Benchmarking Open Source NoSQL Databases Performance on NLP Queries*

Extended Abstract†

Zhihang Dong
University of Washington
Seattle, WA, USA
zdong@uw.edu

Tongshuang Wu
University of Washington
Seattle, WA, USA
wtshuang@cs.washington.edu

ABSTRACT

NoSQL databases are gaining increased amount of attention due to their powerfulness, flexibility and for many of them, reliability and replicability. This popularity results in a number of open-source NoSQL data management tools. However, there is not a "golden standard" on which one to choose over others. Different tools exhibit varying features and running performance, which makes it difficult for software architects to select an appropriate one. Here, we start the inventory of benchmarking performance on NLP tasks with a medium-sized Q&A document. Specifically, we focus on two simplest while fairest NLP tasks: bi-gram and tokenization statistics. We conduct a comparative study of some of the most popular NoSQL databases, including CouchDB, MongoDB, OrientDB, Redis, Neo4j and Cassandra. We first benchmark the running performance of these tools with two tasks described above on the Stanford Question Answering Dataset (SQuAD). We then benchmarked the transferring efficiency using various formats of data. Our contribution is two-fold. First, for software architects, we point out potentials of running performance optimizations. Second, for artificial intelligence scientists, we provide a "road map" of selecting the most appropriate NoSQL database management tools.[1]

CCS CONCEPTS

• **Information systems → Query planning**; • **Computing methodologies → Information extraction**;

KEYWORDS

NoSQL, CouchDB, MongoDB, HBase, Redis, Neo4j, Cassandra, Bigram, Tokenization, Statistics

ACM Reference Format:
Zhihang Dong and Tongshuang Wu. 2018. Benchmarking Open Source NoSQL Databases Performance on NLP Queries: Extended Abstract. In *Proceedings of CSE 544 Principles of Database Management (544'18).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.475/123_4

*Produces the permission block, and copyright information
†The full version of this paper will be available early March
[1]This is tentative, we may add/reduce the number of NoSQL DBMS tested.

1 INTRODUCTION

In the past decade, natural language processing has grown exponentially thanks to the development of deep learning, linguistics and data mining. Applications of natural language processing are attributed to the rich, well-labeled and linguistically-sound "big" data. Therefore, appropriate handling of large text corpus and potentially noisy, high-dimensional text data is indispensible to the success of development on natural language processing. For this endeavor, cloud storage and advancement on data management system have played a crucial role. At the same time, the growing open-source NoSQL database management tools provide exciting opportunities for AI scientists, especially those who want a query-based text processing. The use of term 'NoSQL' can be understood as "open-source, distributed, non-relational databases" that deemphasize SQL-style querying as the objective of data stores [16]. Typically, such data management systems come with the advantage of simple and flexible data models that are not relational, and the abilities to provide high availability (such as CouchDB) and horizontal scalability over various commondity servers. These characteristics are product-friendly, and more importantly, suitable for cloud data management systems. Since many of the NLP research requires processing of large-scale datasets "on cloud", NoSQL options are considered ideal data stores for tasks powered by natural language processing.

Given the diversity of NoSQL data stores, the main challenge described in this study can be summarized as "which NoSQL database should I use." Unfortunately, with the development of shoppable items in NoSQL database market, there is not a uniformly correct answer. That is because many NoSQL data stores are designed with objectives to excel at one or two specific tasks. In subsequent discussions of these NoSQL databases in Section 2, this is referred as "data models." In this paper, we benchmark a set of NoSQL data stores using two simple NLP tasks: tokenization statistics and bigram. A token is the smallest measurable-unit-of-interest in NLP typically consisting of the name (of the token itself) and possibly an assigned value. Tokenization statistics in our paper simply means a frequency statistics of tokens. A bigram is then two adjacent elements from a given set of tokens that typically consist of syllables, letters and words. With the statistics of which token is likely preceding/following another, bigrams generate the conditional probability of a token given the preceding token, which is following the conditional probability on Equation 1 below.

$$P(W_i|W_{i-1}) = \frac{P(W_{i-1}, W_i)}{P(W_{i-1})} \quad (1)$$

Here, the probability $P(\cdot)$ of a token W_i given the preceding token W_{i-1} is the probability of their bigram divided by the probability of that preceding token.

Hence, we benchmark a set of NoSQL database tools under different umbrellas of data models with the task of data import, transfers, tokenization statistics and bigram[2]. The spirit is to bring up NoSQL databases of different data models to compare. For the purpose of conciseness, we only choose the most representative one or two from each data model. Specifically, OrientDB [6] is a document-based graph database written in Java; CouchDB [2] is a document and bigtable stores database written in Erlang; Cassandra [1] is a large-dataset store database written in Java; Redis [5] is a fast, disk-backed in-memory database written in C; MongoDB [3] is a JSON document store written in C++; Neo4j [4] is a graph database written in Java. We will give a more detailed description of each NoSQL product in Section 2.

The rest of this paper is organized as follows. Section 2 presents the background and related work. Section 3 discusses our benchmarking platform and methodology. We include our experiment results in Section 4, which followed by our discussion on the experiment results in Section 5. We conclude the paper and suggest directions for future work in Section 6.

2 BACKGROUND AND RELATED WORK

In this section, we start with a few most relevant papers in benchmarking studies of NoSQL database systems following by a brief argument of new insights brought by this paper. Then, we give an overview of NoSQL databases participated in this benchmarking study. Finally, we give a brief background introduction of the NLP tasks we experiment with.

2.1 Related Literautre

There are many insightful benchmarking studies in various aspects of NoSQL databases [7, 10, 16, 22]. Perhaps the most relevant paper is by Grolinger et al. (2013) [16], which compared the querying capabilities, partitioning, replication, consistency and concurrency control capabilities as well as security features of numerous NoSQL data stores (such as Redis, Riak, Cassandra, HBase, DynamoDB, Neo4J, VoltDB, CouchDB, MongoDB) in impressive details. Interested readers should look at their comparisons for a more in-depth overview of different NoSQL products. Our comparison at Table 1 will only incorporate to a small amount of them. On the other hand, there have been significant development on how database queries help NLP-related tasks. Instead of using a chronological order, I used a logic of AI-toolkit development. Zhang et al. (2013) [23] discussed how to use familiar data-processing languages to complete a variety of statistical inference works. Conditional random fields appear as the probabilistic models for segmenting and labeling sequence data and tabular data extraction [19, 20], which were arguably the start of machine learning applications on natural language. So far, natural language processing relies on a well-performed database system that "contextualize" tables of text data [14]. Another key development of NLP-serving databases are the emergence of passion on data mining. In this [15] cornerstone paper, the author discussed how building a complete language model for a language using all the text available on the Web is possible using the data mining techniques. There are many applications that attempt to "connect the dots between database designs and natural language processing. Actually, similar efforts can be traced as early as to 1998, where WordNet [13] emerges, which is a "universal database" of word senses. Other examples include the "common sense database" introduced by Eagle et al. (2003) [12]. Bollacker et al. (2008) [9] created a "Freebase" that creates graph database for structuring human knowledge collaboratively.

Although these studies demonstrate the potential of using data management system to empower natural language processing tasks and research, these applications are narrow in the sense that they are so specifically designed for a particular task and not so generalizable. There have yet been many papers discussing how general relational databases or NoSQL databases could potentially accelerate NLP tasks and their engineering. They are more of "database structures embedded to the specific data format and processing tasks". Then, it comes to the key paper of Jain and Howe (2018) [18]. In this paper, the authors discussed how to use vector representations learning methods from NLP to accelerate SQL queries with generalized workload analytics. When NLP aid the database management questions of query optimization, we finally arrive at our question: what is the best database product for data in NLP tasks. A good answer yet exists for this. This motivates us to find the best database product that outperforms in three dimensions:

- Expressiveness: What they can do with text data? Whiledocument storage is necessary, what types of query this product supports with a reasonable effciency and interests for large text data?
- Speed: How soon are the jobs such as import, processing and analysis expected to conclude?
- Compatibility: How well does the structure in a given product translate/transfer to others?

Our benchmarking study develops around these three objectives. Since we do not have time to examine every NoSQL database, we want to have a representative (while small) sample of NoSQL databases. In the next paragraph, I will describe the reasoning of our inclusions.

2.2 NoSQL Data Models

As described, the family of NoSQL products can be further classified based on the data model they employ [16]. In this paper, the same classification scheme provided by Hecht and Jablonski [17] is used. Specifically, we are looking at the most representative and popular NoSQL products of each data model that include key-value stores (Redis), column-family stores (Cassandra), document stores (CouchDB, MongoDB) and graph databases (OrientDB, Neo4j). The next few paragraphs provide a brief introduction of each of the data model, and Table 1[11] provides a summary of various characteristics (such as querying capabilities, partitioning, replication and consistency) for each of the NoSQL product.

2.2.1 Key-Value Pairs. Key-value stores provide a data model that entirely comprised of key-value pairs [17], which formulate

[2]I hope we have time to do updates by the end of the quarter

[11]Legends: [3]: CLI-supported; [4]: API-supported;[5]: Eventual Consistency; [6]: Strong Consistency [7]: thrift-interface-supported; [8]: Consistent Hashing; [9]: Range-Partitioning [10]: Configurable *: With modification

Table 1: Comparison of Different NoSQL products in this Benchmarking Study

NoSQL Database	Language	MapReduce	API Support	Partitioning	Replication	Consistency
Redis	C	No	C [3], A [4]	No	Master-Slave	E [5], S*[6]
Cassandra	Java	Yes	C, A, T[7]	C [8], R[9]	Masterless	S
MongoDB	C++	Yes	C, A	R	Master-Slave	C[10]
CouchDB	Erlang	Yes	A	C	MultiMaster	E
Neo4j	Java	No	C, A	No	Master-Slave	E
OrientDB	Java	No	C, A	No	MultiMaster	E

an associative dictionary. Each pair has the key identify the value associated with the retrieving key. Such value can be integer, string, array or an object. Therefore, this is a schema-free data model. The advantage of using key-value stores is its high efficiency of distributed data storage, while the limitation of using key-value stores is its unsuitability of handling data-level queries and indexing as the only way for key-value stores to perform queries is through keys [16]. That means for highly relational and structured data, key-value stores may not be a viable choice.

2.2.2 Column-Family Data Stores. Column-family data stores are mostly inspired by Google Bigtable [11, 16], in which data are aggregated into columns affiliated to different several column families. The logic was presented in Figure 1, which is a graph quoted from Grolinger et al. (2013) [16]. Here, each row consists of a unique row key serving as the primary key, as well as a few column families, which is varying by row. Column-family data stores are stronger in indexing and querying than the key-value stores counterpart, but still requiring relations to be implemented in the client application [16].

2.2.3 Document Stores. Implying by their name, document stores locate documents inside the data store by a key. The representation of documents uses JSON or format of similar ilk. For example, MongoDB [3] contains another grouping method called collection, in which each document has a unique key. The advantages of using document format is the possibilities of incorporating complex data structures and functionality. Each document may have its own structure, which provides great flexibility. This is considered a huge advantage over the relational database management system.

2.2.4 Graph Database. Graph databases use graph theory, which use vertices to represent objects and edges to build relationship links (by connecting such vertices) [8] to connect related objects. The advantages of using graph databases is obvious: such databases are ideal to handle interconnected data, which could potentially become very useful in text data. On the other hand, the graph databases are less efficient in other aspects of database performance compared to its counterparts in Section 2.2.1 - 2.2.3.

2.3 Two Simple NLP tasks

Here, we briefly review the definition and purposes of the two simple NLP tasks that we will perform as the experiment sets in this study.

2.3.1 Tokenization Statistics. If we are given a character sequence and a defined document unit, tokenization chopped such predefined sequences and documents up into pieces, which is referred as *tokens*. At the same time, this function throws away certain characters, such as punctuation. Here is an example of tokenization:

Input: Oh! Dear, Enjoy your show! Show your power!;

Output: [Oh] [Dear] [Enjoy] [your] [show] [show] [your] [power]

These tokens are often loosely referred to as terms or words. As defined by the Stanford NLP Group[24], a *token* is 'an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing'. In the example above, *your* and *show* have token count of 2, while others have count of 1.

In our experiments, we refer tokenization solely as counting tokens from a post-treatment JSON/BSON file, which has by-and-large broke documents into tokens and saved them as new fields. The tokenization task is to extract them from different data store types, count them and give a summary statistics however a user prefer.

2.3.2 Bigram. Bigram is the process of handling the set of co-occuring tokens within a predefined sample space $\mathcal{D}$. When computing the bi-grams, it is just as easy as to move pointer one token forward. In such way, we can generate a set of candidate combinations as $\mathcal{X}$, which helps generating probability distribution function using the Equation 1 above on Page 1. For example:

Input: $\mathcal{D}$ = Andy eats sweet corns!;

Output: $\mathcal{X}$ = [Andy eats] [eats sweet] [sweet corns]

Each two word combinations here have $P = \frac{1}{3}$ of occurence. In our experiment, we solely refer such token combinations as the word-combinations (no punctuations).

3 EXPERIMENTS

3.1 Data

The dataset we are using for this experiment is the Stanford Question Answering Dataset (SQuAD) [21], which is a "new reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage. With 100,000+ question-answer pairs on 500+ articles, SQuAD is significantly larger than previous reading comprehension datasets" [21]. Under the time constraints of this paper, we use solely the well-labeled developer dataset. The information about this dataset can be referred from Table 2. More information regarding the dataset can be assisted with Appendix A1.

Table 2: Squad Dev Dataset Information

Item	Count
Article	48
Paragraphs	2,067
Questions	10,570
Answers	37,575
Total Possible Tokens	152,318

3.2 Pre-Experiment Processing and Structures

While MongoDB and CouchDB can handle JSON files directly, other datasets need assistance digesting such dataset. For Redis, since it only consider key pairs, we remove all identifying information of higher-level keys but only the lowest document-level data. In other words, we only leave each paragraph with tokens nested in the simplest form (no compound) of JSON format. We then use the ReJSON module from Redis to load the JSON file. For Cassandra, since we need to make sure if every document serves as a column and each text (token) serves as a sub-level column nested within higher-level columns. We make a separate arrangement to make those higher-level structures into temporary tables to make the CQL writing more convenient. Finally, the toughest part of pre-experiment processing is for the graph databases, for which we need to manually label the relationships between each two tokens and within different levels of higher-level documents, according to the information provided in Appendix A1.

3.3 Tokenization Statistics: Sample Queries

3.3.1 Sample Tokenization Query in Key-Value Pairs Stores. Since tokenization counting and statistics assume nothing about the internal relationship between token and sentences/paragraphs. It is probably extremely fast to do it with key-value pairs. Essentially the solution is a one-line code producing how many words (values) are with the text "keys".

```
./redis-cli KEYS "text:*" | wc -l
```

3.3.2 Sample Tokenization Query in Column-Family Stores. Cassandra uses the Cassandra Query Language (CQL) as its query language. Here is an example of looking at the frequency given a 'part of speech (POS)' in the series of questions:

```
SELECT psid, count(text *) as token
FROM question
GROUP BY psid
ORDER BY token
LIMIT 50
```

3.3.3 Sample Tokenization Query in Document Stores. Here is an example of how to do tokenization statistics in MongoDB with an example of counting the token 'what' through all collections (equivalent entries) in the dataset 'questions'.

```
db.questions.aggregate([
{$match: {text: {$eq: "What"}}},
{$count: "passing_scores"}
])
```

3.3.4 Sample Tokenization Query in Graph Database. In graph database such as Neo4j, the count is representated as the "relationship" of "have" when a given condition exists. In our case, we treat each scenario as an aggregate, and count the frequency of a given token exists in these aggregates.

```
MATCH (n { name: 'when' })-[r]->() # Example
RETURN type(r), count(*)
```

3.4 Tokenization Statistics: Bigram

3.4.1 Sample Tokenization Query in Key-Value Pairs Stores. It is not possible to write bigram queries with key-value pairs stores because the internal relations of parts within a text were not preserved. However, it is possible to do bigram if we can rename the text key into a sequenced text key. This violates the benchmarking standard, so we cannot count the post-treatment running time as a valid one.

3.4.2 Sample Tokenization Query in Column-Family Stores. It is easy to write bigram on CQL: essentially the bigram is simply calculated by the frequency of distinct text component following the prior text component. However, the disadvantage is that we need to loop through this query a lot of times :

```
SELECT DISTINCT(text 2) as follow, COUNT(text2) as freq
FROM question
WHERE text 1 = 'Prior'
GROUP BY text2
ORDER BY freq
```

3.4.3 Sample Tokenization Query in Document Stores. Here is an example of how to do tokenization statistics in MongoDB with an example of counting the token 'what' through all collections (equivalent entries) in the dataset 'questions'.

```
db.questions.find(text1:
{$match: {text1: {$eq: "What"}}})
```

3.4.4 Sample Tokenization Query in Graph Database. In graph database such as Neo4j, the count is representated as the "relationship" of "have" when a given condition exists. In our case, we treat each scenario as an aggregate, and count the frequency of a given token exists in these aggregates.

```
MATCH (text2)<-[:FOLLOWS]-(ans:text3)
WHERE text3 !="\whitespace"
RETURN text2, text3, count(text3)
```

There is another situation where text 3 does equal to a whitespace and we need to manually adjust that. For the simplicity of showing the ideas of codes, we just assume text3 is not whitespace.

4 RESULTS

Table 3 provides the experiment results of data import. As discussed in Section 3.2, for all NoSQL products, we import a JSON file or file of similar ilk(BSON, for example) with minimal necessary modification. As we have discussed in Section 3, the import speed in Redis is meaningless since it is an in-memory datastore. The differences of data import for the NoSQL products are not trivial: MongoDB has the fastest import while CouchDB has the slowest import, with latter takes two times more of runtime, so we suspect that data

Table 3: Benchmark Experiment 1: Relational NLP Files Upload

NoSQL Database	Time Elapsed for Full Import (s)
Redis	NA
Cassandra	4.471
MongoDB	3.928
CouchDB	8.514
Neo4j	4.690
OrientDB	6.055

Table 4: Benchmark Experiment 2: Tokenization Statistics

NoSQL Database	Time Consumed for all Queries (s)
Redis	2.545
Cassandra	3.960
MongoDB	5.389
CouchDB	9.030
Neo4j	9.525
OrientDB	9.066

store method is not the major cause of import efficiency: rather, the system design and the language that built these systems are playing more important roles. MongoDB and CouchDB, on the other hand, provides the easiest data import for users, especially since they take BSON/JSON files directly without a separate API. In this benchmarking example, we can incorporate higher-level document as a collection, and focus on individual, independent tokens. It is also possible that since our dataset is not very large, the system design and other factors interfere the influence of datastore methods. Table 4 introduces the tokenization statistics runtime. It is notable that Redis was incredibly fast on tokenization. Actually, Redis runs equally fast in paragraphs with the tokenizer looks like the codes below:

```
def tokenize(content):
words=set()
for match in WORDS_RE.finditer(content.lower()):
-- iterator of all words over the content
word = match.group().strip("'")
if len(word)>= 2:
words.add(word)
return words - STOP_WORDS
```

Based on Table 4, graph data stores are not working very well with the tokenization statistics task: because we need to arbitrarily rely on relationships of `'belong-to'` for a token to a certain `'document'`. Column-family data stores such as Cassandra work also very well here since we only need to loop through each `'document'` subcolumn to extract all text tokens, which is very efficient with a JSON data of text format. Similarly, document stores provide a quick extraction of tokens but through each document independently.

Finally, it is easiest to provide bigram through graph database because idea of *n*-gram is essentially looking at relations, which are defined by the edge between one token (as a node) and another. For this reason, as shown on the Section 3.4.4., we can directly query the `'follows'` relationship and map the token combinations from 'follow' relation-pairs. Therefore, as evident by the Experiment 3 from Table 5, the graph data stores is the best option for such queries. Using column-family data stores to run this query can become problematic, since we have to iterate the same token for a maximum number of times, and we do not have information to assume regarding when to stop.

Table 5: Benchmark Experiment 3: Bigram (full 9 pairs)

NoSQL Database	Time Consumed for all Queries (s)
Redis	NA[12]
Cassandra	80.01
MongoDB	38.54
CouchDB	—
Neo4j	27.47
OrientDB	30.02

5 CONCLUSIONS

5.1 Discussion

This paper provides a simple benchmarking experiment with six NoSQL database stores. We found, not surprisingly, a different data store outperforms others for each task, since each represents a different data model. Indeed, the limitation of this short paper makes us unable to discover the strengths of some NoSQL products we nominate here, such as CouchDB, which makes the availability across different user-end platforms really easy but underperformed in many metrics we have here. Actually, as the developers of CouchDB admitted, using CouchDB as a combined standalone database and application server is not a great option [2]. For their documentation, they named many significant limitations to a pure CouchDB web server application stack, such as "fully-fledged fine-grained security, robust templating and scaffolding, complete developer tooling" [2]. For other database stores, the limitation is significant, too. For example, graph databases such as Neo4j can be great in tasks involving *dependency*, which is actually pretty common in NLP tasks such as parsing and tagging since the existence of grammar. However, such structure also makes the data storage and query more complicated than necessary in simpler case such as counting.

At the same time, column-family store database makes the token-based statistics really easy, so the column-family store database such as Cassandra shines when we have tasks such as token statistics and n-gram where classification is crucial. Document stores such as MongoDB makes the data retrieval tasks incredibly faster than its peers. Both products also have a great API support. Typical key-value approaches does not work very well in either case, and the opportunities to explore this expansion becomes the topic for future research.

The key here is, we now have an ecosystem of modules and frameworks for developers to choose from. We believe that web developers should pick "the right tool for the right job". For example, considerusing CouchDB as your database layer, in conjunction with any number of other server-side web application frameworks, such as the entire Node.JS ecosystem, Python's Django and Flask, PHP's

Drupal, Java's Apache Struts, and more, and using Neo4j when the underlying text dependency is important. The combination of multiple NoSQL database stores make it possible to use the combined strengths of them and make the NLP work more efficient.

5.2 Limitations

For this study, we have three major limitations yet answered.

- Does the two queries representative enough in terms of natural language processing tasks?
- How can we prove we have optimized the query codes for the task we intended to do with each NoSQL database?
- Is it fair to compare across these platforms when import data are not in the same format? a.k.a. what is a fair game for this benchmarking

It is evident though that NLP tasks involve more heavily on predictions, explorations and analytics should not rely on query tasks. However, we choose the two tasks because those characteristics on tokenization and bigrams are relatively less important, and perhaps a bit more trivial. It is not a bad news that query language are quite capable and competitive in doing this. The comparability of initial files we load into the six database products may constitute a problem, but as discussed in Section 3.2., we *have to* pre-process the data.

Another possible discussion is whether we should benchmark relational database (such as postgre). We don't think relational database (such as postgre) is good at this work because our data structure is a quite sophisticated, nested stHructure, which naturally producing relationship edges. Scholars interested in this area are suggested to consider the use of datalog in this case, which might be helpful.

A APPENDIX

A.1 Data Overview

articles
'The article info.'
- 'aid': <int> Article id
- 'title':<str> the title

paragraphs
'Paragraph info.'
- 'aid': <int> Article id
- 'pid': <int> Paragraph id
- 'qid': <str> Question id (use to get 'aid' and 'pid'); only happens when a paragraph is edited. Default -1.
- 'vid': <int> Version id
- 'doc': <dict[]> The token wise array with rich info.
- 'idx': <int> the idx of the span in doc
- 'text': <str> text of one word
- 'ner': <str> named speech recognition
- 'lemma':<str> lemma in lower case
- 'pos': <str> part-of-speech tag in detail
- 'tag': <str> simplified
- 'whitespace':<str> the space
- 'sid': <int> sentence id

questions
'Basic question info'
- 'qid': <str> Question id (use to get aid and pid)
- 'vid': <int> Version id
- 'doc': <dict[]> The token wise array with rich info.
- 'idx': <int> the idx of the span in doc
- 'text': <str> text of one word
- 'ner': <str> named speech recognition
- 'lemma':<str> lemma in lower case
- 'pos': <str> part-of-speech tag in detail
- 'tag': <str> simplified
- 'whitespace':<str> the space
- 'sid': <int> sentence id

ACKNOWLEDGMENTS

N/A

REFERENCES

[1] 2018. Apache Cassandra. (Feb. 2018). Retrieved Feburary 24, 2018 from http://cassandra.apache.org/
[2] 2018. Apache CouchDB. (Feb. 2018). Retrieved Feburary 24, 2018 from http://couchdb.apache.org/
[3] 2018. MongoDB for GIANT Ideas | MongoDB. (Feb. 2018). Retrieved Feburary 24, 2018 from https://www.mongodb.com/
[4] 2018. The Neo4j Graph Platform | The No. 1 Platform for Connected Data. (Feb. 2018). Retrieved Feburary 26, 2018 from https://neo4j.com/
[5] 2018. Redis. (Feb. 2018). Retrieved Feburary 24, 2018 from https://redis.io/
[6] 2018. Why OrientDB| Open Source NoSQL Multimodel Database | OrientDB. (Feb. 2018). Retrieved Feburary 24, 2018 from https://orientdb.com/why-orientdb/
[7] Veronika Abramova and Jorge Bernardino. 2013. NoSQL databases: MongoDB vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering*. ACM, 14–22.
[8] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1.
[9] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. AcM, 1247–1250.
[10] Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *Acm Sigmod Record* 39, 4 (2011), 12–27.
[11] F Chang, J Dean, S Ghemawat, WC Hsieh, DA Wallach, M Burrows, T Chandra, A Fikes, and R Gruber. 2006. Bigtable: A distributed structured data storage system. In *7th OSDI*, Vol. 26. 305–314.
[12] Nathan Eagle, Push Singh, and Alex Pentland. 2003. Common sense conversations: understanding casual conversation using a common sense database. In *Proceedings of the Artificial Intelligence, Information Access, and Mobile Computing Workshop (IJCAI 2003)*.
[13] Christiane Fellbaum. 1998. *WordNet*. Wiley Online Library.
[14] Vidhya Govindaraju, Ce Zhang, and Christopher Ré. 2013. Understanding tables in context using standard NLP toolkits. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Vol. 2. 658–664.
[15] Gregory Grefenstette. 2007. Conquering language: Using NLP on a massive scale to build high dimensional language models from the web. In *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, 35–49.
[16] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. 2013. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications* 2, 22 (Dec. 2013), 1–24. https://doi.org/10.1186/2192-113X-2-22
[17] Robin Hecht and Stefan Jablonski. 2011. NoSQL evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE, 336–341.
[18] Shrainik Jain and Bill Howe. 2018. Query2Vec: NLP Meets Databases for Generalized Workload Analytics. *arXiv preprint arXiv:1801.05613* (2018).
[19] John Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. (2001).
[20] David Pinto, Andrew McCallum, Xing Wei, and W Bruce Croft. 2003. Table extraction using conditional random fields. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM, 235–242.

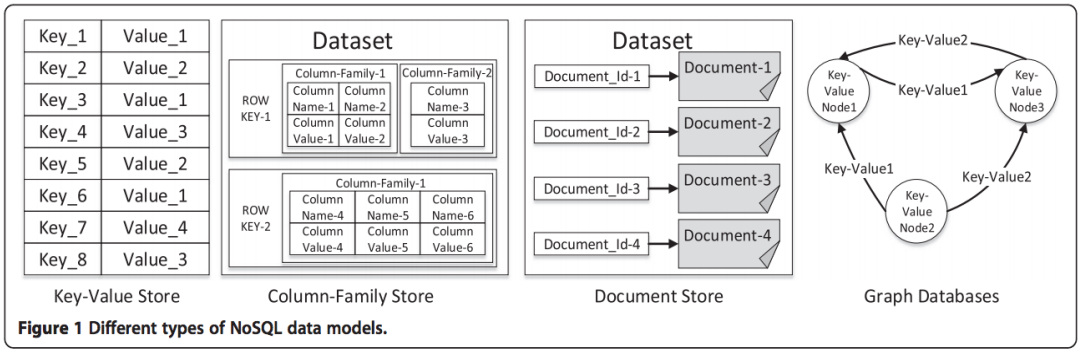

Figure 1 Different types of NoSQL data models.

Figure 1: Logic of Different Data Models, by [16]

[21] Pranav Rajpurkar. 2018. The Stanford Question Answering Dataset. (Feb. 2018). Retrieved March 9, 2018 from https://rajpurkar.github.io/SQuAD-explorer/

[22] Bogdan George Tudorica and Cristian Bucur. 2011. A comparison between several NoSQL databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE, 1–5.

[23] Ce Zhang, Vidhya Govindaraju, Jackson Borchardt, Tim Foltz, Christopher Ré, and Shanan Peters. 2013. GeoDeepDive: statistical inference using familiar data-processing languages. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 993–996.