Teaching Data Models with TriQL

Abdussalam Alawini
Leyao Zhou
Lujia Kang
alawini@illinois.edu
leyaoz2@illinois.edu
lujiak2@illinois.edu
The University of Illinois at Urbana-Champaign
USA

Ping-Che Ho
pingrogerche1995.ho@gmail.com
Pure Storage
USA

Peilin Rao
peilinr@andrew.cmu.edu
Carnegie Mellon University
USA

Abstract

With the abundance of database systems implementing various data models, such as the relational, graph, and document-oriented models, learners often find it challenging to understand the trade-offs between different data models and to decide which database system they should learn and why. Additionally, most introductory database courses focus on the predominant relational model for teaching database design and programming, and do not discuss other emerging databases. While the relational database systems still play a vital role in modern data systems, especially with the emergence of NewSQL, it is crucial to introduce students to databases implementing other data models. In this paper, we introduce TriQL, a system for helping novices learn the schema and query languages of three major database systems, including MySQL (a relational database), Neo4J (a graph database), and MongoDB (a document-oriented database). TriQL offers learners a graphical user interface to design and execute a query against a generic database schema without requiring them to have any database programming experience. TriQL follows an interactive approach to learning new database models, supporting a dynamic and agile learning environment that can be easily integrated into database labs and homework assignments.

Keywords

database education, SQL, Neo4J, MongoDB

ACM Reference Format:

Abdussalam Alawini, Leyao Zhou, Lujia Kang, Ping-Che Ho, and Peilin Rao. 2022. Teaching Data Models with TriQL. In *1st International Workshop on Data Systems Education (DataEd'22), June 17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3531072.3535320

DataEd'22, June 17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9350-8/22/06…$15.00
https://doi.org/10.1145/3531072.3535320

1 Introduction

With the emergence of database systems implementing non-relational data models, such as graph, document-oriented, and key-value stores, beginner learners often find it challenging to decide what database model they should learn. Experienced developers also struggle to understand new database models as different models have different data structures and query languages. Thus, introductory database courses should introduce students to the abstract concept of a data model (i.e., the notation for describing data or information and generally consists of three parts 1) structure of the data, 2) operation on the data, and 3) constraints on the data) and provide them with the opportunity to compare and contrast the structure, operations and constraints of databases implementing different data models.

Several educational institutions realized the importance of introducing students to other data models and have adopted a curriculum that includes relational and NoSQL (Not only SQL) databases [Fowler et al. 2016; Guo et al. 2016; Li et al. 2016; Mohan 2018]. However, most database courses teach each data model separately using independent labs and homework assignments without providing students with insights into the tradeoffs between different data models [Alawini 2022; Davidson 2020; Kristin Tufte 2014]. Such limitation hinders students' ability to generalize their knowledge to learning new data models.

In this paper, we introduce TriQL (i.e., Tribus linguis query, Latin for three query languages), a system for helping novices and learners with limited database experience learn the structures (schema) and query languages of three major database systems, including MySQL (a relational, SQL-Structured Query Language, database), Neo4J (a graph database), and MongoDB (a document/collection-oriented database). Our system also helps learners explore structural and query language variations among different data models. Our system uses advanced database techniques, such as data integration and

logical programming, to capture the core data operations common between the relational, graph and document-oriented data models, including selection (filtering data), projection (redefining the output schema), grouping and aggregation.

TriQL offers learners a query builder interface for users to design and execute a query against a generalized database schema without requiring them to have any database programming experience. Our generalized schema can capture applications' business logic while embedding properties of different data models. TriQL converts the generated user-query into three database query languages: SQL, Cypher (Neo4J's query language), and MongoDB. The user will then examine each generated query and can view its results on the native database. For instance, TriQL produces an interactive Neo4J graph for Cypher queries, allowing users to interact with the resulting graph giving them the same experience working with the native Neo4J database engine. Enabling users to examine their queries on three database models help them quickly learn the three query languages and understand the tradeoffs among different database models.

The paper is organized as follows. Section 2 discusses related work and introduces MongoDB and Neo4J databases. The TriQL system architecture is detailed in section 3. Section 4 presents our future plans and we conclude the paper in section 5.

2 Background

In this section, we first discuss related work. Then, we introduce DataLog, an intermediate logical database language we use to capture the generic user query, and provide a quick tutorial on MongoDB, and Ne04J[1].

2.1 Related Work

While there is not much research on students' difficulties while learning different database models, there are multiple reports of instructors including other database models into their database courses [Fowler et al. 2016; Guo et al. 2016; Li et al. 2016; Mohan 2018]. Mohan reported experiences of a database education curriculum that incorporated NoSQL [Mohan 2018]. In Mohan's work, students were exposed to several NoSQL paradigms and had a set of projects, lab and research assignments to complete using the knowledge they gained during the course. However, their course did not provide labs for exploring the trade-offs between different database models. Other NoSQL databases have also been incorporated into university curricula. For example, Fowler et al. reported their experience in two database courses with teaching CouchDB, a NoSQL data management system that uses JavaScript as its query language [Fowler et al. 2016]. They mainly focused on measuring students' improvement of understanding NoSQL systems.

Researchers have proposed several tools for teaching databases. SQLator, proposed by Sadiq et al. [Sadiq et al. 2004], is an SQL learning tool that attempts to evaluate student queries. It compares SQL queries to plain English prompts to verify whether a student-written query is correct. SQLator uses a 'workbench' of tools, including a multimedia tutorial and a collection of practice databases. While this provides students with additional resources, it is focused on the relational database and SQL, and thus inadequate for teaching the tradeoffs of the relational, graph, and document-oriented databases. Other database learning tools include WebSQL[Allen 2000], an interactive system for executing SQL queries, and MDB[Hilles and Naser 2017], a tool for teaching MongoDB. These tools also focus on helping students learn a particular database system, and none of these tools combines relational and NoSQL. TriQL is different as it allows students to learn query languages, focusing on teaching students the trade-offs between the various data models and query languages.

TriQL uses DataLog to capture the core of relational, graph-based, and document-oriented query languages. Datalog is a powerful logical and declarative programming language [Minker 1997]. Due to its simplicity and query expressiveness, DataLog became the standard choice for intermediate query language generated based on the user's input [Alawini et al. 2018; Wu et al. 2018, 2019]. In Datalog, each formula is a function-free Horn clause, and every variable in the head of a clause must appear in the body of the clause. Such simple yet powerful formalization can maintain the query logic and express it in several database models, enabling our system to be flexible, easily plugable, and extendable with different database technologies.

We briefly demonstrate the syntax of Datalog queries through two examples. We use the following university database as a running example. This database has three relations (tables):

Course(course_id, Name, Instructor);
Has (course_id, student_id);
Student (student_id, FirstName, LastName, Age, Year, Major)

The following query (Q_1) will find the major of a student named "James Smith":

```
output(E) :- Student(A, B, C, D, E, F), B = "James", C = "Smith"
```

`Student(A, B, C, D, E, F)` is the definition of the `Student` relation with each variable corresponding to each field in the `student` relation. For example: `A` represents `student_id`, `B` represents `FirstName`, `C` represents `LastName`. This query has two conditions: `B = "James"` and `C = "Smith"` and projects (i.e., outputs) the result of `E`, which is the `Major` attribute.
A more complex query (Q_2) that finds, for each course, the number of students majoring in ECE:

```
output(B, V0) :- V0 = Course(_, B, _), COUNT(D):
{Course(A, B, C), Student(D, E, F, G, H, I), Has(J, K),
A = J, D = K, I = "ECE"}
```

This is an example of the aggregation operation. In order to find the number of ECE students in every class, we first select tuples with the ECE major. Then, we JOIN `Course` and `Has` on `course_id`, and `Student` and `Has` on `student_id`. Finally, we GROUP BY `course_name` in `Course` table. `V0` is the result of the COUNT operation, which counts the `student_id`. The output is `course_name` and `V0`.

2.2 Introduction to Neo4J and Cypher

Cypher is the query language used to query Neo4j databases. It has a similar syntax to SQL, with declarative pattern-matching

[1]SQL and the relational model are well known, so we do not introduce them here.

features added for querying graph relationships. As one of the most popular graph databases, Neo4j stands out among the current graph models for its performance, simplicity, and powerful query language [Fernandes and Bernardino 2018; Guia et al. 2017]. We briefly demonstrate the syntax of Neo4j structure and queries. Figure 1 shows the Neo4J database equivalent to the database from our running example. It has two types of nodes: `Course` and `Student`, and one relationship `HAS`. Notice that the `HAS` relationship was represented as a relation (table) in the relational database.

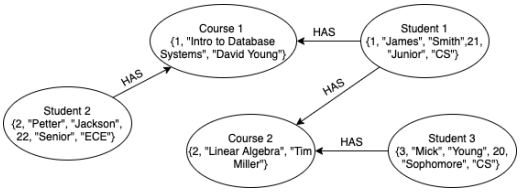


Figure 1: The Neo4J (graph) version of the university database presented in Section 2.1

Figure 2 shows the Cypher query for Q_1 and Q_2. The Cypher version of Q_1 (Figure 2(a)) is similar to Q_1's SQL version. We first find the `Student` nodes and use the `WHERE` clause to select students with the name 'James Smith'. Then, we `RETURN` (output) the `Major` property of the student. Figure 2(b) shows the Cypher query equivalent to Q_2 and demonstrates Neo4J's power in querying interconnected data. The graph pattern matching clause shown in the `MATCH` finds all `Students` measuring ECE and taking a class (represented by the relationship `Has`). The `RETURN` clause group the result by course name and COUNT `student_ids` per course name. Notice that Cypher does not use explicit GROUP BY cluase. If the `RETURN` clause contains any aggregate function (such as `COUNT()`), it will group by all listed attributes in the `RETURN`.

```
MATCH (s:Student)
WHERE s.FirstName = "James"
    AND s.LastName ="Smith"
RETURN s.Major
```

(a) Q_2: for each course, find the number of students majoring in ECE

```
MATCH (s:Student {Major:"ECE"})-[:HAS]->(c:Course)
RETURN c.name, COUNT(s.student_id) AS count_student
```

(b) The Cypher queries corresponding to the DataLog queries presented in Section 2.1

Figure 2: Examples of Cypher (Neo4J queries)

2.3 Introduction to MongoDB

MongoDB [MongoDB, Inc. 2019] is a document-oriented NoSQL database that stores JSON-like data in documents with flexible schema, removing the need for pre-defining the structure before inserting the data into documents. A MongoDB consists of a set of `Collections`, which consists of a set of `Documents`, which consists of a set of key-value pairs. We show examples of MongoDB documents and code snippets that exhibit the basic syntax of MongoDB. Note that the `_id` attribute is an indexed attribute that must be included in every object. Figure 3 shows a MongoDB database equivalent to our running example's database.

```
Course:
{
   "course_id": 1,
   "Name": "Intro to Database System",
   "Instructor": "David Young"
}
```

```
Student:
{
   "student_id": 2,
    "FirstName": "Peter",
    "LastName": "Jackson",
    "Age": 22,
    "Year": "Senior",
     "Major": "ECE"
}
```

```
Has:
{
    "course_id": 1,
    "student_id": 2
}
```

Figure 3: The MongoDB version of the university database presented in Section 2.1

MongoDB databases can be queried from many different programming languages, but the most straightforward interface is through MongoDB's JavaScript shell. The MongoDB's version of Q_1 is shown below.

```
db.Student.find(
    { FirstName: "James", LastName: "Smith" },
    { _id: 0, Major: 1})
```

The find operation takes two arguments (selection and projection) and returns documents in a collection or view, and returns a cursor to the selected documents. In the query above (Q_1), the second line represents the selection part of the query. The selection condition finds a document with `FirstName` 'James' and `LastName` 'Smith'. The third line (`{_id:0, Major:1}`) returns `Major` and hides (`_id:0`) the _id property.

MongodB offers a variety of complex data operations. For example, `$match` is an operator that takes conditions as arguments to filter and produce documents that have met set conditions. The `$unwind` is an aggregation operator, which takes a reference to an array and produces multiple objects from the array elements. Another aggregation operator `$group`, which takes an `_id` field as its first argument, continually takes fields combined with accumulator operators to perform basic operations, such as `$sum`, on the collection. Finally, `$project` is an operator that takes fields as its arguments, then adds, renames, excludes, or includes the specified fields in the resulting collection.

Below we list the MongoDB version of Q_2. This query uses the `aggregate` pipeline on the `Course` collection. The `$lookup` operation finds documents in the `Has` collection that match on the `course_id`. Then, the `$unwind` creates a document for each element in the newly constructed array `has`. The same two operations will be repeated again to connect `has` to `Student` based on the `student_id` property. Next, we use the `$MATCH` operator to select students majoring 'ECE'. Then, the `$group` operator groups documents by the course name; and finally the `$project` operator outputs the course name and the size of the student array.

```
db.Course.aggregate([
    {$lookup:
        {   from: "HAS",
            local_field: "course_id",
            foreign_field: "course_id",
            as:"has" }
    },
    {$unwind: "$has"},
    {$lookup:
        {   from: "Student",
            local_field: "has.student_id",
```

```
            foreign_field: "student_id",
            as:"student" }
    },
    {$unwind: "$student"},
    {$match:
        { Student.Major: "ECE" }
    },
    {$group:
        {   _id: {name: "name"},
            students: {$push: "$student.student_id"} }
    },
    {$project: {_id: 1, count_student: {$size: "$students"}}}
])
```

3 System Overview

Figure 4 shows the main components of TriQL. Once a user submits a generic query (represented as a JSON, JavaScript Object Notation, structure) via the Query Builder Interface (QBI), the Intermediate Query Generator (IQG) converts the user query into DataLog. Then, the Schema and Query Translator (SQT) 1) transforms the generic database (JSON) schema shown to the user into MySQL, Neo4J, and MongoDB, and 2) concurrently converts the DataLog query into SQL, Cypher, and MongoDB. Finally, TriQL executes each generated query on its corresponding database engine and shows native result to the user on the Query Result Interface. The user can then examine the three generated queries along with their outputs. They can also modify their query using TriQL's QBI and resubmit it again to see the effects of their changes. Such an interactive approach is beneficial for users to learn by examples in a dynamic and agile fashion.

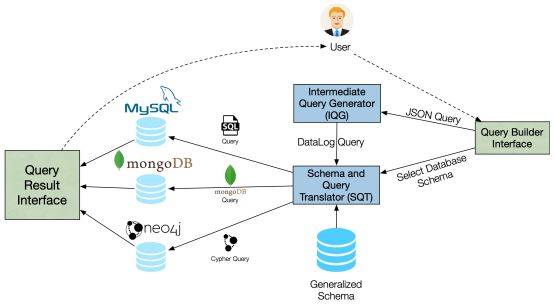


Figure 4: TriQL System Architecture: The Query Builder for building queries using a GUI; the Intermediate Query Generator converts user queries to DataLog; The Schema and Query Translator generates the schema of the three database and coverts the DataLog query into SQL, Cypher and MongoDB

3.1 Generalized Schema

The database schema describes data entities and their relationships based on the real-world application's underlying business logic. Database systems vary significantly in how they represent data. For example, the relational database has a fixed schema where users must define the data structure before uploading it into the database. In contrast, other databases, such as MongoDB and Neo4J, have a more flexible schema where users can upload the data without worrying about its structure. Such variations in database structures make it challenging for novices to learn database programming.

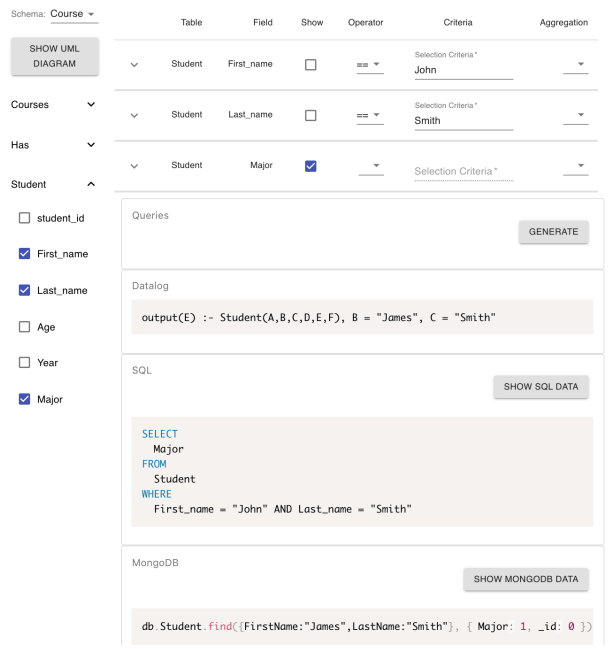
In this paper, we introduce our generalized JSON-based (JavaScript Object Notation) schema (GS), capable of capturing properties of relational (SQL) and NoSQL databases. We choose JSON to represent our generalized schema because of its expressive and straightforward structure. JSON is also widely supported by many programming languages.

The ability to generalize the structure of databases has several key learning outcomes: 1) learners can easily understand the data entities and how they are connected without having to learn complex data definition languages, 2) because we capture various database properties in GS, TriQL can easily transform GS into other database structures, and 3) our GS provides learners with the ability to examine the properties of relational, graph and document databases.

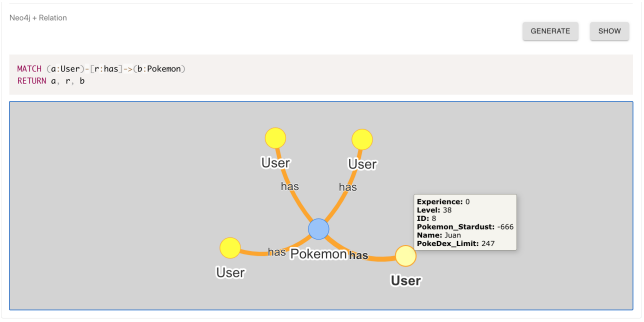
The general schema stores information of the underlying data models using two primary substructures: ENTITY and RELATION. Entities contain attributes that describe real-world objects; relationships capture connections between these entities. Each relationship captures the *relationship cardinality* (i.e., one-to-one, one-to-many, or many-to-many), and the *relationship direction* for directed relationships. Such a simple yet powerful representation of structure allows learners to quickly identify entities and their attributes and relationships between these entities. Additionally, capturing information such as relationship direction and cardinality allows our system to transform this schema into relational, graph, or document databases without human intervention.

3.2 Query Builder and Result Interface

TriQL's Query Builder Interface (QBI) allows users to construct queries using a user-friendly Graphical User Interface (GUI). The GUI allows users to select a database schema from a set of preloaded databases. Users can examine the conceptual design by clicking the "show UML diagram" button, which displays the design as a Unified Modeling Language (UML) diagram. Once the user is comfortable with the database schema, they can use the QBI to query the database. They can select entities and attributes (fields), define selection criteria over attributes, and decide whether to return an attribute with the output. The user must use the "Show" checkbox to choose the queries' attribute or aggregations output. They can also define grouping and aggregation functions. After clicking the "Generate" button, the QBI sends the user query as a JSON structure to TriQL's intermediate (DataLog) query generator. The generated Datalog query is then translated into SQL, MongoDB, and Neo4J queries, displayed in the Query Result Interface (see Figure 5). To see each translated query's output, users can click the "show data" buttons to see the query result in its native structure. Figure 5(b) shows an example of TriQL's Query Result Interface showing a Cypher query in its native Neo4J database. Users can interact with the graph and visualize the properties of the nodes and edges.



(a) TriQL's Query Builder and Query Result Interfaces. The QBI allows users to construct the query using a user-friendly GUI and the Query Result Interface shows the query result in its native database.



(b) An example of TriQL's Query Result Interface showing a Cypher query in its native Neo4J database. Users can interact with the graph and visualize the properties of the nodes and edges.

Figure 5: TriQL User Interface

3.3 Intermediate Query Generator

The Intermediate Query Generator (IQG) translates the JSON query received from the query builder into DataLog. The process starts with converting entities into Datalog relations. For instance, the user input from Figure 2 will output relation **tmp**(Major). Next, the IQG creates a mapping between entities' attributes and the Datalog variable. Using the relations and mappings, the IQG can now generate a Datalog query that matches the user query. Here is the Datalog query for Q_1:

```
output(E) :- Student(A, B, C, D, E, F), B = "James", C = "Smith"
```

The head of the DataLog query (`output(E)`) specifies the attribute(s) that the user would like to output. The query body contains the input relation (`Student`) and the conditions for filtering the tuples (records). The query, the generated intermediate relations, and dictionary are passed to the Schema and Query Translator (SQT), which converts the DataLog to the corresponding SQL, Cypher, and MongoDB queries.

3.4 Schema and Query Translator

The Schema and Query Translator (SQT) has two subsystems: Schema Generator and Query Translators. The schema generator converts the JSON-based Generalized Schema of the selected database and converts it into relational (MySQL), graph (Neo4J), and document (MongoDB) databases. The Query translators subsystem receives the DataLog and translates it into into the equivalent SQL, Cypher, and MongoDB queries.

The Query Translator uses predefined rules to convert a DataLog query into SQL, Cypher and MongoDB queries (See Figure 5). The generated queries can capture the primary data querying operations, including *selection*, *projection*, *grouping*, and *aggregation*. Once the three queries are generated, each query will be executed in its corresponding database and the data will be returned to the user in its native structure.

4 Ongoing and Future work

We are currently working on making the TriQL service publicly available online to collect and analyze users' interactions with the service. This data we plan to gather will help us understand how learners use TriQL to learn databases.

Most importantly, we plan to use TriQL in our database course (CS 411) to evaluate its learning effectiveness. We will have students use TriQL as part of lab and homework assignments. We will then conduct quantitative and qualitative analysis to measure the impact of TriQL in students' understanding of database schema and query languages. To that end, we are now integrating TriQL in two lab assignments. TriQL lab 1, which precedes the SQL, MongoDB, and Neo4J labs, introduces students to the generalized schema and the query builder interface. The first part of this experimental lab would help students explore the generalized schema of a real-world application. Examining a generic schema helps students understand the data entities and their connections without worrying about understanding any particular database's structure. The second part of this lab focuses on teaching students the principal data querying operations, including selection, projection, grouping, and aggregation. Using the TriQL QB interface, students can immediately query the database without any prior knowledge of any database programming language.

TriQL lab 2, which will succeed all SQL, MongoDB, and Neo4J labs, will include open-ended questions that encourage students to use TriQL to solve problems and reflect on the differences between the relational, graph, and document-oriented models and their query languages. We will design this lab to showcase the advantages and disadvantages of each data model. For example, students can work on a scenario in which data entities are highly connected. Cypher

(graph) and the graph model of Neo4J would be more effective in such a scenario compared to SQL or MongoDB.

Lab 2 will also help student learn about the structure of the three database. Because students can examine their query results on its native structure, they can quickly learn the tradeoffs among different database structures. Additionally, this lab will require students to reflect on the difference among the three data models. Students will use TriQL to view the generalized schema, the database conceptual design (viewed as a Unified Modeling Language chart), and the native structure of each of the three databases. For instance, students can view the Data Definition commands for the relational database.

We plan to improve TriQL's functionality by allowing users to submit queries in the native database query language (without using the GUI) and see the equivalent query in another database query languages. We are testing a data import function that allows users to upload a new database to TriQL by uploading their SQL DDL (.sql) files to TriQL. Additionally, we aim to develop an API to integrate TriQL with online assessment tools, such as PrairieLearn. Another future direction is to extend TriQL to support other modern data models, including Key-Value, Column-Family and Array databases.

5 Conclusions

We developed TriQL, a system for helping novices learn three major database systems, including relational (MySQL), graph (Neo4J), and document-oriented (MongoDB), and their query languages. Learners can explore the trade-offs among data models and the different querying paradigms, including the abstract declarative paradigm of SQL and Cypher and the imperative paradigms of MongoDB shell. We also discussed how TriQL can be integrated into an introductory database curriculum as part of the database programming lab assignments.

Acknowledgments

This work was partially funded by NSF (Award number: 2021499)

References

Abdussalam Alawini. 2022. *Database Systems.* https://alawini.web.illinois.edu/teaching/database-systems/

Abdussalam Alawini, Susan B Davidson, Gianmaria Silvello, Val Tannen, and Yinjun Wu. 2018. Data Citation: A New Provenance Challenge. (2018).

Grove N Allen. 2000. WebSQL: An Interactive Web Tool for Teaching Structured Query Language. *AMCIS 2000 Proceedings* (2000), 384.

Susan Davidson. 2020. *Data Management in the Cloud.* https://www.seas.upenn.edu/~cis550/

Diogo Fernandes and Jorge Bernardino. 2018. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB.. In *DATA*. 373–380.

Brad Fowler, Joy Godin, and Margaret Geddy. 2016. Teaching Case: Introduction to NoSQL in a Traditional Database Course. http://jise.org/Volume27/n2/JISEv27n2p99.html

José Guia, Valéria Gonçalves Soares, and Jorge Bernardino. 2017. Graph Databases: Neo4j Analysis.. In *ICEIS (1)*. 351–356.

Minzhe Guo, Kai Qian, and Li Yang. 2016. Hands-on labs for learning mobile and NoSQL database security. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. IEEE, 606–607.

Mohanad M Hilles and Samy S Abu Naser. 2017. Knowledge-based intelligent tutoring system for teaching mongo database. (2017).

David Maier Kristin Tufte. 2014. *Data Management in the Cloud.* http://datalab.cs.pdx.edu/education/clouddbms-win2014/page.php?content=index

Lei Li, Kai Qian, Qian Chen, Ragib Hasan, and Guifeng Shao. 2016. Developing Hands-on Labware for Emerging Database Security. In *Proceedings of the 17th Annual Conference on Information Technology Education* (Boston, Massachusetts, USA) *(SIGITE '16)*. Association for Computing Machinery, New York, NY, USA, 60–64. https://doi.org/10.1145/2978192.2978225

J. Minker. 1997. Logic and Databases: Past, Present, and Future. *AI Mag.* 18 (1997), 21–47.

Sriram Mohan. 2018. Teaching NoSQL Databases to Undergraduate Students: A Novel Approach. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 314–319. https://doi.org/10.1145/3159450.3159554

MongoDB, Inc. 2019. MongoDB. https://www.mongodb.com/

Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Joe Lin. 2004. SQLator: An Online SQL Learning Workbench. *SIGCSE Bull.* 36, 3 (June 2004), 223–227. https://doi.org/10.1145/1026487.1008055

Yinjun Wu, Abdussalam Alawini, Susan B. Davidson, and Gianmaria Silvello. 2018. Data Citation: Giving Credit Where Credit is Due. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 99–114. https://doi.org/10.1145/3183713.3196910

Yinjun Wu, Abdussalam Alawini, Daniel Deutch, Tova Milo, and Susan Davidson. 2019. ProvCite: Provenance-Based Data Citation. *Proc. VLDB Endow.* 12, 7 (March 2019), 738–751. https://doi.org/10.14778/3317315.3317317