query reordering and DNNs were tailored to our 27 primitives. These optimizations should be able to be calculated for the specific data set for a ClarityNLP institution, and updated as Solr data sets change, taking into consideration measures such as term frequency, term frequency-inverse document frequency (tf-idf), and potential for downselection.

5. Further evaluate Luigi for resource contention. Over time, changes in query performance would randomly spike, even though no other processes were running on the evaluation server. Further evaluation is needed to understand and mitigate resource contention around Luigi jobs.

6. For Redis caching, additional enhancements should be made to mark the cache as stale, so that values are refreshed, in the case of system updates to algorithms or other configuration changes.

7. For Solr pre-computation, tools should be integrated into ClarityNLP to run segmentation as documents are ingested into Solr, and should support any additional improvements to segmentation. In addition, it may be value to pre-compute other lower-level NLP tasks, such as part-of-speech tagging.

10.3 Evaluation

In future experiments, we would like to include metrics to evaluate ClarityNLP performance, including: CPU time, system usage, and processing time of each step in an NLPQL job (this information is available, but we did not include these times in our evaluation).

11. Conclusion

Our experiments to optimize ClarityNLP were primarily centered around caching, NLPQL query optimization, shifting data locality and DNNs. Our optimizations were focused on areas not currently in active development on the ClarityNLP project. We showed that given certain query patterns, significant speedup is achievable. This project is our first step in optimizing the ClarityNLP platform. We believe these optimizations will contribute to further adoption of ClarityNLP in the clinical research space and lead to improvements in rapid, computation-based clinical phenotyping.

References

[1] Explosion AI. 2018. spaCy: Industrial-strength Natural Language Processing in Python. https://spacy.io/. (Accessed on 10/24/2018).

[2] Steven Bird and Edward Loper. 2004. NLTK: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 209 N. Eighth Street, Stroudsburg PA 18360, USA, 31.

[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[4] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Int. Res.* 16, 1 (June 2002), 321–357. http://dl.acm.org/citation.cfm?id=1622407.1622416

[5] Mayo Clinic. 2017. Apache cTAKES - clinical Text Analysis Knowledge Extraction System. http://ctakes.apache.org/index.html. (Accessed on 10/24/2018).

[6] Hamish Cunningham. 2002. GATE, a general architecture for text engineering. *Computers and the Humanities* 36, 2 (2002), 223–254.

[7] eCQI Resource Center. 2018. Clinical Query Language. https://ecqi.healthit.gov/cql-clinical-quality-language. (Accessed on 09/23/2018).

[8] David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering* 10, 3-4 (2004), 327–348.

[9] Python Software Foundation. 2018. Higher-order functions and operations on callable objects. https://docs.python.org/3/library/functools.html

[10] The Apache Foundation. 2017. FAQ - Apache OpenNLP. https://opennlp.apache.org/faq.html. (Accessed on 10/24/2018).

[11] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. 2000. PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals. *Circulation* 101, 23 (2000), e215–e220.

[12] Clinical Trials Transformation Initiative. 2018. AACT Database: Clinical Trials Transformation Initiative. https://www.ctti-clinicaltrials.org/aact-database. (Accessed on 10/25/2018).

[13] Georgia Tech Research Institute. 2018. Clarity NLP. https://github.com/ClarityNLP. (Accessed on 09/21/2018).

[14] Georgia Tech Research Institute. 2018. Clarity NLP Documentation. https://clarity-nlp.readthedocs.io/en/latest/. (Accessed on 09/22/2018).

[15] Georgia Tech Research Institute. 2018. ICEHAMMER User Guide. https://github.gatech.edu/ICEHAMMER/UserGuide. (Accessed on 09/19/2018).

[16] Alistair EW Johnson, Tom J Pollard, Lu Shen, H Lehman Li-wei, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific data* 3 (2016), 160035.

[17] Daniel Kang, Peter Bailis, and Matei Zaharia. 2018. BlazeIt: Fast Exploratory Video Queries using Neural Networks. *CoRR* abs/1805.01046 (2018). arXiv:1805.01046 http://arxiv.org/abs/1805.01046

[18] Redis Labs. 2018. https://redis.io/

[19] Qi Li, Kristin Melton, Todd Lingren, Eric S Kirkendall, Eric Hall, Haijun Zhai, Yizhao Ni, Megan Kaiser, Laura Stoutenborough, and Imre Solti. 2014. Phenotyping for patient safety: algorithm development for electronic health record based automated adverse event and medical error detection in neonatal intensive care. *Journal of the American Medical Informatics Association* 21, 5 (2014), 776–784.

[20] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1493–1508. https://doi.org/10.1145/3183713.3183751

[21] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. Association for Computational Linguistics, 209 N. Eighth Street, Stroudsburg PA 18360, USA, 55–60.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[23] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[24] Rachel L Richesson, Jimeng Sun, Jyotishman Pathak, Abel N Kho, and Joshua C Denny. 2016. Clinical phenotyping in selected national networks: demonstrating the need for high-throughput, portable, and computational methods. *Artificial intelligence in medicine* 71 (2016), 57–61.

[25] Ergin Soysal, Jingqi Wang, Min Jiang, Yonghui Wu, Serguei Pakhomov, Hongfang Liu, and Hua Xu. 2017. CLAMP–a toolkit for efficiently building customized clinical natural language processing pipelines. *Journal of the American Medical Informatics Association* 25, 3 (2017), 331–336.

[26] Eugene Tseytlin, Kevin Mitchell, Elizabeth Legowski, Julia Corrigan, Girish Chavan, and Rebecca S Jacobson. 2016. NOBLE–Flexible concept recognition for large-scale biomedical natural language processing. *BMC bioinformatics* 17, 1 (2016), 32.

[27] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, ACM, New York, NY, 1009–1024.

[28] Meliha Yetisgen, Prescott Klassen, and Peter Tarczy-Hornoch. 2014. Automating Data Abstraction in a Quality Improvement Platform for Surgical and Interventional Procedures. *eGEMs* 2, 1 (2014), 1–7.

[29] Kazuki Yoshida, Daniel H Solomon, and Seoyoung C Kim. 2015. Active-comparator design and new-user design in observational studies. *Nature Reviews Rheumatology* 11, 7 (2015), 437.

[30] Sheng Yu, Abhishek Chakrabortty, Katherine P Liao, Tianrun Cai, Ashwin N Ananthakrishnan, Vivian S Gainer, Susanne E Churchill, Peter Szolovits, Shawn N Murphy, Isaac S Kohane, et al. 2016. Surrogate-assisted feature extraction for high-throughput phenotyping. *Journal of the American Medical Informatics Association* 24, e1 (2016), e143–e149.

A Sample NLPQL

```
//phenotype name
phenotype "Ejection Fraction Values"
version "1";

//include Clarity main NLP libraries
include ClarityCore version "1.0" called
Clarity;

termset EjectionFractionTerms:
  ["ef","ejection fraction","lvef"];

define EjectionFraction:
  Clarity.ValueExtraction({
    termset:[EjectionFractionTerms],
    minimum_value: "10",
    maximum_value: "85"
    });

//logical Context (Patient, Document)
context Patient;

define final LowEFPatient:
    where EjectionFraction.value <= 30;
```

B Feature to Term Mappings

Feature	Terms
female	"woman", "women", "female", "girl", "girls", "pregnant", "menopausal", "postmenopausal"
male	"man", "men", "male", "boy", "boys"
asian	"asian"
black	"black", "african american"
native american	"native american", "american indian", "alaska native"
pacific islander	"pacific islander", "native hawaiian"
white	"white", "caucasian"
glucose	"glucose"
potassium	"potassium"
docusate	"docusate"
heparin	"heparin"
magnesium sulfate	"magnesium sulfate"
acetaminophen	"acetaminophen"
pantoprazole	"pantoprazole"
metoprolol	"metoprolol"
furosemide	"furosemide"
hypertension	"hypertension", "high blood pressure"
chf	"congestive heart failure","chf", "ccf - congestive cardiac failure","chf - congestive heart failure","congestive cardiac failure","congestive heart disease","congestive heart failure"
afib	"atrial fibrillation", "atrial fibrilation", "a fib", "afib", "atrial fib", "atr fibrillation", "atr fibrilation", "atr fib", "auricular fibrillation", "auricular fib", "aflutter", "atrial flutter"
diabetes	"diabetes","dm"
renal failure	"renal failure"
high cholesterol	"hyperlipidemia", "high blood cholesterol", "high cholesterol"
uti	"urinary tract infectious disease","uti", "urinary tract infection"
gerd	"gastroesophageal reflux","gerd"
arteriosclerosis	"arteriosclerosis"
respiratory failure	"respiratory failure"

C Primitives Ranked by % Downselection (MIMIC-III)

Category	Primitive	Count	%	Rank
race	pacific islander	0	100.00	1
conditions	chf	107	99.98	2
race	native american	47	99.90	3
medications	pantoprazole	46089	98.93	4
conditions	gerd	6326	98.87	5
conditions	uti	6578	98.83	6
medications	docusate	52667	98.78	7
medications	acetaminophen	54257	98.74	8
conditions	respiratory failure	8524	98.48	9
medications	magnesium sulfate	80516	98.13	10
medications	heparin	80977	98.12	11
medications	furosemide	104238	97.58	12
conditions	arteriosclerosis	14541	97.41	13
conditions	high cholesterol	14627	97.40	14
conditions	afib	14680	97.39	15
medications	metoprolol	114518	97.34	16
conditions	diabetes	15796	97.19	17
conditions	renal failure	16678	97.03	18
race	asian	1691	96.37	19
conditions	hypertension	22779	95.94	20
medications	potassium	204724	95.24	21
medications	glucose	212345	95.06	22
medications	sodium chloride	346605	91.94	23
race	black	3875	91.67	24
gender	female	20399	56.15	25
gender	male	26121	43.85	26
race	white	33899	27.13	27

D Query Primitive Co-occurrence Frequencies

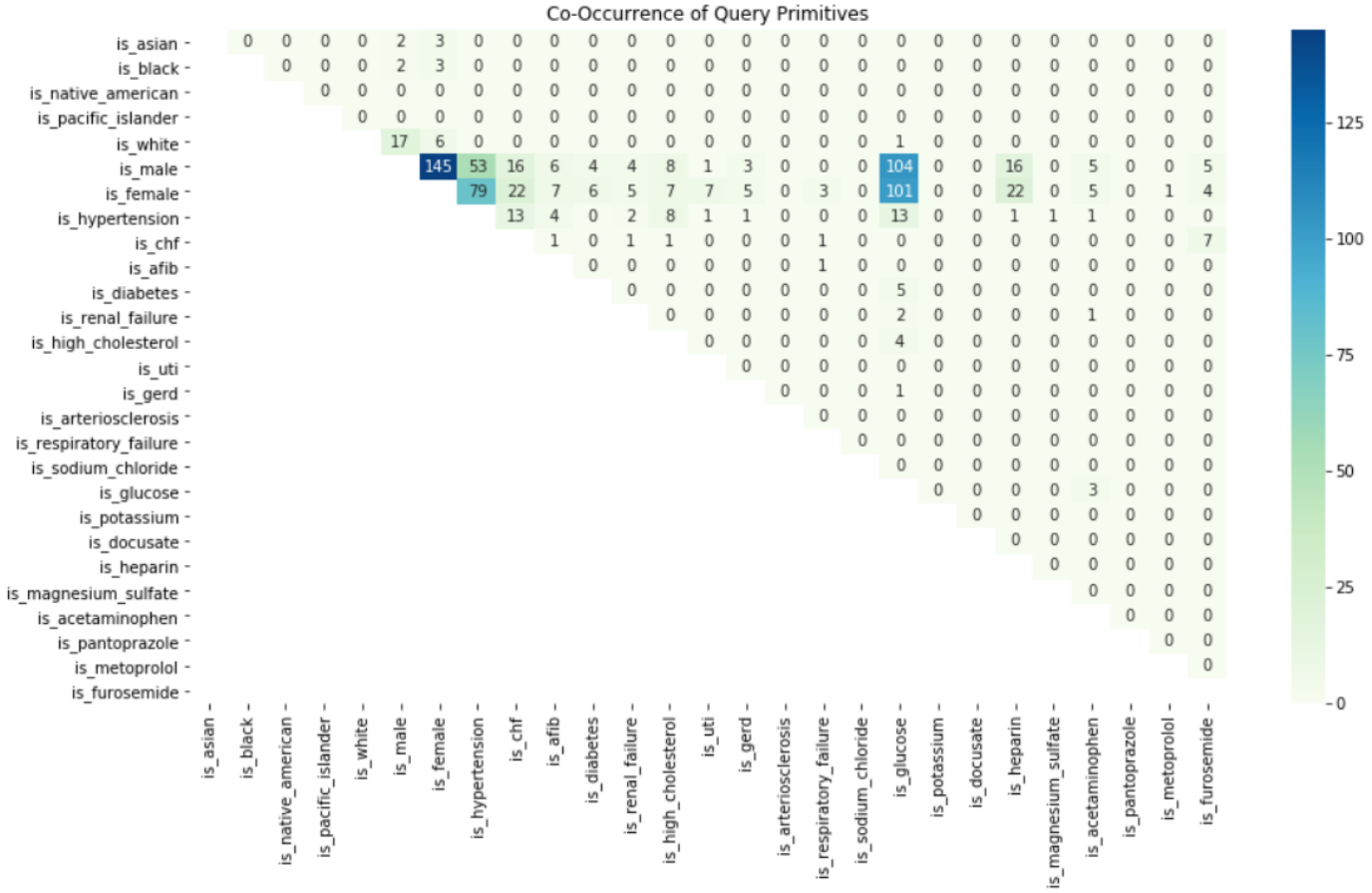

Figure 10. Query Primitive Co-Occurrence Counts Within Synthetic NLPQL Corpus

E Configuration Settings

	category	lru cache	luigi workers	batch size	precomputed seg	reordered nlpql	chained queries	redis cache	results status
baseline	N/A	0	4	25	0	0	0	0	complete
batch_size_10	single opt	0	4	10	0	0	0	0	complete
batch_size_100	single opt	0	4	100	0	0	0	0	complete
chained_queries	single opt	0	4	25	0	0	1	0	complete
lru_cache	single opt	1	4	25	0	0	0	0	complete
precompute_nlp_tasks	single opt	0	4	25	1	0	0	0	complete
redis	single opt	0	4	25	0	0	0	1	complete
reorder_nlpql	single opt	0	4	25	0	1	0	0	complete
workers_size_5	single opt	0	5	25	0	0	0	0	complete
all_except_chaining	combo (4 opts)	1	4	25	1	1	0	1	complete
lru_cache_precomp_reorder	combo (3 opts)	1	4	25	1	1	0	0	complete
precomp_reorder_redis	combo (3 opts)	0	4	25	1	1	0	1	complete
reorder_redis	combo (3 opts)	0	4	25	0	1	0	1	complete
reorder_lru_cache	combo (2 opts)	1	4	25	0	1	0	0	complete

	: baseline setting
	: optimization

Figure 11. Configuration Flag Settings

F Average Primitive-Level DNN Results

Average Primitive-level DNN results (3-fold stratified cross validation with scaling and SMOTE; Hidden layers of size 128, 64, 10, 2; sigmoid outer layer)

primitive	avg_accuracy	avg_precision	avg_recall	avg_f1_score	avg_num0_ytrain	avg_num1_ytrain	avg_num0_ytest	avg_num1_ytest	avg_num0_ypred	avg_num1_ypred
asian	0.9985	0.0000	0.0000	0.0000	13332.6667	0.6667	6666.3333	0.3333	6657.3333	9.3333
black	0.9977	0.0000	0.0000	0.0000	13331.3333	2.0000	6665.6667	1.0000	6652.3333	14.3333
is_magnesium_sulfate	0.9975	0.0000	0.0000	0.0000	13328.6667	4.6667	6664.3333	2.3333	6652.3333	14.3333
is_dm	0.1807	0.0079	0.8397	0.0157	13229.3333	104.0000	6614.6667	52.0000	1169.6667	5497.0000
is_afib	0.4916	0.0058	0.5225	0.0116	13259.3333	74.0000	6629.6667	37.0000	3275.6667	3391.0000
is_chf	0.7747	0.0058	0.2118	0.0112	13254.6667	78.6667	6627.3333	39.3333	5187.0000	1479.6667
is_acetaminophen	0.9565	0.0016	0.0317	0.0031	13290.6667	42.6667	6645.3333	21.3333	6396.6667	270.0000
is_respiratory_failure	0.9906	0.0034	0.0256	0.0060	13307.3333	26.0000	6653.6667	13.0000	6616.3333	50.3333
is_uti	0.9311	0.0023	0.0784	0.0045	13299.3333	34.0000	6649.6667	17.0000	6222.0000	444.6667
white	0.9288	0.0005	0.0741	0.0010	13314.6667	18.6667	6657.3333	9.3333	6200.0000	466.6667
is_sodium_chloride	0.5620	0.0007	0.5000	0.0014	13321.3333	12.0000	6660.6667	6.0000	3747.0000	2919.6667
is_renal_failure	0.9364	0.0008	0.0317	0.0015	13290.6667	42.6667	6645.3333	21.3333	6263.0000	403.6667
is_potassium	0.3262	0.0097	0.6329	0.0191	13195.3333	138.0000	6597.6667	69.0000	2156.6667	4510.0000
is_pantoprazole	0.6629	0.0047	0.4048	0.0090	13277.3333	56.0000	6638.6667	28.0000	4424.6667	2242.0000
is_male	0.3098	0.0098	0.6990	0.0193	13204.6667	128.6667	6602.3333	64.3333	2039.3333	4627.3333
is_metoprolol	0.5363	0.0082	0.4746	0.0161	13228.0000	105.3333	6614.0000	52.6667	3578.3333	3088.3333
is_hypertension	0.1250	0.0126	0.8920	0.0249	13166.0000	167.3333	6583.0000	83.6667	767.3333	5899.3333
is_high_cholesterol	0.9261	0.0056	0.1021	0.0105	13280.6667	52.6667	6640.3333	26.3333	6195.0000	471.6667
is_gerd	0.8269	0.0024	0.0825	0.0047	13276.6667	56.6667	6638.3333	28.3333	5536.6667	1130.0000
is_glucose	0.2494	0.0132	0.7204	0.0260	13147.3333	186.0000	6573.6667	93.0000	1621.3333	5045.3333
is_female	0.2638	0.0078	0.7121	0.0154	13224.0000	109.3333	6612.0000	54.6667	1735.3333	4931.3333
is_heparin	0.4783	0.0084	0.5616	0.0164	13238.0000	95.3333	6619.0000	47.6667	3183.0000	3483.6667
is_docusate	0.3407	0.0057	0.6552	0.0113	13256.6667	76.6667	6628.3333	38.3333	2259.6667	4407.0000
is_furosemide	0.7873	0.0031	0.2101	0.0061	13286.6667	46.6667	6643.3333	23.3333	5262.0000	1404.6667

Figure 12. Average Primitive-Level DNN results