FACILITATING CLINICAL PHENOTYPE DEVELOPMENT AT SCALE:

Optimizing the ClarityNLP Platform Using Clinical Trial Data

Christine Herlihy & Charity Hilton

Fall 2018

CS 8803-DDL
Georgia Tech

Problem Statement & Importance

- **Computational phenotyping**: development of algorithms to identify patients with particular observable traits associated with a disease from a broader clinical population.
  - Critical step in pharmaceutical R&D lifecycle.
  - Vital for development of precision medicine.
- **ClarityNLP** is an existing open-source platform used by clinical researchers to build and run patient phenotypes using structured and unstructured data.
  - Queries written in NLPQL (custom declarative language).
  - Query runtime varies based on dataset size and complexity of component NLP algorithm(s).
  - Built primarily for NLP functionality, not performance.
- **Objective**: implement a series of optimizations to reduce expected runtime and system resource usage.

HIGH-LEVEL OVERVIEW OF OUR APPROACH

- **Synthetic Query Generation**
  - Develop a representative corpus of synthetic NLPQL queries that we can run against available patient records and/or synthetic data to ensure non-trivial computational load.
  - Use synthetic queries to evaluate performance of baseline system.
- **Implementation of Platform Optimizations**
  - **Tier I Optimizations:** Foundational improvements, including:
    - Chaining of queries
    - Caching based on hash of query and database contents at time *t*
    - Sequencing of NLPQL operations based on downselection potential
    - Pre-computing and indexing common NLP tasks in Solr
  - **Tier 2 Optimizations:** Exploratory/open-ended, including:
    - Cache commonly co-occurring query predicates.
    - Train DNNs to compute query results (e.g., Boolean set membership) using vector-valued representation of the NLPQL queries as input.
    - Train a model to assign an optimal number of Luigi workers and/or Luigi batch-size based on system constraints and job criteria.
    - Use in-memory databases for active or recent jobs.
- **Analysis of Results**
  - Accuracy varies by optimization configuration; on average, we achieve an average speedup of 21.89x and a max speedup of 66.23x relative to baseline.

TIER I OPTIMIZATIONS IMPLEMENTED

	Optimization	Description	Implemented?	Integrated?
1	Allow query chaining	Run queries such that queries where all predicates contain only 'AND' are chained. Results from NLPQL 1, must complete before NLPQL 2 is ran, and the document set of query 2 is limited to only matches found in query 1.	1	1
2	Implement least recently used (LRU) caching	Cache commonly used objects in queries, such as documents and Python objects preloaded with models and regular expressions. Cache results of NLPQL queries given a hash of query parameters. This was implemented using the cachetools library with a max size of 5000.	1	1
3	Reorder NLPQL clauses based on downselection potential	Pre-compute downselection of each query primitive. Then, reorder synthetic query so that sub-clauses containing primitives with greater downselection potential get executed first.	1	1
4	Create additional indices	We evaluated existing indices. At this time all commonly used keys are indexed. However, a future application might be to use a deep learning model to determine where additional indexes would be helpful. In addition, our evaluation is primarily focused on computation and writes, while indexes generally help on read applications.	0	0
5	Shift computation closer to data source	Note: There has been effort on the main ClarityNLP project to implement this with MongoDB, but we did not evaluate it for this project.	0	0
6	Precompute and index common NLP tasks in Solr	Two common tasks in the ClarityNLP pipeline are segmentation of notes by section and sentence. The former uses a custom ClarityNLP library and the latter uses spacy. Both are computationally intense. For every document in the evaluation Solr index, we ran and stored the sections and sentences as arrays in Solr, which can be retrieved rather than re-computed.	1	1

TIER II OPTIMIZATIONS IMPLEMENTED

	Optimization	Description	Implemented?	Integrated?
1	Cache of commonly occurring query predicates	Take synthetic query corpus and compute pairwise cooccurrence of each query primitive; cache query results of commonly cooccuring primitives. Cooccurrence frequency was determined to be too low for this optimization to result in speedup. We could tweak this in future query generation.	1	0
2	Train DNNs to compute query primitive results	Train a separate deep neual network to classify a given note as 0 or 1 for each query primitive. We used stratified sampling and SMOTE to address class imbalance, but for many primitives, high recall comes at the expense of precision. High accuracy can be achieved by simply predicting 0 each time. Thus, the high-recall DNNs might be more useful as a first-step filter.	1	0
3	Train a model to dynamically assign optimal number of Luigi workers and/or Luigi batch size	Luigi works are somewhat constrained by CPUs in a system. However, in theory ClarityNLP + Luigi could be evaluated by comparing the same NLPQL query across different configurations of ClarityNLP batch sizes and Luigi workers. We ran additional configurations to capture performance, but did not build a model for this task.	0	0
4	Use in-memory database for active/recent jobs	Redis is a commonly used in-memory key-value data store. This activity primarily compares Redis against a LRU cache as discussed above. Redis is slightly less amenable to data in ClarityNLP, as most cached items require some serialization.	1	1

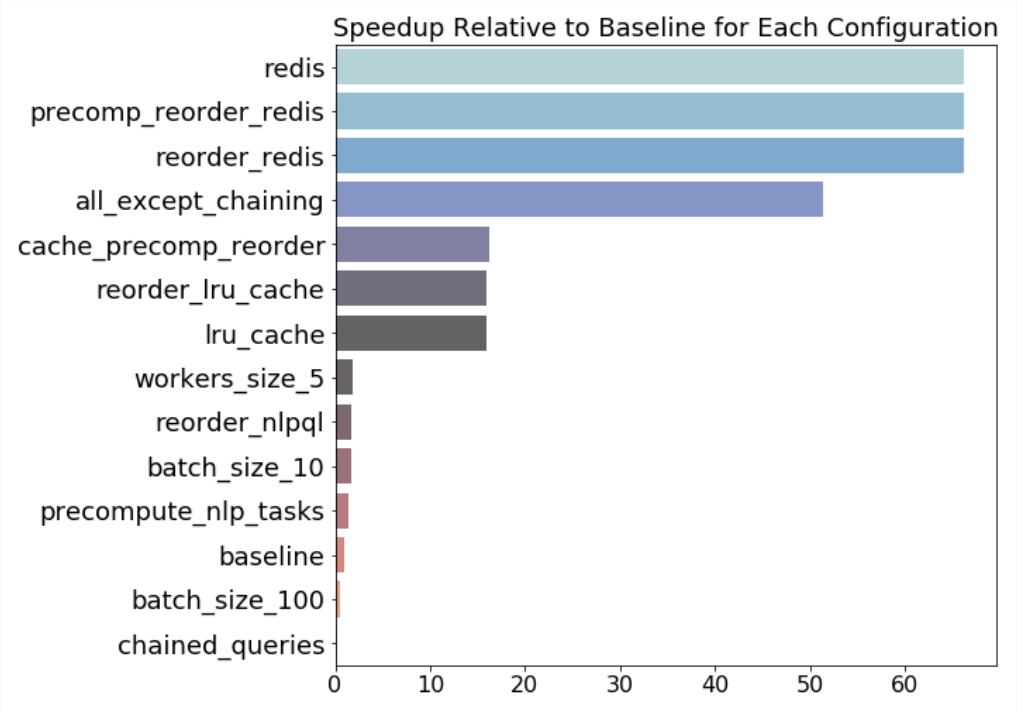
SUMMARY OF OPTIMIZATION CONFIGURATIONS

	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

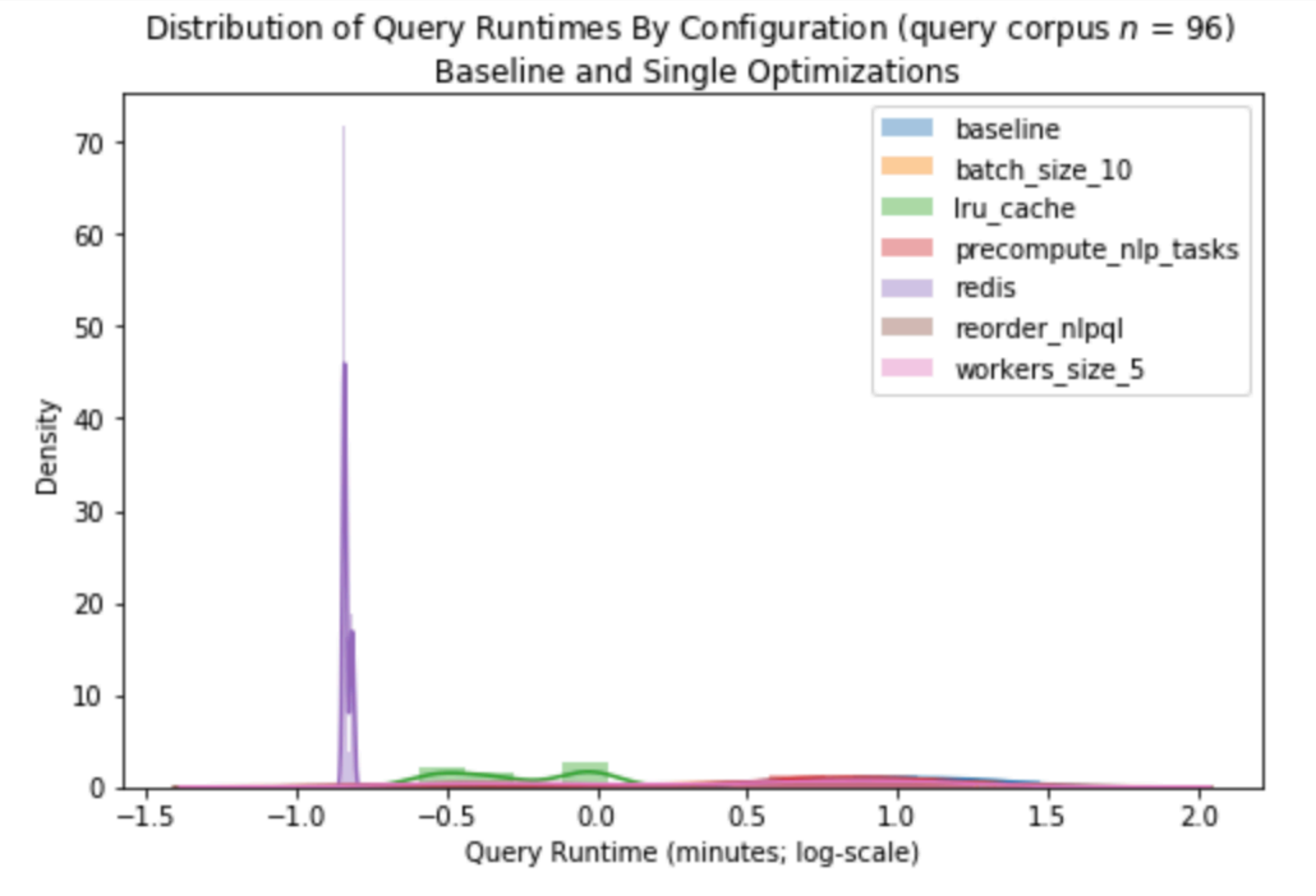
	: baseline setting
	: optimization

RESULTS: SPEEDUP ACHIEVED RELATIVE TO BASELINE PLATFORM

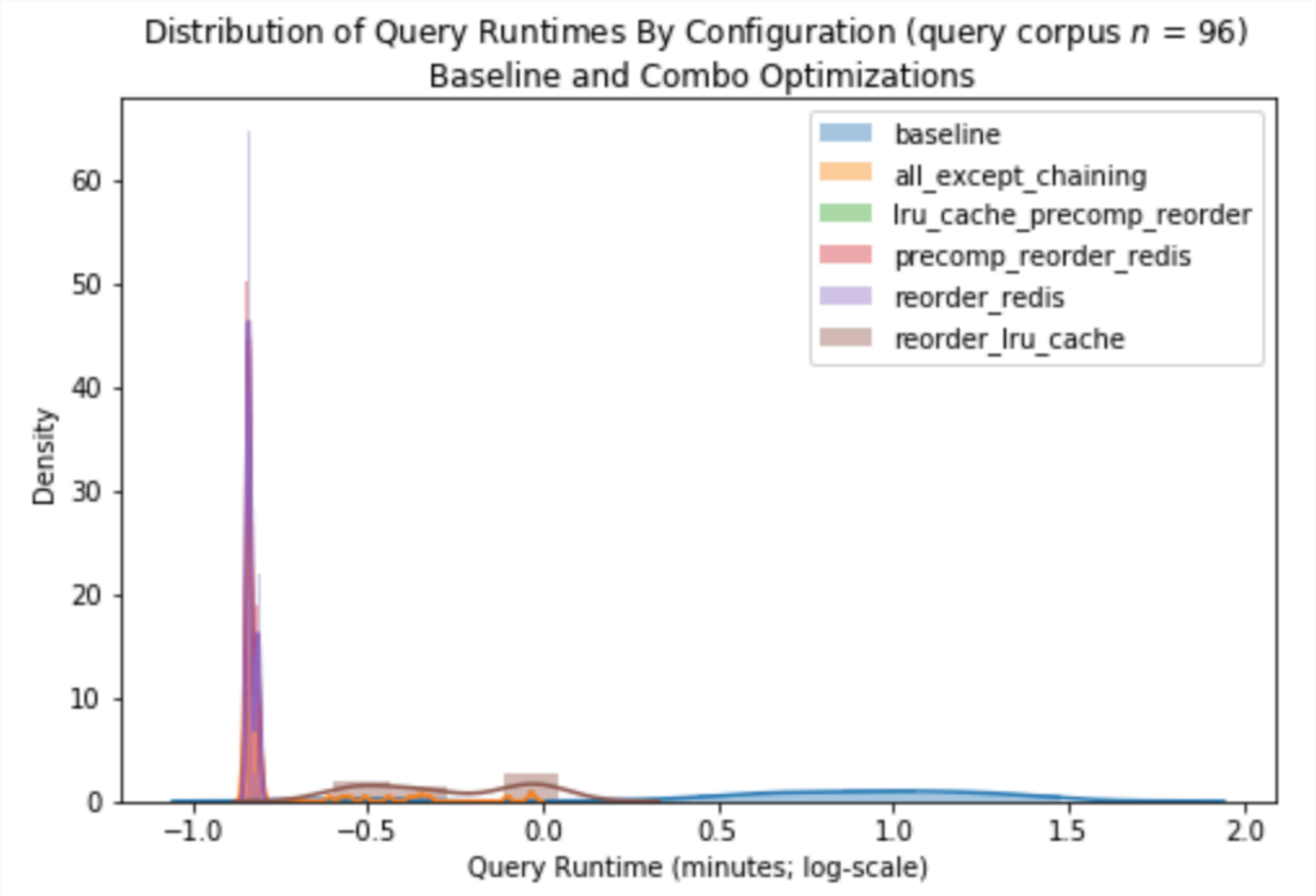
	config_name	average_runtime (h-m-s)	speedup
0	redis	00:00:08.824728	66.230891
1	precomp_reorder_redis	00:00:08.828026	66.206152
2	reorder_redis	00:00:08.830098	66.190617
3	all_except_chaining	00:00:11.372663	51.392503
4	cache_precomp_reorder	00:00:36.075572	16.201258
5	reorder_lru_cache	00:00:36.541302	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



RESULTS: DISTRIBUTION OF SYNTHETIC QUERY RUNTIMES ($n = 96$)



RESULTS: DISTRIBUTION OF SYNTHETIC QUERY RUNTIMES ($n = 96$)



VALIDATION: DO QUERIES RUN ON OPTIMIZED PLATFORM MATCH BASELINE RESULTS?

- Most runs of queries return similar results. Most of the differences are due to missing job results. This should be studied further.
- Chained queries had serious bugs. It needs a full re-implementation before it can be validated.
- 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, they are not the same exact 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

- **Improve performance of DNNs** used to predict note-level labels for query primitives, and, if possible, integrate into pipeline as a computationally inexpensive downselection mechanism.
- **Finalize results** for all configurations considered (a final set of combination configs are still running).
- To re-integrate our fork to the main repo, we need **realistic primitives**. For this, we must develop standard termsets and query stubs to encourage convergence and make frequency stats more useful. Clinical texts and/or regulatory guidelines can serve as a baseline for features to include in some cases.
- A more sophisticated **constraint parser** will allow us to handle more complex queries, including those with nested sub-clauses.

https://xkcd.com/1421/