Fast Lexical-Semantic Analysis of Syntactic n-grams in Myria

CSE544 Final Project

Leila Zilles
University of Washington
lzilles@cs.washington.edu

ABSTRACT

The Google syntactic n-grams dataset is a linguistic resource containing billions of detailed annotations on short strings of text and the relationships between words within them. The dataset is available for download as a large (hundreds of GBs) set of flat files, but no open tools for working with the data have been released to date. As a result, extracting information from this data currently requires custom solutions by any user attempting to analyze it. In this project, we show that by loading this data into a distributed DBMS and using an efficient query plan, it is possible to make a wide range of general queries over it in short amounts of time. We also create a Python API for building common types of queries over the data, effectively abstracting the database itself away from the user while still allowing them to perform a wide range of queries, making it both faster and simpler to analyze and collect information about this dataset.

1. INTRODUCTION

Like many fields, the area of natural language processing (NLP) has benefited tremendously from the use of statistical methods for learning more accurate models of linguistic phenomena from massive data. Typically, the data used for machine learning in NLP consists of a relatively small set of texts annotated for a specific task (since the cost of obtaining large amounts of labeled data is high), or large amounts of entirely unstructured text that is processed automatically. One form of data that is useful for NLP is n-grams data, which provides counts over sequences of words found in large corpora of unstructured text. N-grams data itself is a compressed form of linguistic data, since it strictly represents language in terms of sequences of words rather than in entire texts. However, even when constrained to sequences of up to only a few words and reasonable frequency filtering, the size of a web-scale n-grams dataset is on the order of hundreds of gigabytes and billions of n-grams, and the size of this data only grows when linguistic annotations.

One such annotated web-scale n-grams dataset is the Google syntactic n-grams dataset, released in 2013 [4]. Though this annotated n-grams dataset contains massive amounts of useful information, it is not designed for ease of use; nothing beyond the simplest surface-level queries are possible over the raw data, and many end users may opt for specialized schemas that only account for the information specific to their application in order to better handle the size of the dataset. For this project, we transform the syntactic n-grams corpus into a form such that queries regarding grammatical and lexical context can be made simple and fast, rather than requiring multiple scans of flat files and complex queries specific to the given data format. This necessitates first transforming the data from a nested column format to a flat relational format. Because the relations contain of billions of rows, it is also necessary to use a parallel DBMS to handle the large amount of data. In particular, Myria DB [5] is a strong choice for this data, not only because it is a parallel DMBS but also for its emphasis on handling data specfically meant for statistical analysis.

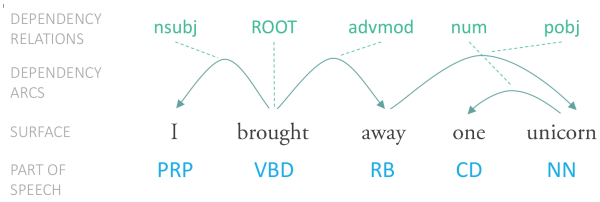

Figure 1: Diagram of a syntactic n-gram.

2. BACKGROUND AND MOTIVATION

In this section, we describe the structure of the n-grams data and how we place this data into a relational schema.

2.1 The syntactic n-grams dataset

The Google syntactic n-grams corpus is a multi-gigabyte dataset containing several different forms of linguistic information on sequences of words between 1 and 5 tokens long, with frequency counts over a corpora of around 3.5 million English books [4]. At the heart of this data is not simply counts over word sequences, but information in the form of dependency parses. Each word sequence itself includes metadata about the syntactic structure, modeled as a tree with labeled edges. A single sample parse is shown in Figure 1. The data is distributed with each n-gram as a row, and the data for each token in the n-gram is represented within nested columns within each row.

Accessible analysis of this data would be extremely valuable to both computational linguists and researchers in natural language processing for both academic study and as an external resource for feature extraction in NLP tools. However, the format that the dataset is provided in is not ideal for any kind of data processing other than aggregating information over individual head words. In particular, this data is a rich

source of contextual information, both for the grammatical uses of a particular word (e.g., finding differences between how a word is used as a subject vs as an object), as well as what words are used in certain lexical contexts (e.g., given the context "the X was late", what are the top nouns that could be X?"). Being able to answer questions such as these quickly and efficiently could greatly aid NLP applications that are currently semantics-deficient by providing a large source of data outside of "gold" training examples alone to determine what language works well in what contexts.

2.2 Schema

The relationalized schema for the data is very simple. Each n-gram is comprised of unique tokens, which themselves contain information on the structure of the n-gram itself. In contrast to previous work, we choose to represent each token as a row in a dedicated *Token* relation, since the n-gram annotations can be represented information embedded in each token. A diagram of the schema is shown in Figure 2. *TokenIn* is not represented in an explicit relation, but the unique n-gram ID referenced as a foreign key in each token as `nid`, which is the primary key for the *Ngram* relation. The other attribute of note is `headid`, which references the unique token ID `tid`. We also create two relations to map the string representations of dependency arc types and part-of-speech tags to integer IDs, in order to store them more compactly in the relation. However, we chose to store surface strings as-is in the *Token* relation in order to avoid the need to either (a) do a distributed join every time we wanted to query for a surface string match, or (b) replicate a large "vocabulary" relation across every worker to make word IDs back to strings.

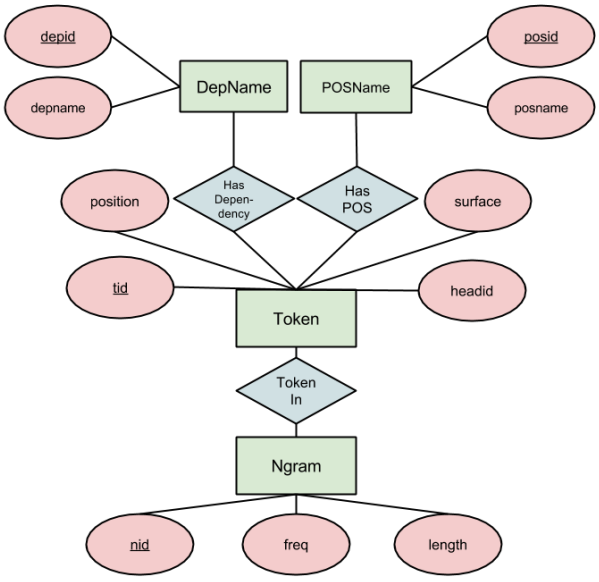

Figure 2: Entity-relationship diagram for syntactic n-grams data.

A major issue that arises from this representation is that it increases the size of an already large dataset, since every token in each n-gram is represented as a unique row. That is, for every n-gram added to the database, n token rows are also added. In a dataset that contains over a billion n-grams, this makes the *Token* relation a factor of billions of rows large. The size of the database necessitates the use of a parallel DBMS for efficient querying.

3. APPROACH

A general outline of our approach is as follows:

1. Clean, transform, and ingest n-grams data into Myria.
2. Partition and index the data in a "smart" way.
3. Create general custom query plans that take advantage of the partitioning.
4. Expose a set of possible queries on the dataset to the user via a Python API.

3.1 Physical data tuning

One aspect of this data that can be used to our advantage is the fact that the two most practical joins – specifically, between the *Token* and *Ngram* relations on the `nid` attribute, and the self-join on *Token* where `parent.tid=child.headid` – can be confined to single workers in Myria with the correct partitioning of the data. Therefore, we horizontally hash partition the data across workers on `nid` to colocate all the token data relevant to each unique n-gram. A further advantage of this horizontal partitioning is that our tuples should be spread fairly uniformly with regards to every other attribute, providing some protection against stragglers due to data skew on a particular node.

We also create a number of indexes on the PostgreSQL storage layer to speed up what we expect to be especially common lookups. The indexes we create are:

- *Ngram* `(nid)`
- *Token* `(nid)`
- *Token* `(tid)`
- *Token* `(headid)`
- *Token* `(surface, position, depid, posid)`

Finally, we replicate the very small relations *DepName* and *POSName* across every worker's database to enable doing the basic joins for specifying a dependency or part-of-speech tag with a string on a local level.

3.2 Custom query plans

In order for the DMBS to take full advantage of the physical tuning of our data, the query planner must be aware of the placement decisions we made, such as knowing that we hash partitioned on a join attribute and some of our tables are fully replicated on each worker. Unfortunately, the current version of Myria has no way to specify this to the query planner, so although our indexes help speed up some lookups, the overall runtime of each query is still on the order of minutes as the planner always includes a distributed join. We avoid the effects of this by specifying custom JSON

query plans to pass to Myria, rather than letting Myria attempt to optimize the queries itself.

The general outline of each query plan is as follows:

1. Each worker runs the `DbQueryScan` operator, containing the raw SQL query (constructed using the functions described in Section 3.3).
2. The intermediate results of the raw query as executed on each worker are sent to a single worker via the `CollectProducer` and `CollectConsumer` operators.
3. *(Optional; not used in full n-gram queries.)* The `SingleGroupByAggregate` (or `MultiGroupByAggregate`) operator is used to group the full set of results by one (or more) attributes, summing them over the n-gram frequencies.
4. The results are ordered by descending n-gram frequency using the `InMemoryOrderBy` operator. We assume that size of our result set is small enough to allow this. (If the query is *not* a histogram query, results are also sorted over n-gram ID and token position in order to mediate n-gram reconstruction on the application level.)
5. The grouped and sorted results are stored in new relation in Myria using `DbInsert` (this is the temporary *OUTPUT* relation by default, but can also be user-specified).

This query plan would not be sufficient for any queries on the dataset that required a global join, and would fail entirely without the assumption our data is hash partitioned on all join attributes specified in the raw query. Though theoretically some queries that necessitate a join across the entire database exist (e.g., "grafting" two or more dependency trees together based on some attribute), we believe these queries are both sufficiently rare and complex enough that they would ultimately be better handled on the application layer.

3.3 Python API

Rather than require (or allow!) users to manually specify the SQL queries they wish to run over the n-grams dataset themselves, we implement a Python API which automatically builds queries for common questions we believe users would want to ask about this data. The two main goals we sought to address in creating this API were:

1. Do not require the user to know anything about the underlying structure of the database.
2. Provide functions that allow the user to easily specify constraints on both the individual tokens of an n-gram, as well as any relationships that may exist between these tokens.

To this end, we devised three query building functions that take token or n-gram constraints specified by the user as their parameters. These functions output a list of tokens, a list of n-grams (reconstructed from token tuples), or an ordered histogram of tuples, depending on which is used. A SQL query built by these functions is passed into an appropriate JSON query plan, specifically in the `DbQueryScan` operator. After the query has finished running, the results are both stored in the database, as well as downloaded and converted to Python objects. We describe in detail and provide example use cases for each of the functions below. Samples of the SQL queries that these functions build can be found in Table 1.

3.3.1 get_child_tokens, get_parent_tokens

The purpose of the `get_child_tokens` and `get_parent_tokens` functions is to allow the user to obtain a list of tokens (that is, individual words) that occur within n-grams as children (or parents) of some other token. Because the results for an extremely general child/parent query can be large (e.g., all of the children of any token with the ROOT dependency type), we require the user to specify at least the surface string of the parent (or child) to make this query manageably selective. The user is also optionally able to specify the part-of-speech and dependency type of both the child and parent, as well as the surface of the child (or parent).

Some examples of questions that can be answered through using `get_child_tokens` or `get_parent_tokens` are:

- What adjectives are used to describe the noun "bear"?
- What words are used as subjects of the verb "grow"?
- What past-tense verbs is the adverb "lazily" used to describe?

3.3.2 get_contexts

Of the three functions exposed to the user, `get_contexts` is by far the most powerful. This function allows the user to specify constraints on any (or all) tokens of an n-gram. Specifically, the user is able to specify a surface string, part-of-speech tag, or dependency type for any token present in the n-gram, as well as define the dependency arcs and relative (or absolute) positions of the tokens in the n-gram. Any of these attributes can also be "wildcarded" by not providing an argument for that attribute, or setting the attribute for specific tokens to "None". In this way, this function supports any search from a simple string match to more complex queries on arbitrary dependency structures present an n-gram.

Some examples of questions that can be answered through using `get_contexts` are:

- What n-grams contain the word "notebook"?
- What n-grams contain the word "notebook" as the direct object of a verb?
- What n-grams contain two adjectives, both describing the same noun?
- What n-grams start with "cat", end with "mouse", and have a verb in between?

Scenario	Sampled Words
Top 1,000	call, who, give, title, version
Top 10,000	chosen, fashion, slide, carpet, treaty
Top 100,000	malate, concoctions, ideologies, disaffected, interspersed

Table 2: Words queried for the surface string search experiment.

- What n-grams match a fully specified dependency structure?

3.3.3 get_histogram

The `get_histogram` function returns a histogram on the frequencies of a single user-specified token attribute, given constraints on the properties of a token and (optionally) its relationship to another token. Many of the histograms returned by this function could be computed at the application level using results from `get_child_tokens` or `get_parent_tokens`, but `get_histogram` does not require a surface string argument, which allows the user to collect more general statistics about the tokens in the database.

Some examples of questions that can be answered through using `get_histogram` are:

- What are the part-of-speech frequencies of the word "whisper"?
- What are the counts of the dependency relationships between the words "cat" and "chased"?
- What are the frequencies of the child dependencies of "flower" when used as a verb?
- What are the part-of-speech frequencies of words that precede verbs?

4. EVALUATION

In our final setup, we deployed Myria using PostgreSQL as the storage layer, with 30 workers across 10 nodes. Each worker was given a maximum heap size of 8gb. Due to limited space on the cluster as well as time constraints, we only loaded the subset of syntactic n-grams with a frequency of 50 or above into Myria. The final dataset we tested on contained approximately 500 million n-grams, with over 1.5 billion rows in the *Token* relation. Though it does not comprise of the entire n-grams dataset as we originally hoped, we believe our experiments still remain relevant, as all of our optimized queries should scale linearly with the number of workers.

4.1 Searching n-grams by surface string

Intuitively, one of the simplest queries one could make on the syntactic n-grams dataset is selecting all of the n-grams containing a specific surface string (i.e., some word) at any position within them. Such a query would also be fairly common, as NLP applications often use information about the different contexts a word appears in to build features for classifiers, for instance.

It is important to note that the n-grams data, being based on language, has an inherent skew – just like language itself. Words are not distributed uniformly, but instead follow Zipf's law, which states that a word's frequency is inversely proportional to its rank. It follows that one would observe vastly different behavior in the query times of common words vs. rarer words.

With this in mind, we chose to characterize our surface string queries in three different scenarios: one where we search for words sampled from the top 1000 frequent surface tokens in the English language, one with tokens sampled from the top 10,000, and one with tokens sampled from the top 100,000. The exact words we sampled for querying in each scenario are shown in Table 2.

Then, we executed queries in these scenarios using the three following query plan strategies in order to compare their relative performance:[1]

1. **MYRIA:** The query is run with the query plan generated by Myria after parsing the SQL query built by the `get_contexts` function, with only the minimal changes necessary for the parser to understand it. This query is run on the hash partitioned, indexed relations. (Unlike the custom query plans, it does not include any grouping or ordering operators, and thus its results are slightly optimistic.)
2. **NO INDEXES:** The query is run with our custom query plan on relations that have been hash partitioned, but not indexed.
3. **OPTIMIZED:** The query is run with our custom query plan on relations that have been both hash partitioned and indexed.

We show our results in Figure 3. In every case, the custom query plans run much faster than the plan generated by Myria, which takes a relatively fixed amount of time regardless of the scenario. For extremely common words (top 1000), we see that the indexes provide no performance gain and in some cases may have hurt due to a large number of non-sequential disk reads. However, we see a noticeable improvement for the indexed relations as the words we sample are from larger pools. In cases where the surface string has very high selectivity – that is, for most words – our queries can run in a fraction of a second, resulting in a speedup factor of 330 times over that of the original Myria query plan.

On one hand, it is not ideal that the most common words of English are also the most likely to result in slow queries. However, this could also be used to our advantage: relatively few queries will actually take this long, and tokens or even n-grams containing common words could be cached in a separate relation that would be faster to scan than the

[1]A "worst case" baseline would involve using Myria out-of-the-box, with round-robin partitioning and no storage layer indexes. We did not run these experiments because they would have taken too long, but anecdotally, these queries ran for around 20 minutes or more before returning an answer.

Function	Query
`get_contexts`	*What n-grams contain the word "notebook"?* `get_contexts(words=["notebook"])` SELECT ng.freq, tk.nid, tk.tid, tk.surface, tk.position, tk.posid, tk.depid, tk.headid FROM Token tk, Ngram ng WHERE tk.nid IN (SELECT tkk.nid FROM Token tkk, Token tk0 WHERE tk0.position=0 AND tk0.surface='notebook' AND tkk.nid=tk0.nid) AND tk.nid=ng.nid;
	What n-grams contain the pattern "cat [past-tense verb] mouse", where "cat" and "mouse" are nouns? `get_contexts(words=["cat", None, "mouse"], posids=[11, 28, 11], relative_position=True)` SELECT ng.freq, tk.nid, tk.tid, tk.surface, tk.position, tk.posid, tk.depid, tk.headid FROM Token tk, Ngram ng WHERE tk.nid IN (SELECT tkk.nid FROM Token tkk, Token tk0, Token tk1, Token tk2 WHERE tk0.surface='cat' AND tk0.posid=11 AND tkk.nid=tk0.nid AND tk1.position=tk0.position+1 AND tk1.posid=28 AND tkk.nid=tk1.nid AND tk2.position=tk1.position+1 AND tk2.surface='mouse' AND tk2.posid=11 AND tkk.nid=tk2.nid) AND tk.nid=ng.nid
`get_parent_tokens`, `get_child_tokens`	*What verbs is the adverb "lazily" used to describe?* `get_parent_tokens('lazily', parent_posid=28, child_posid=20, child_depid=40)` SELECT ng.freq, tk.surface, tk.posid, tk.depid FROM Token ch, Token tk, Ngram ng WHERE ch.surface='lazily' AND ch.depid=40 AND ch.posid=20 AND tk.posid=28 AND tk.tid=ch.headid AND tk.nid=ng.nid;

`get_histogram`	*What are the frequencies of surface strings used as objects of the verb "take"?*
	`get_histogram('surface', output_predicates={'is_child': True, 'depid': 14},`
	`input_predicates={'surface': 'take', 'posid': 27})`
	SELECT SUM(ng.freq), tkout.surface AS surface_out FROM Token tkin, Token tkout, Ngram ng
	WHERE tkin.nid=ng.nid AND tkout.nid=ng.nid AND tkin.surface='take' AND tkin.posid=27
	AND tkout.depid=14 AND tkin.tid=tkout.headid GROUP BY surface_out;

Table 1: Some example queries generated by the query building functions with their natural language equivalents and the function calls used to generate them, simplified for space. (Joins on the the dependency type and part-of-speech label tables are not shown here, but exist in the actual implementation.)

entire dataset. For example, an extension of our work could allow the user to store the results of a large query in a new relation, and then specify that they are running queries over that relation rather than the full dataset. Though we do not implement this, it would not be difficult to incorporate this functionality into our existing framework.

4.2 Other types of queries

Searching for n-grams containing a single surface string is a relatively simple query, containing one subquery to match tokens with this surface string, with the full query selecting all of the tokens in matching n-grams. Other queries we we do may require more joins, depending on the complexity of the pattern specified by the user. In test queries run during development, we found that many of our more complicated queries took a matter of several minutes on Myria, but only a matter of seconds to a few minutes on our most optimized plan.

With this anecdote and the results of the previous section in mind, we believe that similar benchmarking tests on these queries would show an even more drastic gain in performance over Myria's query plan alone, though this is left to be shown more empirically.

We report approximate query times for more complex queries below. Though these results are less formal than the previous section, they give an approximate sense of how long it takes to run other queries in our optimized system, and demonstrate that most of the runtimes are viable.

- **Aggregation queries**
  - Part of speech and dependency histograms over entire dataset: 6 minutes
  - Attribute statistics over a surface string: < 1 minute
- **Context queries**
  - Children or parents of token, with surface + other constraints: < 1 minute
  - Surface sequences with wildcards (e.g., "cat * mouse"): < 5 seconds
  - Dependency tree match: < 5 minutes

5. RELATED WORK

A number of tools have been devised to aid NLP researchers in leveraging large n-gram datasets easily and effectively. [1] and [2] both describe methods of searching the *Google Web 1T n-grams* dataset, which is an (unannotated) n-grams dataset with counts over sequences of words up to length 5. In particular, [1] presents a generalized query language that takes advantage of external resources for conducting similarity searches over the n-grams, but little attention is given to the performance of this language. [2] present a system called Linggle, which stores the n-grams in an HBase NoSQL database and enable search over synonym matches also using external resources.

One of the most basic types of queries over an n-gram dataset are wildcard searches – that is, searching for n-grams where one or more of the words are not fixed, e.g. "the dog is *". [6] describes a number of methods for dealing with n-grams data in the form of flat files, one of which involves "rotated n-grams", which stores a separate copy of each n-gram from different starting points. Work that places the n-gram data in a database (e.g. [1], [3]) acknowledges this need to query over n-grams by context by computing indices computed over each word and its position.

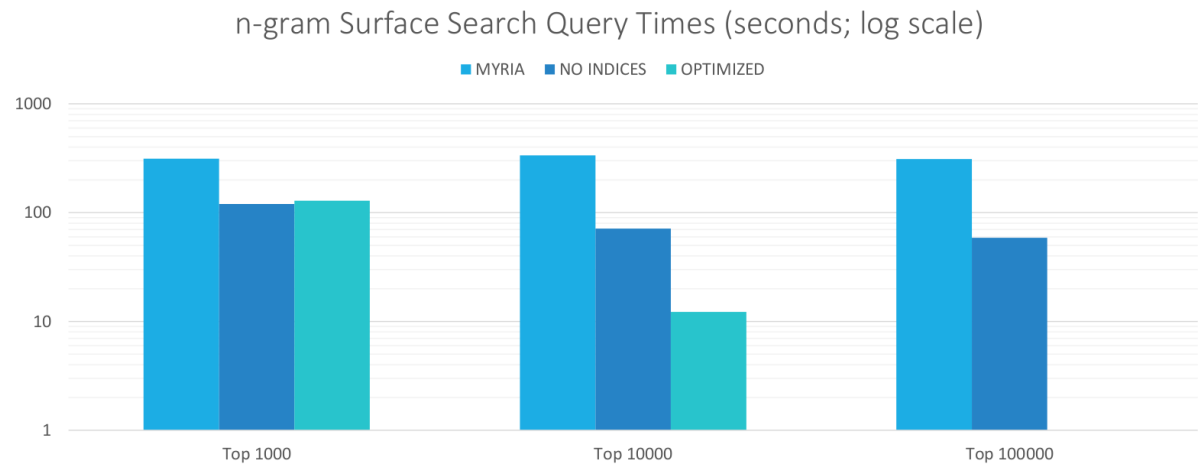

Figure 3: A comparison of the runtime of a query for n-grams containing a surface token, over words sampled from increasingly larger sets. (The lack of a bar for optimized on the rightmost chart is due to the fact that queries took less than a second on average.)

The Google Books N-gram Viewer provides an interactive query interface that allows the use of wildcards and part-of-speech, and the data behind it is described in [7]; however, the paper does not describe the architecture behind querying such a large dataset. Furthermore, to our knowledge, there has not yet been any published work on similar tools for the syntactic n-grams dataset, which contains more detailed, higher-order dependencies than [7].

6. CONCLUSIONS AND FUTURE WORK

In this project, we loaded data into a distributed DBMS, Myria, and created a Python API to enable general linguistic queries over a very large n-grams dataset. We found that in order to get reasonable performance from the database, it was necessary to tune the data well and manually write distributed query plans that took advantage of the particular way we chose to lay out the data. The fact that we were eventually able to achieve fast performance on this dataset was heartening, but also demonstrates some of the current difficulties in using modern distributed DBMSs. Optimizing query plans and physically tuning distributed databases still requires a lot of knowledge about both the nature of the data and the functionality of the database, and simplifying this process is a broad but extremely important research direction. If Myria could automatically create query plans as performant as the ones hand-written for this project, this would be very exciting!

From the NLP perspective, there are still a few important queries that are not supported by our API. Currently the only kind of query regarding words in the n-gram requires an exact match of the word. For instance, a query for n-grams containing "bet" will not match n-grams containing "betting". Ideally, the system could be extended to match words on lemmas, perhaps via an additional attribute, a custom data type/UDFs, or a user option in the API to search for all surface forms of a particular word (e.g., an OR for every verb conjugation of "bet").

This project specifically dealt with querying a read-only dataset. However, what if we also had new parses streaming in from a web scraper? In this case, it may be necessary to change our choice of database and think carefully about how to organize the data in this scenario, especially since our current performance is so reliant on indexes.

7. REFERENCES

[1] M. Aleksandrov and C. Strapparava. Ngramquery - smart information extraction from google n-gram using external resources. In N. Calzolari, K. Choukri, T. Declerck, M. U. Doğan, B. Maegaard, J. Mariani, J. Odijk, and S. Piperidis, editors, *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC-2012)*, pages 563–568, Istanbul, Turkey, May 2012. European Language Resources Association (ELRA). ACL Anthology Identifier: L12-1429.

[2] J. Boisson, T.-H. Kao, J.-C. Wu, T.-H. Yen, and J. S. Chang. Linggle: a web-scale linguistic search engine for words in context. *ACL 2013*, pages 139–144, 2013.

[3] A. Carlson, T. M. Mitchell, and I. Fette. Data analysis project: Leveraging massive textual corpora using n-gram statistics. Technical report, DTIC Document, 2008.

[4] Y. Goldberg and J. Orwant. A dataset of syntactic-ngrams over time from a very large corpus of english books. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*, pages 241–247, Atlanta, Georgia, USA, June 2013. Association for Computational Linguistics.

[5] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 881–884. ACM, 2014.

[6] D. Lin, K. Church, H. Ji, S. Sekine, D. Yarowsky, S. Bergsma, K. Patil, E. Pitler, R. Lathbury, V. Rao, K. Dalwani, and S. Narsale. New tools for web-scale n-grams. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may 2010. European Language Resources Association (ELRA).

[7] Y. Lin, J.-B. Michel, E. L. Aiden, J. Orwant, W. Brockman, and S. Petrov. Syntactic annotations for the google books ngram corpus. In *Proceedings of the ACL 2012 system demonstrations*, pages 169–174. Association for Computational Linguistics, 2012.