NLonSpark: NL to SQL translation on top of Apache Spark

Apostolos Glenis
Athena Research Center
Greece
aglenis@athenarc.gr

Georgia Koutrika
Athena Research Center
Greece
georgia@athenarc.gr

ABSTRACT

Traditional ways of interfacing with databases such as SQL although declarative are not intuitive enough for non-expert users. Because of this would be helpful if users of a database management system could pose their queries in natural language. As datasets grow larger and larger traditional ways of storing and querying data become obsolete and developers gravitate towards distributed processing frameworks such as Apache Spark. In this paper we present NLonSpark, a framework for Natural Language to SQL translation and execution built on top of Apache Spark. We have also introduced a series of optimization that significantly reduce execution time compared to the naive implementation.

CCS CONCEPTS

• **Information systems → Data management systems**;

KEYWORDS

NL query interfaces, Spark, interactive exploration

ACM Reference Format:
Apostolos Glenis and Georgia Koutrika. 2020. NLonSpark: NL to SQL translation on top of Apache Spark. In *Workshop on Human-In-the-Loop Data Analytics (HILDA'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3398730.3399195

1 INTRODUCTION

As more and more data becomes available, a growing number of end users need to query them. A large portion of those users do not have any knowledge of the database schema or the database query language (usually SQL). Thus, they would rather express their queries in natural language (NL) and have a tool that understands the underlying database schema and converts their query to a valid SQL query. The vision of *NL interfaces to databases* is to make data more accessible for a wide range of non-tech savvy end users with the ultimate goal to 'talk' to a database (almost) like to a human.

Furthermore, with the growing volume of data, it becomes increasingly difficult to store and analyze them on a single machine. Rather than running pre-defined queries to create static dashboards, business analysts and data scientists want to explore their data by asking a question, viewing the result, and then either altering the initial question or drilling into results. This interactive query process over large amounts of data requires distributed processing frameworks such as Apache Spark [20] that can respond and adapt quickly. However, existing NL interfaces assume a relational database management system, and take advantage of its capabilities, including schema information and indexes. To the best of our knowledge, natural language query interfaces over Spark are still missing, and the implications and challenges of porting existing technology from the database to work over Spark have not been examined.

In this paper, we present NLonSpark, a framework that aims to fuse Natural Language-to-SQL translation and execution with the Spark distributed processing framework. The main aim of NLonSpark is to provide a Natural Language Interface on datasets that need to be queried using Apache Spark. To achieve this goal, we modified NaLIR [9], a state-of-the-art NL interface, to use Apache Spark instead of a traditional DBMS. In the process, we ported the most time-consuming part of NaLIR, which used the DBMS, to Apache Spark. We investigate the challenges and tradeoffs of the problem, and we present a series of optimizations that aim to mitigate shortcomings of Apache Spark compared to a traditional DBMS. To the best of our knowledge, this is the first attempt for Natural Language-to-SQL translation and execution on top of a distributed processing framework.

2 RELATED WORK

Several approaches to NL interfaces have been proposed by the database community, including early keyword-based systems [6, 8, 12, 13], and latest NL-based approaches, such as NALIR [9]. In their core, most approaches assume some kind of graph, either a schema graph that represents the database relations as nodes and the joins between them as edges [8, 12, 13] or a tuple graph where the tuples become the nodes [6]. Then, answers to a query are defined as sub-graphs over the complete graph, comprising a subset of the relations and tuples that contain the query keywords and are connected by the joins between them. NALIR [9] is the first to introduce a syntactic parse tree for query representation. We will describe its architecture in more detail in Section 3.2.

In SQLizer [16], the user query is translated into a query sketch (skeleton) using semantic parsing techniques. Type-directed program synthesis is used to complete the sketch into a well-typed SQL query. Finally, repair techniques are used to repair the initial sketch and capture the structure of the desired, final, query.

Deep learning (DL) techniques have been recently proposed for translating NL queries to SQL queries (e.g., [5, 11, 15, 17, 21]). For instance, Seq2SQL [21] is a deep neural network that is trained using a mixed objective combining cross-entropy losses and reinforcement learning rewards for in-the-loop query execution on a database. SQLNet [15] uses a sketch-based approach to generate a SQL query from a sketch. The sketch aligns to the syntactical structure of a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HILDA'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8022-5/20/06…$15.00
https://doi.org/10.1145/3398730.3399195

SQL query. A neural network is then used to predict the content for each slot in the sketch. DialSQL [5] is a dialogue-based structured query generation framework that uses human guidance to boost the performance of existing algorithms by identifying potential errors in the generated SQL query and asking users multi-choice questions to fix it. The authors test their framework using SQLNet as a black box and report performance increase from 61 to 69 % .

For this work, we chose NaLIR for the following reasons:

- It uses a natural language understanding module that does not employ neural networks that are time-consuming to train and more cumbersome to deploy.
- The natural language understanding module understands the structure of the sentence and this might lead to better understanding of complex sentences.
- It can infer more effectively joins when needed compared to DL-based methods (the majority of them handle a single table).
- Existing DL-based approaches do not consider the actual data, hence the resulting SQL, even if it is syntactically correct, it may not be semantically correct, and hence executable over the underlying data [7]. NaLIR will produce queries that can produce results over the underlying data.

3 PRELIMINARIES

3.1 Apache Spark

Apache Spark [20] is a unified engine for distributed data processing. Spark uses the MapReduce [3] programming model but extends it with an abstraction called Resilient Distributed Datasets (RDDs) [18]. Using RDDs, Spark can express a vast number of workloads that previously needed separate processing engines. Example workloads can involve SQL [1, 4], streaming workloads [19], machine learning [10] and graph processing [14].

Apart from RDDs, another Spark abstraction that is relevant to our use-case is Spark DataFrames which are RDDs of records with a known schema. Spark DataFrames get most of their functionality from the DataFrame abstraction for tabular data in R and Python and essentially model a database table. Dataframes provide methods for filtering, computing new columns, and aggregation. In Spark Dataframes, these operations map down to operations of the Spark SQL engine and use all available optimizations, such as predicate push-down, operator reordering, and join algorithm selection.

3.2 Overview of NaLIR

Figure 1 describes NaLIR's architecture.

As we can see in the figure, the *Dependency Parser* takes as input the NL query and uses the Stanford Parser [2] to generate a dependency parse tree, where nodes represent the terms and edges represent the linguistic relationships between them.

The *Parse Tree Node Mapper* maps elements of the parse tree to database elements, i.e., relation names, attributes, and values. To do so, it uses *the database schema graph and use indexing facilities both provided by the underlying RDBMS*. The identification of the select, operator, function, quantifier and logic elements in the query is independent from the database being queried; a dictionary prefilled with a set of phrases – for example, the function node *COUNT* corresponds to the phrase *"number of"* – is utilized.

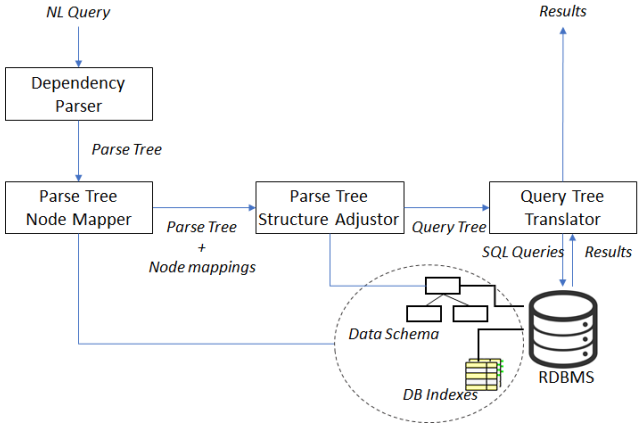


Figure 1: The architecture of NaLIR

Then, both the Parse Tree and the Node Mappings are fed into the *Parse Tree Structure Adjustor* that generates intermediate representations of the query, the *Query Trees*. The *Parse Tree Structure Adjustor* corrects any possible flaws in the structure of the parse tree to make it grammatically valid. To rank candidate query trees, three factors are taken into account: (i) the number of parse tree nodes that violate the grammar, (ii) the mappings between the parse tree nodes and the database elements that can help to infer the desired structure of a parse tree, and (iii) the similarity of a resulting tree to the original linguistic parse tree.

Finally, the *Query Tree Translator* takes as input the Query Tree and produces the final SQL query that is fed to the RDBMS for execution. During the translation, there are two possible cases. On one hand, when the query tree does not contain function nodes or quantifier nodes the translation is quite straightforward. On the other hand, when the query tree contains function nodes or quantifier nodes, the target SQL statements will contain *subqueries*. NaLIR uses the concept of a *block*, which represents a nested subquery, to clarify the inner scopes, and translates the tree one block at a time, starting from the innermost level.

4 NLONSPARK

NLonSpark is implemented on top of Spark and Spark SQL. Spark SQL is a module in Spark which integrates relational processing with SparkâĂŹs functional programming API. It supports querying data either via SQL or via the Hive Query Language.

NLonSpark is based on NaLIR. Porting NaLIR on Spark, we faced the following challenges:

- NaLIR is tied to the underlying DBMS to provide schema information and fetch the underlying data. In our case, we had the additional problem that Spark SQL does not incorporate any schema information. Thus, we need to manually inject schema information about the dataset.
- The Node Mapper is the most time-consuming component because it tries to map query terms to database elements. After porting NaLIR over SPARK, the performance was not on par with the original NaLIR because NaLIR used full-text indices provided by the RDBMS for the query term mapping and text

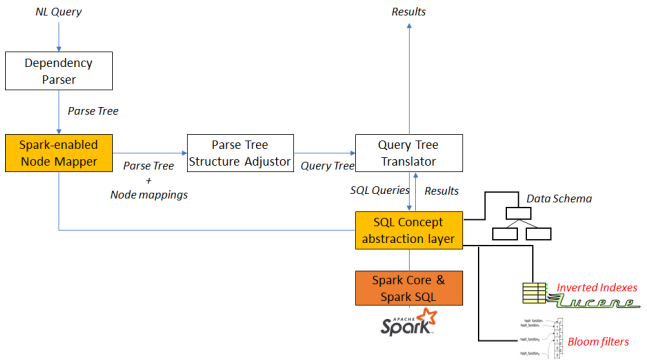


Figure 2: The architecture of NLonSpark

similarity routines. Spark, on the other hand, does not provide textual indexes that we could use.

In a nutshell, NaLIR's reliance on foreign keys and need for low latency string lookups make it a challenging fit for a batch-processing system like Spark. In what follows, we examine how we address these challenges making a first step towards supporting Natural Language-to-SQL translation and execution over Spark.

To port NaLIR on top of Spark and tackle the first challenge, we changed the Node Mapper functionality. Instead of communicating with an RDBMS, it now communicated with an SQL concept abstraction layer that provides the functionality of an RDBMS.

The *SQL concept abstraction layer* achieves two primary goals. First, it provides the necessary schema information that the RDBMS provided for the original NaLIR. Second, it provides indexing capabilities that Spark lacked, by implementing inverted indexes functionality on top of Apache Spark using lucene4s and Spark operations instead of relying on an RDBMS.

The *Spark-enabled Node Mapper* performs these operations:

- *Full Text search*: Since Spark lacked inverted indexes, we implemented this functionality on the SQL concept abstraction layer. We also added bloom filters to avoid inverted index searches where possible.
- *Columns summations*: This functionality provides similarity amongst numerical columns and was implemented using the Dataframe API of Spark.
- *Column name similarity*: This was provided by the SQL concept abstraction layer without calling Spark since it is not a processing-heavy operation.

The available indexes of the SQL abstraction layer take two forms: (a) bloom filters, and (b) inverted indices. We used bloom filters to all textual columns in the dataset to reduce full-table scans of the dataset. We created inverted indices for large textual fields to speed up full-table scans when they do occur. One example where we utilized a full text index is for the text of reviews (in the Yelp dataset that we used in our experiments). The two types of indexes were needed to reach performance that was on par with the centralized implementation.

Bloom filters use a hashing function to determine if an element is part of a set. Bloom filters are a probabilistic data structure that returns that an element is "possibly in set" or "definitely not in set". This means that when the bloom filter returns that the element is not in the set, we avoid a full-table scan. However, if the bloom filter returns that the element is possibly in the set, we have to perform a full-table scan to determine if the element actually exist in the set. For the bloom filter implementation, we used Spark's built in bloom filters.

Complex text fields that contain a lot of common words will cause the bloom filters to return "possibly in the set" often and thus triggering a full table scan. We implemented inverted indices for those complex textual fields. More specifically, for each partition of the target field in the dataset, we created an inverted index using lucene4s and Spark's mapPartitions primitive. So when the time comes for a full table scan, NLonSpark queries each partition of the target field and returns the results from the saved inverted index.

Figure 2 shows the architecture of NLonSpark. The major differences are:

(1) the NodeMapper functionality now runs on top of Spark.
(2) Spark now provides via an abstraction layer information about the database schema.
(3) Spark now provides the textual indices.

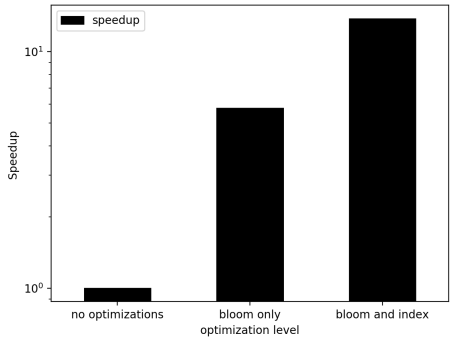
Table 1: How parallelism was achieved in the cluster

Number of instances	Number of cores per instance	Parallelism
9	14	126
4	14	56
2	14	28
1	14	14
1	7	7
1	1	1

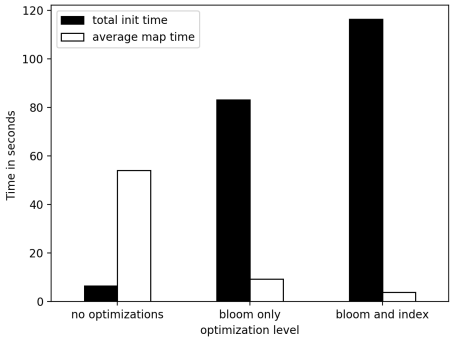
5 EVALUATION

For our evaluation, we used the YELP dataset. The dataset was converted to ORC from JSON prior to running the experiments. The conversion process was mostly easy since Spark inferred the datatype for each field automatically. What we had to do manually is to feed the foreign key constraints for the dataset to the SQL concept abstraction layer. The SQL abstraction layer uses Spark's type information for the Dataframe. Hence, we expect that for JSON datasets with relatively few foreign key constraints the process of moving the dataset to Spark will be straightforward. Furthermore, this process is not very different from when creating a new database in a relational database system, where one has to define the foreign key constraints. Hence, one can think that when creating a new dataset on Spark, this information also needs to be provided, albeit in a different way, since on Spark, this process is not provided as an inherent mechanism. Thus, we do not consider this as a stumbling block for supporting Natural Language-to-SQL translation and execution over the Spark distributed processing framework.

We used a cluster of virtual machines with the following specifications: 16 cores and 32 GB of RAM. On the YARN cluster we had a total of 14 virtual cores per instance and 9 instances total. Table 1 summarizes how each level of parallelism was achieved.



(a) Speedup provided by the various optimizations



(b) Effects of the various optimizations in the initialization and execution time of NLonSpark

Figure 3: Effects of optimizations on the execution time

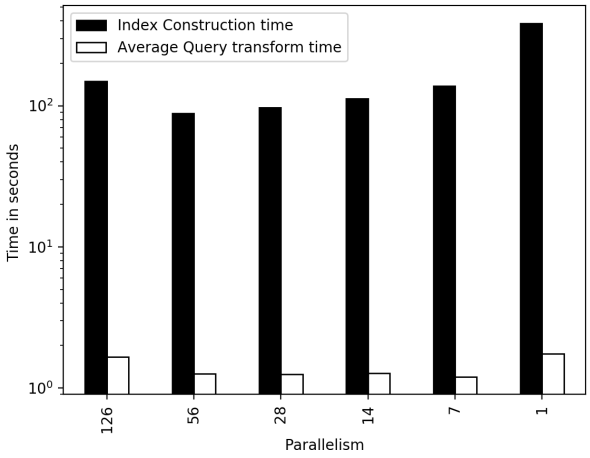


Figure 4: Scalability of NLonSpark for various degrees of parallelism

We evaluated NLonSpark both in terms of scalability and impact of optimizations. Since NLonSpark is based on NaLIR, and we did not change the underlying algorithms, NLonSpark has the same accuracy of NL translation as NaLIR. Hence, we do not report any accuracy numbers here.

As *scalability*, we mean the effect of added parallelism to the total execution time of the application. As *impact of optimizations*, we mean the effect of each optimization on the total execution time of the application. To measure the impact of each optimization in NLonSpark, we kept the degree of parallelism constant and progressively enabled each optimization.

For the scalability and execution time, we used a total of 42 queries. The queries were hand crafted by us and model a vast array of user interactions from simple queries such as "Find the number of users called Michelle" to more difficult ones such as "List the most crowded restaurant on Fridays". For the optimizations experiment, we used a smaller set of queries because without the optimizations NLonSpark execution takes a lot of time. This smaller subset of the previous queries is of modest difficulty where the system managed to find the right answer. Those queries included fields with both small and large textual values, so in some part of the translation the bloom filters failed and the importance of inverted indices becomes apparent. Examples include: "List all the businesses with more than 4.5 stars" and "Find Italian restaurants with music".

As we can see from Figure 3a, bloom filters and inverted indices significantly improve execution time. Using only bloom filters we achieve a speedup of 5.7x whereas using both bloom filters and inverted indices leads to a speedup of 13.7x compared to the un-optimized version.

In Figure 3b, we can see the effects of the various optimizations to the initialization and execution time of NLonSpark. Using only bloom filters, the initialization time goes from 6.4 seconds to 83 seconds while the average execution time for a single query drops from 52 seconds to 9.3 seconds. Adding inverted indices to the application increases the initialization time to 116 seconds and the average execution time for a single query to drop to 3.9 seconds. Both optimizations thus proved worthwhile even when the system is about to execute only a few queries. All the tests concerning the optimizations were conducted with the maximum level of parallelism to allow the un-optimized version to be as fast as possible.

Moving on to scalability, in Figure 4, we can see the effect of parallelism on the index creation and average execution time for a single query. For a single instance equipped with a single core the initialisation time is 389 seconds. For parallelism equal to 7, it drops to 138 seconds, for 14 to 112 seconds and it reaches its best value for parallelism 56 with 88 seconds. Then, for 128 cores it increases to 149 seconds. The average query execution time does not change much with added parallelism mainly because querying the bloom filters and inverted indices takes a negligible amount of time. The added parallelism did however improve the index creation time leading to a speedup of 4.4x.

In summary, a reasonable Natural Language-to-SQL translation and execution functionality can be supported over Spark. The overhead of porting a dataset on Spark (which entails the creation of

foreign key constraints) is manageable. On the other hand, NaLIR's need for low-latency string lookups can be addressed by using a mix of bloom filters and inverted indices, which reduces the execution time providing substantial speedup.

6 DISCUSSION AND FUTURE WORK

In our attempt to add natural language capabilities to Apache Spark we learned the following:

- The primitives available in Spark SQL were enough to create an SQL abstraction layer so that NaLIR can work with Spark instead of an RDBMS.
- The extensibility and flexibility of Apache Spark allowed us to create the necessary structures to speed up execution significantly.
- Spark built-in optimization primitives such as bloom filters significantly boost performance in some scenarios.

As future work, we plan to adapt NLonSpark to run on multi-application environments. The dataset is a Dataframe and the indexes are RDDs so there are projects such as Apache Livy [1], Apache Ignite [2] and Alluxio [3] that support sharing RDDs and Dataframes amongst applications. In this way, different users can start different instances of NLonSpark that will share the dataset and indexes. One of the main challenges is that Spark cluster utilization is sensitive to the resource manager. Resource managers decide how to allocate cluster resources among the many users and applications contending for them, and this affects application performance. In this case, we need new techniques for tuning Spark resource management to optimize goals like speed and fairness as well as for sharing both NL queries, their translations, and user feedback (apart from data and indexes) among multiple users and applications.

7 CONCLUSIONS

In this paper, we have introduced NLonSpark, a system for Natural Language to SQL translation and execution built on top of Apache Spark. Using NLonSpark, the end user can query Apache Spark datasets using Natural Language. As part of our implementation, we built an abstraction layer to alleviate NaLIR's dependency on a traditional DBMS. We have also implemented and evaluated a series of optimizations that significantly reduce the execution time on NLonSpark compared to the naïve implementation, providing a cumulative speedup of 13.7x.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 1383–1394.

[2] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. 2006. Generating typed dependency parses from phrase structure parses.. In *Lrec*, Vol. 6. 449–454.

[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[4] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 689–692.

[1] https://livy.apache.org
[2] https://ignite.apache.org
[3] https://www.alluxio.io

[5] Izzeddin Gur, Semih Yavuz, Yu Su, and Xifeng Yan. 2018. Dialsql: Dialogue based structured query generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1339–1349.

[6] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. 2007. BLINKS: ranked keyword searches on graphs. In *ACM SIGMOD* . ACM, 305–316.

[7] Ionel-Alexandru Hosu, Radu Cristian Alexandru Iacob, Florin Brad, Stefan Ruseti, and Traian Rebedea. 2018. Natural Language Interface for Databases Using a Dual-Encoder Model. In *Proceedings of the 27th International Conference on Computational Linguistics, COLING 2018, Santa Fe, New Mexico, USA, August 20-26, 2018*, Emily M. Bender, Leon Derczynski, and Pierre Isabelle (Eds.). Association for Computational Linguistics, 514–524.

[8] Vagelis Hristidis and Yannis Papakonstantinou. 2002. Discover: Keyword search in relational databases. In *VLDB*. 670–681.

[9] Fei Li and HV Jagadish. 2016. Understanding natural language queries over relational databases. *ACM SIGMOD Record* 45, 1 (2016), 6–13.

[10] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[11] Tianze Shi, Kedar Tatwawadi, Kaushik Chakrabarti, Yi Mao, Oleksandr Polozov, and Weizhu Chen. 2018. IncSQL: Training Incremental Text-to-SQL Parsers with Non-Deterministic Oracles. *CoRR* abs/1809.05054 (2018). http://arxiv.org/abs/1809.05054

[12] Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2008. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal* 17, 1 (2008), 117–149.

[13] Sandeep Tata and Guy M Lohman. 2008. SQAK: doing more with keywords. In *ACM SIGMOD*. ACM, 889–902.

[14] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.

[15] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436* (2017).

[16] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[17] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir R. Radev. 2018. TypeSQL: Knowledge-Based Type-Aware Neural Text-to-SQL Generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, Marilyn A. Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, 588–594.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[19] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM, 423–438.

[20] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[21] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017). arXiv:1709.00103 http://arxiv.org/abs/1709.00103