SMEAGOL: A Static Code Smell Detector for MongoDB

Boris Cherry
Namur Digital Institute
University of Namur, Belgium
boris.cherry@unamur.be

Csaba Nagy
Software Institute
USI, Lugano, Switzerland
csaba.nagy@usi.ch

Michele Lanza
Software Institute
USI, Lugano, Switzerland
michele.lanza@usi.ch

Anthony Cleve
Namur Digital Institute
University of Namur, Belgium
anthony.cleve@unamur.be

***Abstract*—MongoDB is one of the most popular NoSQL database engines. To foster scalability, it provides multiple features such as schema-less data storage or sharding. However, those new features introduce additional considerations for the maintainer to be careful, which might lead to erroneous implementation choices often referred to as code smells or antipatterns. Detecting and fixing those code smells can play a crucial role for developers in their maintenance efforts.**

We present SMEAGOL (SMEll and Antipattern detection for monGOdb appLications), a static analysis tool to detect MongoDB code smells in JavaScript applications. SMEAGOL relies on CodeQL and detects code smells by analyzing and extracting all the necessary information (*e.g.,* data structure) from the database access code of the application. We demonstrate it by examining the evolution of MongoDB code smells in five popular open-source projects, showing promising results.

***Video link*: https://youtu.be/h4Xbp9dIFt0**

***Repository link*: https://github.com/bocherry/SMEAGOL_tool**

***Index Terms*—Code smells, Static analysis, MongoDB, JavaScript**

I. Introduction

NoSQL ("Not Only SQL") datastores have gained popularity as data persistence solutions for database applications, with MongoDB being the most popular one, according to the DB-engines ranking.[1] MongoDB stores JSON-like documents in collections and offers appealing features such as built-in aggregation pipelines, sharding for data distribution, or schema-less modeling for increased flexibility. Nevertheless, these features come with considerations that require careful attention when maintaining MongoDB code, to avoid erroneous design or unoptimized queries [1].

Researchers studied various approaches to assist MongoDB developers, such as supporting normalization [2], query optimization [1], or security issues [3], [4] like the critical NoSQL injection [5], [6]. Many problems stem from maintainability or quality issues, often originating from poor design or implementation choices. Code smells, or antipatterns, are well-known indicators of such problems [7], [8], and they are also well-studied in the context of database applications. Previous work considered data smells [9], DB schema smells [10], ORM (Object-Relational Mapping) smells [11], and SQL antipatterns or smells [12]–[15].

MongoDB has published a blog series on Schema Design Anti-Patterns [16]. There are also books on "best practices" with suggestions on code constructs to be avoided [17]–[19].

However, practitioners lack tool support to detect such issues, and to the best of our knowledge, there is no tool detecting MongoDB smells in the applications' source code.

We aim to fill this gap with SMEAGOL (SMEll and Antipattern detection for monGOdb appLications), a rule-based static analysis tool to detect code smells in MongoDB. SMEAGOL supports JavaScript, the most popular programming language used with MongoDB [20]. It is built on CodeQL,[2] a multi-language static analyzer with sound dataflow analysis in JavaScript. We support the two most popular libraries using MongoDB on npm[3]: MongoDB NodeJS native driver[4] and Mongoose,[5] a MongoDB-ODM (Object Document Mapper).

For the code smell definition, we rely on Mahajan *et al.*'s work [1], MongoDB's official blog series [16] and books on MongoDB best practices [17]–[19]. We implemented 8 smells that can be detected from the source code of JavaScript applications. For each, we designed detection rules and implemented them as CodeQL queries. We also extracted data structure information from the queries to refine our detection.

We demonstrate SMEAGOL by analyzing five popular open-source projects in which we found 1,623 smell instances.

II. Example

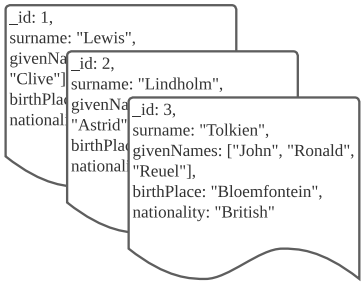

Fig. 1. Author collection example

[1]https://db-engines.com/en/ranking

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

[3]https://www.npmjs.com/search?ranking=popularity&q=Mongodb

[4]https://www.npmjs.com/package/mongodb

[5]https://www.npmjs.com/package/mongoose

```
1  const { MongoClient } = require("mongodb");
2  const uri = "mongodb://localhost:27017";
3  const client = new MongoClient(uri);
4  async function run() {
5    try {
6      await client.connect();
7      const db = client.db("main");
8      let britishAuthor = await db
9        .collection("writers")
10       .findOne({ nationality: "British" });
11     console.log(britishAuthor.surname);
12   } catch (err) {
13     console.log(err);
14   } finally {
15     await client.close();
16   }
17 }
18 run().catch(console.dir);
```

Listing 1. Database access code example with antipattern

```
7  /* same as line 1-7 */
8    let britishAuthor = await db
9      .collection("writers")
10     .findOne({ nationality: "British" }{ surname: 1 });
11   console.log(britishAuthor.surname);
12 /* same as line 11-18 */
```

Listing 2. Database access code example fixed

MongoDB stores BSON (Binary JSON) *documents* in *collections*. A *document* has field-value pairs holding relevant information for a given object.

Fig. 1 shows an example collection of documents representing fantasy authors, each with 4 attributes: surname, givenNames, birthplace and nationality. Listing 1 shows JavaScript code with a query to find a document in this collection.

First, the program imports and initializes a `MongoClient` from the MongoDB Native driver (Line 1-3). Then, it connects to the database (Line 7), performs a query to retrieve a single document based on its nationality attribute (Line 8-10) in order to print its surname property (Line 11), and closes the database connection (Line 15).

There is an antipattern in this code snippet: The application code queries more data than it uses. As suggested by the *Practical MongoDB* book [17], this unnecessarily clutters the retrieved document. We can use MongoDB projections[6] to fix it and specify the fields to return from a fetched document.

To introduce a projection in a *find* query, one must add an object with the field to retrieve followed by a 1, as in Listing 2.

III. SMEAGOL

Fig. 2 describes SMEAGOL's workflow. First, SMEAGOL needs to analyze the project to create its CodeQL database. Then, it runs the defined queries against the project to check for potential smell instances. Finally, it outputs the locations of the smell instances into various formats (SARIF, CSV, etc.).

To explain the process, we introduce the necessary concepts of the CodeQL analysis, we explain how we detect smells in MongoDB queries using CodeQL, and finally, we present the list of smells supported by SMEAGOL.

[6]https://www.mongodb.com/docs/manual/tutorial/project-fields-from-query-results/

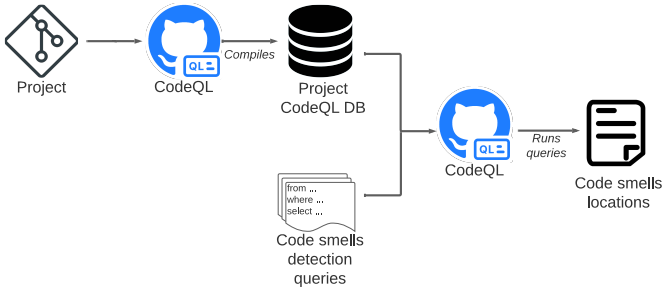

Fig. 2. SMEAGOL workflow

```
1  from CallExpr ce
2  where ce.getCalleeName() = "findOne"
3  select ce, "This is a call to a findOne function"
```

Listing 3. CodeQL query example

1) CodeQL Analysis: We use CodeQL[2] as our static analysis framework. It supports multiple languages, including JavaScript, our target language. CodeQL stores the analysis results of a project in a relational database where each table corresponds to an AST (Abstract Syntax Tree) element alongside analysis data. This database can then be queried in QL, a declarative, object-oriented query language with SQL-like syntax. CodeQL also implements analysis techniques, such as dataflow or taint analyses.

Listing 3 shows a simple yet powerful CodeQL query to find every call to a function *findOne*.

First, the `from` clause specifies the AST element to query, in this case a function call. Then, the `where` clause defines the constraint to apply to the element. The query checks whether the name of the function being called is *findOne*. Finally, the `select` clause specifies the output element alongside an alert message.

Being object-oriented, CodeQL supports class "inheritance", which is actually a logical reuse system facilitating the grouping of similar elements and leveraging the implementation complexity.

2) Analyzing MongoDB Queries in CodeQL: As a motivating example, let us assume we have two functions *foo* and *bar* performing document insert in the database, where the inserted document lies in the first and second arguments, respectively. If we were to implement it in a single CodeQL query, we would have to handle this variability in the `where` clause, which can get confusing, as showcased by Listing 4.

```
1  from CallExpr insertQuery, ObjectExpr insertedDocument
2  where
3      (insertQuery.getCalleeName() = "foo" and insertQuery
           .getArgument(0) = insertedDocument) or
4      (insertQuery.getCalleeName() = "bar" and insertQuery
           .getArgument(1) = insertedDocument)
5  select insertedDocument, "An inserted document"
```

Listing 4. CodeQL query variability example

```
1  abstract class InsertQueryCall extends CallExpr {
2      abstract ObjectExpr getInsertedDocument() {} }
3
4  class FooCall extends InsertQueryCall {
5      FooCall() { this.getCalleeName() = "foo" }
6      override ObjectExpr getInsertedDocument() {
7          result = this.getArgument(0) } }
8
9  class BarCall extends InsertQueryCall {
10     BarCall() { this.getCalleeName() = "bar" }
11     override ObjectExpr getInsertedDocument() {
12         result = this.getArgument(1) } }
```

Listing 5. CodeQL class example

Listing 5 shows an example implementation to handle this issue. First, we define an abstract class (Lines 1-2) to group the two related Objects. The `extends` keyword indicates an inheritance from the CallExpr type, allowing it to reuse its values and predicates. It also introduces an abstract predicate, `getInsertedDocument`, which we override in subsequent classes to obtain the document. Then, we define two classes: FooCall and BarCall (Lines 4-7 and 9-12). The characteristic predicate (Lines 5 and 10) defines the constraints related to their class, here, function callee names "foo" or "bar". Now that the domain is restrained, we can override the predicate based on the function and obtain the inserted document from the relevant argument position (Lines 7 and 12).

To retrieve all the inserted documents in a given project, we need to query the abstract class, as highlighted by Listing 6. Note that the query has gained clarity and has higher abstraction than the one from Listing 4.

```
1  from InsertQueryCall iqc
2  select iqc.getInsertedDocument(), "An inserted document"
```

Listing 6. CodeQL class usage example

This feature helps handle our task variability. Indeed, we need to extract various information sources, such as index declarations, collection names or attributes associated with a query, coming from different method calls or Mongoose schema declarations. As such, we could abstract it and simplify the queries while handling the fluctuations in the background.

Another feature we extensively use is dataflow analysis, as we do not assume that developers explicitly define the inserted document(s) in the query (*e.g.,* as object expressions). Hence, we track the documents' accesses occurring before their insertion to gather information on the collections' attributes.

3) MongoDB Code Smells: To establish the list of smells to detect, we used several sources we already knew prior to the study, including Mahajan *et al.*'s work on NoSQL energy efficiency [1], three books [17]–[19] and a blog series of MongoDB developers [21]–[24].

We defined the following 8 smells from the sources: *Case-insensitive queries without matching index (S1)*, *Index intersection rather than compound index (S2)*, *Querying too much data (S3)*, *Negation in queries (S4)*, *Uncovered queries (S5)*, *Sorted monkeys (S6)*, *Multiple schemas in a file (S7)* and *Using a document only for ID field (S8)*.

In summary, three smells concern index misusage (*S1*, *S2* and *S5*), three describe incorrect use of a query (*S3*, *S4* and *S6*), and two indicate modeling issues (*S7*, *S8*), with *S7* being Mongoose-specific.

Due to limited space, we describe the smells in detail in the companion repository.[7]

To detect the smells, we followed a cycle of defining the global detection rule associated with a smell, then implementing this detection in subsequent CodeQL queries and evaluating it against the global detection rule. The implementation contains 1356 lines of code and defines 58 classes.

All the detection rules and their implementations are available in the companion repository.[7]

IV. Case Studies

We demonstrate SMEAGOL with case studies on five open-source JavaScript projects chosen with over 1k stars on GitHub, that use either MongoDB Native Driver or Mongoose. We ran our tool against the latest commit of the default branch. We used one monthly commit as an evolution proxy and ran SMEAGOL against each version.

Table I presents an overview of the results. We manually reviewed a representative sample ($n = 98$) to assess SMEAGOL's accuracy, looking for potential false positives. The analysis confirmed that 73% of the detected instances were true positives after excluding two ambiguous cases.[8] The number in parentheses is $\frac{TP}{TP+FP}$.

During this inspection, we found that *HabitRPG/habitica* did not declare its indexes in the codebase. Instead, the developers document the indexes in a markdown file,[9] making the index declaration process invisible for SMEAGOL and, thus, triggering false positive warnings. Such an approach is risky in maintaining the indexes as the developers must keep this documentation up-to-date.

SMEAGOL detected at least one smell in each project, and the *Querying too much data (S3)* and *Uncovered queries (S5)* smells appeared in all projects.

While we could not detect *Index intersection rather than compound index (S2)* smells, our manual inspection revealed an instance.[7] Indeed, while the query uses three different indexed attributes (`userId`, `challenge.id` and `challenge.taskId`), this has no positive impact on this query as they would need to be declared in the same index.

No instances of *Case-insensitive queries without matching index (S1)* and *Index intersection rather than compound index (S2)* could be found.

Using a document only for ID field (S8), *Multiple schemas in a file (S7)* and *Sorted monkeys (S6)* were found 2, 7 and 1 times, respectively. In the case of *S8*, one was a false positive as the project used `Object.assign()` to copy a document's fields, and SMEAGOL does not support its detection. For the other one,[7] we could confirm that the inserted document only had one field, its `_id`.

[7]See companion repository link in the abstract.

[8]Two queries used projection with fields coming from a function's argument. We excluded them as it was impossible to tell from the code whether they included too many fields.

[9]https://github.com/HabitRPG/habitica/blob/develop/migrations/docs/mongo-indexes.md

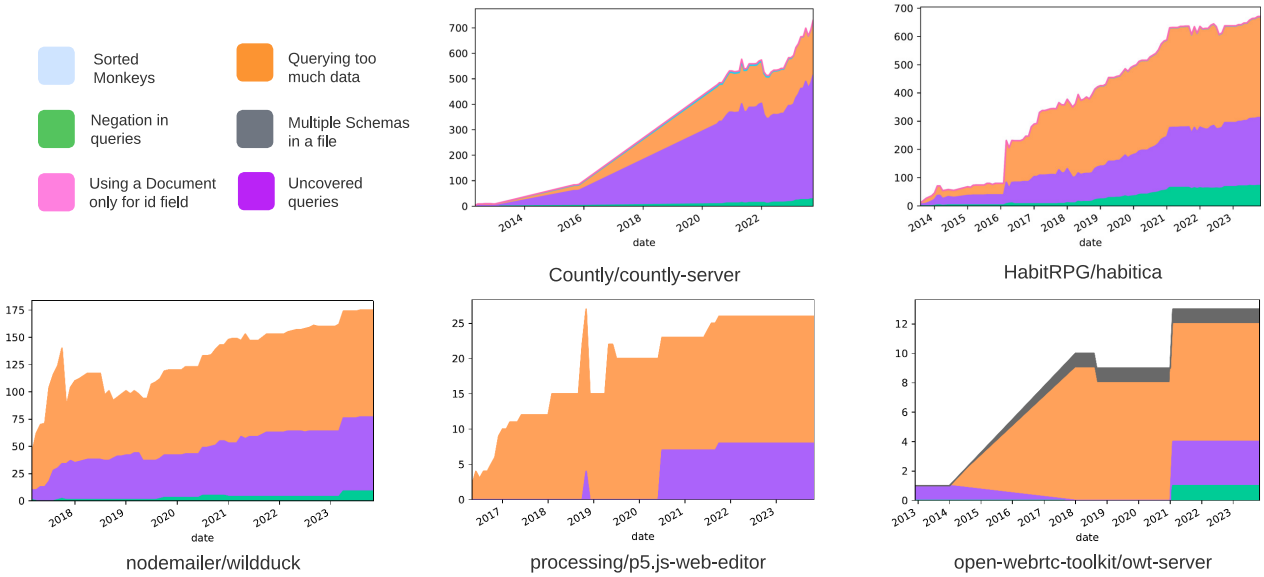

Fig. 3. Code smells evolution over time

TABLE I
PROJECTS RESULTS

Project	Stars	LOC	S1	S2	S3	S4	S5	S6	S7	S8
Countly/countly-server	5,289	290,290	0	0	217 (4/9)	29 (5/5)	490 (10/10)	1 (1/1)	0	1 (1/1)
HabitRPG/habitica	10,708	116,261	0	0	355 (8/10)	72 (7/7)	241 (4/10)	0	2 (2/2)	1 (0/1)
nodemailer/wildduck	1,755	53,145	0	0	98 (3/9)	9 (5/5)	68 (5/6)	0	0	0
processing/p5.js-web-editor	1,078	13,557	0	0	18 (2/5)	0	8 (4/5)	0	0	0
open-webrtc-toolkit/owt-server	1,057	49,568	0	0	8 (5/5)	1 (1/1)	3 (3/3)	0	1 (0/1)	0

The *Uncovered queries (S5)* smell was detected in every project, suggesting that the projects did not systematically declare indexes for each query.

Querying too much data (S3) was also detected in each project. This smell has the most instances (696). Interestingly, we found numerous cases in test files. An example in *HabitRPG/habitica*[7] illustrates this: a `User` document is retrieved on line 2083, and until the unit test ends on line 2085, only one attribute is accessed: `items.mounts`. To gain test performance, a projection should be used to return only this particular field.

Fig. 3 reports the result of the evolution analysis for the five projects. We observe a constant increase in the number of smell instances across all projects. The two largest projects, *Countly/countly-server* and *HabitRPG/habitica* count the highest proportion of *Uncovered query (S5)*. However, it may be deliberate by the maintainers: as the number of indexes grows within a system, so does the memory size they occupy. Developers might consider a trade-off between storage space and execution time, which is impossible to assess statically. *nodemailer/wildduck* has a similar proportion of *Uncovered queries*. Nevertheless, this can be a good indicator for developers who may not know whether a query is appropriately indexed, and it can help them diagnose a slow query.

We can also spot a sizeable prominence of the *Querying too much data (S3)* smell in all projects.

It can be explained by developers' ignorance about the projection concept or, while not being accessed, documents being saved as such by a user-defined function. Further investigations are needed to figure out the main cause. *processing/p5.js-web-editor* and *open-webrtc-toolkit/owt-server* experience abrupt evolution due to the low number of data points.

Overall, SMEAGOL allowed us to uncover an evolution of code smells with a constantly growing number of instances in the five projects. It may indicate that developers are unaware of those smells and require guidance to discover and fix them.

V. RELATED WORK

Several approaches and tools exist for identifying issues in database communication.

Van Den Brink *et al.* analyzed SQL queries within PL/SQL, COBOL, and Visual Basic systems [25]. Nagy *et al.* proposed SQLInspect to detect the SQL antipatterns defined by Karwin [12] within Java programs [14]. Chen *et al.* focused on detecting antipatterns in systems utilizing Object-Relational Mapping (ORM) [26]. In their subsequent research, they explored the performance implications of redundant data accesses [11] and investigated how web applications could be enhanced by refactoring performance antipatterns [27]. Lyu *et al.* developed SAND, a static analysis tool to detect SQL antipatterns in mobile apps [28].

Yang *et al.* relied on dynamic analysis to identify ORM performance antipatterns in Ruby on Rails applications [29]. Later, they introduced PowerStation, an IDE plugin for RubyMiner, which detects ORM inefficiencies and suggests fixes to developers [30]. Yan *et al.* proposed a static analysis method to pinpoint and address ORM issues in Ruby on Rails applications [31].

The closest existing tool is MongoDB Atlas, a cloud database service for database deployment and management.[10] Among its many features, it can detect MongoDB Schema Design Anti-Patterns [16] directly from the data. In contrast to Atlas, SMEAGOL statically detects smells in the *application source code*, without accessing the data stored in the database.

VI. Conclusion

We presented SMEAGOL, a static analysis tool to detect MongoDB code smells in JavaScript projects that use the Native MongoDB Driver or Mongoose. It relies on multiple CodeQL queries, each dedicated to detecting a particular smell. We defined 8 smells from a research paper, books and official MongoDB articles. SMEAGOL leverages CodeQL dataflow analysis and classes to extract information about the document data structure from the database access code, further refining the detection process. Applying SMEAGOL to five open-source projects allowed us to detect instances of each code smell, and to observe a high prevalence of two smells and a constant growth in smell instances over time.

While SMEAGOL showed its potential through a preliminary evaluation, more work needs to be done to increase its usefulness for developers. For instance, CodeQL supports many programming languages, and we implemented SMEAGOL for JavaScript. Each language has peculiarities, but the core API is generic, and the approach could also be adapted to other languages. In the future, we plan to extend the number of projects analyzed to conduct a more comprehensive evaluation, assess its limitations, and have a better glimpse at the community's MongoDB code smell awareness. We also plan to extend the supported code smell catalog systematically.

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] D. Mahajan, C. Blakeney, and Z. Zong, "Improving the energy efficiency of relational and NoSQL databases via query optimizations," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 120–133, 2019.

[2] A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in MongoDB," in *Proc. of IACC'14*, 2014, pp. 416–421.

[3] J. Kumar and V. Garg, "Security analysis of unstructured data in NOSQL MongoDB database," in *Proc. of IC3TSN'17*, 2017, pp. 300–305.

[4] S. Wen, Y. Xue, J. Xu, L.-Y. Yuan, W.-L. Song, H.-J. Yang, and G.-N. Si, "Lom: Discovering logic flaws within MongoDB-based web applications," *International Journal of Automation and Computing*, vol. 14, no. 1, pp. 106–118, Feb. 2017.

[5] A. Ron, A. Shulman-Peleg, and A. Puzanov, "Analysis and mitigation of NoSQL injections," *IEEE Security & Privacy*, vol. 14, no. 2, pp. 30–39, 2016.

[10] https://www.mongodb.com/atlas

[6] V. Sachdeva and S. Gupta, "Basic NOSQL injection analysis and detection on MongoDB," in *Proc. of ICACAT'18*, 2018, pp. 1–5.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[8] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *EMSE*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018.

[9] A. Shome, L. Cruz, and A. v. Deursen, "Data smells in public datasets," in *Proc. of CAIN'22*, 2022, pp. 205–216.

[10] T. Sharma, M. Fragkoulis, S. Rizou, M. Bruntink, and D. Spinellis, "Smelly relations: Measuring and understanding database schema quality," in *Proc. of ICSE'18 (SEIP)*. ACM, 2018, pp. 55–64.

[11] T. H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *TSE*, vol. 42, no. 12, pp. 1148–1161, Dec. 2016.

[12] B. Karwin, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*, 1st ed. Pragmatic Bookshelf, 2010.

[13] S. Shao, Z. Qiu, X. Yu, W. Yang, G. Jin, T. Xie, and X. Wu, "Database-access performance antipatterns in database-backed web applications," in *Proc. of ICSME'20*, 2020, pp. 58–69.

[14] C. Nagy and A. Cleve, "SQLInspect: a static analyzer to inspect database usage in Java applications," in *Proc. of ICSE'18*, 2018, pp. 93–96.

[15] N. Bessghaier, A. Ouni, and M. W. Mkaouer, "A longitudinal exploratory study on code smells in server side web applications," *Software Quality Journal*, vol. 29, no. 4, pp. 901–941, Dec. 2021.

[16] L. Schaefer and D. Coupal, "A summary of schema design anti-patterns and how to spot them." [Online]. Available: https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-summary/

[17] S. G. Edward and N. Sabharwal, *Practical MongoDB*. Apress, 2015.

[18] S. Bradshaw, E. Brazil, and K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media, 2019.

[19] K. Chodorow, *50 Tips and Tricks for MongoDB Developers: Get the Most Out of Your Database*. O'Reilly Media, Inc., 2011.

[20] P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "An empirical study of (multi-) database models in open-source projects," in *Proc. of ER'21*, 2021.

[21] MongoDB, "Mongodb operations best practices," Jun. 2017. [Online]. Available: https://s3-ap-southeast-1.amazonaws.com/tv-prod/documents/null-10gen-MongoDB_Operations_Best_Practices.pdf

[22] D. C. Lauren Schaefer, "Case-insensitive queries without case-insensitive indexes," May 2022. [Online]. Available: https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-case-insensitive-query-index/

[23] "The mean stack: Mistakes you're probably making with mongoosejs, and how to fix them," https://www.mongodb.com/blog/post/the-mean-stack-mistakes-youre-probably-making.

[24] C. Sarrazin, "Raiders of the anti-patterns: A journey towards fixing schema mistakes in mongodb," https://www.slideshare.net/mongodb/mongodb-world-2019-raiders-of-the-antipatterns-a-journey-towards-fixing-schema-mistakes-in-mongodb.

[25] H. Van Den Brink, R. Van Der Leek, and J. Visser, "Quality assessment for embedded SQL," in *Proc. of SCAM'07*, 2007, pp. 163–170.

[26] T. H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proc. of ICSE'14*. IEEE, May 2014, pp. 1001–1012.

[27] B. Chen, Z. M. Jiang, P. Matos, and M. Lacaria, "An industrial experience report on performance-aware refactoring on a database-centric web application," in *Proc. of ASE'19*, 2019, pp. 653–664.

[28] Y. Lyu, S. Volokh, W. G. J. Halfond, and O. Tripp, "SAND: A static analysis approach for detecting SQL antipatterns," in *Proc. of ISSTA'21*. ACM, 2021, pp. 270–282.

[29] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How not to structure your database-backed web applications: A study of performance bugs in the wild," in *Proc. of ICSE'18*. IEEE, 2018, pp. 800–810.

[30] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "PowerStation: Automatically detecting and fixing inefficiencies of database-backed web applications in IDE," in *Proc. of ESEC/FSE'18*, 2018, pp. 884–887.

[31] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proc. of CIKM'17*, 2017, pp. 1299–1308.