Static Analysis of Database Accesses in MongoDB Applications

Boris Cherry*, Pol Benats*, Maxime Gobert*, Loup Meurice*, Csaba Nagy†, Anthony Cleve*
*Namur Digital Institute, University of Namur, Belgium
{boris.cherry|pol.benats|maxime.gobert|loup.meurice|anthony.cleve}@unamur.be
†Software Institute, Università della Svizzera italiana, Switzerland
csaba.nagy@usi.ch

***Abstract*—The increasing data volume and the variety of data formats of modern data-intensive systems unveiled the boundaries of traditional relational database management systems. NoSQL technologies aim to fulfill shortcomings through numerous features such as allowing unstructured, schema-less data storage. However, new features also pose challenges to software engineering techniques that used to work well for relational databases.**

In this paper, we present an approach to retrieve database accesses in JavaScript applications that use MongoDB. The approach handles JavaScript's highly dynamic and typeless nature through heuristics to avoid collision with third-party libraries. The aim is to identify the part of the source code involved in the database communication as the first step towards additional static analysis approaches. We evaluated the approach on an oracle of 307 open-source projects and reached a precision of 78%. We demonstrate potential use cases of the approach through case studies on the evolution of open-source systems.

***Index Terms*—Database Accesses, Static Analysis, NoSql, MongoDB, JavaScript, NodeJS**

I. Introduction

NoSQL ("Not Only SQL") systems emerged to tackle the limitations of relational databases. They offer attractive features such as scalability with scale out, cloud readiness, and schema-less data models [1]. New features come at a price, however. For example, schema-less storage allows faster data structure changes, but the absence of explicit schema can result in *multiple* implicit schemas co-existing in the same system. The increased complexity makes developers' operational and maintenance burdens heavier [2], [3].

Several efforts have been made to address the challenges of NoSQL systems. A popular purpose is to support schema evolution in the schema-less NoSQL environment [4]. For example, researchers study automatic schema extraction [5], schema generation [6], optimization [7], and schema suggestions [8]. Behind the scenes, such approaches mainly rely on a static analysis of the source code or the data when it is available. For the source code, they operate on the part of it that implements the database communication.

This paper addresses the problem of retrieving database accesses from the source code of JavaScript applications that use MongoDB. This is the first critical step for further static analysis techniques.

We target MongoDB, the most popular NoSQL technology on DB-Engines Ranking[1] and JavaScript, the programming language where MongoDB is used the most frequently [9]. According to a recent empirical study [9], about half (52%) of the database-dependent JavaScript projects use a document store (*i.e.,* MongoDB), and only about a third of them (35%) rely on a relational database.

Static analysis of JavaScript is known to be extremely difficult. Existing techniques [10], [11] usually struggle to handle the excessively dynamic features of the language [12], and approaches with type inference [13], data flow [14], or call graphs [15] need to balance between scalability and soundness.

We need a sound approach that scales with potentially large system code bases. MongoDB queries in JavaScript are typically constructed through the APIs of database access libraries. The most popular ones[2] are the native MongoDB Node Driver[3] and Mongoose ODM.[4] Therefore, we look for the usage of these APIs in JavaScript projects. We use CodeQL,[5] a powerful semantic code analysis engine that provides the syntax tree of the analyzed project and performs a sound dataflow analysis. We define heuristics to improve the precision of identifying APIs of database access libraries.

We evaluated the accuracy of our approach on an oracle of 307 open-source projects and reached promising results achieving a precision of 78%. Our approach is the first step towards additional analyses of database access API usage in JavaScript applications. It is required, for example, to analyze the evolution of such systems [16], help their developers propagate schema changes [17], or identify antipatterns [18]. We demonstrate potential use cases on two interesting case studies where we assess the evolution of open-source systems.

Paper Organization. Sections II and III present the background and the details of the approach. Section IV describes its evaluation on open-source projects. Section V demonstrates the approach through two case studies. Section VI presents the related work, then we conclude and discuss potential future directions in Section VII.

[1] https://db-engines.com/en/ranking
[2] https://www.npmjs.com/search?q=mongodb&ranking=popularity
[3] https://docs.mongodb.com/drivers/node/current/
[4] https://mongoosejs.com/docs/
[5] https://codeql.github.com/

This is the author's version of the work. It is posted for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1–5, 2022, doi: 10.1109/SANER2022.2022.00111.

II. Background

MongoDB is a document-oriented datastore. Its central concept is a *document* that stores all information for a given object. Every document can be different, unlike in a relational database, where tables have to conform to a specific schema. Documents can also embed and reference other documents, similar to a *join* in a relational database. A group of documents is called a *collection*. MongoDB stores data in BSON (Binary JSON), optimized for speed, space, and flexibility.

The most popular npm libraries to work with MongoDB are Mongoose ODM and MongoDB Node Driver.[2] MongoDB Node Driver is offered by MongoDB for Node.js applications as the native driver. Its API provides the basic operations to query the database. Mongoose is an ODM (Object Document Mapping) layer on the MongoDB Node Driver.

```
1  const mongoose = require("mongoose");
2
3  let SmartphoneSchema = new mongoose.Schema({
4      name: String,
5      price: Number,
6      inStock: Boolean
7  });
8
9  const Smartphone = mongoose.model("smartphones",
       SmartphoneSchema);
10
11 module.exports = Smartphone;
```

Listing 1. Mongoose schema definition example

```
1  Smartphone = require("./smartphones.js");
2
3  // ...
4  iPhone = new Smartphone("iPhone 13 Pro", 999, true);
5  await iPhone.save();
6
7  // ...
8  iPhones = await Smartphone.find({name: /iPhone/});
```

Listing 2. Mongoose query example

Listing 1 presents a typical schema definition in Mongoose. First, the `mongoose` module is included using the built-in `require` function. Then a schema is created through the `mongoose.Schema(...)` API call. Everything in Mongoose starts with a `Schema` that is mapped to a MongoDB collection and defines the structure of the documents within that collection. A `Model` is needed to work with a `Schema` in Mongoose. Line 9 creates a `Model` in Listing 1. Finally, the model gets exported to be used externally (line 11).

Listing 2 shows an example usage of the model exported in Listing 1. The model is imported using the `require` function. An instance of a model is a `Document` in Mongoose. It can be created and saved in various ways. An example can be seen on line 5, using the `Document.save()` method of the iPhone Document instance. Finally, line 8 shows a simple query to find documents.

Both Mongoose and MongoDB Node Driver have well-defined APIs to operate with the database. Our goal is to identify the part of the source code where they use these APIs to create, query, modify or delete data. In general, the aim is to identify every statement that operates with the database. This enables us to further analyze the database access code.

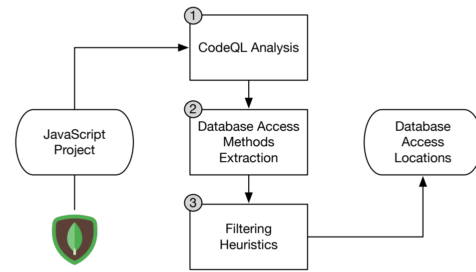

Fig. 1. Approach overview

III. Approach

Fig. 1 presents an overview of the main steps of the approach. First, we analyze a JavaScript project with CodeQL, and then we run our queries to extract the database access methods. In a second step, we apply filtering heuristics to improve the precision by eliminating method calls in potential conflict with other APIs. The outcome is a list of source code locations accessing the database with details of the access (*e.g.,* API used, receiver, context).

1) CodeQL Analysis: As mentioned in the introduction, state-of-the-art tools exist to perform static analysis for JavaScript. We choose *CodeQL*[6] mainly for the following reasons: (1) it can parse JavaScript projects conforming to recent ECMAScript language specifications (which is not the case for many other tools); (2) it provides an abstract syntax tree (AST) that can be queried in an SQL-like query language; (3) it has scalable data flow analysis; (4) it supports multiple languages, making our approach reusable, *e.g.,* to TypeScript; (5) finally, it is freely available for research and open-source.

CodeQL takes as input the source files of the project. It parses them and builds an internal database that can be queried in QL, an SQL-like, declarative, object-oriented query language.[7] A CodeQL analysis can be customized by defining new predicates and classes to be used in the queries. Therefore, the next step is to implement queries and predicates to extract database access methods in JavaScript applications. We made our source code available in an *online appendix*.[8]

2) Database Access Methods Extraction: We look for method calls invoking a database access method of the MongoDB Node Driver or Mongoose. We gathered method signatures from reference guides of MongoDB Node Driver 3.6 and Mongoose 5.12.8. From this list, we selected the methods that access the database for one of the following operations: (1) creates a new collection/document; (2) updates the content of documents or a collection; (3) deletes documents from a collection; (4) accesses the content of documents.

Overall, we collected 179 methods, 74 from MongoDB Node Driver and 105 from Mongoose. The complete list of the methods is also available in our *online appendix*.[8]

[6] https://codeql.github.com/

[7] https://codeql.github.com/docs/ql-language-reference/about-the-ql-language/

[8] https://github.com/bocherry/saner22-online-appendix

CodeQL provides type inference to approximate types for JavaScript expressions and variables. It also has a call graph that represents the caller-callee relationship between functions. Yet, due to the highly dynamic nature of JavaScript, these are often incomplete; thus, we cannot rely on them to identify database access API usage. We implement a naive approach that looks for method calls with identifiers matching the method names in our API list. However, only matching method names would produce noise and mistakenly detect methods from other APIs. To minimize this noise, we apply filtering heuristics in the following step.

3) Filtering Heuristics: The names of some API methods would likely collide with other methods defined internally or in external libraries. For example, the Mongoose `Model.create(...)`[9] API saves one or more documents to the database. A naive approach can mix this with the `create(...)` method of JavaScript's `Object`, *i.e.,* `Object.create(...)`. We call these *collisions* in the rest of the paper. Similar collisions can generate significant noise in our approach. Thus, we apply heuristics to avoid such cases and looked for potentially colliding methods in the MongoDB and Mongoose API documentation.

Many of the heuristics check the immediate type or name of the receiver object of the method call. For example, the receiver object of `iPhone.save()` in Listing 2 is `iPhone`. In the remaining of the paper, we refer to the receiver object as the *receiver*.

Some heuristics also check the parameters of the method calls, *i.e.,* the number of parameters and their types should match when they are available. For example, we collect all the method calls for the `Model.findOne(...)` method of Mongoose with at least one Object parameter and a callback function. This function has two mandatory parameters and two more optional parameters according to the documentation.[10]

Below, we further describe each of our heuristics.

H1: Find call's single parameter should not be a string literal. JavaScript has a standard `Array.find(...)` method. Hence, we define this heuristic to avoid collision with native JavaScript *find* and other well-spread libraries.

H2: The receiver should not be a `Promise` object. This heuristic is applied to all methods to avoid collision with the `Promise` API. Promise is frequently used for asynchronous operations in JavaScript.

H3: A `create(...)` invocation's receiver should not be an `Object` or a subclass of Object that is not a Document. This heuristic avoids collision with native JavaScript Object methods, *e.g.,* `Object.create(...)`.

H4: The receiver should not be "_". Lodash[11] is a JavaScript library, which provides utility functions to work with arrays, numbers, objects, strings, etc. Its standard notation uses the dash ("_") as an identifier. Thus, this heuristic avoids potential collisions with the frequently used Lodash.

[9]https://mongoosejs.com/docs/api/model.html#model_Model.create
[10]https://mongoosejs.com/docs/api.html#model_Model.findOne
[11]https://lodash.com/

H5: The receiver should not be `JQuery.events`, `JQuery` or match the regular expression "`\$\(.*\)`". This heuristic avoids method collisions with the JQuery library.[12]

H6: The file should transitively import MongoDB or Mongoose. Candidate method call should be in a file belonging to an import chain originating from a file importing MongoDB or Mongoose. A file belongs to an import chain if the file (1) contains a MongoDB or Mongoose import, or (2) imports such a file containing a MongoDB or Mongoose import, or (3) transitively imports such a file.

H7: Mandatory parameters should match. Each method invocation should have the same number of mandatory parameters as the method signature in the documentation. The minimum and the maximum number of mandatory arguments are considered when more methods share the same name.

IV. Evaluation

We assess the precision of the approach on an oracle of open-source JavaScript projects. We relied on the dataset of Benats *et al.* [9] and queried JavaScript projects using MongoDB. We queried the dataset for repositories with at least 100 stars and found a total number of 502 projects. We cloned these projects and analyzed them. We identified database accesses with the approach in 307 projects. An overview of the projects can be seen in Table I with descriptive statistics (min, first quartile (Q1), median, third quartile (Q3), and max) of the GitHub stars, files, lines of code, and DB accesses.

TABLE I
Statistics of the projects

	Min	Q1	Median	Q3	Max
Stars	100	230	413	1191	335,035
Files (JavaScript)	2	25.5	61	163	3,880
LOC (JavaScript)	58	1,570	5,034	25,938.5	536,380
DB Accesses	1	6	19	48	2,213

We collected 19,093 database accesses in the oracle. Interestingly, the projects with at least one database access method had a total number of 2,166,866 method calls. Meaning that, on average, about 1 out of 100 method calls were related to database accesses in the projects.

We took a random sample of 818 database access calls representing a confidence level of 90% with a 3% margin of error. Each call was randomly assigned to two authors who checked them manually. The aim was to see if the call indeed belonged to the detected method signature of the identified database access. Suppose, for example, that the approach detected the `Smartphone.find({name: /iPhone/})` call in Listing 2 as Mongoose's `Model.find(Object filter)` method. The two authors manually checked whether `Smartphone` was indeed a `Model` object and if it had overwritten the `find` method.

We did two tagging rounds. Before labeling the 818 methods, we had a trial round on a sample set of 150 methods.

[12]https://jquery.com

The goal of the trial was to avoid potential misunderstandings in the process. As a result, we also improved the heuristics.

When reviewing a method call, the two authors tagged whether it was a false or true positive and added remarks where needed. It was also possible to assign an 'Unclear' tag. This tag aimed to indicate cases when the code was obfuscated or when it was impossible to determine the final method call, for example, due to a POST request. The two taggers worked independently, *i.e.,* they were not aware of the tags of each other. After this tagging, we checked for conflicts, and a third author rechecked the specific cases and discussed them with the two taggers when needed.

Many cases involved deep structures where it was challenging to check the code context manually. 15% (120) of the method calls had conflicts and needed a third reviewer.

At the end of this process, we tagged 179 cases as *false positives*, 619 as *true positives*, and 20 as *unclear*. Excluding the unclear cases, the approach achieved a precision of 78%. The dataset of the oracle is available in the *online appendix*.[8]

V. Case Studies

We demonstrate the approach on case studies of two open-source systems. We ranked the projects in our benchmark according to database accesses and looked at the top 15 projects. Remind that all the projects had at least 100 stars on GitHub. Thus, these projects are not "toy projects" but represent popular JavaScript projects. Here, we analyze two of the top 15 projects to illustrate the application of our approach.

We organize the database access methods into categories: select, update, delete, insert, create, and generic. We use the generic category for methods that do not fall in the categories above or perform multiple operations.

A. Bitcore

Bitcore[13] is an "*infrastructure to build Bitcoin and blockchain-based applications for the next generation of financial technology*." The project has 4.2K stars and 2K forks on GitHub. We selected this repository as an interesting multi-project infrastructure with a MongoDB database in its core. Its GitHub repository is composed of six applications and nine libraries. The developers use tags regularly. Hence we analyze the tagged releases to get a glance at the evolution of Bitcore.

Fig. 2 shows the database access methods in the different releases. The absence of releases before v8.1.0 is due to a massive change in the evolution of the project. Before that version, the repository held a standalone application and the other applications were in separate repositories, which they later joined into multiple sub-projects of this repository.

The most represented database operation is *select* with 170 distinct method calls. One can also see a major change in the number of database accesses around v8.16.2. Taking a closer look at it reveals that a commit[14] adds numerous models and methods interacting with it. It is a new feature, Bitcore "*can now sync ETH and get wallet history for ERC20 tokens*."

[13]https://github.com/bitpay/bitcore

[14]https://github.com/bitpay/bitcore/commit/d08ea9

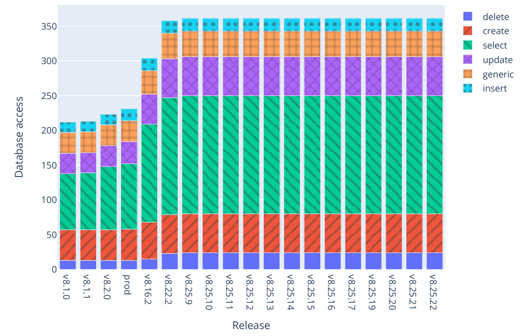

Fig. 2. Evolution of database accesses in Bitcore

We also analyzed the three applications which were split in the repository, namely, bitcore-node, bitcore-client, and wallet-service. The most significant part of the database operations is performed in the bitcore-node application.

B. Overleaf

Overleaf[15] is a well-known "*online real-time collaborative LaTeX editor.*" We selected this project as an interesting candidate to represent a front-end Web application. They do not use tags or releases, hence, we take one commit per month to analyze its evolution.

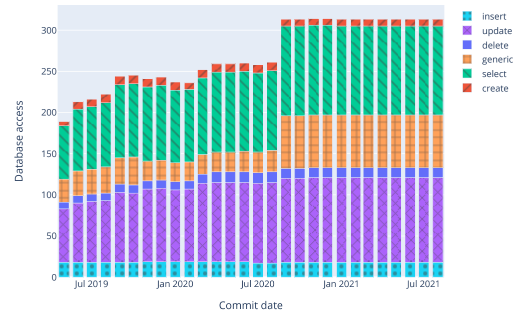

Fig. 3. Evolution of database accesses in Overleaf

Fig. 3 shows the results of our analysis. The repository was created in 2007. Still, we can see that the tool did not detect database access before May 2019. Our manual investigation revealed that the project's back-end used CoffeeScript in this early period.[16] CodeQL does not support this language. Thus its analysis is missing from the history.

This project uses much more *update* queries than Bitcore. Indeed, the proportion of updates 34% (108) is around the same as selects 32% (103). There is also an abrupt change in database accesses between September and October 2020. Interestingly, Overleaf was actually migrated from MongoJS to MongoDB Node Driver.

Overall, the approach worked well to get a picture of the evolution of the two systems, and we could also spot interesting major events in their histories.

[15]https://github.com/overleaf/web (Notice that since our initial dataset, the project has been migrated to overleaf/overleaf.)

[16]https://github.com/overleaf/web/commit/82f7e4

VI. Related Work

Many researchers tackled the challenges of analyzing JavaScript through static approaches [12]–[15]. Sun *et al.* recently presented a summary of such approaches, their challenges, and recent trends [12]. Interesting to us are the JSAI [10] and TAJS [11] analysis frameworks. Both are state-of-the-art tools to analyze JavaScript code. However, CodeQL had more benefits for our needs. Also interesting to note recent machine learning models to predict types in JavaScript, *i.e.,* NL2Type [19].

There are a few studies in the realm of reverse engineering of MongoDB applications. In particular, they extract models from the JSON document database [5], [20]–[22]. Some approaches also deal with schema generation [6], optimization [7], and schema suggestions [8]. Also interesting to us is the work of Störl *et al.*, who studied schema evolution and data migration in a NoSQL environment [4]. In this context, the closest work to us was done by Meurice *et al.*, who implemented an approach to extract the database schema of MongoDB applications written in Java. They also applied their method to analyze the evolution of Java systems.

The approaches above rely on the data instead of the source code or do not support JavaScript. To the best of our knowledge, there is no other approach to identify MongoDB database accesses in JavaScript applications despite their frequent usage and popularity in this development environment. We analyze JavaScript through a robust analysis framework, CodeQL, and tackle the challenges of the dynamic language through multiple heuristics.

VII. Conclusion

Extracting database accesses from source code is the first inevitable step of different approaches that target applications working with databases, *e.g.,* when supporting code comprehension, schema inference, testing, or data migration. Current approaches for NoSQL systems mostly take the database as input. However, the source code is often the only available (and reliable) documentation about the system's data structures. We target exactly this source of information in JavaScript applications where document stores are used frequently. In this context, the approach is particularly useful in mining software repositories, where these technologies, namely JavaScript and MongoDB, are prevalent today.

JavaScript is a highly dynamic language and extremely difficult to handle with static analysis. We also have a few limitations to deal with. Currently, we support MongoDB Node Driver and Mongoose. If the application uses another library or a web service for the database access, we will miss it. Our heuristics might also be strict in some cases. For example, JavaScript does not require checking parameter types, or one can define a method without importing a MongoDB library, thus being removed by our import chain heuristic.

Our approach opens the possibility for many potential future research directions. In particular, we plan to extend the analysis to the context of the database access. For example, we would track the parameters' value and obtain information about the data structure. The approach could be used to infer the database schema, visualize database accesses, or analyze the evolution of schemas co-existing in NoSQL applications.

Acknowledgments. This research was supported by the Fonds de la Recherche Scientifique (F.R.S.-FNRS) and the Swiss National Science Foundation (SNF), under the PDR project INSTINCT (35270712).

References

[1] McKnight, "NoSQL Evaluator's Guide," 2014.

[2] S. Scherzinger, E. C. De Almeida, F. Ickert, and M. D. Del Fabro, "On the necessity of model checking NoSQL database schemas when building SaaS applications," in *Int. Workshop on Testing the Cloud (TTC 2013).* ACM, 2013.

[3] K. W. Alger and D. Coupal. Building with patterns: The polymorphic pattern. [Online]. Available: https://www.mongodb.com/developer/how-to/polymorphic-pattern/

[4] U. Störl, M. Klettke, and S. Scherzinger, "NoSQL schema evolution and data migration: State-of-the-art and opportunities," in *23rd Int. Conf. Extending Database Technology, (EDBT 2020)*, 2020, pp. 655–658.

[5] F. Abdelhedi, A. Brahim, H. Rajhi, R. Ferhat, and G. Zurfluh, "Automatic extraction of a document-oriented NoSQL schema," in *23rd Int. Conf. Enterprise Information Systems*, 2021.

[6] P. Gómez, R. Casallas, and C. Roncancio, "Automatic schema generation for document-oriented systems," in *Database and Expert Systems Applications*. Springer, 2020, pp. 152–163.

[7] M. J. Mior, "Automated schema design for NoSQL databases," in *2014 SIGMOD PhD Symposium*. ACM, 2014, p. 41–45.

[8] A. A. Imam, S. Basri, R. Ahmad, J. Watada, and M. T. González-Aparicio, "Automatic schema suggestion model for NoSQL document-stores databases," *Journal of Big Data*, vol. 5, 2018.

[9] P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "An empirical study of (multi-) database models in open-source projects," in *40th Int. Conf. Conceptual Modeling (ER 2021)*. Springer, 2021, pp. 87–101.

[10] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for JavaScript," *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-Nove, pp. 121–132, 2014.

[11] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," *Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pp. 17–31, 2014.

[12] K. Sun and S. Ryu, "Analysis of JavaScript programs: Challenges and research trends," *ACM Comput. Surv.*, vol. 50, no. 4, aug 2017.

[13] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Static Analysis*. Springer, 2009, pp. 238–255.

[14] M. Madsen and A. Møller, "Sparse dataflow analysis with pointers and reachability," in *Static Analysis*. Springer, 2014, pp. 201–218.

[15] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *2013 Int. Conf. Software Engineering*. IEEE, 2013, p. 752–761.

[16] L. Meurice and A. Cleve, "Supporting schema evolution in schema-less NoSQL data stores," in *24th Int. Conf. Software Analysis, Evolution and Reengineering (SANER 2017)*. IEEE, 2017, pp. 457–461.

[17] A. Afonso, A. da Silva, T. Conte, P. Martins, J. Cavalcanti, and A. Garcia, "LESSQL: dealing with database schema changes in continuous deployment," in *2020 IEEE 27th Int. Conf. Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 138–148.

[18] C. Nagy and A. Cleve, "SQLInspect: A static analyzer to inspect database usage in Java applications," in *40th Int. Conf. Software Engineering: Companion*, ser. ICSE '18. ACM, 2018, p. 93–96.

[19] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring JavaScript function types from natural language information," in *Int. Conf. Software Engineering*. IEEE, 2019, pp. 304–315.

[20] A. A. Brahim, R. T. Ferhat, and G. Zurfluh, "Model driven extraction of NoSQL databases schema: Case of MongodB," in *11th Int. Joint Conf. on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, 2019, pp. 145–154.

[21] E. Gallinucci, M. Golfarelli, and S. Rizzi, "Schema profiling of document-oriented databases," *Inf. Systems*, vol. 75, pp. 13–25, 2018.

[22] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, "Parametric schema inference for massive JSON datasets," *The VLDB Journal*, vol. 28, no. 4, pp. 497–521, 2019.