Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

DOCTOR OF SCIENCES

Empirical Investigation and Static Detection of Code Smells in Applications using Document-Oriented Datastores

Cherry, Boris

Award date:
2024

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Mar. 2025

Empirical Investigation and Static Detection of Code Smells in Applications using Document-Oriented Datastores

Boris Cherry

Jury

Prof. Serge Demeyer
University of Antwerp, Belgium

Prof Michele Lanza
Università della Svizzera italiana, Switzerland

Prof. Benoît Vanderose
University of Namur, Belgium

Prof. Xavier Devroey
University of Namur, Belgium

Dr. Gilles Perrouin (President)
University of Namur, Belgium

Prof. Anthony Cleve (Supervisor)
University of Namur, Belgium

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Supervised by Prof. Anthony Cleve

University of Namur

PReCISE Research Center

Couverture: ©Presses universitaires de Namur

©Presses universitaires de Namur & Boris Cherry
Rue Grandgagnage 19
5000 Namur (Belgique)
pun@unamur.be - www.pun.be

Tous droits de reproduction, traduction, adaptation, même partielle, y compris les microfilms et les supports informatiques, réservés pour tous les pays.

ISBN : 978-2-39029-201-2
Dépôt légal: D/2024/1881/13
Imprimé en Belgique

“Il faut imaginer Sisyphe heureux”

— ***Albert Camus, Le mythe de Sisyphe (1942)***

Abstract

The recent increase in data volume changed the landscape of database-centric applications by unveiling the limitations of relational databases, especially in terms of performance, storage or distribution. Consequently, alternative databases, namely the NoSQL databases, emerged by proposing new ways to scale with the ever-increasing data volume by relaxing usual relational modeling features. This had led developers to rely on a document-oriented datastore, as being part of this technology family, for their primary database. However, this shift has introduced new challenges as optimal document database modeling remains largely unknown to maintainers. Consequently, developers may introduce antipatterns in their application that would hinder program maintainability and performance.

Our central challenge is to assist developers in maintaining and evolving software systems relying on Document-Oriented Datastores. Identifying code smells or antipatterns in application code interacting with a document-oriented database is difficult due to varying performance outcomes based on data volume and query structure. Also, there is currently little research attention to the specific document-oriented antipatterns symptoms/code smells.

Furthermore, identifying the interaction points between application programs and document-oriented databases is challenging. Indeed, those programs are for the most part written in JavaScript which is known to be hard to statically analyse. Accurately detecting method calls, the standard database access form, remains an unsolved problem in JavaScript static analysis as for building sound call graphs.

Finally, detecting code smells or antipatterns instances requires knowledge of the underlying data structure, which is often not directly accessible in open-source projects due to privacy concerns. Also, unlike relational databases where data structures are explicit in (embedded) query strings, document fields may not be explicitly present in every query, making it harder to analyze and inspect these systems effectively.

The main purposes of this thesis are to a) empirically build a structured catalog of code smells and antipatterns relative to document-oriented databases; b) automatically detect instances of those code smells and antipatterns directly from the application code.

Keywords: Document-Oriented databases, Static Analysis, Code Smells, Antipatterns, Reverse engineering

RÉSUMÉ

L'augmentation récente du volume de données a transformé le paysage des applications centrées sur les bases de données en révélant les limites de la modélisation relationnelle, notamment en termes de performance, stockage et distribution de donnée. Par conséquent, des modèles de bases de données alternatifs, les bases de données NoSQL, ont émergé en proposant de nouvelles manières de s'adapter à l'augmentation constante du volume de données en assouplissant les caractéristiques habituelles de la modélisation relationnelle. Cela a conduit certains développeurs à s'appuyer sur une base de données orientée document, catégorie des bases de données NoSQL, comme base de données principale. Cependant, ce changement a introduit de nouveaux défis, car la modélisation optimale des bases de données orientées document reste largement inconnue des mainteneurs. Par conséquent, les développeurs risquent d'introduire des antipatterns dans leurs applications, pouvant nuire à la maintenabilité et aux performances du programme.

Notre défi central est d'aider les développeurs à maintenir et faire évoluer leurs systèmes interagissant avec des stockages orientés documents. Identifier les mauvaises pratiques ou antipatterns dans le code de ces systèmes est difficile en raison de performances variables dépendant du volume de données et de la structure des requêtes. De plus, il y a actuellement peu de recherches sur les antipatterns ou mauvaises pratiques spécifiques aux bases de données orientées document.

En outre, extraire les points d'interaction entre un programme d'application et une base de données orientée documents est difficile. En effet, ces systèmes sont pour la plupart écrits en JavaScript, un langage reconnu comme difficile pour l'analyse statique. Détecter précisément les appels de méthode et construire des graphes d'appel fiables restent des problèmes non résolus dans l'analyse statique de JavaScript, alors que les applications utilisent les appels de méthodes pour interagir avec les bases de données orientées document.

Enfin, détecter les mauvaises pratiques ou antipatterns nécessite de connaitre la structure de donnée sous-jacente au programme, qui n'est souvent pas directement accessible en tant que tel dans les projets open-source, notamment pour des raisons de confidentialité. De plus, contrairement aux bases de données relationnelles où les structures de données sont explicitées dans les requêtes, les champs composant les documents peuvent ne pas être explicitement présents dans chaque requête, ce qui rend l'analyse et l'inspection de ces systèmes plus difficiles.

Les principaux objectifs de cette thèse sont de a) développer empiriquement un catalogue complet des code smells et antipatterns relatifs aux bases de données orientées document; b) détecter automatiquement les instances de ces code smells et antipatterns directement dans le code source de l'application.

Mots clés : Bases de données orientée document, Analyse statique, Code Smells, Antipatterns, Rétro-ingénierie

ACKNOWLEDGEMENTS

This work marks the end of my PhD student journey that was initiated back in September 2020. What started as “why not?” adventure ended up being packed with insightful learnings, enriching experiences and unforgettable meeting but also fierce adversity and complex challenges. I had to give it my all to get through it but I surely could not have done it without all the support I received during this journey.

The first companion I would like to thank is Anthony, my supervisor. He’s the one who embarked me on this wonderful journey, I could not be more thankful for this life-changing opportunity. But he was not just a mere initiator, he also supported for the whole duration of the trip, highlighted my achievements, and cheered me up when I was at my lowest. He also took me on awesome conferences trip where he always made sure I was in the best condition to meet people or just enjoy the food and many drinks (with modération). So Anthony, I sincerely want to thank you for all those years, meetings, guidances, trips and for always keeping your faith in me when I needed it the most.

My second companion is Csaba, my navigator. During the darkest time of the never-ending quarantines which made the first half of our journey, he was there to guide me through the storms. His limitless stamina helped me greatly to reach the goals and I could always rely on his quick feedbacks to get to the next point. So Csaba, I was really glad to meet you and have you around for the research and especially for all those fun times and beers we shared together.

And the final crew member I want to deeply thank is my partner, Claire. She always found the way to talk me out of my self-doubts and lack of confidence. She was (and still is) my first supporter for all the challenges I can attend and the Sun to outshine my grumpiness. So Claire, I love you and really proud to have you in my life. Moreover, I would like to thank my mother and my sister to have encouraged me to try this journey and cheered me up for the whole duration. Also, thank to both of my dogs, Kaly and Pixa for their eternal love and support.

I would also want to thank all the office co-workers I have met: thank you Maxime Gobert for the jeudredis, Pol for all those lunches, Loup for his quick wisdom clues, Jean for his presence and especially Maxime André for all those memorable moments in the office, lunches and in Lisbon. I have a special thought for Xavier who guided me through research during my master thesis and has been a fervent supporter ever since.

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).

Also, thank you to all the wonderful people I met during my time in Namur Computer Science Faculty: Thibaut, all the Jérôme's, Xavier, Sacha, Valentin, all the Antoine's, Maxime, Pierre, Alix, Jules, Simon and all the others. Thank also to my steering committee members, Prof. Michele Lanza and Prof. Benoît Vanderose for their accompaniment and relevant questions during my thesis. Additionally, thanks to the remaining members of my jury, Prof. Serge Demeyer, Prof. Xavier Devroey and Dr. Gilles Perrouin for their time and attention. Finally, I would like to say a big thank you to my friends who proof read this work, Mathieu, Thibaut, Jérôme, Maxime and Ludovic.

To all the people I have exchanged with during this wonderful journey, I am grateful to have met you and I hope to soon see you around!

CONTENTS

Contents **xi**

List of Figures **xiii**

List of Tables **xv**

Acronyms **xvii**

Preface **xix**

Introduction **xxi**
Context . . . xxi
Problem Statement . . . xxii
Goals . . . xxiii
Research Questions . . . xxiv
Contributions . . . xxiv
Publications . . . xxiv
Structure of the thesis . . . xxv

1 Background **1**
1.1 NoSQL . . . 1
1.2 Static Analysis . . . 7
1.3 Concluding remarks . . . 11

2 State of the Art **13**
2.1 Database access extraction . . . 13
2.2 Code smell . . . 15
2.3 Database communication smell detection . . . 17
2.4 Concluding remarks . . . 20

I Contributions **23**

3 Database accesses extraction **25**
3.1 Introduction . . . 25
3.2 Approach . . . 26

3.3 Evaluation . . . 41
3.4 Case Studies . . . 45
3.5 Threats to Validity . . . 47
3.6 Conclusion . . . 48

4 Code smell catalog **49**
4.1 Introduction . . . 49
4.2 Background . . . 50
4.3 Method . . . 52
4.4 RQ_1: MongoDB Code Smell Catalog . . . 56
4.5 RQ_2: Source Types' Classification . . . 81
4.6 Discussion . . . 84
4.7 Conclusion . . . 88

5 SMEAGOL **91**
5.1 Introduction . . . 91
5.2 Tracked code smells . . . 93
5.3 SMEAGOL . . . 96
5.4 Results . . . 103
5.5 Conclusion . . . 110

II Postface **111**

6 Conclusion **113**
6.1 Summary of the contributions . . . 113
6.2 Future Directions . . . 115

Bibliography **119**

List of Figures

1.1 Document example 2
1.2 Collection example 3

3.1 Approach overview 27
3.2 Example of import chain 38
3.3 Evaluation setup 40
3.4 Labeller sheet screenshot 42
3.5 Evolution of database accesses in Bitcore 46
3.6 Evolution of database accesses in Overleaf 47

4.1 Countries collection 51
4.2 Cities collection 51
4.3 Countries collection with embedding 53
4.4 Overview of the Multivocal Literature Mapping Approach 54
4.5 MongoDB Code Smell Classification 57
4.6 Parallel categorization of source category and smell type 80
4.7 Network representation of smells and sources 82
4.8 Induced smell per category 84
4.9 Distribution of sources according to publication year 85

5.1 Countries collection 92
5.2 SMEAGOL workflow 97
5.3 SMEAGOL individual query execution 98
5.4 Detection query definition process 103
5.5 Benchmark building overview 105

List of Tables

2.1 Database communication smell detection approach 20

3.1 MongoDB Node driver database access methods 28
3.2 Mongoose database access methods 30
3.3 Project statistics 41

4.1 Source Types classification 83

5.1 MongoDB database access methods 94
5.1 MongoDB database access methods 95
5.1 MongoDB database access methods 96
5.2 Benchmark projects statistics 104
5.3 Retrieved code smells instances 106
5.4 Projects containing code smells ($n = 340$) 108
5.5 Top 20 projects in terms of smells occurrences 108
5.6 Data structure information for projects having at least one instance ($n =$ 340). 109

ACRONYMS

DBMS	DataBase Management Systems	xxi, 11, 115
DOM	Document Object Model	33, 36, 37
ERP	Enterprise Resource Planning	43, 107
LA	Linguistic Antipatterns	17, 18
MLM	Multivocal Literature Mapping	xxv, 49, 50, 53, 56, 81, 87, 88, 114
NoSQL	Not only SQL	1, 14, 25, 49
ODM	Object Document Mapper	11, 117
OOP	Object-Oriented Programming	xxii, 15
ORM	Object-Relational Mapping	xxiii, 16, 18, 19
SLR	Systematic Literature Review	7, 16

Preface

INTRODUCTION

Context

Database-Centered Systems

Database-centered systems play a crucial role in the operation of organizations. They are at the center of crucial functions such as administration or production. Such systems are characterized by one or more DataBase Management Systems (DBMS) to handle large volumes of data while providing guarantees such as the one evoked by the CAP (Consistency Availability Partition tolerance) or BASE (Basically Available, Soft state, Eventual consistency) theorem. Those DBMS are accessed by application programs for them to store, retrieve, and manipulate data. This symbiotic relationship places the database as the system's central component and critical point of care and maintenance.

Document-Oriented Datastores

Document-oriented datastores are databases that store their data as documents. A document is characterized by its fields and associated values, that can also be other fields, de facto enabling document embedding. Supported value types depend on the datastore implementation, which can then support many document formats such as YAML, JSON or XML.

Unlike relational databases, document-oriented databases do not impose a strict data schema. Each database may utilize different query languages, including native programming language methods in MongoDB, a dedicated query language in BaseX, and an SQL-like language in Couchbase. Moreover, horizontal scaling in document-oriented databases is typically achieved through sharding, which involves distributing related documents across different shards based on specific field values. This approach differs from the one used by relational databases, which often balance data across replicas at a physical level. Additionally, while data normalization is critical in relational data modeling, per respect to the Second and Third Normal Form, this property can be violated by the document embedding mechanism. Consequently, habits and patterns acquired by a relational developer cannot be directly transposed to document database modeling.

In the recent years, document-oriented databases have emerged as a popular alternative to relational modeling for handling growing data volumes. MongoDB is currently one of the most widely used document-oriented database platform, ac-

cording to the "*DB-engines*" ranking [1]. This was also confirmed by a recent empirical study about database technology usage in open-source projects [25].

Code Smells and Antipatterns

In software development, the term "*code smells*" refers to symptoms in the application code that may indicate deeper quality problems. These are not bugs or defects resulting in a crash or malfunction, but rather observable characteristics of the code suggesting underlying issues that could harm system maintainability. Code smells often manifest as recognizable patterns, such as duplicate code, too long methods, or poor naming conventions.

William J. Brown *et al.* defined an antipattern in its book "*AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*" [29] as "*a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences*". In contrast to design patterns that are optimal solutions, antipatterns refer to solutions that ultimately lead to harmful consequences, such as limited performance or complexity overhead. Brown *et al.*'s definition includes four main components: the root cause (*e.g.,* misunderstanding a design principle) that leads to the antipattern, the symptoms that help identify the antipattern in the system, the impacts of these antipatterns, and a refactored solution to resolve them. Antipatterns can manifest in various aspects of a system, including development, architecture, and project management. A notable antipattern in Object-Oriented Programming (OOP) is the "*God Class*", which consists in a single class handling too much responsibility over an application's code, compared to the other classes.

Recognizing and identifying those issues in their application code is a common practice for software development teams. Consequently, their automated detection can guide the development process and the quality assurance policy. Therefore, tracking and prioritizing the removal of the right antipatterns and code smells is a crucial task for developers.

Problem Statement

The global increase of data volume brought some developers to adopt Document-Oriented Datastores to achieve better scaling and higher flexibility compared to traditional relational databases. However, as those databases drop traditional features of the relational paradigm, they also introduce new challenges. Unlike relational modeling which benefits from many years of research and a traditional place in Computer Science education, optimal document database modeling and interactions often remain unknown to most of the maintainers. Consequently, those maintainers may introduce antipatterns or make mistakes in their programs while interacting with their database, which could ultimately hinder program maintainability or performance. This motivates the central challenge of this thesis, which is: **How can we support developers in improving their programs interactions with document-oriented datastores?**

Gathering the different code smells or antipatterns developers may introduce in their codebase when interacting with document database is challenging. Indeed,

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

similar modeling decisions may lead to different performance evaluations as query execution time heavily depends on the data volume and structure. For example, in MongoDB, while declaring indexes helps speed up the reading performance, they also use memory and burdens the data modification speed, making the recommendation task more complex. Also, maintainers concerned about avoiding antipatterns or code smells may be faced with the challenge to gather an exhaustive overview. Academic and developers' communities tend to produce small-scale and scattered reports of those undesirable behavior. Therefore a structured catalog of such smells and antipatterns is currently missing.

Furthermore, identifying and interpreting relevant database accesses also pose a challenge. One of the most popular languages used to develop open-source systems nowadays is JavaScript. Those systems typically access their database through MongoDB driver method calls. Accurately extracting such method calls from a program source code proves to be challenging, given the dynamic nature of the language. Whether it is to soundly build complete call graphs or faithfully identifying method call signatures, the community still misses silver bullet methods for JavaScript static analysis. Thus, statically identifying document datastores database accesses remains an unsolved problem.

Finally, accurately detecting code smells or antipatterns in the application code interacting with a document-oriented database also constitutes a challenge. Indeed, determining if a specific database access suffers from an antipattern or corresponds to a code smell instance requires a deep knowledge of the underlying data structures. However in open-source systems and for privacy reasons, the data is not often publicly available and if we want to assist those projects developers, the application code is sometimes the only available source of information. In contrast to relational querying, where the table and column names appear explicitly in query strings or ORM annotations, some document fields may not be mentioned in any query or can be only visible in the surrounding code.

Goals

This dissertation focuses on database-centered applications using document-oriented datastores, and address the topic of code smells and antipatterns in MongoDB database accesses. More precisely, our main goals are to:

(i) Provide a structured catalog offering a global overview of MongoDB database communication smells in database accesses, modelling and application contexts.

(ii) Automatically detect those smells in JavaScript systems to help developers in their maintenance tasks.

To achieve those goals, we formulate our thesis:

> In order to help developers improving how their applications interact with their underlying database, we undergo an empirical investigation of document-oriented database code smells and antipatterns and their automatic detection in JavaScript applications.

Research Questions

This thesis research is guided by the following research questions:

RQ1: How to automatically and statically extract the database access fragments of application programs using a document-oriented datastore?

RQ2: How to extract implicit data structure information from the database access fragments?

RQ3: How are the MongoDB code smells and antipatterns discussed by the community?

- **RQ3.1:** What types of MongoDB code smells and antipatterns have been identified by the community?
- **RQ3.2:** Where are discussed those code smells and antipatterns?

RQ4: How to automatically detect code smells and antipatterns in application programs using MongoDB datastore?

Contributions

The major contributions of this thesis include:

- A generic approach to statically distinguish target method call from noise in JavaScript systems (*RQ1*);
- A static analysis approach to extract MongoDB database accesses from JavaScript systems (*RQ1*);
- An approach to infer data structure information from document-database access code (*RQ2*);
- A comprehensive catalog of MongoDB code smells and antipatterns (*RQ3.1*)
- An investigation of the community description and discussions concerning MongoDB code smells and antipatterns (*RQ3.2*);
- A tool automatically detecting MongoDB code smells and antipatterns from JavaScript systems source code (*RQ4*).

Publications

The content of this thesis is based upon, reuses, and extends the following peer-reviewed publications by the author:

Conferences

[35] Boris Cherry, Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of database accesses in MongoDB applications. In **2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)**. IEEE, mar 2022

[36] Boris Cherry, Jehan Bernard, Thomas Kintziger, Csaba Nagy, Anthony Cleve, and Michele Lanza. A multivocal mapping study of MongoDB smells. In **2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024)**. IEEE Computer society, 2024 (in press)

[37] Boris Cherry, Csaba Nagy, Michele Lanza, and Anthony Cleve. SMEAGOL: A static code smell detector for MongoDB. In **2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024)**. IEEE Computer society, 2024 (in press)

Structure of the thesis

Chapter 1 introduces the central concepts used in this thesis.

Chapter 2 surveys the state of the art in regards to the structure of the thesis, in order to identify research gaps related to our research questions.

Chapter 3 details an approach to statically extract database interactions between a JavaScript system and a MongoDB database and evaluates it on several open-source systems. Then it proceeds to analyze the evolution of database accesses of two popular open-source systems.

Chapter 4 presents an Multivocal Literature Mapping (MLM) study of MongoDB code smells and antipatterns by also relying on grey literature. It also organizes the results of the study into a structured catalog and discusses the information sources describing those code smells.

Chapter 5 presents SMEAGOL, a static detection tool to identify MongoDB code smells and antipatterns in JavaScript applications. It explains how the implementation was conducted and how information about the data structure was extracted and exploited during the code smell detection process. Finally, it presents the application of the tool to a benchmark including a large set of open-source projects and discusses the obtained results.

Chapter 6 revisits our research questions, summarizes our contributions, and anticipates the possible future works in the field of this thesis.

CHAPTER 1

BACKGROUND

1.1 NoSQL

Not only SQL (NoSQL) databases are a set of software systems designed to store data without the need of a strict data schema, unlike relational database management systems. While sharing a common naming, those systems offer different modeling, management and retrieving solutions for different issues. Hect and Jablonski [59] classified such systems into 4 categories : *key-value stores*, *column family stores*, *graph databases* and *document stores*. For each category, we give implementations examples of those systems from the database-ranking website db-engines [1].

Key-Value Datastores Key-value stores manage data as a collection of key-value pairs. Each key is unique and is associated with a value. Value types can range from primitive types to collections holding multiple fields. These databases are efficient for simple read-write operations, such as caching or session storage. Examples of key-value stores include Redis [10] and Memcached [6].

Column Family Datastores Wide-column stores organize data in tables consisting of rows and columns, where each row is uniquely identified by a key. Columns are grouped into sets known as column families, and each row can have a varying number of columns within these families. Unlike traditional relational databases, the schema in wide-column stores can be flexible within each row, accommodating a diverse range of data types and structures. Such stores are useful to query large volumes of data. Cassandra [1] and Hbase [3] are examples of such systems.

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

```
_id: 1,
surname: "Tolkien”,
givenNames: [“John”, “Ronald”, “Reuel”],
birthPlace: {
      country: “Orange Free State”,
      city: “Bloemfontein”
}
```

Figure 1.1: Document example

Graph Databases Graph databases manage data as a collection of nodes and edges, the first representing entities and the latter the relationships between them. Each node can contain properties in key-value form. Edges, connecting these nodes, can also carry additional information about the nature of the relationship between entities, including direction and weight. This structure is suited for data having many relationships. Notable examples consist of Neo4j [9] and Memgraph [7].

Document-Oriented Datastores Document-oriented datastores rely on documents to store their data, which are then organized into collections. Each document is a structured data record which can come in various formats, such as XML, YAML or JSON. As such, documents hold key-value pairs whose type range depends on the document format. This allows documents to have a hierarchy of nested data. Documents can also have different structures and fields without needing to adhere to a strict schema. Those databases are effective at managing semi-structured data. The two biggest fishes in the pond are MongoDB [99] and Couchbase [2]

1.1.1 MongoDB

MongoDB is a document-oriented datastore. Its central concept is a *document* that stores all information for a given object. Each document consists of a JSON object, which is a set of a key-value pairs. Being JSON, it allows to work with multiple JavaScript types: string, number, boolean, array or document.

Fig. 1.1 introduces a *document* example, representing the famous fantasy writer, *J.R.R. Tolkien*. All the information displayed here comes from his Wikipedia page [2]. First it has an internal `_id` field; for the sake of simplicity we will use an integer for the rest of this thesis but behind the scenes, MongoDB uses a dedicated representation, the ObjectId [3]. It is composed of 12 bytes, 4 representing the timestamp, another 5 are randomly generated and the last 3 are incremented from a random value. This ensures the uniqueness of document identifiers across the whole database. Then, a `surname` field has a string value, storing the author's surname. Following it, we have the `givenNames` attribute which is a list containing every author's given names. Finally, we have a `birthPlace` field which is an embedded document holding two fields, the birth `country` and `city`.

Multiple related documents can be stored in a common *collection*. While they are analogous to a table in relational databases, they don't have to conform to a strict

[2] https://en.wikipedia.org/wiki/J._R._R._Tolkien
[3] https://www.mongodb.com/docs/manual/reference/method/ObjectId/

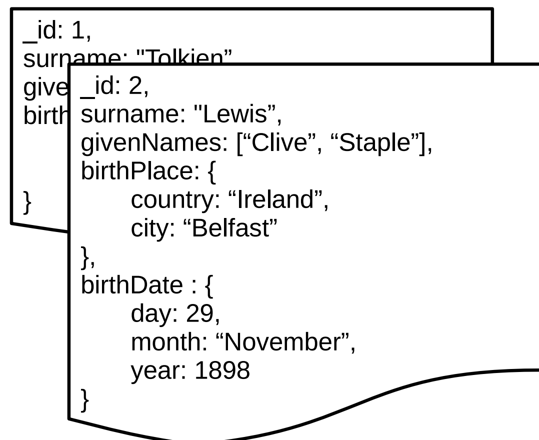


Figure 1.2: Collection example

schema and every document can be different. As showcased in Fig. 1.2, we can see a document bearing the _id value 2, representing another famous fantasy writer C.S. Lewis. [4] Notice that it possesses one additional field, `birthDate` which is an embedded document corresponding to the author's birth date.

MongoDB also provides an API to retrieve or insert documents. The retrieval part relies on a *filter* JSON objects to select specific documents. Listing 1.1 shows an example of how to select documents having "*Lewis*" as value for the `surname` field.

Listing 1.1: Filter definition example

```
1 {
2       "surname": "Lewis"
3 }
```

It is also possible to use *operators* to introduce additional logic into our queries. For example, we could select `author` documents having 2 or more given names. To do that, we can use the `$size` and `$gt` operators. The first one allows to query array fields by the number of elements while the latter queries a number element.

Listing 1.2: Filter operators definition example

```
1       {
2           "givenNames": { "$size" : { "$gt" : 2}}
3       }
```

Listing 1.2 illustrates the usage of those operators. It is worth noting that operators all start with a `$` sign. In this example, we select documents with `givenNames` field being an array and having at least 3 elements.

[4]https://en.wikipedia.org/wiki/C._S._Lewis

Index

Indexes are a MongoDB mechanism to improve read query performance. In an index-less scenario, MongoDB scans a complete collection. However, if the *field* used in the *filter* object has been thoroughly indexed, MongoDB will actually scan less documents. [5]

Behind the scenes, MongoDB stores the indexed fields of a collection in a dedicated binary tree. The values are stored in ascending or descending order depending on the direction given by the index. Listing 1.3 illustrates an index definition.

Listing 1.3: Index definition example

```
1 {
2     "givenName": 1
3 }
```

In this example, the `givenName` field from our author collection is indexed. The given value that is either 1 or -1, indicates respectively an ascending index or descending index. In this case, the value is 1, making it an ascending index.

Projection

The default behavior of MongoDB when querying is to return all fields from matching documents. To optimize data transfer, one can use a selection object to specify which fields to include or exclude in the returned results. [6]

The location of the projection document depends on the driver programming language however they always adopt the format described by Listing 1.4

Listing 1.4: Projection object example

```
1 {
2     "surname": 1,
3     "birthDate": {
4         "year": 1
5     }
6 }
```

Here, the projection object retrieves only the fields `surname` and `year` from the embedded document `birthDate`. The inclusion is indicated by the value `1`, to exclude a field the `0` value is preferred, as portrayed by Listing 1.5.

Listing 1.5: Projection object example with field exclusion

```
1 {
2     "givenNames": 0
3 }
```

In this example, every field from the document is retrieved but the `givenNames` one.

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

JavaScript drivers

On the commonly used package manager npm, the two most popular libraries to work with MongoDB are Mongoose ODM and MongoDB Node Driver. [7] 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.

Here, we briefly introduce the two APIs.

Listing 1.6: MongoDB insert query example

```
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          await db
9              .collection("authors")
10             .insertOne(
11             {
12                 surname: "Lindholm",
13                 givenNames : ["Margaret", "Astrid"],
14                 birthPlace : {
15                     country : "United State",
16                     city : "Berkeley"
17                 }
18             });
19     } catch (err) {
20         console.log(err);
21     } finally {
22         await client.close();
23     }
24 }
25 run().catch(console.dir);
```

Listing 1.6 illustrates a code snippet of an insert with the MongoDB driver. First, the program imports and initializes a `MongoClient` from the MongoDB Native driver (Lines 1-3). Then, it connects to the database (Line 7), performs a query to insert a single document into the authors collection (Lines 10-18) and closes the database connection (Line 22).

Listing 1.7: MongoDB find query example

```
1  /* same as lines 1-7 */
2      let britishAuthor = await db
3          .collection("authors")
4          .findOne({ surname: "Lindholm" });
5      console.log(britishAuthor.surname);
6  /* same as line 19-26 */
```

Listing 1.7 shows how to retrieve documents with the MongoDB driver. The client and database connection initialization is similar to the insert example. The code snippet retrieves a single document based on its `nationality` attribute (Lines

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

8-10) in order to print its `surname` property (Line 11). The database disconnection is also similar to the Listing 1.6.

Mongoose [8] is a Node.js library that provides an object modeling environment on top of MongoDB. As an ODM (Object-Document-Mapper), it abstracts part of the schema creation, data validation, and CRUD operations processes. We describe its functioning with an example.

Listing 1.8: Mongoose schema definition example

```
1  const mongoose = require("mongoose");
2
3  const AuthorSchema = new mongoose.Schema({
4      surname: String,
5      givenNames: [String],
6      birthPlace: {
7          country: String,
8          city: String
9      }
10 });
11
12 const Author = mongoose.model("authors", AuthorSchema);
13
14 module.exports = Author;
```

Listing 1.8 showcases an example of Mongoose Schema definition. A `Schema` is the object used by Mongoose to define the expected structure of a document in a given collection. Note that this structure is purely indicative and the developer needs to manually specify the validation [9].

First, this code imports the Mongoose library (Line 1). Then a schema is created through the `mongoose.Schema()` call (Line 3). This method takes as an argument the structure of the schema, in the form of a JavaScript object. Each object property corresponds to a field inside a document and the associated value describes related information, including its type.

To be used, a `Schema` needs to be compiled into a `Model`. A `mongoose.model()` call creates a `Model` (Line 12) which is later exported (Line 14).

Listing 1.9: Mongoose query example

```
1  Author = require("./authors.js");
2
3  // ...
4  tolkien = new Author({
5      surname: "Tolkien",
6      givenNames: ["John", "Ronald", "Reuel"],
7      birthPlace: {
8          country: "Orange Free State",
9          city: "Bloemfontein"}});
10
11 await tolkien.save();
12
13 // ...
14 tolkien = await Author.find({surname: "Tolkien"});
```

[8] https://mongoosejs.com/

[9] The interested reader can learn more about it here : https://mongoosejs.com/docs/validation.html

Listing 1.9 shows an example usage of the model exported in Listing 1.8. The model is imported using the `require` function (Line 1). In Mongoose, the instance of a model is a `Document`. As for the official MongoDB library, Mongoose offers multiple ways to query and create Documents. The `Author` constructor call illustrates the creation (Lines 4-9), the document content being passed as argument of the call as an object. Then, the `.save()` method call performs the actual asynchronous document insertion in the collection (Line 11).

A query is used to retrieve the document (Line 14). It uses the `.find` method call and as for the MongoDB native driver, its first argument is the object filter specifying the desired constraints. Here, it will select each document with the field `surname` having *Tolkien* as a value, retrieving at least the document we have just inserted.

1.2 Static Analysis

In their book "*Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*", Erder and Purat define static program analysis as "*the analysis of computer software that is performed without actually executing programs*" [46]. Although numerous techniques can fall under the qualification of static analysis, we will describe only the ones used in the context of this work.

Dataflow analysis is a static analysis technique that evaluates the value of a given variable by navigating through the instructions that access or modify this variable and analyzing how it propagates through the program. Li et al. conducted a Systematic Literature Review (SLR) on static analysis of Android Apps and highlighted five fundamental static analysis techniques [82], among those stood the taint analysis, which is a subset of dataflow analysis. The taint analysis differs from other dataflow techniques as it tracks whether a tainted object, the source, flows to a defined program point, the sink. Use cases include SQL injection detection, where a function taking user input may directly flow into a function executing SQL code without being sanitized, leading a to security vulnerability.

Static analysis with CodeQL

CodeQL [63] is a semantic code search and analysis tool developed by Semmle, a company acquired by GitHub in November 2019. This tool also features advanced static analysis techniques such as vulnerability tracking or taint tracking. It was made free for open-source and academic research shortly after. It supports multiple programming languages such as Java, JavaScript, C or Python and can apply dataflow and taint analysis techniques. While initially being aimed at security analysis and vulnerability detection, it can also be used as a standard static analyzer.

This tool models a project codebase as a relational database which can be later queried by a dedicated query language, namely QL [10]. In this database, tables correspond to elements of the syntax tree, such as method call or variable declarator. QL being object-oriented, each table is a class which has its dedicated predicates and attributes which can be inherited by its children.

We present here an illustrative example to showcase CodeQL.

[10] https://codeql.github.com/docs/ql-language-reference/

Listing 1.10: CodeQL query example

```
1 import javascript
2 from MethodCallExpr methodCall
3 where methodCall.getMethodName() = "insertOne"
4 select methodCall, "This is a call to an insertOne method"
```

Listing 1.10 is an example query retrieving every call to an *insertOne* method.

First, the `import` clause indicates the language package too use, which is JavaScript. Then, the `from` clause specifies the AST element to query, which is method calls here. The `where` clause defines the constraints to apply on the element. Here, the only constraint is that the name of the method is "*insertOne*". Finally, the `select` clause specifies the output element alongside a message.

If we were to run this query on the code snippet illustrated in Listing 1.6, it would give us the location of the insert method (Line10) alongside the message *This is a call to a insertOne method.*

To also track a Mongoose method to insert documents, we would need also to detect invocations of the Mongoose *save* method.

A naive approach would be to extend the constraints in the `where` clause so that it includes also save as a method name, as showcased in Listing 1.11.

Listing 1.11: CodeQL query example including save method call

```
1 import javascript
2 from MethodCallExpr methodCall
3 where methodCall.getMethodName() = "insertOne" or methodCall.getMethodName
      () = "save"
4 select methodCall, "This is a call to an insertOne or save method"
```

However, the semantics of the methods we are trying to access are different. For MongoDB, the inserting document is passed as parameter of the insertOne function. For Mongoose, the inserted document is first defined then saved.

Consequently, we can introduce classes to regroup different program fragments under the same abstraction.

Listing 1.12: CodeQL class example

```
1  abstract class InsertQuery extends MethodCallExpr {}
2
3  class InsertOneQuery extends InsertQuery {
4      InsertOneQuery() {
5          this.getMethodName() = "insertOne"
6      }
7  }
8  class SaveQuery extends InsertQuery {
9      SaveQuery() {
10         this.getMethodName() = "save"
11     }
12 }
```

First, we define an abstract class to regroup the different insert methods (Line1). The `extends` indicates the base class used, in this case `MethodCallExpr`. Then, we extend that class for each of the tracked query (Line3 and 8). Inside each of these classes, we define the constraints specific to the class in the characteristic predicate (Lines 4-5 and 9-10) as in a `where` clause. Now, we can make a query strictly equivalent to Listing 1.11 but a shorter, as showcased in Listing 1.13.

Listing 1.13: CodeQL class query example

```
from InsertQuery insertQuery
select insertQuery, "This query performs an insertion"
```

Now, we could also want to extract the inserted document. Now that we have a separate class for each method name, we can define a predicate in each class which extracts the inserted document.

Listing 1.14: CodeQL class example with the inserted documents

```
abstract class InsertQuery extends MethodCallExpr {
    abstract ObjectExpr getInsertedDocument()
}

class InsertOneQuery extends InsertQuery {
    InsertOneQuery() {
        this.getCalleeName() = "insertOne"
    }

    override ObjectExpr getInsertedDocument() {
        result = this.getArgument(0)
    }
}
class SaveQuery extends InsertQuery {
    SaveQuery() {
        this.
        getCalleeName() = "save"
    }

    override ObjectExpr getInsertedDocument() {
        result = any(NewExpr docCreation | docCreation.flow().(DataFlow::
            NewNode).getASuccessor+().asExpr() = this.getReceiver() |
            docCreation.getArgument(0))
    }
}
```

Listing 1.14 shows an example on how to extract the inserted document from the two different methods. First, we have to declare the method in the abstract class (Line 2). Concerning *insertOne*, as the document lies in the first function parameter, we can extract this parameter (Line 7). For the sake of the explanation, we assume here that the method don't use a variable reference here. On the contrary of *save*, Mongoose first requires to initialize a Model with a Document to then save it. As such, we need to perform a DataFlow analysis up until the document definition (Line 21). To do that, we first use an aggregation to define a temporary element, a NewExpr which references an expression that instantiates an Object in JavaScript [11]. Then, we access the corresponding dataflow element with the `flow` method and navigate through its successors with the `getASuccessor` method transitively, thanks to the + symbol on it. Finally, we try to unify one of its successors with the receiver of the save method call (in the case of Listing 1.9 it would be the variable reference `tolkien`) which is the object corresponding to the document.

[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new

Listing 1.15: CodeQL class query example

```
1  from InsertQuery insertQuery
2  select insertQuery.getInsertedDocument(), "This is an inserted document"
```

Listing 1.15 shows how to query this class. Notice that the selected elements are the documents, not the method calls themselves.

CodeQL also offers dedicated API's for taint tracking analysis. We explain this feature with an example using the Mongoose driver.

Listing 1.16: Mongoose multiple Schemas example

```
1   const mongoose = require("mongoose")
2
3   const citySchema = new mongoose.Schema({
4       name: String,
5       zipCode: Number,
6       population: Number,
7       area: Number
8   })
9
10  const countrySchema = new mongoose.Schema({
11      name: String,
12      cities: [citySchema],
13      languages: [String],
14      currencyName: String
15  })
16
17  export const countryModel = mongoose.model('Country', countrySchema)
```

Listing 1.16 shows the creation and export of the `Country` model. We can observe that its corresponding schema embeds the `citySchema` as an array under the `cities` property. Indeed, this corresponds to a subdocument, as explained in Section 1.1.1. However, only the `countrySchema` is used to build the model.

Consequently, we cannot use an approach that would simply track calls to the `Schema` method of the `mongoose` driver; we actually need to track that the schema is used in the `Model` constructor.

Listing 1.17: CodeQL taint analysis example

```
1   class SchemaToModelConfiguration extends TaintTracking::Configuration {
2       SchemaToModelConfiguration() {
3           this="SchemaToModelConfiguration"
4       }
5       override predicate isSource(DataFlow::Node node) {
6           exists(CallExpr ce | ce = node.asExpr() and ce.getCalleeName() = "
                Schema")
7       }
8       override predicate isSink(DataFlow::Node node) {
9           exists(CallExpr ce | ce = node.asExpr() and ce.getCalleeName() = "
                model")
10      }
11
12  }
13
14  from SchemaToModelConfiguration config, CallExpr schemaCall, CallExpr
        modelCall
15  where config.hasFlow(schemaCall.flow(), modelCall.flow())
```

```
16 select schemaCall, "This schema is used in a model constructor"
```

Listing 1.17 shows an example on how to perform such a task. First, it creates a class to configure the dataflow (Line 1). The class consists of 2 predicates: `isSource` and `isSink`, describing respectively the taint object to start the dataflow analysis and the sink object, being the dataflow destination (Line 5 and 8). Those two are defined as being function call with the callee name *Schema* and *model* (Lines 6 and 9).

To use this configuration, this program declares three variables: `config` being the configuration that has just been defined, `schemaCall` and `modelCall` for the source and the sink of the taint tracking. As a constraint, we use the predicate `.hasFlow` to check whether there is a dataflow between the source and the sink. Finally, we simply select the source element as it flows through the sink.

1.3 Concluding remarks

In this chapter, we embarked on a concise exploration of the NoSQL database ecosystem, highlighting the four predominant families of DataBase Management Systems (DBMS). Our focus then narrowed to MongoDB, a document-oriented database of which we introduced the core concepts and features. We then dissected the two most popular MongoDB drivers in Node.js systems, being the MongoDB native Node driver and Mongoose, an ODM. The journey through MongoDB's landscape set the stage for understanding how developers interactions with the drivers can be scrutinized through static analysis. There, we introduced static analysis and focused on dataflow analysis. In that context, we introduced CodeQL as the backbone of our static analysis process. We then introduced each of its features with a motivating example to illustrate its capabilities.

CHAPTER 2

STATE OF THE ART

2.1 Database access extraction

In this section we discuss the works related to our study. First we inspect static analysis in JavaScript and then we look at reverse engineering for MongoDB. We conclude by outlining the research gaps.

2.1.1 JavaScript static analysis

Many researchers tackled the challenges of analyzing JavaScript through static approaches. Sun and Ryu recently presented a summary of such approaches, their challenges, and recent trends [129]. Among six literature topics, static analysis is the most prevalent when analyzing JavaScript systems, especially to detect security vulnerabilities and type-related errors. For future works, authors pointed out the topic *soundiness*, which consists in approximating some language features to achieve better scalability than a completely sound analysis while being sufficiently sound.

Andreasen and Møller presented TAJS, a "*whole-program dataflow analyzer for JavaScript*" [18] programmed in Java, based on the 3rd edition of the language specification while having partial models for the 5th edition. By implementing selective context sensitivity, constant propagation, and branch pruning, they are able to leverage the inherent complexity of JavaScript programs using jQuery. The evaluation consisted of the analysis of 154 programs from the jQuery tutorial, from which the analysis was successful 86 times. They report a substantial improvement in terms of type inference and call graph construction in such programs.

Later, Kashyap *et al.* presented JSAI, a JavaScript abstract interpreter formally specified implemented in Scala, supporting the 3rd edition of the language specification. [72] Their tool context, path and heap sensitivity are user-configurable.

JSAI design is broken down into three main components: a JavaScript intermediate representation with its concrete semantics, abstract semantics yielding sub and configurable analysis and configurable abstract domains for num and string. Authors evaluated their tool on existing benchmarks from V8 and SunSpider plus frameworks that they included. JSAI performance was evaluated on its ability to track types based on prediction error. They reported a roughly similar precision and execution time than the TAJS tool and did not find a "*silver bullet*" configuration.

Among the machine learning models approach tackling type prediction in JavaScript, it is worth noting NL2Type presented by Malik, Patra and Pradel [87]. NL2type exploits natural language information in source code, such as comments, function names, and parameter names as source information to predict type signatures of functions. They train their model using the JSDOC information about a function parameters and returns type. The tool can predict the following types: `string`, `number`, `boolean`, `array`, `object`, `function`, `integer`, `element`, `observable` and `mixed`. In their evaluation, authors included a dataset holding 600,000 data points and achieved different amount of precision and recall depending on the number of suggestions considered, yet still obtaining 84.1% precision and 78.9% when taking the first suggestion, 95.5% and 89.6% when taking into account the first five suggestions.

2.1.2 MongoDB reverse engineering

There are a few studies in the realm of MongoDB applications reverse engineering.

Many approaches extract models from JSON document databases [28, 51, 22, 11]. Among those, we can present the work by Baazizi *et al.* who extracted attribute types from JSON documents datasets using a Map-Reduce approach [22], using Spark [1]. Their approach is actually a two-step process, consisting in the (attribute) individual type inference and a type reduction. The first step, individual type inference, is an association of an attribute value to its corresponding type. The second, type reduction, tries to merge all documents types from the first step. They implemented a parametric approach to balance between precision and brevity, from producing every possible document type to a more compact type description to adjust corresponding to specific requirements.

Mior proposed an approach to optimize schema in NoSQL Databases [93]. To use it, they propose multiple modelings to compare query execution time with a set of query for a given schema. The workload modeling consists of an SQL-like syntax, their physical design one considers only wide-column databases and cost modelling approach using the types of entities and indices, the estimated number of entity types and the estimated cardinality of each field.

Störl, Klettke and Scherzinger introduced way to handle schema evolution and data migration in a NoSQL environment. [128] Among the techniques to capture the NoSQL Schema, they introduced the *Reverse Engineering from Code* one, which relies on "*more involved code analysis*".

In this context, the closest work to us was done by Meurice and Cleve, who implemented an approach to extract the database schema of MongoDB applications written in Java.[92] They manually identified database access methods from the

[1] https://spark.apache.org/

MongoDB Java driver API document and parse the Java code to retrieve calls to those methods. To infer the database schema they considered inserted document objects, search criteria, collection names and usage of retrieved documents. They also applied their method to analyze the evolution of Java systems. To evaluate their approach, they applied it to the backend services of a radio system and could inspect the schema evolution.

2.1.3 Summary and research gaps

TAJS and JSAI, the two main JavaScript static analysis research tools, are unable to parse modern JavaScript, highlighting a gap in the current research literature. Type prediction machine learning model only takes into account generic JavaScript type and not more fine-grained ones such as a Mongoose `Model` or MongoDB `collection`. Also, MongoDB reverse engineering techniques 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. In Chapter 3, we analyze JavaScript through a maintained tool, CodeQL, to stay up-to-date with current JavaScript specification and tackle the challenges of the dynamic language through multiple heuristics.

2.2 Code smell

In 1999, Martin Fowler *et al.* released their famous book "*Refactoring: Improving the Design of Existing Code*". There, he and Beck introduced the notion of code smell as a guide to initiate code refactoring processes. He defined a code smell as "*a surface indication that usually corresponds to a deeper problem in the system*" and listed 22 for developers to look in OOP programs. Since then,many researchers have extended Fowler *et al.* initial list with catalogs or taxonomies for object-oriented code smells: Mika Mäntylä *et al.* proposed a taxonomy to class those 22 smells into 7 categories [88]. Also, it underwent a developer survey to determine the correlation between the different smells. Later, they investigated their effect on software evolvability and developers' perception about them [89]. Also, Raúl Marticorena extended this taxonomy by applying four new dimensions to each code smell: *granularity* which designates the component in which the code smell can be detected, *intra* and *inheritance* indicating if knowledge of other components or inheritance is necessary for the detection and *acc* showing the access level among components.

Many researchers continued to enrich the taxonomy and investigated their effect on software systems. Foutse Khomh *et al.* investigated the impact of code smells on change proneness. By detecting code smells in two large-scale open source systems, they were able to show a negative impact of a code smell on its class change proneness, being more likely to be changed than other classes. [75] Steffen Olbrich *et al.* also drew similar conclusions while enquiring of code smells density impact over a software evolution. Indeed, he also showed that infected classes have a higher change frequency. [106] Aiko Yamashita and Leon Moonen explored the relation between code smells and maintainability [137]. They relied on experts to evaluate the maintainability and then other maintainers who would apply change requests

to those systems. Using the explanations given by the developers, they were able to identify partial code smells. Consequently, they could find which code smell has a negative impact on maintainability.

2.2.1 Smells/Antipatterns in Database Communication

There are some examples of community members releasing antipatterns symptoms or smells catalogs related to data communication.

First, Bill Karwin published a book named "*SQL Antipatterns: Avoiding the Pitfalls of Database Programming*" where he described 24 antipatterns spread across 4 categories: logical database design, physical database design, queries, and application development [71]. For each antipattern, he describes the root cause, how to detect the antipattern and its related fix.

Also, Red Gate Software Ltd. published a booklet of 240 SQL code smells [47]. They briefly describe the code smells and organise them into 8 categories depending on the smell localization: Database design, Table design, Data types, expressions, query syntax, naming, routines and security.

Boyuan Chen *et al.* investigated database antipatterns of Object-Relational Mapping (ORM) from Laravel, a PHP web framework [31]. They build an antipatterns catalog using a literature study and performance monitoring. Their catalog features 17 antipatterns organized into 5 categories: unnecessary computation, inefficient data accessing, unnecessary data retrieval, inefficient computation and inefficient rendering. Each antipattern is linked to a fixing refactoring.

Shudi Shao *et al.* also conducted an SLR to gather SQL database-access performance antipatterns from the research literature [125]. They ultimately found 24 antipatterns, each described by a root cause sentence shortly explaining the antipattern and a single-sentence fix strategy. Then, they exploited bug reports information to infer the presence of these antipatterns in open-source systems.

Junwen Yang *et al.* identified ORM performance antipatterns in Ruby on Rails applications [139]. They exploited commit-fixing information from project bug tracking systems to obtain their antipatterns. The antipatterns were then grouped depending on their inefficiency cause: ORM API misuse or Database design problem. Later, they enriched their catalog with Powerstation, an IDE plugin for RubyMiner, which suggests fixes to the developers [140].

Yan *et al.* proposed a static analysis method to pinpoint ORM performance issues in Ruby on Rails applications [138]. Their antipatterns gathering process consisted of profiling the application database and recognizing recurring performance-dropping patterns. Through this process, they identified 8 antipatterns, which they categorized into 4 groups based on the profiling results that led to their detection.

2.2.2 MongoDB and NoSQL

Despite their popularity, we could not find many published works describing smelly interactions or antipatterns symptoms in NoSQL databases. Paola Gomez *et al.* compared the performance of six MongoDB databases containing the same data set but with different data structuring choices [57]. They proposed 6 different schemas with different levels of embedding ranging from a single collection where a document

contains every other possible entity to a completely normalized Schema. Then, they tested 6 different queries on each schema and could draw different conclusions on the schema structure effect such as heavily nested data is associated with poor performance or that separating the data complexes the access code.

Abdullahi Abubakar Imam *et al.* suggested 23 guidelines for designing a document-oriented database [62]. They obtained those guidelines by a variety of sources, from expert consultations, official guidelines or perceived complexity of certain design choices. The embeddings were grouped into 3 different categories corresponding to MongoDB modelling techniques: embedding, referencing and bitbucketing. Given guidelines were prioritized separately on read and on write operation based on expert consultation. Such guidelines can also be found in the MongoDB Applied Design Patterns book by Copeland *et al.* [40].

By performing a security analysis on a MongoDB system, Jitender Kumar and Varsha Garg were able to highlight various security issues in MongoDB server [77]. Indeed, it does not support natively encryption/decryption nor does it have a password. The authors then recommended that developers use additional mechanism of key generation and encryption for security concerns.

Divya Mahajan *et al.* measured the energy consumption variation resulting of a query optimization in various NoSQL systems They were able to pinpoint a reduction of 99% in the consumed energy when having an indexed query vs not indexed one. They could also identify an energy consumption reduction of 75% when using an aggregation pipeline vs using a map-reduce function.

2.2.3 Summary and research gaps

While the concept of code smells serves as a valuable metaphor for guiding software development and maintenance, there appears to be a lack of empirical research specifically addressing MongoDB antipatterns or code smells. MongoDB published a series of articles in 2022 under the title "*Schemas Design Antipatterns*". Given MongoDB's popularity, numerous sources within grey literature discuss MongoDB antipatterns or code smells. Our objective is to bridge this gap by proposing a catalog of MongoDB Code Smells/Antipatterns. This will be achieved through the execution of a Multi-vocal Literature Mapping (MLM) to systematically capture the available grey literature. That contribution is described in Chapter 4.

2.3 Database communication smell detection

In this section, we present existing approaches and tools to detect code smells or antipatterns in database communication.

Huib van den Brink, Rob van der Leek and Joost Visser proposed an approach to assess the quality of embedded SQL query for PL/SQL, COBOL, Visual Basic and Java systems [132]. They reconstruct the queries from their string value and measure several quality metrics on them. Measured metrics include but does not limit to: number of queries, number of operations or number of variables. The evaluation is made on two systems, a PL/SQL one and COBOL one.

Aghajani *et al.* investigated the presence and impact of 12 Linguistic Antipatterns (LA) in API over their usage in open-source systems [13]. LA designate poor naming

or documentation practices in every software artifacts (*e.g.,* code or documentation). Their methodology involved tracking the usage of 1.6M Java methods (including 33k LA-affected) from 1,642 releases of 76 libraries, across 16k projects, as well as the bug fixes from the files containing those method calls. They found that when an API method call is introduced in a system for the first time, "*the likelihood of introducing a bug is 29% higher if the API is affected by a LA*".

Csaba Nagy *et al.* proposed SQLInspect to detect the SQL antipatterns defined by Karwin [71] within Java programs [101] and an Eclipse extension to detect SQL antipatterns, code smells and metrics from Java programs [101]. It extracts embedded queries used with the JDBC [2] driver and a MySQL or Apache Impala engine. The antipatterns come from the book "*SQL Antiptterns: Avoiding the Pitfalls of Database Programing*" by Bill Karwin. They could evaluate their tool on four large-scale open-source systems. Later, Biruk Asmare Muse *et al.* conducted a larger-scale empirical study on the prevalence, impact, and evolution of four smells detected by SQLInspect in 150 open-source systems [100]. Their findings indicated that two of the four smells were prevalent in the projects. By also tracking generic Java code smells with DECOR [94], they were able to establish the co-occurrence of SQL code smells with these generic code smells. However, they found only a weak association between any specific SQL code smell and the generic code smells. Additionally, they did not find a significant association between SQL code smells and software bugs, and they observed that 80% of the SQL code smells persisted throughout the complete considered evolution of the systems.

Tse Hsun Chen *et al.* proposed an approach to detect antipatterns in Java applications using Object-Relational Mapping (ORM)[32]. In their work, they combine a static and a dynamic approach: they use control-flow and data-flow analysis to extract database accesses and rely on a logger, "log4jdbc" [3] to construct the query. Their subsequent analysis look for two different antipatterns and the evaluation on two open source systems could link system performance drop with the presence of those antipatterns.

Prashanth Dintyala *et al.* presented SQLCHECK, a toolchain dedicated to database applications detection and fixing [43]. The tracked antipatterns were obtained from 2 grey literature sources (one wiki post and a StackExchange post on the most common SQL antipatterns), Bill Karwin's book [71] and a research paper. Their detection algorithm takes SQL queries as an input and perform intra- and inter-query analysis to build context to better refine the detection context. SQLCHECK was able to find 63,058 antipatterns instances across raw SQL statements from 1406 open source projects. They also proposed a ranking system of the detected antipatterns based on multiple metrics gathered on the running system database, which is also used to suggest fixes.

In a follow-up research, Tse Hsun Chen *et al.* explored the performance implications of redundant data accesses [33]. They broke down redundant data accesses into four different performance antipatterns occurring in transactions. To detect the antipatterns, they rely on static analysis to find database accesses and use aspect-oriented programming to obtain the query. They were able to evaluate

[2] https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/
[3] https://github.com/arthurblake/log4jdbc

their approach on two open-source systems and measure the performance impact relative to the antipatterns as well as automatically flagging antipatterns.

Boyuan Chen *et al.* investigated database antipatterns of Object-Relational Mapping (ORM) from Laravel, a PHP web framework [31]. They obtained their antipatterns list from a literature study which are a mix of ORM misuses and redundant/inefficient database accesses. ORM misuses are detected through static analysis while they rely on dynamic analysis to extract the SQL queries and detect the other antipatterns. They used dynamic analysis as a mean to uncover new antipatterns and evaluated their approach on an industrial database-centric web application but did not report the detected instances.

Junwen Yang *et al.* proposed a static analysis approach to identify ORM performance antipatterns in Ruby on Rails applications [139]. The antipatterns were obtained from a systematical study of performance-fixing commits. The static analysis implementation was not discussed in the paper but they were able to find at least one instance of every antipattern among their 12 selected projects. Later, they reused this approach for PowerStation, an IDE plugin for RubyMiner, which statically detects ORM inefficiencies and suggests fixes to developers [140].

Yan *et al.* proposed a static analysis method to pinpoint and address ORM issues in Ruby on Rails applications [138]. Their static analysis relies on a custom static analyzer building the Action Flow Graph, which combines dataflow and control-flow information originating from ORM action. With that, they were able to build SQL queries from the ORM actions and profiling query inefficiencies. Then they applied their approaches to 27 web applications and detected many ORM actions that could further be optimized.

Yingjun Lyu *et al.* developed SAND, a static analysis tool to detect SQL antipatterns in Anrdoid mobile apps [83]. Its antipattern detection system relies on an abstraction over the process of extracting a SQL statement from a database interaction. From there, SAND builds the possible SQL statements and relationships between tables. They evaluated their approach on 1,000 Android projects.

MongoDB released Atlas, a cloud database service for database deployment and management [98]. While initially made to assist developers in their modelling, querying and deployment tasks, it is also able to detect antipatterns. Indeed, MongoDB described several Schema antipatterns in a dedicated blog series. Their tool is able to detect those antipatterns directly from the data.

2.3.1 Summary and gaps

Table 2.1 shows the different approaches and tools to detect database communication smells, alongside its target **data model** (*e.g.,* relational, document-oriented), **analysis scope** which designates which software artifact is analyzed, **target language** indicates the programming language, **detection output** the result of the analysis and **Analysis**, that tells whether the approach performs static or dynamic analysis, or a mix of the two.

We can see that most of the presented approaches extract embedded SQL queries in various ways (from ORM method calls, query string concatenation, database accesses, logger) and then detect SQL queries antipatterns. Indeed, except for Van den Brink [132] that produces metrics, they all look to detect antipatterns.

Table 2.1: Database communication smell detection approach

Approach	Data model	Analysis Scope	Target Language
Van den Brink *et al.* [132]	relational	Query	PL/SQL, COBOL, Visual Basic, Java
SQLInspect [101]	relational	Query	Java
Tse Hsun Chen *et al.* (2014) [32]	relational	Query	Java
Tse Hsun Chen *et al.* (2016) [33]	relational	Query	Java
Boyuan Chen *et al.* [31]	relational	Query, ORM	PHP
SQLCHECK [43]	relational	Query	/
Junwen Yang *et al.* [139]	relational	Query, ORM	Ruby
Yang *et al.* [139]	relational	ORM	Ruby
PowerStation [140]	relational	ORM	Ruby
Yan *et al.* [138]	relational	ORM	Ruby
SAND [83]	relational	Query	Android
Atlas [98]	document-oriented	Data	/

Approach	Detection output	Analysis
Van den Brink *et al.* [132]	metrics	static
SQLInspect [101]	code smells, antipatterns, metrics	static
Tse Hsun Chen *et al.* (2014) [32]	antipatterns	hybrid
Tse Hsun Chen *et al.* (2016) [33]	antipatterns	hybrid
Boyuan Chen *et al.* [31]	antipatterns	hybrid
SQLCHECK [43]	antipatterns	static
Junwen Yang *et al.* [139]	antipatterns	static
Yang *et al.* [139]	antipatterns	static
PowerStation [140]	antipatterns	static
Yan *et al.* [138]	antipatterns	static
SAND [83]	antipatterns	static
Atlas [98]	antipatterns	dynamic

Additionally, none of them are designed for JavaScript. However, except for the industrial tool Atlas, there is no approach detecting database communication smell for document-oriented or NoSQL data model.

Concerning the Atlas tool, it requires access to the data, which can be sensitive or not always available.

With our tool SMEAGOL defined in Chapter 5, we aim to fill that gap by proposing a tool able to detect code smells and antipatterns for document-oriented data models, for JavaScript systems as it is the most frequently used language alongside this data model.

2.4 Concluding remarks

In this chapter, we surveyed the state of the art for the three main contributions of this thesis.

Concerning database accesses extraction, while JavaScript static analysis has seen many researches, none of the published tool is able to parse up-to-date Java-

Script syntax. Also, in the field of MongoDB Reverse engineering, the most part relies on the data, while it is not always available in open-source systems. The only approach relying on the source code uses Java. Consequently, our approach extracts database accesses from the source in JavaScript applications, using CodeQL.

Regarding the code smells/antipatterns catalog, we can see many approaches for relational databases in various languages. However, none actually exists for MongoDB. Additionally, we could see that code smells, even in the database communication field, affect the software system.

Finally, we could observe many approaches tackling the detection of code smells or antipatterns via the reconstruction of embedded SQL query and for Java program. However, this approach cannot be applied to document-oriented datastores models. We then chose to base our approach on a static detection tool for JavaScript programs.

Part I

Contributions

CHAPTER 3

DATABASE ACCESSES EXTRACTION

3.1 Introduction

NoSQL 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 [91]. 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 [123, 15].

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 [128]. For example, researchers study automatic schema extraction [11], schema generation [58], optimization [93], and schema suggestions [62]. 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.

Our approach 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 [25]. According to a recent empirical study [25], 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 [72, 18] usually struggle to handle the highly dynamic features of the lan-

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

guage [129], and approaches with type inference [65], dataflow [84], or call graphs [49] 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 evaluate the accuracy of our approach on an oracle of 307 open-source projects and reaches 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 [92], help their developers propagate schema changes [12], or identify antipatterns [102]. We demonstrate potential use cases on two interesting case studies where we assess the evolution of open-source systems.

3.2 Approach

Fig. 3.1 presents an overview of the approach. It consists of 3 main phases : **Compilation**, **Database accesses name extraction** and **Candidates filtering**. First, we analyze a JavaScript project with CodeQL and compile it into an internal representation, a CodeQL database. In the second phase, we extract every method name corresponding to a database access in MongoDB or Mongoose. Lastly, we collect every method call matching those names and use filtering heuristics on those calls to improve the precision. The main point of those heuristics is to avoid name conflicts 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).

Compilation

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 dataflow 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.

We also introduced CodeQL in the technical background Section 1.2. CodeQL takes as input the source files of the project. It parses them and builds an inter-

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

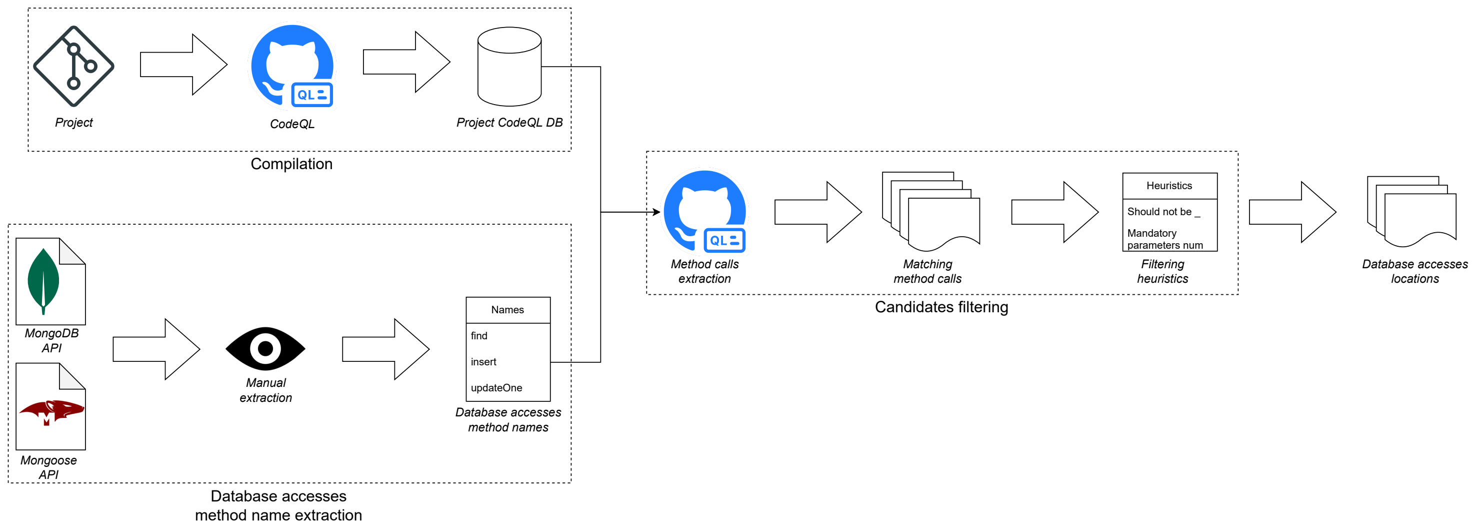


Figure 3.1: Approach overview

nal database. This internal database can then be queried by CodeQL queries as described in Section 1.2.

Database accesses name extraction

We look for method calls invoking a database access method of the MongoDB Node Driver or Mongoose. One author, Boris Cherry, gathered method signatures from reference guides of MongoDB Node Driver 3.6 and Mongoose 5.12.8 which were the up-to-date version at the time of the research. From this list, we select 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 collect 179 distinct method names, 81 methods from MongoDB Node Driver and 105 methods from Mongoose. Table 3.1 presents the database access methods from MongoDB and Table 3.2 from Mongoose. We also feature the **Receiver Type** to differentiate methods having a similar name. They are also available in the dedicated **online appendix** [7].

Table 3.1: MongoDB Node driver database access methods

Method Name	Receiver Type
command	Admin
validateCollection	Admin
bulkExecute	BulkOperationBase
execute	BulkOperationBase
find	BulkOperationBase
insert	BulkOperationBase
raw	BulkOperationBase
aggregate	Collection
bulkWrite	Collection
count	Collection
countDocuments	Collection
createIndex	Collection
createIndexes	Collection
deleteMany	Collection
deleteOne	Collection
distinct	Collection
drop	Collection
dropAllIndexes	Collection
dropIndex	Collection
dropIndexes	Collection
ensureIndex	Collection
estimatedDocumentCount	Collection
find	Collection
findAndModify	Collection

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

findAndRemove	Collection
findOne	Collection
findOneAndDelete	Collection
findOneAndReplace	Collection
findOneAndUpdate	Collection
geoHaystackSearch	Collection
group	Collection
indexes	Collection
indexExists	Collection
indexInformation	Collection
initializeOrderedBulkOp	Collection
initializeUnorderedBulkOp	Collection
insert	Collection
insertMany	Collection
insertOne	Collection
isCapped	Collection
listIndexes	Collection
mapReduce	Collection
reIndex	Collection
remove	Collection
rename	Collection
replaceOne	Collection
save	Collection
update	Collection
updateMany	Collection
updateOne	Collection
aggregate	Db
collection	Db
collections	Db
command	Db
createCollection	Db
createIndex	Db
dropCollection	Db
dropDatabase	Db
ensureIndex	Db
eval	Db
executeDbAdminCommand	Db
listCollections	Db
renameCollection	Db
delete	FindOperators
deleteOne	FindOperators
remove	FindOperators
removeOne	FindOperators
replaceOne	FindOperators
update	FindOperators
updateOne	FindOperators
upsert	FindOperators

collection	GridStore
withSession	MongoClient
execute	OrderedBulkOperation
find	OrderedBulkOperation
insert	OrderedBulkOperation
raw	OrderedBulkOperation
execute	UnorderedBulkOperation
find	UnorderedBulkOperation
insert	UnorderedBulkOperation
raw	UnorderedBulkOperation

Table 3.2: Mongoose database access methods

Method Name	**Receiver Type**
create	Model
createCollection	Model
hydrate	Model
deleteMany	Model
deleteOne	Model
remove	Model
deleteMany	Query
deleteOne	Query
remove	Query
execPopulate	Document
save	Document
insertMany	Model
save	Model
populate	Query
bulkWrite	Model
populate	Document
$getAllSubdocs	Document
get	Document
aggregate	Model
count	Model
countDocuments	Model
distinct	Model
exists	Model
find	Model
findById	Model
findOne	Model
geoSearch	Model
mapReduce	Model
populate	Model
where	Model
$where	Query
all	Query

and	Query
box	Query
centerSphere	Query
circle	Query
count	Query
countDocuments	Query
distinct	Query
elemMatch	Query
equals	Query
exists	Query
find	Query
findOne	Query
get	Query
near	Query
nor	Query
or	Query
orFall	Query
polygon	Query
where	Query
count	Aggregate
findByIdAndDelete	Model
findByIdAndRemove	Model
findOneAndDelete	Model
findOneAndRemove	Model
findOneAndDelete	Query
findOneAndRemove	Query
findByIdAndUpdate	Model
findOneAndReplace	Model
findOneAndUpdate	Model
findOneAndReplace	Query
findOneAndUpdate	Query
overwrite	Document
replaceOne	Document
update	Document
updateOne	Document
replaceOne	Model
update	Model
updateMany	Model
updateOne	Model
replaceOne	Query
update	Query
updateMany	Query
updateOne	Query
addFields	Aggregate
deleteModel	Mongoose
model	Mongoose
add	Schema

pick	Schema
remove	Schema
collection	Connection
createCollection	Connection
deleteModel	Connection
dropCollection	Connection
dropDatabase	Connection
model	Connection
$ignore	Document
set	Document
cleanIndexes	Model
discriminator	Model
ensureIndexes	Model
estimatedDocumentCount	Model
validate	Model
estimatedDocumentCount	Query
append	Aggregate
immutable	SchemaType
index	SchemaType
required	SchemaType
sparse	SchemaType
text	SchemaType
unique	SchemaType
createIndexes	Model

Candidates filtering

CodeQL provides type inference to approximate types for JavaScript expressions and variables, a mechanism that attempts to deduce the types associated with expressions and variables within JavaScript code. It also constructs a call graph, a representation of the invocation relationship between functions, highlighting the caller and callee dynamics. Despite these sophisticated features, the inherent dynamism of JavaScript introduces a layer of complexity, compromising these analytical constructs. This limitation notably affects the accurate identification of MongoDB database accesses and we cannot rely on them.

However, we still need to precisely identify the MongoDB and Mongoose database accesses which are method calls. To do that, we implemented an initial approach where we systematically gather each method call which identifier aligns with our predefined database access methods list (see Table 3.1 and Table 3.2). Nevertheless, this initial approach can introduce noise as it may inadvertently identify method calls from unrelated APIs. To address this challenge and refine the precision of our database access identification process, we have formulated and applied a set of 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(...)`[8]

[8]https://mongoosejs.com/docs/api/model.html#model_Model.create

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*. Similar collisions can generate significant noise in our approach. Thus, we apply heuristics to avoid such cases and look for potentially colliding methods in the MongoDB and Mongoose API documentation.

Many of the heuristics check the method call receiver immediate type or name. For example, the receiver object of `Author.find()` in Listing 1.9 is `Author`. 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.[9]

In the next section, we present our heuristics.

Filtering heuristics

H1 *Find call's single parameter should not be a string literal.* As explained in the Background Section 1.1.1, the method indicated for retrieving documents in both MongoDB and Mongoose is respectively `collection.find(...)` [10] and `Model.find(...)` [11]. However, jQuery, an "*HTML document traversal and manipulation*" [4] library has a `.find(selector)` method [12] which uses a string parameter to match tree elements. We find an illustration of the necessity of this heuristic from a repository showcasing applications using Azure DevOps Project, *microsoft/devops-project-samples*.[13] A summary of this snippet is showcased in Listing 3.1.This project's maintainers used the jQuery find methods in conjunction with the string argument *:input* to select elements that can take user input from a web page DOM. Hence, we define this heuristic to avoid collision with this method and other well-spread libraries.

Listing 3.1: Collision with a `jQuery.find` invocation

```
185  $(selector).find(":input").filter("[data-val=true]").each(function () {
186      $jQval.unobtrusive.parseElement(this, true);
187      });
```

To filter out those specific cases, we defined a CodeQL predicate reported in Listing 3.2:

Listing 3.2: Predicate to filter `jQuery.find` invocations

```
1  predicate findAxioms(MethodCallExpr mce) {
2      ( mce.getMethodName() = "find" and mce.getNumArgument() = 1 ) implies
           not (mce.getAnArgument() instanceof StringLiteral)
3  }
```

[9] https://mongoosejs.com/docs/api.html#model_Model.findOne
[10] https://www.mongodb.com/docs/drivers/node/current/usage-examples/find/
[11] https://mongoosejs.com/docs/api/model.html#Model.find()
[12] https://api.jquery.com/find/
[13] https://github.com/microsoft/devops-project-samples/blob/d17f1a31d36eed9c813d3526c41c7e73019fbf9a/python/bottle/webapp/Application/static/scripts/jquery.validate.unobtrusive.js#L185

This predicate consists of a logical implication that a MongoDB database access should respect, which is indicated by the keyword `implies`. The first part unifies with a method having 1) the name find and 2) strictly one argument. The second part is to make sure that this specific argument is not a string literal. Consequently, this predicate holds when a method call bearing *find* as an identifier and one parameter does not use a literal string.

H2 *The receiver should not be a `Promise` object.* Promise is the dedicated mechanism for parallelism in JavaScript. Mozilla web docs [8] give this definition concerning those: "*The `Promise` object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.*". [14] Being an object, a promise also comes with its dedicated methods including `Promise.all()`, that has a collision with one of our tracked method identifiers `Query.all()`. While the `Query.all()` method is used to introduce a `$all` operator as chained method [15], that checks if a document field value is an array that contains the given elements, the `Promise.all()` [16] method takes as parameter a Promises iterable and return a single Promise and fail if one of the promise fail.

An example of this collision can be found in the open source project *eshengsky/iBlog.* [17] There, we can see that the maintainers use `Promise.all()` to perform two actual Mongoose queries, similarly to a transaction in relational modelling, except that `Promise.all` does not have a failure propagation feature, thus exposing the database consistency. Listing 3.3 shows this specific code.

Listing 3.3: Collision with a `Promise.all` call

```
432 const data = await Promise.all([
433     Guestbook.find(query, {}, options).exec(),
434     Guestbook.countDocuments(query).exec()
435     ]);
```

The dedicated CodeQL code, a predicate, is shown in Listing 3.4.

Listing 3.4: Predicate to filter `Promise.all` invocations

```
1 predicate allAxioms(MethodCallExpr mce) {
2     mce.getMethodName() = "all" implies not(mce.getReceiver().toString().
          matches("Promise"))
3 }
```

As for the predicate dedicated to the first heuristic, it is made of a logical implication. The first part makes sure the method name is *all* and the second that the *receiver* identifier of this method call is not "*Promise*", the basic Promise object identifier. To summarize, this predicate holds when a method call having the identifier *all* is not made on a receiver which identifier is *Promise*.

[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[15] https://mongoosejs.com/docs/api/query.html#Query.prototype.all()
[16] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all
[17] https://github.com/eshengsky/iBlog/blob/03be36c425cad5bf3b2312c58fff4cedbfb7a45c/server/proxy/admin.ts#L432

H3 *A `create(...)` invocation's receiver should not be an `Object` or a subclass of Object that is not a Document.* On top of the insertion example showcased in Section 1.1.1, Mongoose has an alternative way to insert a document into a collection: `Model.create(docs)`, which lets you insert one or more documents into a given model. However, JavaScript has a standard method to create a new object from an existing one, also bearing the identifier `Object.create(...)`.

We find an example of such a collision in the open source system, *jones2000/HQChart* a project to build charts in many programming languages. [18] Listing 3.5 depicts the relevant code line. Here, we can see that the project developers use the method `Object.create` to propagate the `null object` through the variable `watchers` and the property `_computedWatchers`.

Listing 3.5: Collision with a `Object.create` call

```
3048  var watchers = vm._computedWatchers = Object.create(null);
```

In order to avoid considering such cases as Mongoose document insertion, we build the query featured in Listing 3.6.

Listing 3.6: Predicate to filter `Object.create` invocations

```
1  predicate createAxioms(MethodCallExpr mce) {
2      ( mce.getMethodName() = "create" and mce.getNumArgument() = 1 ) implies
           not (  mce.getReceiver().toString().matches("Object") )
3  }
```

Likewise the two preceding heuristics, this heuristic uses an implication. The first implication proposition holds when the candidate method call name is *create* and has one parameter while the second holds if the identifier of the method call's receiver identifier does not correspond to `Object`. To recapitulate this predicate, it holds when the method call name is *create* and has one parameter but its receiver is not *Object*.

H4 *The receiver should not be "_".* Lodash[19] is a JavaScript utility library that facilitates the manipulation of data structures, including arrays, objects, and strings, as well as the implementation of utilities not present in the JavaScript Standard. To achieve this, the library offers around 300 functions, many of which having a collision with our set of method names such as `find` [20] or `remove` [21]. A display of such a collision can be observed in the open source project *Strider-CD/strider* which is a CI/CD server. Listing 3.7 highlights this collision.

Listing 3.7: Collision with a _.find call

```
57  function add(project, email, accessLevel, inviter, done) {
58      User.findOne({ email: email }, function (err, user) {
59          if (err) {
60          return done(err);
```

[18] https://github.com/jones2000/HQChart/blob/e71b787d40a348225ea746163f5dfc7f5c032d63/webhqchart.demo/demo/content/js/vue.js#L3048

[19] https://lodash.com/

[20] https://lodash.com/docs/4.17.15#find

[21] https://lodash.com/docs/4.17.15#remove

```
61          }
62
63          if (user) {
64          const p = _.find(user.projects, function (p) {
65              return p.name === project.toLowerCase();
66          });
67              if (p) {
68                  return done('user already a collaborator', true);
69  //...
70  }
```

In this example [22], this project performs a Mongoose `findOne` query on the `User` Model. In this query callback, which is a function to be applied on this query result, maintainers use the Lodash `_.find` method to search through the `user.projects` array for a project that matches the project given as argument, complementary to the Mongoose query. Here, we can observe that the identifier "`_`" was used to reference the Lodash library.

To fix those cases, we decided to abuse this standard usage of Lodash. Its standard notation uses the dash ("`_`") as an identifier. Thus, this heuristic avoids potential collisions with the frequently used Lodash library, as illustrated by Listing 3.8.

Listing 3.8: Predicate to filter Lodash invocations

```
1  predicate loadashAxioms(MethodCallExpr mce) {
2      not mce.getReceiver().toString().matches("_")
3  }
```

This predicate is composed of a single statement making sure that it holds when the textual representation of a given method call receiver does not match the string "`_`". As such, this heuristic makes sure that a candidate database access receiver is never the identifier "`_`", which is enough to discard standard Lodash usages.

H5 *The receiver should not be `JQuery.events`, `JQuery` or match the regular expression "`\$\(.*\)`".* As explained in the first heuristic, we need to make sure we do not inadvertently take the `.find` method from jQuery as a database access. However, we soon discover that this method is not the only one from jQuery with which we could have a collision. Such methods include `.add` [23] which creates a jQuery object, having a collision with the `Schema.add` [24] method which adds a new field to a Mongoose Schema or `.remove` [25] used to remove elements from the DOM colliding with `collection.remove` [26] being used to remove every document from a collection.

An illustration of such a collision can be found in the *Countly/countly-server* open source project that is a product analytics platform for user actions. [27] Listing 3.9 reveals such a collision.

[22] https://github.com/Strider-CD/strider/blob/bb3ec49415b9a48b497191ccb3b04e598859c9a2/apps/strider/lib/routes/collaborators/api.js#L64
[23] https://api.jquery.com/add/
[24] https://mongoosejs.com/docs/api/schema.html#Schema.prototype.add()
[25] https://api.jquery.com/get/
[26] https://www.mongodb.com/docs/manual/reference/method/db.collection.remove/
[27] https://github.com/Countly/countly-server/blob/c8ef4c131a3ac0cdab7f38d83e922ba9d163df28/frontend/express/public/javascripts/countly/countly.template.js#L1067

Listing 3.9: Collision with a `jQuery.remove` call

```
1067  $(".menu-title").remove();
```

In this code snippet, we can see a jQuery query on the DOM elements from the class “*menu-title*” and then a deletion of those elements through the `.remove` method. Notice that it is not possible to differentiate this method call from a `collection.remove` one as they possess the same number of parameters, we then need a way to filter out the jQuery method call using the receiver.

To solve it, we again decide to exploit the standard usage of jQuery. Indeed, jQuery main function identifier is “$” or when using methods from the specific “*jQuery.*” or “*jQuery.events*”, we can observe that those prefixes are likely to be used as receiver for the method call. Our heuristic predicate implements this in Listing 3.10

Listing 3.10: Predicate to filter jQuery methods invocations

```
1  predicate jqueryAxioms(MethodCallExpr mce) {
2      not ( mce.getReceiver().toString().matches("$(%)") or mce.getReceiver()
           .toString().matches("jQuery%") )
3  }
```

This predicate is true if the textual representation of the receiver from the method call does not match either “*$*(%)” (which is equivalent to the regular expression “`\$\(.*\)`”) or either “*jQuery*” or either “*jQuery.events*”. As a result, this heuristic ensures that jQuery method calls that does not deviate from their conventional application are not mistakenly included.

H6 *The file should transitively import MongoDB or Mongoose.* Sometimes, we cannot filter out a candidate method call based on its file as the receiver would pass the above heuristics or respect the required parameters. However, we can still estimate if a given file can have access to the library MongoDB or Mongoose. We consider a file having an access to MongoDB or Mongoose if it belongs to import chain originating from a file importing MongoDB or Mongoose. A file belongs to an import chain if the file

(i) contains a MongoDB or Mongoose import, or

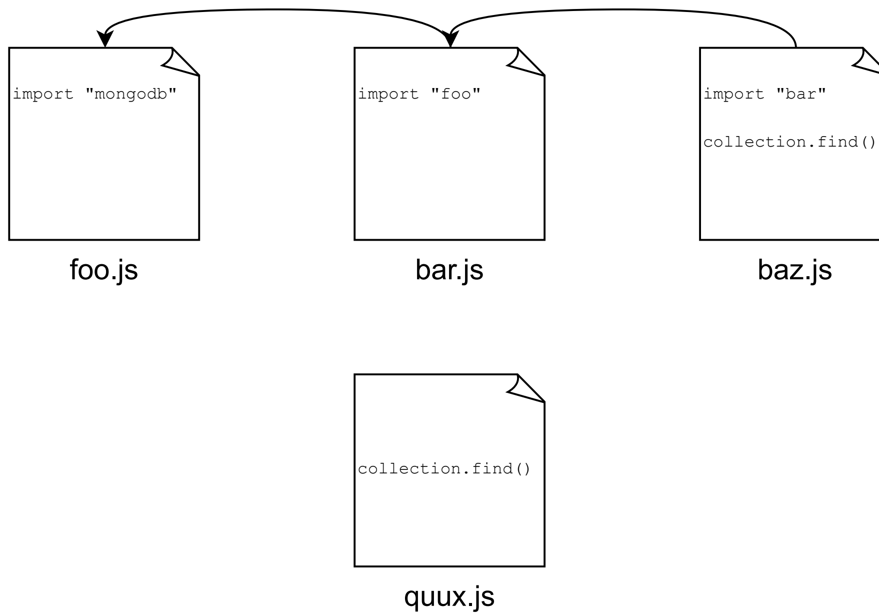
(ii) imports such a file containing a MongoDB or Mongoose import, or

(iii) transitively imports such a file

Fig. 3.2 demonstrates an import chain.

First, the file “*foo.js*” imports the MongoDB native Node.js driver. Trivially, this file has access to the MongoDB driver and falls under our definition first case. Then, “*bar.js*” imports “*foo.js*”. As this last file has access to MongoDB, importing it give “*bar.js*” access to this driver and corresponds to the second case of our definition. Finally, the “*baz.js*” file imports “*bar.js*”. As it imports a file having access to MongoDB, ‘*baz.js*” also gains access to the MongoDB library and the `collection.find()` could be a database access. This represents the third case of our definition. However, the `collection.find()` from the “*quux.js*” file cannot be a MongoDB database

Figure 3.2: Example of import chain



access as it does not belong to an import chain and we can filter it out from the candidates.

To implement the verification of this heuristic, we implemented the two predicates exemplified in Listing 3.11.

Listing 3.11: Predicates to represent import chain

```
1   predicate moduleImportChain(Module m1, Module m2) {
2       m1.getAnImportedModule() = m2 or
3       moduleImportChain(m1.getAnImportedModule(), m2)
4   }
5   predicate importAxioms(MethodCallExpr mce) {
6       exists( Module mongoModule, Module mceModule |
7           mceModule.getFile() = mce.getFile() and
8           ( mongoModule.getAnImport().getImportedPath().getValue().matches("
                mongodb") or mongoModule.getAnImport().getImportedPath().
                getValue().matches("mongoose") ) and
9           moduleImportChain(mceModule, mongoModule)
10
11      )
12      or
13      exists( Import i | ( i.getImportedPath().getValue().matches("mongodb")
            or i.getImportedPath().getValue().matches("mongoose") ) and i.
            getFile() = mce.getFile() )
14  }
```

The first predicate, `moduleImportChain`, describes the import chain logic. Two modules, which represent JavaScript file that can import or export from other modules, are in the same import chain if one imports the other (as verified by the line 2) or if one recursively import the other, (as per line 3).

The second one, `importAxioms`, implements our definition logic. The case corresponding to (i) can be found on line 13 where we use the "`exists`" formula to check whether there is an import statement which imports either MongoDB or Mongoose and is located in the same file as the candidate database accesses. The implementation of the cases (ii) and (iii) can be found on lines 6-9. There, it checks if the module corresponding to the candidate database access (line 7) is in an import chain originating from a module (line 8) which imports either MongoDB or Mongoose.

Hence, this heuristic filters out method calls that cannot access the MongoDB or Mongoose driver via the evaluation of an import chain.

H7 *Mandatory parameters should match.* It is also possible to filter out a possible database access without knowledge of the methods with which we could have collisions. As an illustration, we introduce this code snippet [28] from *dan-divy/spruce*, an open-source social networking platform. The snippet is illustrated in Listing 3.12.

Listing 3.12: Collision with a `.raw` call

```
76    var random_id = guid.raw();
```

In our manual investigation of the MongoDB and Mongoose API, we identify the `BulkOperationBase.raw` method as a database access. While the bulk allows to execute multiple write operations serially, the raw method adds an operation to an existing bulk. Therefore, it requires at least one parameter, the operation to add. Nevertheless, the call in Listing 3.12 does not yield any parameter, making it impossible for it to be a `BulkOperationBase.raw` call and we can then remove it from the set of candidates.

To exploit this, by exploiting the Mongoose and MongoDB API we generate a CodeQL predicate for each database access method identifier where we verify whether a candidate has at least the number of required parameters and does not exceed the total possible number of parameters. Listing 3.13 show an example for the `BulkOperationBase.raw` method.

Listing 3.13: Predicate to check the number of arguments for a get call

```
1   predicate rawGenAxioms(MethodCallExpr mce){ (mce.getMethodName() = "get")
        implies (mce.getNumArgument() >= 1 and mce.getNumArgument() <=2)}
```

Here, the predicate is composed of an implication. The first part holds if the method call name matches with the candidate being verified, which is "*get*" and the second part evaluates to true if the number of arguments is between 1 and 2. By generating such predicate for every database access method, we are able to check the number of arguments for every identifier.

Thus, with this heuristic, we are able to filter out candidates whose number of arguments do not match with the corresponding database access method.

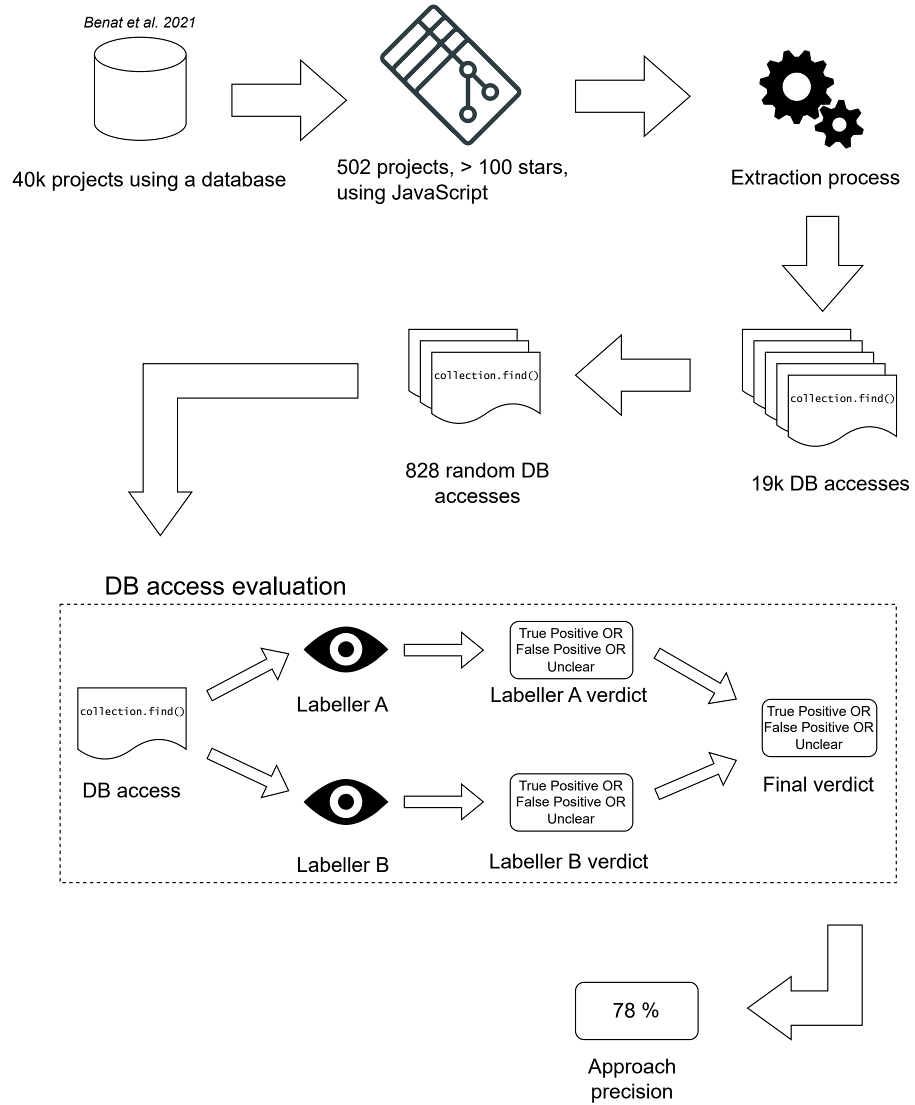


Figure 3.3: Evaluation setup

3.3 Evaluation

Fig. 3.3 shows the evaluation setup to assess the precision of the approach. First we rely on the dataset of Benats *et al.* [25] who built a relational database containing 40 thousand open source systems using a database, from multiple Libraries.io [5] dumps. We queried this dataset to obtain projects having at least 100 stars, using JavaScript and declaring MongoDB as a dependency. We ended up having 502 projects, Table 3.3 breaks down the projects statistics in terms of Github stars, files, lines of code and database accesses. Then, for each project, we cloned it and applied our approach to retrieve database accesses, giving us a total of 19 thousand database accesses.

Table 3.3: Project statistics

	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

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 labellers who checked them manually. The people who participated in the labelling process had at least a master degree in computer science, they were: Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy and Boris Cherry. The aim was to see if the call indeed belonged to the detected method signature of the identified database access. The labelling process was hosted on a Google Sheet. Each labeller was given a dedicated sheet to evaluate database accesses.

Fig. 3.4 is a screenshot of an actual labeller's evaluation sheet. We can observe multiple columns. **ID** specifies the database access internal id, **access_link** gives a github hyperlink leading to the database access location at a given commit [29], **method_name** and **receiver** are textual representation of respectively the tracked method name and the receiver on which it was called, **possible_qualified_names** and **documentation_url** identified the possible retrieved method(s) and pointed out to the relevant documentation artifact while **driver_name** precised the involved driver (mongodb or mongoose), additional location information is provided by **start_line** and **start_column**, **true_positive** is the column where labellers indicated whether the retrieved database access was adequately retrieved by filling it with "*TRUE*" or "*FALSE*" and **remarks** was used to indicate reflections on complex cases.

[28] https://github.com/dan-divy/spruce/blob/40f2d8cdb04f50cf1096f28d376d6a4528f62e4f/routes/settings.js#L76

[29] Example: https://github.com/bitpay/bitcore/blob/cf5074cbcbef27148751139d1831c69c386f1240/packages/bitcore-wallet-service/src/lib/storage.ts#L1652

ID	access_link	method_name	receiver	possible_qualified_names	documentation_url	driver_name	file_path	project_name	start_line	start_column	true_positive	remarks
820	https://github.com/bit	find	this.db ... EME	Model.find / Query.find / BulkOperationBas	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m http://mongodb.github.io/node-m http://mongodb.github.io/node-m http://mongodb.github.io/node-m	mongodb / mongoos	packages/bitcore	bitpay/bitcore	1652	5	TRUE	
269	https://github.com/Str	updateOne	Job	Document.updateOne / Model.updateOne	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m http://mongodb.github.io/node-m	mongodb / mongoos	apps/strider/lib/in	Strider-CD/strider	149	15	TRUE	
35	https://github.com/Op	and	aModelListQue	Query.and	https://mongoosejs.com/docs/ap	mongoose	libs/modelQuery.	OpenUserJS/OpenUser	252	5	TRUE	Cannot be 100%
940	https://github.com/no	findOne	this.da ... 'dkim	Model.findOne / Query.findOne / Collection	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m	mongodb / mongoos	lib/dkim-handler.j	nodemailer/wildduck	231	9	TRUE	Cannot be 100%
556	https://github.com/Ea	model	mongoose	Mongoose.model / Connection.model	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap	mongoose	models/wTrack.js	EasyERP/EasyERP_op	70	5	TRUE	
633	https://github.com/crc	save	this	Document.save / Model.save / Collection.s	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m	mongodb / mongoos	lib/models/user.ts	crowi/crowi	304	5	TRUE	
195	https://github.com/rsc	findOneAndUpdate	this._collection	Model.findOneAndUpdate / Query.findOne	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m	mongodb / mongoos	lib/agenda/save-j	rschmukler/agenda	131	28	TRUE	Function signatu
866	https://github.com/no	findOne	db.data ... sag	Model.findOne / Query.findOne / Collection	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m	mongodb / mongoos	lib/api/messages	nodemailer/wildduck	1193	37	TRUE	Cannot be 100%
358	https://github.com/no	collection	db.database	Connection.collection / GridStore.collection	https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m http://mongodb.github.io/node-m	mongodb / mongoos	lib/api/messages	nodemailer/wildduck	755	50	TRUE	Cannot be 100%
72	https://github.com/Ea	findByIdAndRemove	Opportunity	Model.findByIdAndRemove	https://mongoosejs.com/docs/ap	mongoose	handlers/opportu	EasyERP/EasyERP_op	1242	13	TRUE	
351	https://github.com/rai	collection	this.conn	Connection.collection / GridStore.collection	https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m http://mongodb.github.io/node-m	mongodb / mongoos	app/backends/m	rain1017/memdb	85	12	TRUE	
49	https://github.com/ab	command	program	Admin.command / Db.command	http://mongodb.github.io/node-m http://mongodb.github.io/node-m	mongodb	src/index.js	abecms/abecms	156	0	FALSE	Collision with ht
879	https://github.com/Op	findOne	Script	Model.findOne / Query.findOne / Collection	https://mongoosejs.com/docs/ap https://mongoosejs.com/docs/ap http://mongodb.github.io/node-m	mongodb / mongoos	controllers/issue.	OpenUserJS/OpenUser	413	3	TRUE	
481	https://github.com/bui	create	this	Model.create	https://mongoosejs.com/docs/ap	mongoose	book/8-end/serve	builderbook/builderbook	227	14	TRUE	

Figure 3.4: Labeller sheet screenshot

Another sheet was also used to aggregate every results and detect eventual conflict between labellers.

We did two labelling rounds. Before labeling the 818 methods, we had a trial round on a sample set of 150 methods unrelated to the evaluation set. 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 labellers tagged whether it was a false or true positive and added remarks where needed. It was also possible to assign an unclear tag for cases too complex. 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 each other's tag. After this labelling, 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.

In the following, we provide an example for each type of final result: *True positive*, *False positive* and *Unclear*.

True positive Our first example takes place in the project *Trustroots/trustroots*, a website to share plans and hosts for travelers [30]. Listing 3.14 presents the context surrounding this database access.

Listing 3.14: True positive example from *Trustroots/trustroots*

```
206 /* save the newly created message */
207 message.save(err => {
208     if (err != null) {
209     console.log(err);
210     } else {
211     /* Message was saved successfully */
```

Here, the labellers had to determine whether this call was a `Document.save` or `Model.save` one, which is the default way to insert a Document in Mongoose. The call is located on line 207 and is invoked upon the variable `message`. By navigating through the code, we reach this variable instantiation on line 172 which is made thanks to the `Message` constructor call that itself comes from a Mongoose `.model` call. The Mongoose API informs us that this method returns the constructor for the model specified by the parameter. Consequently the `Message` constructor relates to a model and its instantiation `message` corresponds to a Document. Then, it is indeed a `Document.save` call which qualifies as database access and is labelled as a true positive.

False positive The second example occurs in the *EasyERP/EasyERP_open_source* project, an open source ERP [31]. We present the related code snippet in Listing 3.15.

[30] https://github.com/Trustroots/trustroots/blob/ce9e1bafbcce9adcfe069ddbc010bf53d317f091/bin/fillTestData/Messages.js#L207

[31] https://github.com/EasyERP/EasyERP_open_source/blob/693c2531b8bcb110c4ab8ef649895a5f47c3aed8/handlers/channels.js#L816

Listing 3.15: False positive example from *EasyERP/EasyERP_open_source*

```
816 IntegrationsService.findOneAndUpdate(_query, model, {
817     upsert: true,
818     dbName: db,
819     new   : true
820 },
```

The possible methods for this candidate database access are Mongoose `Model.findOneAndUpdate` and `Document.findOneAndUpdate`, and MongoDB `collection.findOneAndUpdate`. Even though the method call signature seems off as there is a parameter called `model` and the model should be the receiver of this call and not a parameter, we need to learn more about the actual receiver of the call, `IntegrationsService` to conclude on its validity. On line 17, we can see that `IntegrationsService` comes from another module from the project. After inspection of this module [32], we can see that the `findOneAndUpdate` method is in fact a user-defined function acting as a wrapper around the `Model.findOneAndUpdate` call. In conclusion, this candidate was labelled as false positive because we decided not to include wrappers even though it commits a database access.

Unclear The third and last example happens in the *davellanedam/node-express-mongodb-jwt-rest-api-skeleton*, a JavaScript REST API skeleton [33]. You can find the candidate database access in Listing 3.16.

Listing 3.16: Unclear example from *davellanedam/node-express-mongodb-jwt-rest-api-skeleton*

```
13 user.save((err, result) => {
14     if (err) {
15         return reject(buildErrObject(422, err.message))
16     }
17     if (result) {
18         return resolve(true)
19     }
20     })
```

Similar to the true positive example, this call could originate from either `Document.save` or `Model.save`. This receiver, `user` reveals to be a parameter of the function where the call is located. Because of the dynamic nature of JavaScript, any value could be passed as this parameter thus making it impossible to conclude on the possible values `user`. We then conclude that this case is “Unclear” as it depends on the usage of the function `checkLoginAttemptsAndBlockExpires` to verify if that parameter is a model or a document.

As a side note, we did survey the usage of this function and found that this project’s maintainers use it with a model as parameter. [34]

[32] ttps://github.com/EasyERP/EasyERP_open_source/blob/693c2531b8bcb110c4ab8ef649895a5f47c3aed8/services/integration.js#L216

[33] https://github.com/davellanedam/node-express-mongodb-jwt-rest-api-skeleton/blob/28a9aee8341ed64c6bf3b731ea3f9885f3745946/app/controllers/auth/helpers/checkLoginAttemptsAndBlockExpires.js#L13

[34] https://github.com/davellanedam/node-express-mongodb-jwt-rest-api-skeleton/blob/28a9aee8341ed64c6bf3b731ea3f9885f3745946/app/controllers/auth/login.js#L25

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%.

Concerning the duration of the labelling process, it varied between labellers but the whole process of labelling then resolving conflicts took about 3 weeks. The dataset of the oracle is available in the *dedicated appendix*.[35]

3.4 Case Studies

We demonstrate the approach on case studies of two open-source systems. We rank the projects in our benchmark according to the amount of database accesses and look 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.

3.4.1 Bitcore

Bitcore[36] 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 select 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. 3.5 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[37] 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*."

We also analyze 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.

[35] https://github.com/bocherry/saner22-online-appendix
[36] https://github.com/bitpay/bitcore
[37] https://github.com/bitpay/bitcore/commit/d08ea9

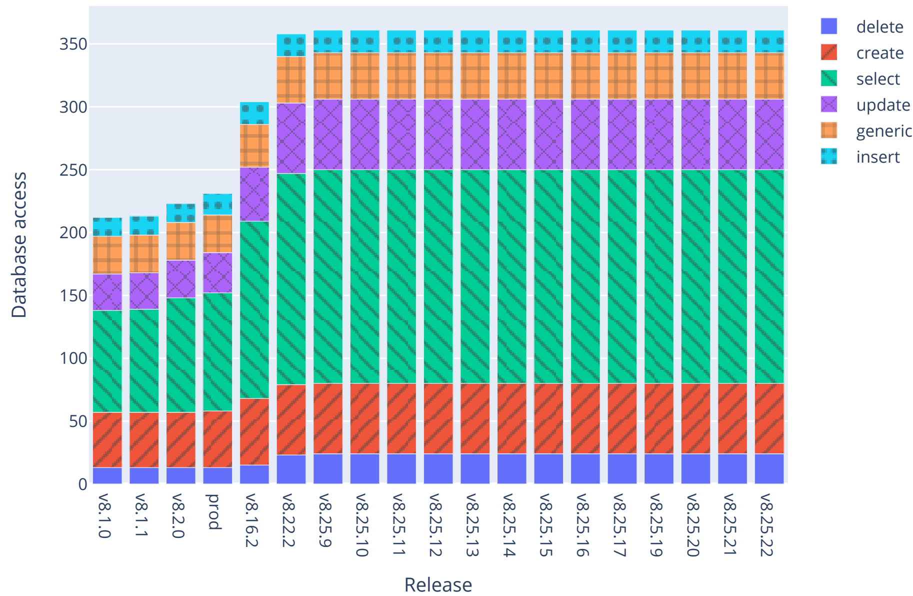


Figure 3.5: Evolution of database accesses in Bitcore

3.4.2 Overleaf

Overleaf[38] is a well-known "*online real-time collaborative LaTeX editor.*" We select 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. 3.6 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.[39] 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 had 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. It could also be used to assist developers in their maintenance tasks by identifying the locations of every database accesses according to their type (*e.g.*, all the write operations) and help them navigate through the code.

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

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

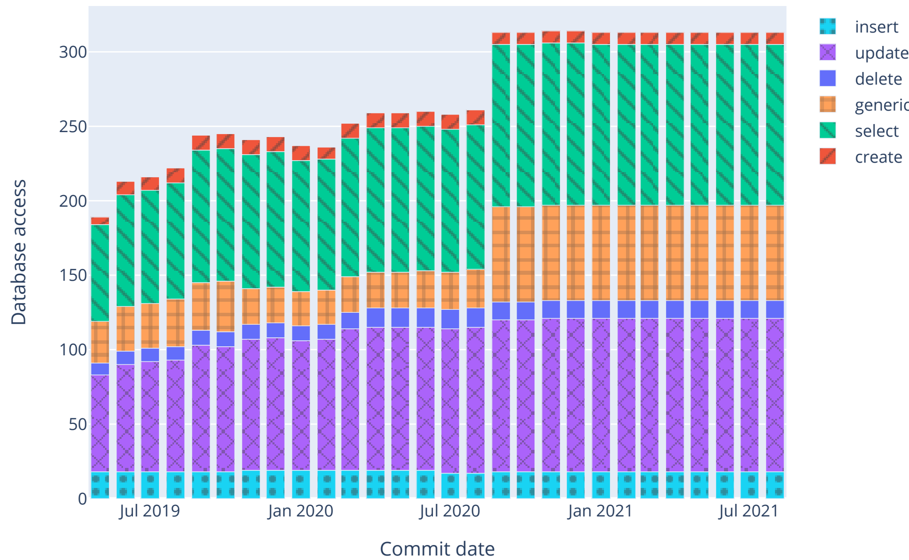


Figure 3.6: Evolution of database accesses in Overleaf

3.5 Threats to Validity

In this section, we discuss threats to validity of our approach.

Construct validity Our approach relies on CodeQL [63], a static analysis tool to retrieve database accesses which are method calls. By nature, this might miss highly dynamic structure especially in a dynamic language like JavaScript.

Also, we depend on filtering heuristics to exclude irrelevant method calls. Our sixth heuristic (import chain) might be too strict as JavaScript does not require checking parameter types, one can define a method without importing a MongoDB library, thus being removed by our heuristic.

We only consider database accesses from the MongoDB Node Driver and Mongoose. If an application uses another library or a web service for the database access, we will miss it.

Internal validity In our evaluation we rely on manual code inspection, which exposes us to subjectivity. In an effort to mitigate it, a labeller did not know the conclusion of the other labeller on the candidate database access and they could label a candidate as “Unclear” if they were not sure of their conclusion. We also made sure that each labeller had experience with JavaScript programs and MongoDB.

External validity Our approach is tailored for JavaScript systems using MongoDB or Mongoose. While some heuristics are generalizable to identify method calls such as the sixth (import chain) or seventh (mandatory number of parameters) heuristic, the others are only relevant for our context.

Additionally, the considered projects all come from the dataset from the study of Benats *et al.* [25] which itself originate from Libraries.io. Other projects might have shown different collisions with different libraries.

3.6 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 frequently used. In this context, the approach is particularly useful in mining software repositories, where these technologies, namely JavaScript and MongoDB, are prevalent today.

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.

3.6.1 Roadmap

In this chapter, we presented a static analysis approach to automatically extract MongoDB database accesses locations from the source code. This approach is designed for JavaScript applications accessing a MongoDB database through the Mongoose or MongoDB native node drivers.

The following chapter, Chapter 4, will introduce a MongoDB code smells/antipatterns catalog and reuse this approach in an experimental setup to detect instances in open source systems.

This approach will also be at the core of our code smells/antipatterns detection tool, SMEAGOL, described in Chapter 5.

CHAPTER 4

CODE SMELL CATALOG

4.1 Introduction

NoSQL data stores have become popular backends of database applications [124, 25, 73]. MongoDB is the most popular NoSQL data store, according to DB-Engines Ranking.[1] Unlike relational databases (DBs), MongoDB is a document store that handles data as a series of JSON-like documents, offering attractive features to developers, such as improved flexibility and horizontal scalability. This comes at the cost of some critical features of relational DBs, such as referential integrity [123]. In the worst case, these differences can lead to erroneous constructs, runtime errors, or data loss [92].

Researchers studied approaches to assist MongoDB developers. For example, Kanade *et al.* proposed a normalization and embedding approach to improving DB performance [68]. Mahajan *et al.* evaluated the effectiveness of query optimization on energy efficiency [85]. Zhao *et al.* devised an approach to support the migration of a relational DB to MongoDB [142], and Maity *et al.* investigated this migration scenario in the opposite direction [86]. Security also interests researchers [77, 135, 136], especially the critical NoSQL injection [112, 113, 61, 114].

As aborded in Chapter 2, code smells and antipatterns are useful indicators of poor design or implementation choices. Despite its popularity, we found a lack of research on antipatterns or code smells for MongoDB. However, there are numerous discussions on forums, blogs or books.

To better understand the state-of-the-art and practice in MongoDB code smells, we present a Multivocal Literature Mapping (MLM) study. Similar studies have been recently used in software engineering (*e.g.,* [131, 53, 66, 78, 134, 110, 111, 109]) as a form of systematic literature review when data from multiple sources are considered. An MLM study allows extending the scope of a systematic literature review

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

by including "grey" literature, such as blog posts, white papers, and presentation videos. It is especially useful when there is a vast grey literature written by practitioners, as it can help researchers and practitioners locate and synthesize such a large literature [53, 67, 66].

For this purpose, we queried various search engines for sources discussing MongoDB code smells. We looked for online articles, blog posts, presentation materials, books, forum discussions, and directly queried targeted sources such as the official MongoDB blog site [121]. We ran a total number of 72 queries (6 queries on 12 search engines), resulting in an initial set of 1,498 search results. We manually filtered them following a set of inclusion/exclusion criteria. In the end, we identified 174 sources which we manually reviewed to establish a list of 76 MongoDB code smells.

This chapter provides an overview of the list of smells and the grey literature we found with our MLM approach. Such an overview can be helpful for practitioners and researchers as a summary and "index" of what we know about these smells. It can also serve as a starting point for future research on MongoDB code smells.

4.2 Background

In 2019, Garousi *et al.* published a journal paper entitled "*Guidelines for including grey literature and conducting multivocal literature reviews in software engineering*" [52].

They categorize grey literature (GL) using the knowledge of two factors: expertise (author authority and knowledge) and outlet control (extent to which content is produced, moderated or edited in conformance with explicit and transparent knowledge creation criteria). It ranges from books or government reports to tweets, blogs or email

We already provided an introduction to MongoDB modelling in Section 1.1.1, here we introduce an example containing a smell.

Let us assume a MongoDB database consisting of 2 collections: `countries` and `cities`, as showcased in Fig. 4.1 and Fig. 4.2. Each document from the `countries` collection represents a country. It is composed of multiple fields: `name` denoting the English country denomination, the `currency` textual representation, belonging `continent`, `anthem`, spoken `languages` as an array of string and `area`, which is an embedded document holding two fields: `value` and `unit`. Documents from the `cities` collection denote cities. A typical document includes several fields, `name` with the city English name, `population` with the total estimated capita, `timezone` expressed in string, the `mayor`'s surname and name as well as `country` that references the parent country's `_id`.

Listing 4.1 shows a JavaScript code snippet to list every country with their corresponding cities, using the MongoDB Node Driver. First, it imports and initializes a `MongoClient` from the library (Lines 1–3). It connects to the database (Line 7), then queries the `countries` collection and its related `cities` (Lines 9–17). The query uses the standard MongoDB method to perform a left outer join on two collections with the `aggregate()` function and the `$lookup` operator. Finally, it prints the results (Line 18) and closes the connection (Line 22).

In this query, there is a MongoDB smell: `$lookup` is slow and resource-intensive compared to operations that do not need to combine data from multiple collections.

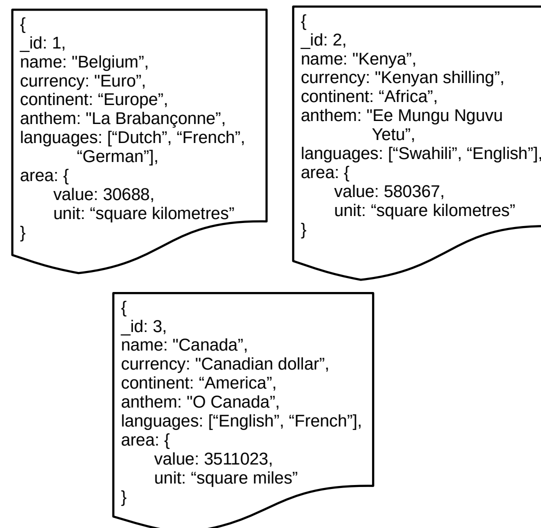


Figure 4.1: Countries collection

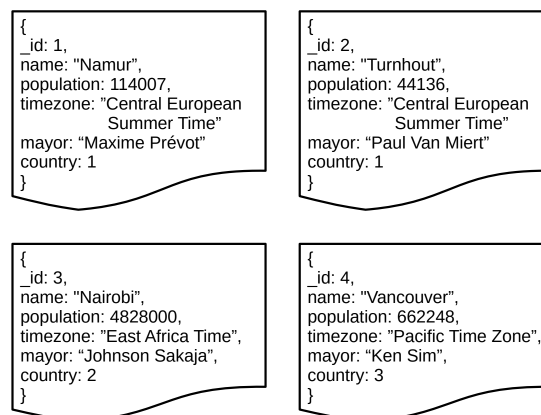


Figure 4.2: Cities collection

The usage of `$lookup` is discouraged in MongoDB, and document embedding is recommended as "*data that is accessed together should be stored together*" [80].

Listing 4.1: Example of *Separating data accessed together* smell

```
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       await db
9           .collection("countries")
10          .aggregate([
11              {
12                  "$lookup": {
13                      "from": "cities",
14                      "localField": "_id",
15                      "foreignField": "country"
16                      "as": "country_cities"
17                  }
18              }
19        ]).forEach(console.log);
20      } catch (err) {
21      console.log(err);
22      } finally {
23      await client.close();
24      }
25  }
26  run().catch(console.dir);
```

An optimized solution embeds cities in their corresponding countries. The query can then be substituted with a simple `find()` method invocation as displayed by Fig. 4.3. The resulting code becomes shorter, requires fewer resources, and runs faster.

Listing 4.2: Fixed access code

```
8   /* ... same as Lines 1-8 */
9   await db.collection("countries").find()
10  .forEach(console.log);
11  /* ... same as Lines 19-25 */
```

4.3 Method

4.3.1 Research Questions

The absence of research on MongoDB smells motivates our research questions:

RQ1 What types of MongoDB smells have been proposed in the community? (**Mapping based on smell types.**)

RQ2 Where are MongoDB smells discussed by the community? (**Mapping based on types.**)

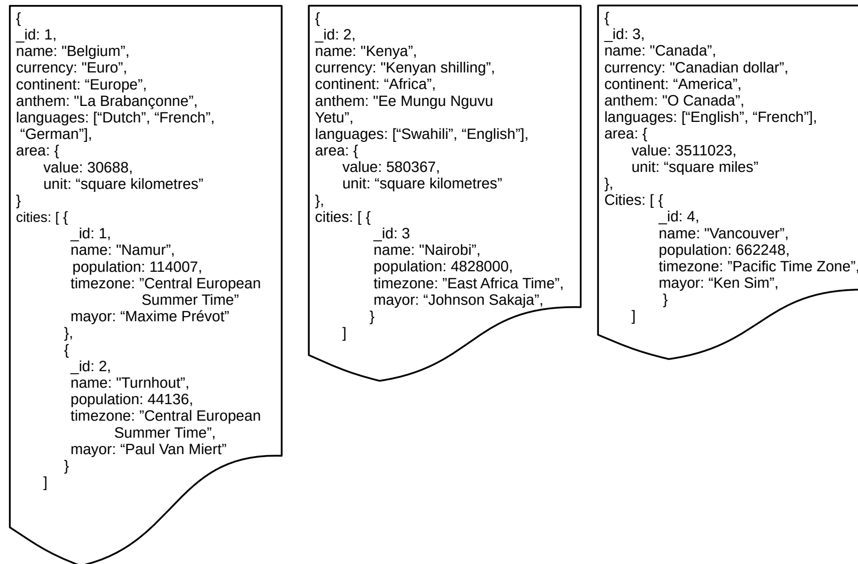


Figure 4.3: Countries collection with embedding

4.3.2 Multivocal Literature Mapping

To answer RQ_1 and RQ_2 we conducted a systematic literature review including *grey* literature, *i.e.*, sources that do not necessarily go through quality control mechanisms (*e.g.*, peer review), such as blog posts, forum messages, and whitepapers. Recent studies have considered similar sources relevant for code smells [53, 133, 66], and the approach has become known as MLM [131, 53].

An MLM aims to classify the body of knowledge in a given area, similar to a systematic review or literature mapping study. MLMs can be extended with follow-up studies. Our goal is to review the literature and classify MongoDB smells. We identify relevant sources, manually examine them to distill candidate smells, then organize them into a catalog.

Fig. 4.4 depicts the main steps of the entire MLM approach.

Source Mining

We mined online sources through search engines of scientific databases (*Google Scholar, Science Direct, IEEE Xplore, ACM Digital Library*), MongoDB forums/blogs (*MongoDB Website, MongoDB Community Forums*), and Google.

To formulate the search string, we first determined a set of potential keywords based on the study's objective. We then conducted trial searches to validate each possible search string as recommended in the guidelines by Kitchenham and Charters [76]. Finally, we used the following search strings:

- MongoDB antipatterns
- MongoDB code smells

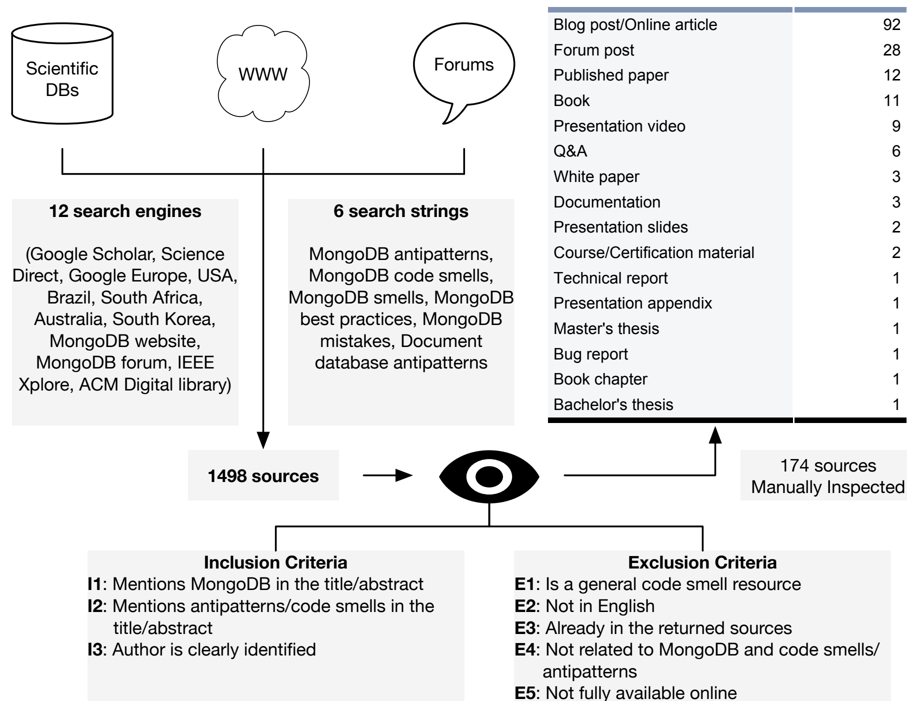


Figure 4.4: Overview of the Multivocal Literature Mapping Approach

- MongoDB smells
- MongoDB best practices
- MongoDB mistakes
- Document database antipatterns

We employed both “Code smells” and “Antipatterns”, as these terms are often used interchangeably by practitioners [130]. We also included “Best practices”, as we found that code smells were often mentioned in conjunction with their violations. Furthermore, we included “Document database antipatterns” as our trial searches discovered relevant matches, but examining its alternatives did not yield further sources.

We ran Google queries multiple times through VPN from five continents (Europe, North America, South America, Oceania, and Asia) to mitigate potential *location bias* [55]. We reviewed the returned results for each search engine and query string, then manually checked each source to determine whether its *content and quality* met our *inclusion and exclusion criteria* (see Fig. 4.4). When it was impossible to decide whether the source should be included based on the title and summary, we opened its link and examined it separately.

As suggested by Garousi *et al.* in their guidelines for conducting multivocal literature reviews [52], we relied on a saturation approach for the *stopping criteria*:

We set a minimum threshold for each query, which was relaxed when needed until the search engine returned no new relevant matches. For example, we looked at the first 20 sources returned by Google in general but extended this limit to 60 for Google Europe, where we performed our first Google searches.

We ran 72 queries (6 searches x 12 engines) and examined 1,498 candidate sources. 154 satisfied our quality, inclusion and exclusion criteria. We kept these sources for manual inspection and added 20 sources through *snowballing*. If a source mentioned one outside our list, we examined that too.

The most relevant source types were the following: *Blog post/Online article* (92), *Forum post* (28), *Scientific paper* (12), *Book/Book chapter* (11), *Presentation video/slides* (9), and *Q&A Post* (6).

Smell Classification

We conducted open coding [127] to identify and classify the smells mentioned in the sources. Such approaches are common in systematic literature reviews and software engineering studies [108, 54, 126, 81] to identify and classify relevant concepts in various sources. Two people, Csaba Nagy and Boris Cherry, independently inspected all 174 sources to collect MongoDB antipatterns/smells according to the following criteria:

- *MongoDB code smells, antipatterns, or bad practices:* Recurring problems and bad practices leading to performance or maintainability issues originating from the database or the database communication.
- *Oppositions to MongoDB's good practices:* Good practices, investigating if ignoring them could lead to performance or maintainability issues.
- *Oppositions between SQL and MongoDB:* Common mistakes that SQL programmers could make when designing/using a NoSQL DB as a SQL database.
- *Transpositions of SQL antipatterns:* Code smells or antipatterns defined for SQL databases that could be extended to NoSQL databases.

We provided a short description for each smell, kept track of its origin, and assessed whether it could be detected from data, schema, or application code.

In total, we collected 242 unique smells (the union of 135 and 140 of each author). The two labelers partially agreed in 72% of the sources, *i.e.,* where both reviewers found at least one common smell for the same source.

We resolved conflicts and organized the smells into categories in the next step. We used *open card sorting*, an established practice for knowledge elicitation and classification [23, 141]. First, the two labelers discussed all 242 smells individually. They merged synonymous smells, excluded too generic ones (*i.e.,* unrelated to source code or MongoDB), and organized them in categories. Two additional authors then revised the final results of this categorization process.

Ultimately, we distilled a catalog of 76 MongoDB smells organized into 11 categories. A final set of 87 sources were related to these smells. The remaining sources did not contain smell or antipattern definitions.

The number of smells also reduced significantly as various sources named smells differently. For example, “Bloated document” had 9 alternatives, *e.g.,* “Large documents”, “crossing 16mb doc size”, or “$project the Elephant”.

Details of this classification process (*e.g.,* sources of smells, merged/excluded smells) are available in the appendix [34].

4.4 RQ_1: MongoDB Code Smell Catalog

Fig. 4.5 presents the catalog of the 76 MongoDB smells resulting from the MLM process, organized into 11 categories. The figure shows the number of sources mentioning each smell in parentheses. We describe each category by presenting examples of smells. Detailed descriptions of all smells, including links to their sources, are available in our online appendix [34].

4.4.1 Aggregation

MongoDB provides *aggregation operations*[2] to process multiple documents in collections. This is an alternative API to the more common `find()` queries. We describe in the following the code smells falling under this category.

Lookup without matching index We introduced the notion of MongoDB index in Section 1.1.1. During the aggregation operation, a lookup is performed on an attribute which has not been indexed. This can result in a performance drop as the whole collection is scanned. Consequently, as there is no index declaration in Listing 4.1 over the `country` field, this aggregation performance may be improved by declaring beforehand an index on that field.

Too many aggregation stages MongoDB does not enforce a maximum number of stages performed in an aggregation. Consequently, one could use as many stages as desired which could severely hinder readability. An aggregation found on the *EasyERP/EasyERP_open_source* repository [3] contains 38 stages, making it harder for the external reader to comprehend the operation performed. The source describing this code smell did not include a strict limit but rather used a subjective term like “*fitting in a page*” [4].

Map-reduce for projection MongoDB offers an API method to perform a map-reduce operation, However, to execute a map-reduce processing the aggregation framework is more effective. In a forum post explicitly asking about common mistakes when using aggregation [5], a developer highlighted the usage map-reduce processing[6] where aggregations would be more efficient.

[2] https://www.mongodb.com/docs/manual/aggregation/

[3] https://github.com/EasyERP/EasyERP_open_source/blob/693c2531b8bcb110c4ab8ef649895a5f47c3aed8/services/jobs.js#L38

[4] https://www.mongodb.com/community/forums/t/what-are-some-of-the-biggest-mistakes-people-make-in-aggregation-pipelines/11803/7

[5] https://www.mongodb.com/community/forums/t/what-are-some-of-the-biggest-mistakes-people-make-in-aggregation-pipelines/11803

[6] https://www.mongodb.com/docs/manual/core/map-reduce/

<u>Aggregation issues</u>: associated with poor query performance.

- Lookup without supporting indexes (1)
- Map-Reduce for projection (3)
- Too many aggregation stages (1)

<u>Design oversights</u>: poor design choices to accelerate design & development, leading to decreased maintainability, bugs, storage waste, and performance issues.

Design oversights in application code
- Immortal cursors (1)
- No dependency injector (1)
- Querying too much data (3)
- Testing only CRUD operations (1)
- Testing queries on the entire ad-hoc big data lake (1)
- The single-person bridge (1)

Design oversights in Data/Schema
- Bloated documents (17)
- Data oriented instead of application oriented (12)
- Flat raw data (2)
- Inconsistent attribute structure (5)
- Multiple schemas in a file (1)
- Repeated immutable data (1)
- Storage of easily calculated values (2)
- Too many collections (13)
- Unbound arrays (24)
- Using a document for "_id" (1)

Design oversights in indexes
- Abusive use of indexes (22)
- Index intersection rather than compound index (1)
- Non-ESR compound indexes (Equality, Sort, Range) (2)
- Prefix index of compound indexes (1)

<u>Performance/Memory issues</u>: misconfigurations or inadequate running environment; associated with poor performance or memory overhead.

- Large read-ahead (2)
- Large WTC (WiredTiger Cache) (1)
- Multiple "mongod" instances (1)
- Running MongoDB in a shared environnement (1)
- Running MongoDB on 32-bit systems (4)
- Unlimited mongos taskExecutor in a container (1)
- Using fast writes (1)
- Using GridFS for small binary data (2)

<u>Sharding issues</u>: poor use of MongoDB partition mechanism, can lead to poor performance, storage waste, and memory overhead.

- Low-cardinality shard key (1)
- Monotonically increasing shard key (3)
- Premature sharding (3)
- Scatter-gather queries (1)
- Unshardable collection (2)
- Working set exceeds memory (7)

<u>Backup issues</u>: increase the risk of data loss and vulnerabilities.

- Manual backups (1)
- No backup budget (1)
- Replicas as backup (3)

<u>Query issues</u>: misuse of the query mechanism; are associated with slow query execution time

- Avoid $Where (1)
- Case-insensitive queries without matching indexes (6)
- Confusing null and undefined (1)
- Large skips for pagination (1)
- Leading wildcard searches on indexed columns (3)
- Negation in queries (2)
- No $elemMatch to match an entire array (1)
- Single update/insert for batches (1)
- Sorted monkeys (1)
- Uncovered queries (5)
- Using $limit without $sort (1)
- Using $map, $reduce and $filter with array fields (1)
- Using limit and skip for pagination (1)

<u>Security issues</u>: inadequate security practices; exposing the database and the whole system to vulnerabilities.

- Forgetting to tie down MongoDB's attack surface (1)
- Improper user credential storage (2)
- No database access control (2)
- No database user policy (3)
- No input sanitizing (1)
- No security patches (1)
- Not using LDAP for passwords rotations (1)
- Server without authentication (3)
- Too much network exposure (4)
- Unencrypted communication (2)
- Unencrypted data (2)
- Using basic passwords (1)
- Using default Mongod ports (1)
- Using unofficial packages (1)

<u>Relational design ghosts</u>: originate from misunderstandings of design principles; can be expected when a programmer familiar with SQL DB design and manipulation recently switched to MongoDB.

- Relying on transactions (2)
- Separating data accessed together (19)
- Storage of empty values (1)
- Use of relational collections (1)

<u>Human-oriented decisions</u>: caused by data modeling choices to foster human aspects, resulting in sub-optimal design choices, leading to performance or storage/ bandwidth overhead.

- Bias toward access patterns (1)
- Human-readable values (2)
- Too long attribute names (2)
- Too long document keys (2)
- Using $ prefixed fields (1)

Figure 4.5: MongoDB Code Smell Classification

An example of such can be found in *bitpay/bitcore-wallet-service* project. [7] Listing 4.3 illustrates this instance.

Listing 4.3: Example of Map-reduce for projection

```
122        self.db.collection(storage.collections.WALLETS)
123        .mapReduce(map, reduce, opts, function(err, collection, stats) {
124          return cb(err);
125        });
```

[7] https://github.com/bitpay/bitcore-wallet-service/blob/1949924/lib/stats.js#L123

We can observe that this project maintainers performers a map-reduce operation on the `storage.collections.WALLETS` collection. The `map` and `reduce` variable references actually point to functions that perform those stages. Actually it is possible to replace this with an aggregation. Indeed, the `map` function matches elements corresponding to a given date while the `reduce` one count them.

Listing 4.4: Aggregation equivalent for Listing 4.3

```
1   db.collection(storage.collections.WALLETS).aggregate([
2   {
3       $match: {
4               day: Date(this.createdOn * 1000),
5               network: this.network,
6               coin: this.coin
7               }
8   },
9       $group: {
10
11          count: { $sum: 1 }
12      }
13  ])
```

4.4.2 Backup

Poor backup strategies can result in permanent data loss; hence, we grouped them into a separate category. In this section we describe the code smells who were identified as being part of this category.

Manual backup In his book "*Mastering MongoDB 6.x*", Alex Giamas listed best practices to follow in order to ensure "*achieve high availability*". He emphasized that regular automated backups should be scheduled and not relying on manual backup as this can hurt consistency.

No backup budget In a linkedin post, Marc Kenig stressed the importance to dedicate a specific backup budget as this operation consumes resources, especially if this needs to be performed on a regular basis. Failing to do so will induce cost overhead or hinder the consistency. [8]

Replicas as backup To provide data redundancy and improve availability, MongoDB provides replica sets. [9] A replica set is composed of multiple mongod instances, one primary node handling all write operations and multiple secondaries. Automatic failover allows the election of a new primary if the current one fails, ensuring high availability. Secondaries replicate the primary's operation log (oplog) asynchronously to keep data consistent. We found that multiple (3) sources insisted on avoiding to rely on this mechanism as backup as they do not protect against data loss originating from human errors, such as accidental database or collection deletion [39].

[8] urlhttps://www.linkedin.com/pulse/big-data-anti-patterns-marc-kenig/

[9] https://www.mongodb.com/docs/manual/replication/

4.4.3 Design oversights

Many smells originate from poor design choices to accelerate design & development. These decisions often lead to decreased maintainability, bugs, storage waste, and performance issues. Because we gathered many smells under this category, we decided to split it into several sub-categories, depending on their origin: *Application*, *Schema* and *Index*.

Application

This category regroups smells that stem from the application code surrounding a database access rather than the access itself.

Immortal cursor In their book "*MongoDB: the definitive guide: powerful and scalable data storage*", Shannon Bradshaw, Eoin Brazil and Kristina Chodorow [27] warned developers of the issues raised by not closing a cursor. Indeed, after performing a find query on a collection, a cursor is created to iterate over the results of this query. If this cursor has not been exhausted, meaning that the results from the query were not completely iterated over or not closed manually via the `Cursor.close()` function, can still unnecessarily consume memory.

An instance of this smell can be found in the *jembi/openhim-core-js* project [10]

Listing 4.5: Immortal Cursor instance from *jembi/openhim-core-js*

```
157        const passportResult = await model.PassportModelAPI.find({
158            email: user.email
159          })
```

Here, maintainers query documents based on the field email and stores the cursor into the constant `passportResult`. However, in the following code, that cursor is neither iterated upon neither closed manually via the `close` method invocation, leading to the cursor hugging memory indefinetely.

No dependency injector A dependency injector is a generic design pattern that allows a program to dynamically inject dependencies instead of hard-coding them, improving system modularity. In the context of Mongoose modelling that relies on Schema definition to insert documents, as explained in Section 1.1.1, where certain modules can accumulate many import statements only for Mongoose models. Here, implementing a dependency injector may help with the system modularity and module readability.

Querying too much data In multiple sources including a book entitled "*Practical MongoDB*" by Shakuntala Gupta Edward and Navin Sabharwal [44] and a StackOverflow post [11], we could observe an antipattern in querying a collection documents without filtering the needed fields. The projection mechanism was explained in Section 1.1.1. Not doing so can generate unnecessary network traffic as unused fields

[10] https://github.com/jembi/openhim-core-js/blob/master/test/unit/usersTest.js#L157

[11] https://stackoverflow.com/questions/13907673/is-transaction-script-an-antipattern-with-nosql-databases-also

are returned. Listing 4.6 illustrates this smell by reusing the collection `countries` from Fig. 4.3

Listing 4.6: Query too much data instance example

```
1   const doc = await db.collection("countries").findOne({name: "Belgium"});
2   console.log("This country currency is " + doc.currency)
```

In this code snippet, we first select a single document which `name` field equals "*Belgium*" (line 1). Then the `currency` field is printed through the `console.log` call (line 2). Therefore, only one field out of the 15 available in the corresponding document is being utilized, leading to excessive cluttering of the application bandwidth.

Testing only CRUD operations In a MongoDB forum post asking the biggest mistakes people make with the aggregation pipeline, a user called "*Michael Höller*" revealed that people tend to test only CRUD operations without consideration for other operations like an aggregation pipeline. [12] This could harm the project validation as this could negatively impact the test suite and its coverage.

Testing queries on the entire ad-hoc big data lake As big data lake grows in size, so does the cost of running and the execution time. Additionally, such queries may not work or return the expected data or even result in a global performance drop. A dedicated subset should be preferred for testing, as suggested by this LinkedIn post [8].

The single-person bridge In a presentation from 2019, Charles Sarrazin a former MongoDB member described many witnessed antipatterns [115]. There, he described what he called a *single-person bridge* that consists in an implementation of a SQL sequence in MongoDB, using a document having a counter field and a *findAndModify* call to increment it. This can severely harm performance as documents are write locked, harming asynchronous document insertions. Listing 4.7 examplifies this smell.

Listing 4.7: Single-person bridge function example

```
1   async function getNextNumber() {
2       let client;
3
4       try {
5           client = await MongoClient.connect(url);
6           const db = client.db("main");
7
8           const result = await db.collection('sequences').findOneAndModify(
9               { name: "sequence" },
10              update = { $inc: { counter: 1 } },
11              upsert = true
12          );
13          return result.value.seq;
14      } catch (err) {
```

[12]https://www.mongodb.com/community/forums/t/what-are-some-of-the-biggest-mistakes-people-make-in-aggregation-pipelines/11803/2

```
15                console.error('Error', err);
16            } finally {
17                if (client) {
18                    await client.close();
19                }
20            }
21        }
```

This code snippet shows the definition of the function `getNextNumber` which is an instance of this smell. We assume that the client has already been initialized and the url is a valid MongoDB connnection one.

This function main query is performed by the `findOneAndModify` method on the sequences collection (Lines 7-10). Thanks to its filter document, it finds a single document matching the value "sequence" for the field `name` (Line 8). It then increments the counter field of that document by 1 (Line 9). If no such document exists, the `upsert` option ensures that a new document is created with the specified initial value (Line 10). The sequence actual value is then returned (Line 11). If any error occurs during the database operations, it is caught in the catch block and logged to the console (Lines 12-14). Finally, the MongoDB client is closed in (Line 18).

Schema

This category focuses on identifying design oversights that occurred during the Schema modeling stage.

Bloated documents Many (17) sources reported that letting individual document size grow hurts the database performance. In their series about MongoDB Schema Anti-Patterns, MongoDB released an article about bloated documents [122]. There, they explained that having too large documents actually hurts memory and that some fields from those documents are not frequently accessed. In those cases, it is indicated to break down them into smaller ones.

Data oriented instead of application oriented A central point according to diverse (12) sources was the modelling philosophy. Indeed, contrary to relational modelling where it is recommended to model first the data according to the many normal forms, MongoDB is centered around data access. In a presentation about "*MongoDB Schema Design Best Practices*", Joe Karlsson, a MongoDB developer, stated that "*How you model your data depends - entirely - on your particular application data access pattern*".

Flat Raw Data "Embedding vs. References" is a critical schema decision in MongoDB, and referencing documents in arrays is a common technique to map one-to-many or many-to-many relationships. Having fields only on the document root level and not utilizing any embedding can lead to a decrease in performance as each root-level field needs to be read during the parsing process. This approach also hinders code readability due to the absence of hierarchy in the document. The Flat Raw Data code smells refer to documents that lack embedding. Additionally, having every field on the root level could indicate an approach similar to table in relational

modelling, similar to smells described in Section 4.4.7. By altering the document structure from Fig. 4.3, Listing 4.8 illustrates an instance of the smell.

Listing 4.8: Flat Raw Data example

```
{
    //...
    "areaValue": 3511023,
    "areaUnit": 'square miles'
    //...
}
```

There, we can see the two attributes `areaValue` and `areaUnit` are on the root level. However, they both qualify the same notion, being the `country` area.

Inconsistent attribute structure Maintaining a consistent document structure is essential in MongoDB modeling. This involves using the same fields repeatedly and ensuring that the order of embedded fields remains consistent. The internal representation of documents, BSON, is influenced by the order of fields in an embedded document. An incorrect field order can lead to difficulties in retrieving the intended document, as noted by Phil Factor in an article on InfoQ. Example of such behavior can be seen in Listing 4.9.

Listing 4.9: Inconsistent attribute order instance example

```
db.collection("countries").find({
    area: {
        unit: "square kilometers",
        value: 30688
    }
});
```

Through the utilization of the Fig. 4.3 collection, the query is designed to fetch the document associated with Belgium by utilizing the respective values for the `area` attribute and unit. However, due to the organization of the document where the `area` subfields are structured in the sequence of `value` followed by `unit` rather than `unit` preceding `value` as expected by the query's filter object, it will not yield any results. As a result, this issue can have a significant impact on the maintainability of the system.

Repeated immutable data This smell occurs when many documents in collection all share a set of field name-value pair. Indeed, if that data is stored but never read or updated, this can cause memory overhead. A solution suggested by Marc Kening would be to move the immutable part in another collection and performing an aggregation when needing to access that data. [8] Listing 4.10 illustrates this smell with a new collection, `books`.

Listing 4.10: Repeated immutable data example

```
{
    "_id": 1,
    "title": "Silmarillion",
```

```
4        "ISBN": 9780618391110,
5        "author": {
6            "surname": "Tolkien",
7            "givenNames": ["John", "Ronald", "Reuel"],
8            "birthPlace": {
9                "country": "Orange Free State",
10               "city": "Bloemfontein"
11               }
12           }
13   },
14   {
15       "_id": 2,
16       "title": "The Fellowship of the Ring",
17       "ISBN": 9780007136599,
18       "author": {
19           "surname": "Tolkien",
20           "givenNames": ["John", "Ronald", "Reuel"],
21           "birthPlace": {
22               "country": "Orange Free State",
23               "city": "Bloemfontein"
24               }
25           }
26   }
```

There, we can see the books document storing the author as a dedicated field. Given the prolific nature of an author like J.R.R. Tolkien, this information is likely duplicated across all documents within the collection. To optimize memory usage, it may be beneficial to move this data into a separate `authors` collection and reference it using a `$join` operation in aggregation, as in Listing 4.11.

Listing 4.11: Repeated immutable data example fixed

```
1    {
2        "_id": 1,
3        "title": "Silmarillion",
4        "ISBN": 9780618391110,
5        "authorId": 1
6    },
7    {
8        "_id": 2,
9        "title": "The Fellowship of the Ring",
10       "ISBN": 9780007136599,
11       "authorId": 1
12   }
```

Storage of easily calculated values Another way to burden a document is to add fields that could be easily computed from others already present. Indeed, as described in "*Bloated Documents*", too large documents may occupy too much storage

space. A possible root cause would be fields that can be computed from others. For example, if we reuse the document Schema from Fig. 4.3 and add a "citiesCount" field which stores the number of cities a country has, just as reported by Listing 4.12

Listing 4.12: Storage of easily calculated values example

```
{
    //...
    "citiesCount": 2
    "cities": [ {
        "_id": 1,
        "name": "Namur",
    //...
}
```

However, since the 'cities' documents are stored as an array, it is feasible to determine the length of this array using the native JavaScript method 'Array.length' and subsequently remove that field from the document. This approach can be viewed as a trade-off between client-side computation and storage space, as discussed by Edward and Sabharwal in their book 12th chapter as [44].

Too many collections In a blog post entitled "*Data Modelling with MongoDB*", Arvind Padmanabhan discusses MongoDB schema design principles [107]. After discussing design differences with relational modeling and design patterns, the post talks about having too many collections and indices: "*Storing numerous collections in a database, some of which are unused, is another anti-pattern*". A MongoDB instance also has an advised maximum number of collections. In an blog post about the "*Massive Number of Collections*", MongoDB employees Lauren Schaefer and Daniel Coupal advise to use 10,000 collections per replica set [119].

Multiple Schemas in a file As explained in Section 1.1.1, Mongoose uses `Model` objects to insert documents and those `Models` which must first be defined with a `Schema`. For Valeri Karpov, a former MongoDB maintainer and user, separate `Schemas` should be kept in separate files to foster maintainability by simplifying navigation through various Schemas [70]. As a result, he recommends adhering to the guideline of one `Schema` per file.

Unbound arrays The most described smell among our sources (23) is "*Unbound arrays*". MongoDB dedicated it an article in their series about schema design antipatterns [118], Alex Giamas discussed the issue in its book "*Mastering MongoDB*" [56] or Kristina Chodorow in her sixth tip "*Do not embed fields that have unbound growth*" in the book "*50 Tips and Tricks for MongoDB Developers*" [38]. When trying to model a one-to-many relationship (as `countries` to `cities`) between MongoDB documents, one way would be to embed the many part of the relationship into the other as an array of documents, such as displayed in Fig. 4.3. However in the case of an unbounded array growth, the document size may reach the maximum of 16 MB or drive up the cost of accessing that document. A solution proposed by Lauren Schaefer and Daniel Coupal is to move the embedded documents into a separate

collection and replace the array of documents by an array of references to those documents. Listing 4.13 shows the application of this solution on the `countries` collection.

Listing 4.13: Example to fix unbound array in `countries` collection

```
{
    //...
    "cities": [1, 2]
    //...
}
```

There, the array of documents is replaced by a list referencing the `_id` of `cities` document related to this one. After this change, an aggregation that performs a join between a `countries` document and its corresponding `cities` document can now be utilized to retrieve information about a country and display its associated cities.

Using a Document only for id field In her book "*50 Tips and Tricks for MongoDB Developers*", Kristina Chodorow warns about using a dedicated document for the `_id` field [38]. Indeed, this field is indexed by default thus indexing a document will result in an index definition for each field this document possesses, cluttering the index storage and hindering maintainability as the `_id` field is immutable. Listing 4.14 shows such an example as an alternate structure for a document from the `countries` collection.

Listing 4.14: Using a Document only for id field smell instance example

```
{
    "_id": {
        "name": "Belgium",
        "currency": "Euro"
    }
}
```

Index

As explained in Section 1.1.1, index is a mechanism to improve query performances. However, oversights can lead to storage overhead or higher query running cost. We describe in the following the code smells from this category.

Abusive use of indexes Abusive use of indexes is the second most described smell in our sources (22). Indeed, each index occupies storage space and slows down updates or insert operations. However, how many index is too many varies depending on a system-specific data and queries. In their article about unnecessary indexes, Lauren Schaefer and Daniel Coupal reported that MongoDB recommends to use at most 50 indexes per collection [120]

Index intersection rather than compound index In a white paper entitled "*MongoDB Operations Best Practices*", MongoDB listed common mistakes while using indexes [95]. In the discussion, they emphasized the preference for utilizing compound indexes over index intersection, along with incorporating a filter document that includes multiple fields.

Listing 4.15 shows an example of a find query over the `countries` collection where the filter document is defined upon two different fields.

Listing 4.15: Query with a filter document holding multiple fields

```
1    db.collection("countries").find( {
2        country: "Belgium",
3        currency: "Euro"
4    });
```

Listing 4.16 corresponds to an intersection index declaration which consists of individually declaring each index.

Listing 4.16: Index intersection declaration

```
1    db.collection("countries").createIndex({ name: 1 });
2    db.collection("countries").createIndex({ currency: 1 });
```

Listing 4.17 is a compound index declaration. Here, the two fields `name` and `currency` are grouped inside the same index declaration, making it a compound index. In this case, the query defined in Listing 4.15 will run faster than with an index intersection declaration such as in Listing 4.16.

Listing 4.17: Compound index declaration

```
1    db.collection("countries").createIndex( {
2        name: 1,
3        currency: 1
4    });
```

Non-ESR (Equality, Sort, Range) compound index Nonetheless, simply using compound index is not enough. Indeed, to design a query that performs an equality, a range and a sort such as in Listing 4.18, the compound index declaration should adhere to the order ESR, which stands for Equality Sort Range.

Listing 4.18: Candidate query for a ESR compound index

```
1    db.collection("countries").find({
2        "area.value" : { $gte: 500000 },
3        "area.unit": "square kilometers" }).
4        sort( { name: 1 } )
```

First, this query searches for document having an `area value` of at least 500,000 (Line 2), which consists in the "Range" part of the query. Then, it looks if this document `area unit` is the square kilometers (Line 3), that corresponds to the "Equality" part. Finally, resulting documents are sorted accordingly to the `name` field (Line 4), being the "Sort" part.

In 2019, Alex Bevilacqua, a MongoDB maintainer, gave a presentation entitled "*The Sights (and Smells) of a Bad Query*" [26]. There, he showed that in the case of a

query performing a ESR, the compound index declaration must respect the order Equality then Sort and Range, such as in Listing 4.19.

Listing 4.19: ESR compound index declaration

```
1   db.collection("countries").createIndex( {
2       "area.unit": 1,
3       name: 1,
4       "area.value": 1
5   });
```

Prefix index of compound indexes Also, some queries that perform their selection based on a single field can use a compound index. Indeed, as reported by MongoDB in their white paper about "*MongoDB Operations Best Practices*", a query can benefit from a single field that is the prefix of a compound index declaration. Consequently, individual index declarations that are prefix of a compound index declaration are unnecessary and can be removed. Listing 4.20 shows a index declaration that is a prefix from the compound index declared in Listing 4.19.

Listing 4.20: Prefix index of compound index example

```
1   db.collection("countries").createIndex( {
2       "area.unit": 1,
3   });
```

In this scenario, the index can be safely removed. However, if the declaration of that particular index had been made on `name` or `area.value` fields, the situation would have been different.

4.4.4 Human-oriented decisions

Fostering human understandability over optimal design choices also produces code smells. We describe in the following the code smells corresponding to this class

Bias toward access patterns One source, Zameer Ansari with his article about "*MongoDB schema design*" posted on hackernoon [20] warns about not being biased toward any particular modelling pattern. Indeed, he advocates to prioritize query performance over shaping the data in standard modeling practices.

Human-readable values Choosing inefficient field formats result in storage / RAM / network bandwidth consumption overhead. Indeed, some (2) sources suggest that this inadequation can stem from fostering human readability over machine optimization, *e.g.*, using a string data type instead of a number or a date, such as illustrated by Listing 4.21.

Listing 4.21: Human-readable value instance

```
1 {
2     "date": "April 1st 2019"
3 }
```

Too long attribute names In a blog post called "*MongoDB Best Practices: Design, Deployment & More*", Esaya Aloto advises against using lengthy field names. Aloto emphasizes that the internal BSON size limit for field names is 125 characters. Additionally, it is important to note that longer field names occupy more space in the database as they are stored with each inserted document [16].

Too long document keys When inserting a document into a collection, MongoDB engine automatically add the `_id` field, whose value is an internal representation, namely the ObjectId. [13] However, developers may be tempted to use other values as ID, such as UUID, according to Marc Kening. [8] Yet, UUID need 16 bytes of storage while Object 12, making UUID usage a storage overhead. Furthermore, Joe Karlsson, an employee at MongoDB, strongly recommends against using UUIDs to mitigate the risk of collisions in a forum post [14].

Using $-prefixed fields Starting from the version 4.0, insertion of a document containing a field starting with the "$" symbol raises a error. Indeed, such field names take the form of operators that are used to refine the searching part as seen in Section 1.1.1. We found an issue raised in the official MongoDB Jira reporting an inconsistency in the verification of this smell between the `insertOne` and `replaceOne` methods. [15]

4.4.5 Performance/Memory

This category addresses various performance issues that arise from deficient configurations. These configurations can negatively affect both the performance of MongoDB operations and the memory usage of the system. We presently discuss those code smells.

Large read-ahead Read-ahead is a Unix kernel feature that allows for loading disk content into cache and read it from there, instead of loading sequentially the data. [16] It can be tuned with a byte offset parameter to load but setting a too large can be problematic, as suggested by a blog post on "MongoDB on AWS: Guidelines and best practices", posted by Amazon Web Services [17]. Indeed, this could cause a memory usage overhead or even prevent remove data that should have been accessed by another application.

Large WTC (WiredTiger Cache) WiredTiger Cache is the mechanism through which MongoDB stores data in memory. It functions in tandem with the mongod instance that automatically uses the leftover memory to store compressed data, which is then uncompressed in the cache to fit the allocated WiredTiger memory allocation. [17] This file buffering system is designed to ensure stable performance by

[13] https://www.mongodb.com/docs/manual/reference/method/ObjectId/
[14] https://www.mongodb.com/community/forums/t/how-to-manage-a-db-with-collections-with-different-fields/98287/8
[15] https://jira.mongodb.org/browse/SERVER-40070
[16] https://man7.org/linux/man-pages/man2/readahead.2.html
[17] https://www.mongodb.com/docs/manual/core/wiredtiger/

preventing data from being read from the disk, provided the buffer zone is sufficiently large. However, increasing the memory allocated to WiredTiger might paradoxically lead to a performance drop. This happens because a larger memory allocation can reduce the buffer zone, thereby increasing the chances of needing to read from the disk. This phenomenon is discussed by Corrado Pandiani in his blog post "*MongoDB Tuning Anti-Patterns: How Tuning Memory Can Make Things Much Worse*".

Multiple "mongod" instances Corrado Pandiani also warns about another antipattern that could be made by wrongly tuning the memory, which would via using multiple instances of mongod, the MongoDB process. Because of the mechanism of grabbing the unused memory to store compressed data, those different mongod processes would be competing for the same resources, and hinder the maintenance process as it would be harder to identify which mongod process used too much memory.

Running MongoDB in a shared environment Following the same logic, it becomes trickier to run MongoDB in a shared environment as the mongod processes would be competing with other processes. Edward and Sabharwal also raised this issue in their book and suggested to run MongoDB on a dedicated server. [44]

Running MongoDB on 32-bit systems "*BEEVA*" an open-source repository containing best practices and recommendations about many software engineering topics suggested not to run MongoDB on 32-bit systems [18] Indeed, 32-bits MongoDB has a 2 GB data limit and such system also have limited memory size. Using such systems is also discouraged by MongoDB API. [19]

Unlimited mongos taskExecutor in a container Improperly configuring the number of taskExecutor threads in a mongos instance may lead to an excessive creation of threads and a surge in CPU usage, as highlighted by Vinicius Grippa in his blog post titled "*MongoDB Best Practices: Security, Data Modeling, & Schema Design*". Grippa points out that these threads are not cgroups (control groups) aware, complicating their real-time monitoring.

Using fast writes Fast writes are counter-productive with MongoDB, as affirmed by Phill Factor in his blog post listing the mistakes related to starting in MongoDB [48]. Indeed, in a fast write context, a MongoDB insertion would return its result before writing it which could lead to an inconsistency in the case of a system crash.

Using GridFS for small binary data The GridFS framework is designed to store documents exceeding the 16 MB size limit. As it manipulates documents larger than the ones admitted in collections, a GridFS query happens in steps : first it fetches the file metadata then the file content. Consequently, using it for documents that would

[18] https://github.com/vaquarkhan/Technology-best-practices/blob/master/nosql/mongodb/README.md

[19] https://www.mongodb.com/docs/v3.2/installation/#deprecation-of-32-bit-versions

fit in standard collections would prove to a performance and storage overhead, as noted by Chodorow. [38]

4.4.6 Query

Queries are prone to various smells. Most sources mentioned problems due to inefficient index usage.

For example, 5 sources stressed the importance of covering queries with indexes because a query without a matching index likely performs a full collection scan instead of an optimized index scan [26]. A case-insensitive query should also be covered with a case-insensitive index (7) [21]. 2 sources also stressed that leading wildcard searches (*i.e.,* regular expressions that are not left anchored or rooted) should be avoided [97, 90].

Avoid large skips In MongoDB, skips are used on a query to pass a given number of documents. This is typically using for pagination, which separate the retrieved data into several pages. Using large skips for pagination could be linked to the processing of many documents and thus performance issues, as reported by Chodorow and Dirloff [27]. In order to avoid using those large skips, authors recommend to sort the query result in reverse or better tuning of the search criteria. Listing 4.22 demonstrates a query that has a skip value of 100 with an empty filter document, resulting in the selection of all documents from the `countries` collection.

Listing 4.22: Large skip instance

```
1     db.collection("countries").find({}).skip(100);
```

Listing 4.23 shows a possible fix for this code smell instance. Instead of skipping documents, it introduces a filtering condition on the `currency` field, which gets rid of the necessity of performing a `skip`.

Listing 4.23: Large skip instance fixed

```
1     db.collection("countries").find({ currency: "Euro" });
```

Avoid $where operator The $where operator allows to pass a JavaScript expression as a string in order to evaluate it and use it to query documents. [20] Listing 4.24 shows a usage of this operator.

Listing 4.24: $where operator usage

```
1     db.collection("countries").find({
2         $where: " this.continent == 'America' "
3     })
```

On line 2 we can see the JavaScript expression `this.continent == "America"` being used to include documents whose `continent` field value is "*America*". In that context, `this` refers to the document on which the JavaScript expression is being executed upon. As reported by Edward [44], this operator is associated with reduced

[20] https://www.mongodb.com/docs/manual/reference/operator/query/where/

performance as it cannot benefit from MongoDB index and needs to be executed on every document. Additionnaly, it can expose the system to security vulnerability as it could allow to execute arbitrary JavaScript code.

Case-insensitive queries without matching indexes In another post of their MongoDB antipatterns series, Schaefer and Coupal noted the issue raised by the mismatch in case-sensitivity of query and its corresponding index. [117] By default, created indexes are case-sensitive. This implies that even if a query is explicitly set as case insensitive, the results will still be influenced by the case sensitivity of the index declaration, because MongoDB performs a index scan.

In the following, we illustrate an instance example. The `countries` collection holds documents as displayed in Listing 4.25. We can spot that the first document `currency` field value is in lower cases while the second starts with a capital letter.

Listing 4.25: `countries` documents example

```
1  {
2      //...
3      "currency": "euro",
4      //...
5      },
6  //...
7  {
8      //...
9      "currency": "Euro",
10     //...
11     }
```

Listing 4.27 shows the corresponding access code. First it defines an index on the `currency` field (Line 1). Then it performs a query with a collation strength of 1, meaning it is case-insensitive (Line 5). However, this query only yields a single document as a result, because of the index default collation.

Listing 4.26: `countries` documents access code

```
1  db.collection("countries").createIndex({ currency: 1 });
2
3  /* ... */
4
5  db.collection("countries").find({ currency: "euro" }, { collation: 1 });
6  /* only returns {"currency": "euro"} */
```

Confusing null & undefined values Using an undefined in a MongoDB document can lead to erroneous behaviors. Indeed, while both null and undefined are present in the BSON specification, the latter is marked as deprecated [21]. We came across a GitHub issue on the *meteor/meteor* repository where one user reported having issues while using undefined in a MongoDB collection. [22] A project maintainer explained

[21] https://bsonspec.org/spec.html
[22] https://github.com/meteor/meteor/issues/1646#issuecomment-29682964

the issue and proceeded to discourage him from using undefined in a MongoDB Document.

Leading wildcard searches on indexed columns Using a leading wildcard to query an indexed column can actually decrease the general performance, as higlighted by Mat Keep and Henrik Ingo in their blog post named "*Performance Best Practices: Indexing*". The author precises the root cause is a complete index scan instead of stopping when retrieving the adequate document, because the leading wildcard. Listing 4.27 illustrates an example of a query using a leading wildcard.

Listing 4.27: Leading wildcard query example

```
1   db.collection("countries").find({ currency: "*uro" });
```

We can see that it uses "*" as a wildcard character. This query will result in a complete index scan as MongoDB will try to unify the wildcard character with every index value.

Using $limit without $sort Using a $limit operator without a $sort one may hinder code maintainability as it causes a non-deterministic behavior. Indeed, as reported by Factor, having a data sample with only $limit will result in an unpredictable documents result. Listing 4.28 shows an example of a query using only a limit operator. In this query, the result is restricted to the first 5 documents from the `countries` collection. Nevertheless those 5 documents can change between executions, even if no new documents were added.

Listing 4.28: Using $limit without $sort instance

```
1   db.collection("countries").find({ continent: "Europe" }).limit(5);
```

Using $map, $reduce and $filter with array fields Using a $map, $reduce, or $filter operator can be less effective than using their projection field counterpart in an aggregation operation, as suggested by Michael Höller in a forum post about "*the biggest mistakes people make in aggregation pipelines?*".

Negation in queries A negation in a query can poorly impact its performance. Indeed, negation operators (such as $not or $ne for not exists) can cause complete a collection scan even if that specific field is indexed. That's becauseMongoDB cannot **index** the absence of a field, especially if the negation is not selective. [95] Listing 4.29 shows an example of a Negation in a query.

Listing 4.29: Negation in query instance

```
1   db.collection("countries").find({ currency: {$not: "Canadian dollar" }});
```

There, the query selects `countries document` whose currency it **not** "*Canadian dollar*".

No $elemMatch to match an entire array Trying to match specific elements within a document array without using the $elemMatch operator may offer undesired behavior as suggested by this blog post [27]. Indeed, if we try to query our `countries` collection if a `cities` element `name` is "*Namur*" and its `mayor` is "*Paul Van Miert*" using a naive query approach as described in Listing 4.30, we actually obtain the document related to Belgium.

Listing 4.30: Naive query to match an array element

```
1   db.collection("countries").find({ "cities.name": "Namur", "mayor" : "Paul
        Van Miert" });
```

This is because the query first matches a `cities` array element whose `name` is "*Namur*" (`_id` 1) and then matches another element whose `mayor` is "*Paul Van Miert*". In order to precisely match a document whose a single `cities` element, we have to use the $elemMatch, as in Listing 4.31.

Listing 4.31: Query to match an array element using $elemMatch

```
1   db.collection("countries").find( $elemMatch: { { "cities.name": "Namur",
        "mayor" : "Paul Van Miert" } });
```

Single update/insert for batches The `db.collection.update()` function only updates a single document by default as reported by Phill Factor. [48] This can lead to migration issues or bugs Listing 4.32 showcases this effect.

Listing 4.32: Single document update

```
1   db.collection("countries").update({}, { $set : { "foo" : "bar" } });
```

This query selects any document and then adds a field called `foo` to set it to a value "*bar*". To perform this operation on every document of the collection, the `multi` option must be set to `true`, as in Listing 4.33.

Listing 4.33: Multiple document update

```
1   db.collection("countries").update({}, { $set : { "foo" : "bar" } }, {
        multi: true });
```

Sorted Monkeys Using a $push operator alongside a $slice or a $sort modifier may result in performance drop as suggested by Sarrazin in his presentation about MongoDB antipatterns. [115] Indeed, performing those sequence of operators implies to replace the complete array with a new one without altering it. A instance of this code smell can be found in Listing 4.34.

Listing 4.34: Sorted Monkey instance

```
1   db.collection("countries").update({"name": "Kenya"}, {
2       $push : {
3           "cities" : {
4               "name": "Mombasa",
5               "population": 1208333,
6               "timezone": "East African Time"
```

```
7                 "mayor": "Abdulswamad Shariff Nassir"
8                 }
9             },
10            $sort : { "name" : 1}
11        });
```

There, we can see the search being done on documents having its `name` field equals to "*Kenya*" (Line 1). Then, it proceeds to add a new element to the `cities` array with the `$push` operator (Line 2-7). Finally, with the $sort modifier, a new array sorted by the `name` field (Line 10).

Uncovered queries Using `fields` in the filter documents that are not indexed is not only inefficient but also is associated with higher energy usage. Divya Mahajan, Cody Blakeney and Ziliang Zong proposed an energy consumption evaluation of multiple database management systems, including MongoDB [85]. In their study, they could observe a decrease of around 50% power consumption from querying an uncovered field (that has not been indexed) VS covered field. Bevilacqua also talked about the potential speedup an index can bring to a MongoDB collection. [26]

4.4.7 Relational design ghosts

This code smell category aims at regrouping code smells originating from confusion between relational modeling practices and MongoDB modeling practices, such as "*Data that is accessed together should be stored together*". Such problems could be expected from developers with a relational modeling background that recently migrated to MongoDB. There are numerous (7) warnings against applying traditional relational modeling techniques to MongoDB. Such sources include *Oren Eini*, founder of RavenDB which is Document Datastore, with its blog post named "*The relational modeling anti pattern in document databases* [45], *Joe Karlsson*, MongoDB employee in an official MongoDB blog post [69] or *Frank David* who highlights common optimization pitfalls in a dedicated blog post [50].

Hence, joins should be avoided as recommended by the *Separating data accessed together* smell in 19 sources [80].

Relying on transactions In relational DBs developers rely on transactions to guarantee ACID (atomicity, consistency, isolation, durability) properties. However, an operation on a single document is already atomic in MongoDB. Moreover, using a transaction to insert documents into different collections might be a sign of "*Separating data that is accessed together*" Thus, transactions indicate accessing data that should not be separated as suggested by Lauren Schaefer in her blog post about "*3 Things to Know When You Switch from SQL to MongoDB*" [116]. Listing 4.35 illustrates this smell with an instance example.

Listing 4.35: Transaction usage

```
1        const { MongoClient } = require("mongodb");
2        const uri = "mongodb://localhost:27017";
3        const client = new MongoClient(uri);
4        const db = client.db("main")
5
```

```
6   async function run() {
7
8       await client.connect();
9       const session = client.startSession()
10
11      try {
12      await session.withTransaction(async() => {
13          let result = await db.collection("countries").insertOne({
14              name: "Cambodia",
15              currency: "Cambodian riel",
16              continent: "Asia",
17              anthem: "Nôkôr Réach",
18              languages: ["Khmer"],
19              area: {
20                  value: 181035,
21                  unit: "square kilometers"
22              }
23          })
24          await db.collection("cities").insertOne({
25              name: "Phnom Penh",
26              population: 2841553,
27              timzeone: "Indochina Time",
28              country: result.id
29          })
30      })
31
32      } finally {
33      await session.endSession();
34      await client.close();
35      }
36  }
37  run();
```

First, it initializes the client and uses the desired database (Line 1-4). Then, `run` function connects to the client and prepares the transaction `session` (Line 8-9). The transaction itself is started with the `session.withTransaction` call (Line 12). In this transaction, 2 insert operations are executed, reusing the separate `countries` and `cities` collections (Lines 13-23 and Lines 24-29). Notice that the second operation reuses the inserted id from the first to references its belonging country. If an exception is thrown during the execution of one of those insert operations, the session would roll back any uncommitted change. In the `finally` clause, the transaction is closed which commits changes that happened during the transaction and the connection to the client is also closed (Line 34).

Separating data accessed together MongoDB general rule of thumb for modeling is "*Data that is accessed together should be stored together*" Indeed, breaking this rule and separating data, which means storing related documents in different collections, implies the use of a $lookup operator which is slow and resource-intensive. This is supported by many sources (19) such as an independent blog post relating "*Another anti-pattern is storing information in separate collections although they're often accessed together*" [107]. An extensive example of this smell can be found in Section 4.2.

Storage of empty values Fields names also occupy storage space as reported by the documentation. [23] Thus, when a document does not have a value for a specific, setting the related field to an empty value (*i.e.,* empty string, null) may be a storage waste according to Marc Kenig. [8] A fix could be to omit the field in the document, there are operators to retrieve documents without a given field, such as $exists. Listing 4.36 shows an insert operation of a document that contains an empty field.

Listing 4.36: Storage of empty value instance

```
1   db.collection("countries").insert({
2       name: "Nauru",
3       currency: "Australian dollar",
4       continent: "Oceania",
5       anthem: "Nauru Bwiema",
6       languages: ["Nauruan", "English"]
7       area: {
8           value: 21,
9           unit: "square kilometers"
10      }
11      cities: []
12  })
```

Here, we insert a document related to the Republic of Nauru, which does not have any official city (according to Wikipedia [24]). We can see the `cities` field associated with an empty array on line 11. To save some space, this document could have instead omitted this field.

Listing 4.37 shows an example on how to retrieve a `countries` document without a `cities` field.

Listing 4.37: Query to retrieve a document without a `cities` field

```
1   db.collection("countries").find({ cities: null })
```

In MongoDB, using null as a value in a filter document for a given **field** will retrieve either document without this **field** or documents having `null` as a value for this **field**.

Use of relational collections In relational modeling, a many-to-many relationship between 2 entities at the conceptual level is most often modeled in the logical schema with an intermediate table between the 2 corresponding tables. This table contains a reference to the 2 other tables. Applying such an approach between 2 MongoDB collections is a code smell. Indeed, to access one collection from another, this would require a sequential usage of 2 $lookup operators, combining their negative impact on performance as suggested by this blog post [42]. To avoid this smell, a concerned maintainer should alter its schema so the intermediate collection is embedded in both of the collections. Listing 4.38 illustrates this code smell with an instance example performing the equivalent of a joint, reusing the classical logical schema tables `Product`, `Order` and `Details`, the latter being the intermediate between the first two.

[23] https://www.mongodb.com/docs/manual/core/data-model-operations/#storage-optimization-for-small-documents

[24] https://en.wikipedia.org/wiki/Nauru

Listing 4.38: Use of relation collections instance

```
1   db.collection("products").aggregate([
2       {
3           $lookup: {
4               from: "details",
5               localField: "_id"
6               foreignField: "productId"
7               as: detailedProducts
8           },
9
10          $lookup: {
11              from: "orders",
12              localField: detailedProducts.order,
13              foreignField: "_id"
14              as: orders
15          }
16      }
17  ])
```

To use the $lookup operators, we first need to perform an aggregation on the `products` collection. Then we match the `details` collection `productId` field with `products _id` field and output the intermediate results in a temporary array `detailedProducts` (Line 3-7). This array contains the documents from the `details` collection that references the current `products` document. Finally, it performs a second $lookup to the `orders` collection, matching the productId references from the `detailedProducts` list with the `_id` from the `products` documents and outputs it in the final array field `orders`.

4.4.8 Security

Security practices, if not respected, exposes the systems to several vulnerabilities that can result in data loss, leakage or system intrusion if exploited by malicious attackers. MongoDB released an official Security Checklist[25] to prevent those vulnerabilities.

Forgetting to tie down MongoDB's attack surface This code smell consists in leaving the default configuration regarding a MongoDB deployment. Indeed with the default configuration, it lacks key components to ensure security such as password or authentication. Phil Factor stressed that this may come from keeping the "*development server*" mentality. [48]

Improper user credential storage Developers may be tempted to hardcode credentials into the project thus inadvertently making them public. Consequently, anyone can access it and connect to the database, as suggested by Onyancha Brian Henry in its blog post named "*Best Practices for MongoDB Security*" [60].

No database access control No database access control designates the absence of measures to restrict access to the database program. Such way include setting up a firewall to allow only the specific application servers or using roles access control

[25] https://www.mongodb.com/docs/manual/administration/security-checklist/

inside the database. Alex Giamas even adds to automate the initialization of those at the startup of the mongod process via the `-auth` parameter [56].

No database user policy This code smell designates in the misconfiguration of the internal database user policy. Charanjit Singh warns about using only one user to handle all the operations and that such policy could be another layer of security if others have been breached [30].

No input sanitizing As for relational modelling and their SQL injection, NoSQL injection can also be an eventuality. This comes from user input not being sanitized that is directly passed into a query which can be worsened, if this user input ends up in a query using a $where operator. Consequently, Nedim Maric advocates to always sanitize user input in applications to avoid such problems [103].

No security patches Security patches are means to fix known vulnerabilities. Failing to apply them regularly exposes the system to automated attacks that could compromise it. A list of security best practices on Satori, a data-security platform insists on being always up to date with the latest MongoDB security update [26]

Not using LDAP for passwords rotations LDAP designates the automated handling of user permission updates when they are promoted or leave the company. Consequently, not setting one may result in unwanted users having access to the database as reported by Onyancha Brian Henry [60].

Server without authentication In an official MongoDB list "*10 mistakes that can compromise your database*", they suggest to use authentication [96]. Indeed, failure to set up authentication can result in unwanted accesses to the database

Too much network exposure Some sources (4) insisted on limiting the network exposure of the database server. Complementary to firewall measures, this includes restricting the number of ports that mongod processes are actually listening to, as suggested by this blog post on Big Data Analytics News [104].

Unencrypted communication This code smell consists in failing to set up the encryption of the database communication with other part of the systems. Those communications could be intercepted by undesired individuals as reported by this article on Satori. [26]

Unencrypted data Encrypting the internal database data allows to protect the data against breaches. MongoDB suggests such encryption at the document level. [96]

Using basic passwords Basic passwords are vulnerable to brute force attacks. Additionally, MongoDB does not offer a lockout mechanism to mitigate such attacks. ObjectRocket, a company specialized into database management and administration advises to use complex passwords to reduce this vulnerability [105].

[26]https://satoricyber.com/mongodb-security/11-mongodb-security-features-and-best-practices/

Using default Mongod ports Automated attacks tend to use the default configuration to target a system, such as the port numbers. Onyancha Brian Henry then suggests changing those port numbers to protect against those attacks. [60]

Using unofficial packages There exist drivers built on top of the MongoDB engine. However, those may not always be up to date with the latest MongoDB security updates. Additionally, such systems may not benefit of the maintenance effort MongoDB has. Consequently, Satori advises only to use the official MongoDB release. [26]

4.4.9 Sharding

Horizontal scaling is achieved in MongoDB via *sharding*, where *shards* (*i.e.,* data subsets) are distributed along a chosen field, the *shard key*.[27]

Low-cardinality shard key In a blog post detailing MongoDB sharding practices, Ankush Thakur highlighted the need to reflect on the need to plan ahead the shard key [19]. Indeed, as documents will be redirected to a specific shard based on that key value, the cardinality that is the "*internal value variations of that specific field among the collection's documents*". Consequently, choosing a low-cardinality shard key, meaning that it does not effectively discriminate a collection documents, could result in shards being unbalanced in terms of documents contained, compromising the horizontal scaling strategy. Poor shard key choices include a Latin letter or a single-digit number while recommended one can be a combination of multiple fields (compound shard key) or a lengthy chain of characters.

Monotonically increasing shard key Another way to poorly choose a shard key is to pick that is monotonically increasing such as a counter. As reported by Kvalheim in his book "*The Little MongoDB Schema Design Book*", choosing such a key will also result in unbalanced shards as eventually, all documents will be stored in the same shard [79]. Indeed, as the shards threshold are fixed, all the newer documents will be distributed to the same location. Likewise *Low-cardinality shard key*, this severely impacts the database distributivity.

Premature sharding The timing to start using shards is also important: sharding introduces new bottlenecks with the documents routing between the several shards. Consequently committing to shards too early may result in overall performance drop. Charanjit Singh, in a blog post named "*5 Mistakes Web Developers Make When Working with MongoDB*", also argues that performance bottlenecks that motivate them to use sharding actually comes from "*a bad schema and (bad) indexing*" [30].

Scatter-gather queries In a sharded environment, scatter-gather queries refer to ones that cannot be routed based on the shard key, which means that the query document filter uses fields that are not the shard key. Such queries must then be broadcast to each shard instead to the relevant one, which adds performance

[27] https://www.mongodb.com/docs/manual/sharding/

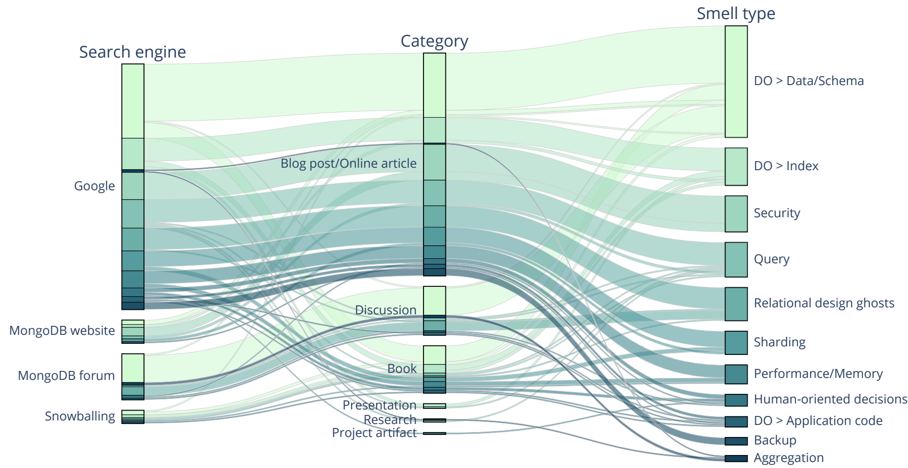


Figure 4.6: Parallel categorization of source category and smell type

overhead. In a blog post about "*Performance Best Practices: Sharding*" Mat Keep and Henrik Ingo reveals that such queries does not scale linearly with the number of shards added to the systems. [74] Consequently, they recommend using shard keys inside queries, except for aggregations.

Unshardable collection An unshardable collection designates a collection that does not contain an adequate shard key. While not every collection should be sharded, a collection that is expected to have a sizable number of documents should be sharded as reported by Kyle Banker *et al.* in their book "*MongoDB in Action*". [24] Therefore, it is recommended that collections with anticipated significant growth should have a sharding key strategy.

Working set exceeds memory The working set pertains to the data and indexes that are stored in memory for efficient access, depending on their usage frequency. If the working set surpasses the memory capacity, it will necessitate accessing physical disk storage, which is significantly slower. This scenario may indicate a suboptimal sharding strategy or serve as a reminder to consider implementing one, as indicated by the "*MongoDB on AWS*" guidelines [17].

RQ1 *What types of MongoDB smells have been proposed in the community?* We identified 11 smell categories with 76 different code smells: *Aggregation issues*, *Design oversight* with subcategories `Application code`, `Data/Schema` and `Indexes`, *Performance/Memory issues*, *Sharding issues*, *Backup issues*, *Query issues*, *Security issues*, *Relational design ghosts*, and *Human-oriented decisions*.

4.5 RQ_2: Source Types' Classification

At the end of the open coding process, we identified 87 sources discussing MongoDB code smells. We grouped them into categories to have an overview of all these sources. Similarly to the classification proposed by Garousi *et al.* [52], we differentiate 'grey' sources such as *Blog posts/Online articles*, *Discussions*, *Presentations*, *Project Artifacts*, and *Course materials*. As our study is not only limited to the 'grey' literature, we also consider 'white' sources such as *Research* papers, Master/Bachelor theses, and *Books*.

Table 4.1 breaks down the source types' classification and the distribution of the sources. It shows the number of sources manually reviewed for each source type and the number of sources with MongoDB code smells.

Most of the sources come from the categories *Blog posts/Online articles* (60%) and *Discussions* (27%), which highlight the community aspect of our MLM study.

Despite our searches for scientific papers on multiple sources, the white literature is minimal in MongoDB smells. While 12 papers passed our initial filtering, after manually reviewing all of them, we found only one paper mentioning MongoDB smells. In this paper, Mahajan *et al.* [85] study the energy efficiency of relational and NoSQL databases via query optimizations. They do not mention the smells explicitly but investigate how indexed vs. non-indexed queries affect the performance in MongoDB. Hence, we included it in our MLM.

We also notice a significant number of books among the sources. These books are *MongoDB in Action* [24] (with a dedicated chapter for Design patterns in its appendix), *The Little Mongo DB Schema Design Book* [79], *Mastering MongoDB 6.x* [56], *Practical MongoDB* [44], *50 Tips Tricks MongoDB Developers* [38], and *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage* [27].

Fig. 4.6 shows the categorization of a source from its `Search engine` to its `Category` and smell `Type`. Again, most smells come from *Blog posts/Online articles* (60%) and *Discussions* (27%). However, some smell types (i.e., `Backup` and `Security`) only have sources from the *Blog post/Online article* category.

Surprisingly, only a small part of the code smell catalog was found using the official MongoDB website (5). However, we still used sources from this website (15) as we reached them from *Google*. The same phenomenon can be observed in the Research category. While the academic search engines did not find relevant sources, we still reached some sources in the Research category through *Google*.

We also investigated how many smells are *induced* by different source types: a source *SO* induces a smell *SM* if *SO* was used as an input in the code smell extraction step to extract *SM*. If a blog post talks about five MongoDB smells and we extracted them, we say it induces five smells.

Fig. 4.8 shows the distribution of the number of induced smells for each source type. As most of the smells are mentioned in blog posts, it is expected that they also induce more smells. It is interesting to notice that `Books` took the first place, which can be attributed to their thoughtful nature.

Another observation is that the median number of induced smells is the closest to one for the *Discussion* category, which acknowledges that a discussion typically revolves around a specific problem or smell.

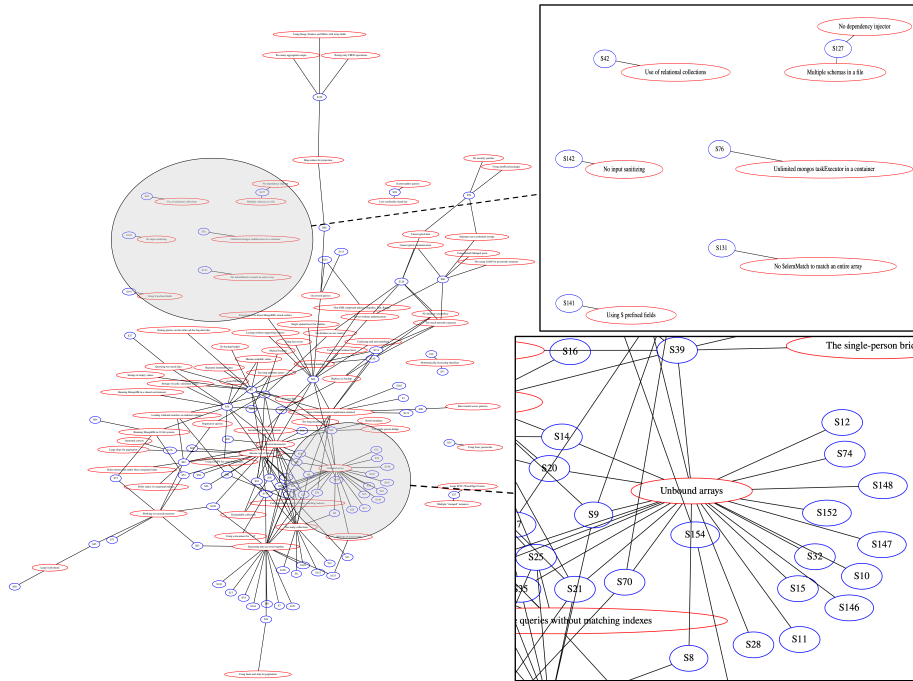


Figure 4.7: Network representation of smells and sources

The source with the highest number of smells was a blog post on *Big Data Anti-Patterns* by Marc Kenig, published in 2019 [8], inducing 13 MongoDB smells. The next notable source was the *Practical MongoDB* book [44] which induced 11 smells of the catalog. The median number of induced smells was 1.5 considering all sources.

To have a broader picture of the relationships between sources and code smells, we constructed a graph that can be seen in Fig. 4.7. The blue nodes are the sources, and the red nodes are the code smells. We use the same *Sx* notation for each source as we identify it in the source list of the online appendix. An edge between source *Sx* and smell *SM* indicates that *Sx* induces *SM*. Globally, we can observe a coupled network between sources and smells, with two notable exceptions. The first one is highlighted by the circle in the top left corner of the figure. These are the isolated source-smell pairs, typically showing Q&A or forum posts about a specific smell. The second one is depicted in the bottom-right circle. It denotes smells mentioned in multiple sources.

For example, 24 sources induce the *Unbound Array* smell, with 11 only about this specific smell. Similar "hot spots" are the *Abusive use of indexes* and the *Separating data accessed together* smells with 22 and 19 sources, respectively.

Finally, Fig. 4.9 shows a bar chart of the sources according to their publication year.

MongoDB keeps track of its release notes starting from version 1.2, released in

Table 4.1: Source Types classification

Category	Source Type	# Sources Reviewed	# Sources w/ Smells
Blog post/Online article	Blog post/Online article	92	50
	White paper	3	2
Discussion	Forum post	28	22
	Q&A	6	2
Presentation	Presentation video	9	1
	Presentation slides	2	1
	Presentation appendix	1	-
Project Artifact	Documentation	3	-
	Technical report	1	-
	Bug report	1	-
Research	Published paper	12	1
	Master's thesis	1	-
	Bachelor's thesis	1	-
Book	Book	11	6
	Book chapter	1	1
Course	Course/Certification material	2	-
Total		**174**	**87**

2009. [28] However, the first release is 0.0.3 from 2008 in its Git repository.[29]

Our earliest sources are from 2010 to 2011. These are the first editions of the books *50 Tips and Tricks for MongoDB Developers: Get the Most Out of Your Database* (2011), *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage* (2010), *MongoDB in Action* (2010), and *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage* (2010).

Only a few, 1–5 sources appear each year until 2018. These are typically blog posts on MongoDB "best practices" such as [39, 14, 64, 16, 48]. [30] Interesting to highlight is the *14 Things I Wish I'd Known When Starting with MongoDB* InfoQ article by Phil Factor [48], published in 2018, which becomes cited by other sources too.

A sharp increase can be observed from 2019 with 7 (2019), 12 (2020), 17 (2021), and 27 (2022) new sources. Interesting to note here is that MongoDB published blog posts on *Performance Best Practices: Indexing* [90] and *Performance Best Practices: Sharding* [74] in 2020, then started a blog series on *MongoDB Schema Design Best Practices* [69] and *Schema Design Anti-Patterns* in 2022. The posts generated related

[28] https://www.mongodb.com/docs/manual/release-notes/
[29] https://github.com/mongodb/mongo/releases/tag/r0.0.3
[30] https://github.com/vaquarkhan/Technology-best-practices/blob/master/nosql/mongodb/README.md

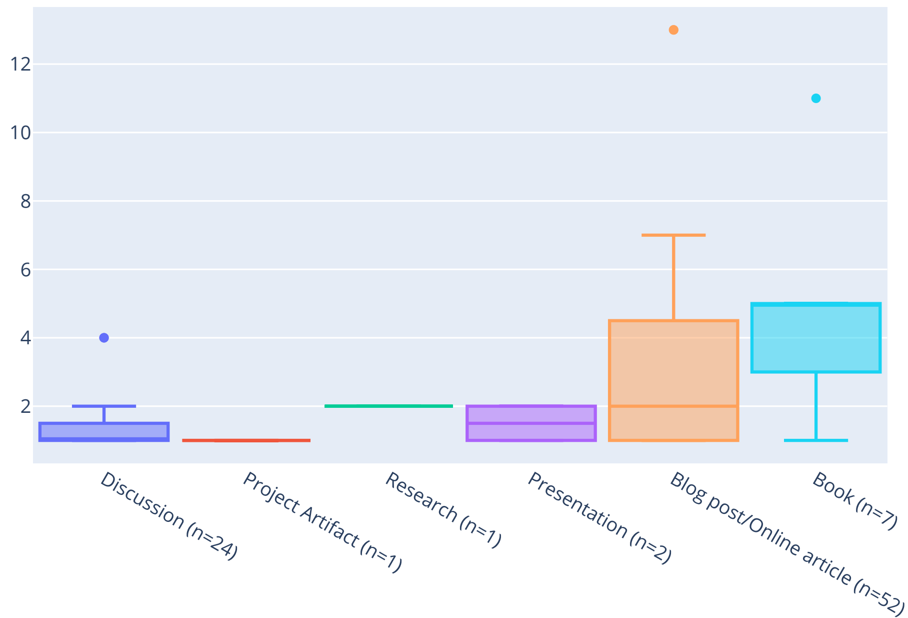


Figure 4.8: Induced smell per category

discussions on their forums, e.g. [31] [32].

> RQ2: *Where are MongoDB smells discussed by the community?*
> Most of the sources come from the categories of *Blog posts/Online articles* (60%) and *Discussions* (27%). While being less numerous, *Books* (8%) tend to hold more smells, contrary to the *Discussions* which mostly revolve around a specific code smell. A sharp increase in published sources can be observed since 2019.

4.6 Discussion

In this section, we discuss implications for researchers and practitioners. We motivate this discussion by presenting example smell instances in open-source systems.

4.6.1 Illustrative examples in open-source systems

As a preliminary attempt to find smells in open-source systems, we implemented static analyzers for the **Relational design ghosts** category. We chose this category given its importance, as many sources referred to these common mistakes made by developers who came to the NoSQL world with a relational database background.

[31]https://www.mongodb.com/community/forums/t/best-way-to-store-products-and-its-attributes-information-in-mongodb/130338

[32]https://www.mongodb.com/community/forums/t/is-it-better-to-make-frequent-read-operations-or-to-pull-all-data-periodically/151429/3

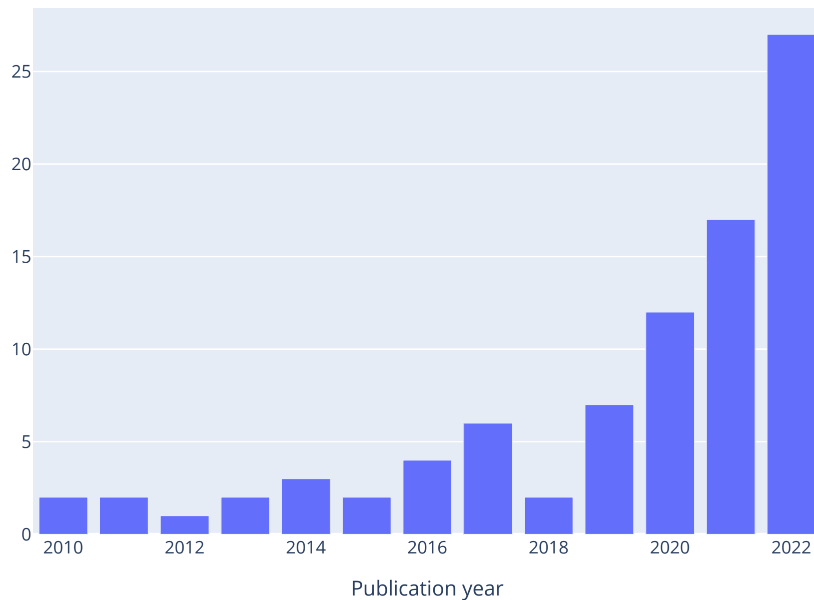


Figure 4.9: Distribution of sources according to publication year

Detection tool

We relied on CodeQL as a static analysis framework which was explained in Chapter 1.2.

We implemented QL queries for the **Use of empty values**, **Separating data accessed together**, **Use of relational collections**, and **Relying on transactions** smells. Their source code is publicly available in our GitHub repository [34].

Listing 4.39: CodeQL query to find *Storage of empty values* instances

```
1   from MethodCallExpr mce, ObjectExpr queryFilter
2   where mce.getMethodName() = "insert" and
3       mce.getAnArgument() = queryFilter and
4       queryFilter.getAProperty().getInit().getStringValue() in ["null", "
            undefined", "", "''", "[]"]
5   select mce, "this inserts holds null values"
```

Listing 4.39 shows an example query of the *Storage of empty values* smell. The query has three main clauses: `from`, `where`, and `select`.

It searches for a method named `insert` (Line 2), the dedicated method to insert documents in MongoDB and Mongoose drivers. It takes the argument of this method call (Line 3), which was selected as an object expression in the from clause. Then it tests the argument for an empty value (Line 4). Finally, the `select` clause lists expressions for the warning message, *e.g.*, `MethodCallExpr` variable in the example (Line 5).

More complicated queries require CodeQL features such as classes, predicates, data flow analysis, or taint tracking. Our implementation of the four smells of the *Relational design ghost* category with two drivers (MongoDB Node Driver and

Mongoose) tries to handle dynamic features of JavaScript in about 700 lines of QL code, available in the online appendix [34].

Smell instances

We analyzed top-starred open-source systems on GitHub (see the online appendix [34]) and found interesting smell instances, which we share as motivational examples for future studies.

Use of relational collection The **voluntarily/vly2** repository hosts the source code of the *Voluntarily*[33] volunteering platform in New Zealand. It is a "*matchmaking platform to connect awesomely skilled volunteers with schools who need a hand.*" We found a `Use of relational collection` smell in its codebase as follows.

The `getMembersWithAttendedInterests` arrow function uses an aggregation on the `Member` collection.[34] This query has 65 lines of code that we briefly summarize here.

First, there is a `$lookup` operator (L14) to perform a left outer join between the `Member` and `InterestArchive` collections using the `person` attribute and outputting the array as `archivedInterests`. Later another `$lookup` (L37) uses this array output to perform a join with the `archivedopportunities` collection. This means that `InterestArchive` was used to access `archivedopportunities`, similar to a join table in relational modeling. The *Use of relational collection* smell is exactly about this, as the `$lookup` operator is known to be slow and resource-intensive [42].

The project's developers implement an M-N relationship through separate collections, representing *members' interests* in *opportunities*. An alternative would be to embed a *member*'s *opportunities* into its corresponding document.

Storage of empty values The **impronunciable/hackdash** repository is a collaborative dashboard to organize hackathon ideas. The method *setCachedPage*[35] inserts a document into the *pages* collection (see Listing 4.40).

The document inserted has a value attribute with an empty string ('') on L89. This is a simple instance of the `Storage of empty values` smell. Due to MongoDB's schema flexibility, the empty attribute can be omitted.

Its absence can be queried/updated later.[36] Consequently, empty values unnecessarily complicate queries. An alternative would be to omit the attribute from the inserted document.

Listing 4.40: An example query with a *Storage of empty values* smell in **impronunciable/hackdash**

```
82    function setCachedPage(url){
83
84      db.collection('pages', function(err, collection) {
85        if (err){ return console.log(err); }
```

[33] https://www.voluntarily.nz/
[34] https://github.com/voluntarily/vly2/blob/master/server/api/statistics/statistics.lib.js/#L8
[35] https://github.com/impronunciable/hackdash/blob/master/seo.js/#L85
[36] https://www.mongodb.com/docs/manual/reference/operator/query/exists/

```
86
87          collection.insert({
88            key: url,
89            value: '',
90            created: new Date(),
91            pending: true
92          }, { w: 0 });
93
94        });
95      }
```

4.6.2 Implications

MongoDB is a relatively "young" 14-year-old NoSQL database compared to relational database management systems, which have been around for over a quarter of a century. The first release of PostgreSQL, for example, dates back to 1996. It is interesting to observe how a discussion in the community emerges around common code smells and antipatterns as the database management system evolves and its popularity increases. We found the first "Best practices" and then "Antipatterns" in books from 2010, when MongoDB was only two years old. However, it took additional eight years to see a more apparent impact on the community.

It is not the purpose of our study to investigate what happened after 2018. Still, it is interesting to observe that it takes a significant amount of time for a community to recognize and start naming its frequent maintainability issues. Once they do so, it brings more attention to such problems and fosters discussions about them. As noticed in previous research, by not knowing the state of practice, practitioners tend to "reinvent the wheel" and use various names for existing smells [53]. We have seen the same effect when we coded all the sources and collected 242 labels to end up with 76 smells after merging conceptually similar or duplicated labels.

We believe we are at the right time for a multivocal mapping study. The numbers indicate that a vast amount of (fresh) knowledge has been gathered in the grey literature of online sources: 76 smells in 87 sources and a sharply increasing trend in new sources. It is time to take a reflective step, map and organize this knowledge. However, the research community still needs to recognize the importance of this field.

There are several ways in which researchers could help practitioners and vice versa. For example, we found many forum discussions where developers seek solutions to these smells. Research studies could help understand the contexts where MongoDB smells occur and cause bugs, quality, maintenance, or additional problems.

A first step in this direction is the study of Mahajan *et al.* [85] on the impact of index usage on energy efficiency. Automated tools and AI approaches could help detect and fix the smells. Our CodeQL queries show promising potential in this direction as a static analysis approach.

4.6.3 Threats to Validity

In this section we reflect on the threats to the validity of our catalog and our MLM methodology.

Construct validity To build our MongoDB code smell catalog, we used various search engines to foster source diversity. In order to mitigate Google location search bias, we relied on VPN (Tor network) set to locations on every continent. Eventhough we considered several academic search engines (Google Scholar, IEEE Xplore, ACM Digital Library), very few sources were kept from those (12). Consequently, our study relies for the most part on “grey” literature and is exposed to related threats to validity. When peer-reviewed literature is unavailable, studying grey literature is common in software engineering. [53]

Internal validity During our process of extracting information from our sources, we encountered challenges regarding the quality and subjectivity of the content. In order to handle that, we applied quality criteria as advised by Garousi *et al.* [52].

Also, the manual smell extraction process from the sources is exposed to subjectiveness. To counteract this, two people (Csaba Nagy and Boris Cherry) executed this task in parallel and merged their finding in the open card sorting phase. That phase is also exposed to subjectiveness.

To mitigate it, we held several sessions where each participant could propose a new smell, delete one or merge several. We also did not finish until both participants agreed on the complete content of the list.

External validity Our study aims to be a global description of the discussed smells by the community. We considered sources from various origins such as blog posts or forum discussions. However, we rely on sources that are indexed by search engines. Unindexed content, such as MOOC or paywalled sources are not considered by this study.

Aditionnaly, as the sources discussing MongoDB code smells are on an increasing trend, our results take place in a certain timeline, which is the period in which it was conducted, during the year 2023. Consequently, MLM studies on MongoDB smells undertaken after this date might yield additional sources and smells.

4.7 Conclusion

We presented a catalog of MongoDB code smells, which we distilled by performing a multivocal literature mapping (MLM) study, using various search engines, gathering 1,498 sources, and manually inspecting 174 sources. The catalog includes 76 smells classified into 11 categories. Many smells induce performance issues, others hinder design and development, and others induce non-trivial security issues.

We opted for an MLM instead of a systematic literature review (SLR) because the field of MongoDB smells is largely unexplored, and peer-reviewed literature simply does not exist yet. However, it is a hot topic among developers, as proven by the numerous “grey literature” sources (developer blogs, forums) that we found in our study. There is also a noteworthy amount of information stemming from books. It all points to topics that are highly relevant for developers and practitioners, but have received little attention from researchers so far. Furthermore, the historical analysis presented in Section 4.6 shows a sharp increase of interest in the topic in recent years.

4.7.1 Roadmap

In this chapter, we built a MongoDB code smells and antipatterns catalog from grey sources. Excepting some smells that are Mongoose-specific, this calaog aims to be language and driver independent.

Concerning the next chapter, Chapter 5, we will describe how we built a static tool on top of the approach described in Chapter 4 detecting some smells reported in this chapter.

CHAPTER 5

SMEAGOL

5.1 Introduction

In Chapter 4, we were able to identify specific MongoDB code smells. Those code smells can occur in a variety of artifacts `type` (*e.g.,* database access, running environment or internal configuration) and also stem from many causes (*e.g.,* inadequate modeling principles or query mechanism misuse). However, those code smells lack a dedicate identification method to support practitioners.

In this chapter, 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. It detects code smells from database accesses by using multiple information sources as 1) index declaration, 2) related collection documents structure and 3) code surrounding the database access. SMEAGOL supports JavaScript, the most popular programming language used with MongoDB [25]. It is built on top of the approach designed in Chapter 3 thus using CodeQL whose functioning was described in Section 1.2 and supporting the two npm MongoDB drivers: MongoDB NodeJS native driver[1] and Mongoose,[2].

Listing 5.1 shows JavaScript code with a query to find a document in this collection containing a code smell, *Querying too much data* smell instance. For this example we reuse the `countries` collection defined in Chapter 4, a reminder can be found in Fig. 5.1.

We already presented MongoDB and driver usage in Section 1.1.1.

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 the *name* field (Line 8-10) in order to print its `currency` one (Line 11), and closes the database connection (Line 15). This is

[1]https://www.npmjs.com/package/mongodb
[2]https://www.npmjs.com/package/mongoose

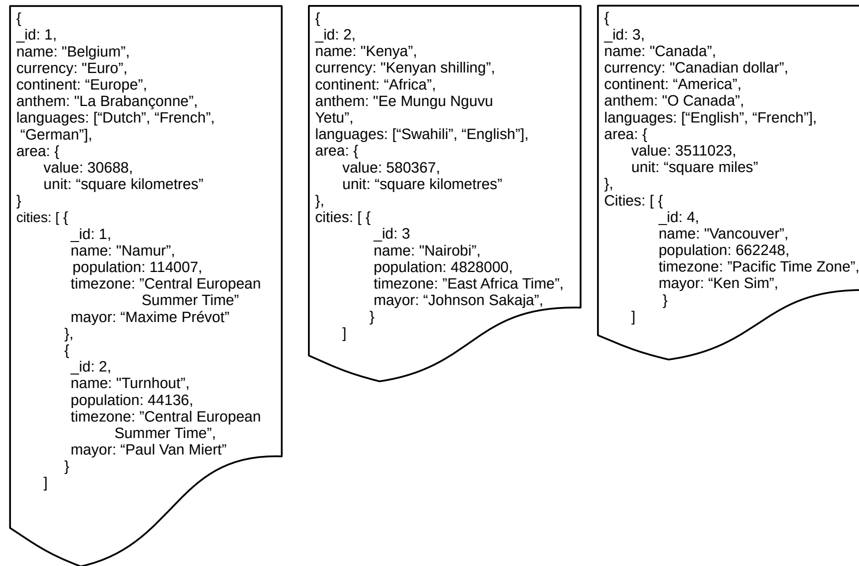


Figure 5.1: Countries collection

Listing 5.1: Database access code example with antipattern

```
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 kenyaCountry = await db
9        .collection("countries")
10       .findOne({ name: "Kenya" });
11     console.log(kenyaCountry.currency);
12   } catch (err) {
13     console.log(err);
14   } finally {
15     await client.close();
16   }
17 }
18 run().catch(console.dir);
```

an instance of the *Querying too much data* code smell. Indeed, as we can see in the Fig. 5.1, each document from the country's collection hold other fields than `currency` although in the following code, only the latter is used.

We can use MongoDB projections[3] to fix it and specify the fields to return from a fetched document, as in Listing 5.2.

Listing 5.2: Database access code example fixed

```
7  /* same as line 1-7 */
8        let kenyaCountry = await db
9          .collection("countries")
10         .findOne({ name: "Kenya" }, { currency: 1 });
11       console.log(kenyaCountry.currency);
12 /* same as line 11-18 */
```

We have already explained the functioning of projections in Section 1.1.1, this code snippet indicates to retrieve only the `currency` field (Line 10).

This chapter presents the internal structure of SMEAGOL and how it handles the information source variety. We also illustrate how we picked the 8 smells to detect. We demonstrate SMEAGOL by analyzing 704 open-source projects in which we found 44,566 smell instances and discuss the retrieved instances.

All the results discussed in this chapter as well as the source code is available in the online appendix [34].

5.2 Tracked code smells

We chose to follow a static analysis method to build on the approach developed in Chapter 3 and to apply our technique to a wide range of open source systems, where the source code is often the only available information. As we opted for this detection strategy, we first classified smells according to its detection scope in the software system. The classification was made in multiple rounds between 2 people, Csaba Nagy and Boris Cherry and stopped when both agreed on it. The detection scope was characterized in 5 levels, we briefly describe them in the following.

(i) *Code* designates when the smell can be directly detected in the application without any insights about the system database structure.

(ii) *Schema* qualifies detection requiring knowledge about the database structure, such as collection names, index or fields structure.

(iii) *Data* detection level needs information about the fields value from the collection documents.

(iv) *Configuration* identifies detection needing to have access to the running environment on which the system runs.

(v) *Process* level requires knowledge about the development process or decisions.

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

Table 5.1 shows the result of this classification. The **Category** and **Code Smell** columns corresponds to the smell category and name, while the **Detection Scope** exposes the result of the classification process.

Table 5.1: MongoDB database access methods

Category	Code Smell	Detection Scope
Aggregation	**Lookup without supporting indexes**	Schema and Code
Aggregation	**Too many aggregation stages**	Code
Aggregation	**Map-reduce for projection**	Code
Backup	Manual backups	Process
Backup	No backup budget	Process
Backup	Replicas as backup	Configuration
DO > Application code	**Immortal cursors**	Code
DO > Application code	No dependency injector	Code
DO > Application code	Testing only CRUD operations	Code
DO > Application code	Testing queries on the entire ad-hoc big data lake	Configuration and Code
DO > Application code	**Querying too much data**	Schema and Code
DO > Data/Schema	**The single-person bridge**	Code
DO > Data/Schema	**Multiple schemas in a file**	Code
DO > Data/Schema	Repeated immutable data	Data
DO > Data/Schema	**Using a document for _id**	Schema
DO > Data/Schema	**Flat raw data**	Schema or Code
DO > Data/Schema	Storage of easily calculated values	Data or Code
DO > Data/Schema	**Inconsistent attribute structure**	Data or (Schema and Code)
DO > Data/Schema	Data oriented instead of application oriented	Process
DO > Data/Schema	**Too many collections**	Code
DO > Data/Schema	Bloated documents	Data
DO > Data/Schema	Unbound arrays	Data and Schema
DO > Index	**Index intersection rather than compound index**	Schema
DO > Index	**Prefix index of compound indexes**	Schema and Code
DO > Index	**Non-ESR compound indexes (Equality - Sort - Range)**	Schema and Code
DO > Index	**Abusive use of indexes**	Schema and Code
Human oriented decisions	Bias toward access patterns	Process
Human oriented decisions	**Using $ prefixed fields**	Schema
Human oriented decisions	Human-readable values	Data
Human oriented decisions	**Too long attribute names**	Data or Code or Schema

Table 5.1: MongoDB database access methods

Category	Code Smell	Detection Scope
Human oriented decisions	**Too long document keys**	Data or Code or Schema
Performance/Memory	Large WTC (WiredTiger Cache)	Configuration
Performance/Memory	Multiple mongod instances	Configuration
Performance/Memory	Running MongoDB in a shared environment	Configuration
Performance/Memory	Unlimited mongos taskExecutor in a container	Configuration
Performance/Memory	Using fast writes	Configuration
Performance/Memory	Large read-ahead	Configuration
Performance/Memory	Using GridFS for small binary data	Data and Configuration
Performance/Memory	Running MongoDB on 32-bit systems	Configuration
Query	**Avoid $Where**	Code
Query	**Confusing null and undefined**	Code
Query	**Large skips for pagination**	Code
Query	**No $elemMatch to match an entire array**	Schema and Code
Query	**Single update/insert for batches**	Code
Query	**Sorted monkeys**	Code or Data
Query	**Using $limit without $sort**	Code
Query	**Using $map $reduce and $filter with array fields**	Schema and Code
Query	Using limit and skip for pagination	Code and Runtime
Query	**Negation in queries**	Code
Query	**Leading wildcard searches on indexed columns**	Schema or Code
Query	**Uncovered queries**	Schema and Code
Query	**Case-insensitive queries without matching indexes**	Schema and Code
Relational design ghosts	**Storage of empty values**	Data or Code
Relational design ghosts	**Use of relational collections**	Code
Relational design ghosts	**Relying on transactions**	Code
Relational design ghosts	**Separating data accessed together**	Code
Security	Forgetting to tie down MongoDB's attack surface	Configuration
Security	No input sanitizing	Code
Security	No security patches	Configuration
Security	Not using LDAP for passwords rotations	Configuration

Table 5.1: MongoDB database access methods

Category	Code Smell	Detection Scope
Security	Using basic passwords	Configuration
Security	Using default Mongod ports	Configuration
Security	Using unofficial packages	Configuration
Security	Improper user credential storage	Configuration
Security	No database access control	Configuration
Security	Unencrypted communication	Configuration and Code
Security	Unencrypted data	Configuration or Data
Security	No database user policy	Configuration
Security	Server without authentication	Configuration
Security	Too much network exposure	Configuration
Sharding	Low-cardinality shard key	Data
Sharding	Scatter-gather queries	Data and Code
Sharding	Unshardable collection	Data or Schema
Sharding	Monotonically increasing shard key	Code
Sharding	Premature sharding	Process
Sharding	Working set exceeds memory	Data

Configuration and process detection-level smells are outside the scope of this chapter, as our tool relies on static analysis. Initially, we considered including data-level smells, but we ruled them out after initial runs showed that very few MongoDB documents had explicit field values. Consequently, we focused on detecting code and schema-level smells, starting with 39 smells. However, after conducting some dry runs, we decided to exclude 7 smells that were too complex to handle. This left us with 32 smells, for which we implemented a detection mechanism. The tracked smells names are highlighted in bold in Table 5.1.

5.3 SMEAGOL

SMEAGOL is a MongoDB rule-based static analysis tool that detects MongoDB code smells in JavaScript applications. Concerning our static analysis framework, we rely on CodeQL that we already introduced in Section 1.2. Fig. 5.2 presents its general workflow.

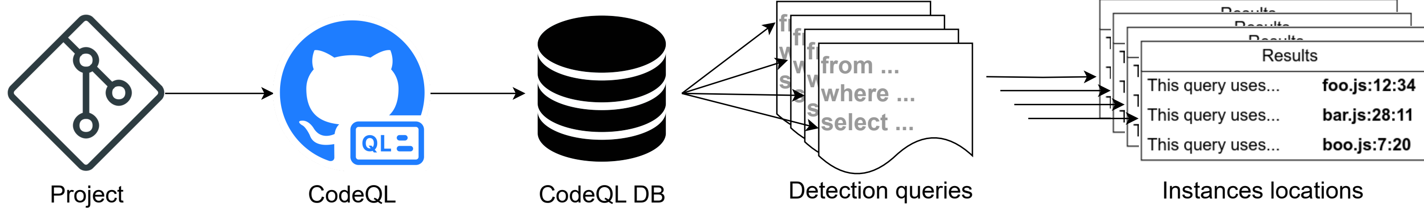


Figure 5.2: SMEAGOL workflow

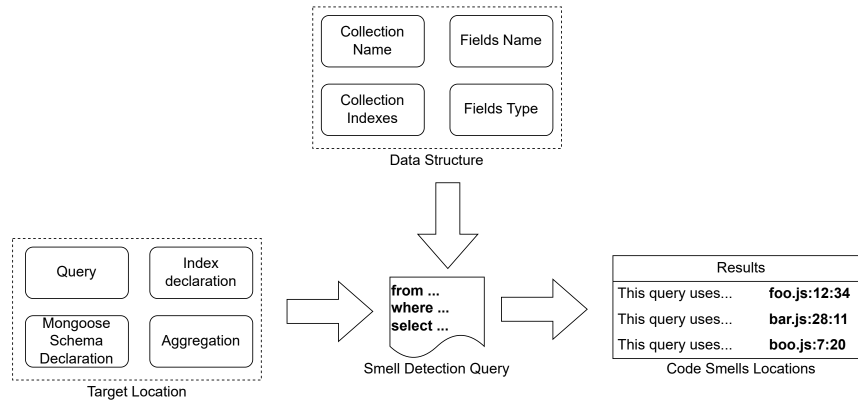


Figure 5.3: SMEAGOL individual query execution

SMEAGOL consists in the sequential execution of the multiple detection queries, each corresponding to a defined code smell. First, SMEAGOL needs to analyze the project to create its corresponding CodeQL database in order. Then, it runs the detection queries against the project CodeQL database to check for potential smell instances. Finally, it outputs the eventual smell instances locations, which can come in various formats (SARIF, CSV, etc.).

Fig. 5.3 presents the execution context of a single detection query in SMEAGOL. First, it processes candidates the target locations, which designates the possible candidates for a given code smell instance. As indicated in the figure, those can occur in various parts of the software system, including queries, index declarations, Mongoose Schemas or aggregations.

Then, it may use information from outside the smell candidate to gain insights about the data structure that the system uses to improve the precision of the detection. This information may also come from various sources and are collection name and indexes or document field names and types. We develop more on this topic in Section 5.3.1.

Afterwards, it runs the detection query to identify the code smell instances from the candidates. There is a detection query for every tracked code smell and we explain their general structure in Section 5.3.2.

Finally, the code smells instances locations are output from the query execution in separated reports for every code smell. Every code smell instance location comes with an explicative message about the smell to assist the maintainers running SMEAGOL.

5.3.1 Schema inference

As said in Section 5.1, we actually need to consider several different sources of information to our schema inference process. In the following, we detail how we gather each data structure information under the same abstraction.

Collection Name

In MongoDB or Mongoose projects, we consider 2 main ways to identify a collection name. The first one relies on the queries receiver, as in Listing 5.3 with the same code snippet as Section 1.1.1.

Listing 5.3: MongoDB find query example

```
1    const britishAuthor = await db
2        .collection("authors")
3        .findOne({ surname: "Lindholm" });
4    console.log(britishAuthor.surname);
```

There, we can use the parameter of the `collection` call, which is `authors` as the name of the collection.

In Mongoose, the collection name, which is the model name (see Section 1.1.1) can be collected during the Model instantiation, as in Listing 5.4.

Listing 5.4: Mongoose model instantiation

```
1    const AuthorSchema = new mongoose.Schema({
2      /* schema fields here */
3    });
4
5  const Author = mongoose.model("authors", AuthorSchema);
```

There, by doing dataflow analysis, we can link schema attributes to a model instantiation and use the first parameter of the `mongoose.model` call.

Collection Indexes

Concerning collection indexes, we also identified 2 ways to extract index information from MongoDB and Mongoose source code. As precised in Section 1.1.1, indexes are declared in MongoDB with a dedicated method call, such as displayed in Listing 5.5.

Listing 5.5: MongoDB index declaration

```
1    await db.collection("authors").createIndex({ surname: 1 });
```

Here, we can obtain the concerned collection similarly to Section 5.3.1 and use the first parameter of the `createIndex` to obtain the indexed field.

In Mongoose, the fields can also be declared as indexed in the Schema declaration, as in Listing 5.6.

Listing 5.6: Mongoose Schema index declration

```
1    const AuthorSchema = new mongoose.Schema({
2      surname: {type: String, index: true},
3      /* other fields here */
4      });
```

There, we infer the indexed nature of a field by checking for the `index` property in the object declaring the field type, in that case on line 2.

Field Name

To extract field names relative to a collection, we use four main sources of information: Mongoose Schema property declaration, explicit inserted documents (from a MongoDB or Mongoose query), query filters and object property access.

As introduced in the Section 1.1.1, a Mongoose Schema declares the shape of its document using a JavaScript object, as illustrated in Listing 5.7.

Listing 5.7: Mongoose Schema Declaration

```
1   const AuthorSchema = new mongoose.Schema({
2       surname : {type: String, index: true},
3       givenNames : [String],
4       birthPlace : {
5           country : String,
6           city : String
7       }
8   });
```

Here, we exploit the property name of the Schema object as field name for the given model name. Consequently, we obtain `surname`, `givenNames`, `birthPlace`, `birthPlace.country` and `birthPlace.city` from this Schema declaration.

We refer to MongoDB explicit document insertion as using a plain object expression as the parameter for the insertion method call. Such an example can be found in Listing 5.8.

Listing 5.8: MongoDB explicit document insertion

```
1   await db.collection("authors").insertOne(
2       {
3           surname : "Lindholm",
4           givenNames : ["Margaret", "Astrid"],
5           birthPlace : {
6               country : "United States",
7               city : "Berkeley"
8           }
9       });
```

An alternative in Mongoose query language can be found in Listing 5.9.

Listing 5.9: Mongoose explicit document insertion

```
1   const Author = require("authorModel.js")
2
3   let authorDocument = new Author({
4       surname : "Lindholm",
5       givenNames : ["Margaret", "Astrid"],
6       birthPlace : {
7           country : "United States",
8           city : "Berkeley"
9       }
10  });
11  await authorDocument.save();
```

We apply here the same approach as for Mongoose Schema declaration to extract the property names from the plain object expression that are directly used as parameter for the MongoDB query in Listing 5.8 (Line 2-7) or for the Mongoose Model instantiation in Listing 5.9 (Line 3-10). Applying it for both queries gave us the field names `surname`, `givenNames`, `birthPlace`, `birthPlace.country` and `birthPlace.city`.

As explained in Section 1.1.1, MongoDB relies on a filter object to select the desired documents in a query. Consequently, each non-empty filter mentions at least one field name from a collection. Extracting this field identifier allows to learn more about a field name belonging to the mentioned collection. Listing 5.10 exemplifies a find query with a non-empty filter object.

Listing 5.10: MongoDB find query with a filter

```
1  let returnedDocument = db.collecton("authors").find({
2    givenNames : {
3      $size: 2
4    }
5  });
```

In this query, we observe that the `givenNames` attribute is used to filter the document search, employing the `$size` operator to match arrays with a specified number of elements, in this case, 2. By extracting the names from the properties at the root of a filter operator, we can derive the field names from the query, such as `givenNames` in this example.

Also, we may encounter queries inserting documents or using filters that do not explicitly expose their value, as in Listing 5.11

Listing 5.11: MongoDB implicit field name

```
1  function getAuthorByFilter(filter) {
2
3    let authorDocument = await db.collection("authors").find(filter);
4
5    console.log(authorDocument. surname );
6  }
```

Here, a function `getAuthorByFilter` is defined with a single parameter, `filter` (Line 1). That single parameter is used as a filter for the `find` query (Line 3). Finally, the `surname` property of the returned document is then output to the console with a `console.log` call (Line 5).

From this example, no information is extracted from the filter regarding any collection field. However, we can infer field names from the property accessed on the variable holding the returned document. We achieve that by performing dataflow analysis on objects returned by a `find` query (from MongoDB or Mongoose) to track every property accessed on it. By following this approach, we can extract `surname` as a field from the collection `authors`.

Field Type

In order to obtain information about a type associated with a given field for collection, we rely on two approaches: Mongoose Schema type declaration and CodeQL

type inference tool. Concerning the Mongoose Schema type declaration, we exploited the field associated type, as showed in Listing 5.12.

Listing 5.12: Mongoose Schema type Declaration

```
1   const AuthorSchema = new mongoose.Schema({
2     surname: { type: String , index: true},
3     givenNames: [String] ,
4     birthPlace: {
5       country: String ,
6       city: String
7     }
8   });
```

They are typically expressed as literals (Lines 3, 5, 6) or as an object expression (Line 2). After extraction of the field types, we obtain the following results: `String` for `surname`, `birthPlace.country` and `birthPlace.city`, and `String` array for `givinNames`.

To infer the type with CodeQL, we rely on the predicate `getTheType`. [4] This predicate tries to infer a primitive type for a given expression, being `string`, `number`, `boolean` or `object`. If the predicate was not able to find a type for a given field, we mark it with the type `any`, as for any type.

5.3.2 Detection queries

Concerning the structure of SMEAGOL, we opted to define a detection query for each code smell. With abstraction for queries or index declaration in separate files, this allowed to only express the detection rule in the file to foster readability and maintainability. Fig. 5.4 illustrates our query definition process.

First, we used as input our code smell catalog obtained in Chapter 4 to define the detection rule for a smell. This detection rule takes the form of a brief plain text description. Here is an example for "*Uncovered query*": "A find query whose filter objects mentions a field that has not been declared as an index, except for the `_id`".

Then, we implement this detection rule in a CodeQL query using the data structure inference library that we defined. The detection query derived from the detection rule corresponding to "*Uncovered query*" can be seen in Listing 5.13

Listing 5.13: Detection query for "*Uncovered query*"

```
1   from Operator queryOperator
2   where
3       not isStronglyIndexed(queryOperator.getFullyQualifiedName(),
            queryOperator.getCollectionName()) and
4       not queryOperator.getFullyQualifiedName() = "_id"
5   select queryOperator.getParentQuery(), "This query is not covered.
        Consider indexing the attribute/querying an indexed attribute"
```

In this query, we do not use the corresponding class directly. Instead, the `from` clause uses the `Operator` class (Line 1), who directly corresponds to an operator

[4] https://codeql.github.com/codeql-standard-libraries/javascript/semmle/javascript/dataflow/TypeInference.qll/predicate.TypeInference\protect\T1\textdollarAnalyzedNode\protect\T1\textdollargetTheType.0.html

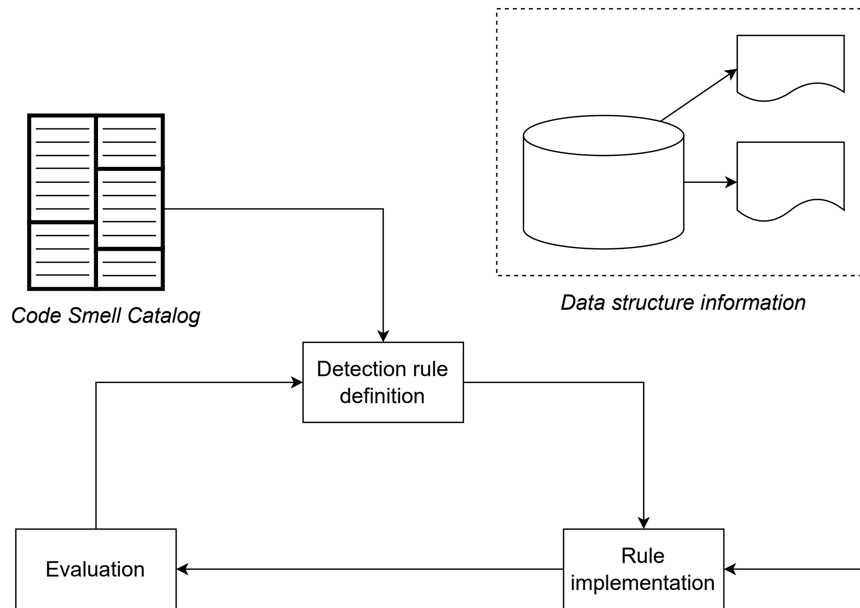


Figure 5.4: Detection query definition process

used in a query filter. Then, in the `where` clause, we use the `isStronglyIndexed` predicate to tell if the field was present in an index declaration somewhere else in the project (Line 3). We also make sure that the tracked field is not the `_id` one. Finally, we retrieve the parent query from the operator using our `getParentQuery` predicate, alongside a small explicative text about the query.

Once the query has been implemented, we evaluate the detection rule by executing the query on few manually picked projects and checked for eventual false positive or whether we missed some instances. We use the output of this phase to refine our detection rules and repeat the cycles as many times as necessary. Once all the queries of our tracked smells had their dedicated detection queries, we launched executed all of them, as detailed in Section 5.4.

5.4 Results

To evaluate our implementation of SMEAGOL, we ran it against a benchmark of 704 JavaScript open source systems using MongoDB or Mongoose to uncover eventual code smells instances. In this section, we describe how we gathered the projects from the benchmark. Then we report the retrieved code smells and discuss eventual thresholds. Finally, we illustrate the results with instance examples from the obtained code smells.

Table 5.2: Benchmark projects statistics

	Min	Q1	Median	Q3	Max
Stars	100	165	306	878	119339
Files	1	25	71	230	16068
LOC	1152	61079	266419	958650	117773569

5.4.1 Benchmark

As the starting point for this benchmark, we initially considered reusing the dataset proposed by Benats *et al.* [25]. However, as it relies on Library.io data dumps, which the most recent ones were released in 2020 and this could hinder its validity as it may still include deprecated or removed projects, or it may miss emerging projects.

Consequently, we decided to rely on "*GitHub Search*", a continuously updated dataset of open source systems proposed by Ozren Dabic, Emad Aghajani and Gabriele Bavota [41]. At the time of submission in 2021, it contained 735,669 projects. It allows for querying projects based on 25 characteristics (*e.g.,* language, name, star numbers).

Fig. 5.5 presents an overview of the benchmark building process.

First, we consulted GitHub Search web portal on 24/01/2024 and queried JavaScript, which is the programming language tracked by SMEAGOL, projects having at least 100 stars, obtaining 26,597 projects. We chose this restrictive threshold of 100 stars to ensure the quality and avoid toy projects. Then, we manually checked each project dependency and controlled if they had MongoDB or Mongoose. To do that, we used Github API with a Python script to obtain each project SBOM (Software Bill Of Materials) file, which recapitulates a project dependencies. In that file, we were able to control the presence of the MongoDB or Mongoose package and filter projects without any of those. At the end of this process, we obtain 704 open-source systems.

Table 5.2 shows the projects statistics repartition in terms of project stargazers, files and total JavaScript lines of code. They are broken down into minimum, first quartile, median, third quartile and maximum.

The stargazers and files statistics were retrieved with Python scripts and the library PyGithub. [5] Each project LOC was measured using the cloc tool, a general tool to count lines of code excluding the blank and comments one. [6]

5.4.2 Reports

To run SMEAGOL on the benchmark projects, we cloned each of them to their most recent commit as of 2nd April 2024. Then, we compiled them into their corresponding CodeQL database, which lasted over 26 hours on a dedicated server. [7] After it, we ran SMEAGOL on every projects, which required 31 hours on the same server. Finally, we gathered every SMEAGOL project report in a common database for analysis using

[5] https://github.com/PyGithub/PyGithub

[6] https://github.com/AlDanial/cloc

[7] CPU: 2x Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz (Max speed 4.00GHz, 14 cores, 28 threads, 35M cache per CPU) RAM: 384GB (24x16 GB DDR4 2133 MT/s)

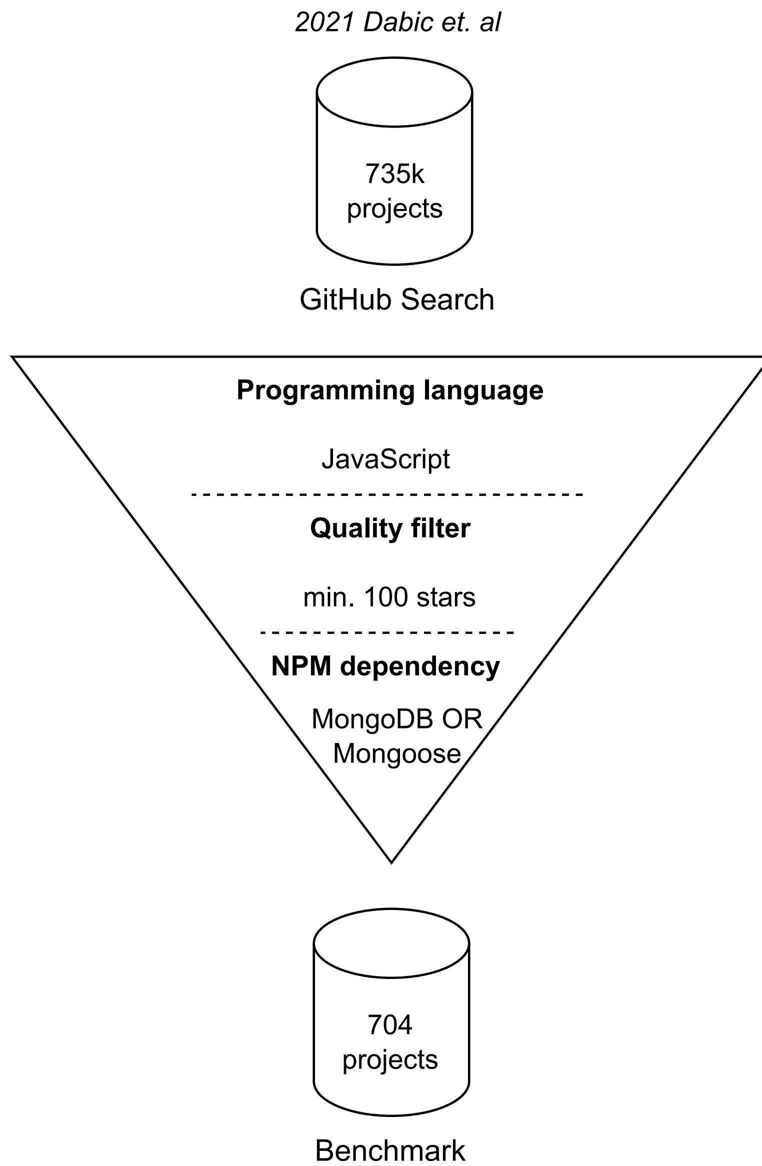


Figure 5.5: Benchmark building overview

Table 5.3: Retrieved code smells instances

Code Smell	Projects	Occurences
Querying too much data	232	13548
Single update/insert for batches	205	4066
Uncovered query	151	15364
Multiple Schemas in a file	87	388
Negation in query	72	1176
Case-insensitive queries without matching indexes	49	8272
Using $limit without $sort	49	258
The single-person bridge	36	332
Repeated immutable data	25	766
Using a document for _id	18	310
Confusing null and undefined	7	26
Map-Reduce for projection	7	32
Avoid $where	4	18
Sorted monkeys	2	6
Index intersection rather than compound index	1	2
Prefix index of compound indexes	1	2

python scripts. Table 5.3 summarizes the occurrences retrieved by SMEAGOL on the benchmark projects. The first column identifies the retrieve Code Smell, the second the number of projects in which it was identified, the latter the total number of instances retrieved.

Overall, we were able to retrieve 44,566 from 340 different projects. Sixteen different smell types of the 32 tracked yielded results.

We can see that the most frequent smell in terms of instances found and projects infected is "*Querying too much data*". An instance from the *nodemailer/wildduck* project [8] is illustrated in Listing 5.14.

Listing 5.14: Querying too much data instance

```
26  let userData;
27  try {
28    userData = await db.users.collection('users').findOne(
29      {
30        _id: user
31      },
32      {
33        projection: {
34          username: true,
35          address: true,
36          specialUse: true
37        }
38      }
39    );
40  }
```

[8] https://github.com/nodemailer/wildduck/blob/ea24b9328b6984db841de86309f1712f100acb97/lib/prepare-search-filter.js#L28

In this code snippet, a query is made on a `users` collection (Line 28), with explicit projections to keep three fields `username`, `address` and `specialUse` (Line 33-36). However, in the following code, none of those fields are actually used. Consequently, the query includes too many fields and should have removed the projection fields.

The "*Uncovered query*" smell, although found in fewer projects than "*Querying too much data*", had more instances. It is worth noting that it may be an intentional design decision not to index a query: 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. 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. In our detection query, we also make the hypothesis that indexes are declared in the source codes. But it might not always be the case. Indeed, during our manual inspection of highly stared projects, we found that the project *HabitRPG/habitica*, a mobile app game to track people's habits, 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.

We were also able to find many instances of "*Multiple Schemas in a file*". Indeed, while this smell does not directly hinder performances, it decreases code readability which may explain its frequent presence. An illustrating example can be found in the project "*Cvmcosta/ltijs*", a tool to improve learning on an application. [10] Listing 5.15 summarizes this instance.

Listing 5.15: Multiple Schemas in a file

```
185 mongoose.model('idtoken', idTokenSchema);
186 mongoose.model('contexttoken', contextTokenSchema);
187 mongoose.model('platform', platformSchema);
188 mongoose.model('platformStatus', platformStatusSchema);
189 mongoose.model('privatekey', keySchema);
190 mongoose.model('publickey', keySchema);
191 mongoose.model('accesstoken', accessTokenSchema);
192 mongoose.model('nonce', nonceSchema);
193 mongoose.model('state', stateSchema);
```

There, we can see the creation of 8 Mongoose models in the same file (Line 185). To improve readability, this project maintainers could create all those models in separated files.

We can also observe many instances (8272) of "*Case-insensitive queries without matching indexes*" while being spread to a limited number of projects (49). In fact, 71% (5902) of all those smells come from a single project, "*easyerp/easyerp_open_source*", an ERP.

In a similar trend without the same size effect, we can class "*Negation in query*" and "*Repeated immutable data*", which were unveiled 1176 and 766 times, in 72 and 25 projects.

[9] https://github.com/HabitRPG/habitica/blob/develop/migrations/docs/mongo-indexes.md
[10] https://github.com/Cvmcosta/ltijs/blob/master/dist/Utils/Database.js#L185

Table 5.4: Projects containing code smells ($n = 340$)

	Min	Q1	Median	Q3	Max
# Instances per project	2	6	23	70	13002

Table 5.5: Top 20 projects in terms of smells occurrences

Project	**Occurrences**	**# Types**
easyerp/easyerp_open_source	13002	8
countly/countly-server	2404	10
habitrpg/habitica	1710	9
socioboard/socioboard-5.0	1536	7
botfront/botfront	1472	8
aquilacms/aquilacms	1406	7
peterhanania/pogy	1016	3
sentinel-official/sentinel	670	4
tdjsnelling/sqtracker	604	5
trustroots/trustroots	502	8
getstream/winds	488	6
mayeaux/nodetube	436	4
forwardemail/forwardemail.net	434	5
nodemailer/wildduck	430	6
fightpandemics/fightpandemics	410	7
duyluonglc/lucid-mongo	346	7
wesbos/learn-node	340	2
sergeyksv/tingodb	326	7
xtremespb/taracotjs	320	5
wafttech/waftengine	316	5

Then, "*Using $limit without $sort*", "*The single-person bridge*" and "*Using a document for _id*" have a similar number of retrieved instances, in a fairly restricted number of projects (49, 36 and 15).

After that, we can form another category of smells that were detected infrequently. It includes "*Confusing null and undefined*" (26 instances in 7 projects), "*Map-Reduce for projection*" (32 instances in 7 projects) and "*Avoid $where*" (18 instances in 4 projects).

Finally, we also arrange "*Sorted monkeys*", "*Index intersection rather than compound index*" and " *Prefix index of compound indexes*" who were detected in one or two projects and for a few number of times.

Table 5.4 breaks down the number of instances a project can have.

We also wanted to determine the number and types of smells in the top 20 projects based on the number of instances. Table 5.5 shows those projects alongside their total instance occurrences and different types of instances.

First, we can see that *easyerp/easyerp_open_source* have many more instances (13,002) than the others while keeping a similar amount of instances kinds. This is due to 2 smells, *Uncovered query* and *Case-insensitive queries without matching in-*

Table 5.6: Data structure information for projects having at least one instance (n = 340).

	Min	Q1	Median	Q3	Max
Fields #	1	10	37	106	3568
Collections #	1	2	3	7	101
Collection size	1	4	10	23	644
Collection depth	0	0	1	1	8
Collection index	1	1	1.5	3	14
Index #	0	1	2	6	133

dexes who were identified respectively 6,548 and 5,902 times. Then, *countly/countly-server* has the biggest number of different kinds. It instances come mainly from *Uncovered query* (1,090) as they declared their index separately from the source code, *Single update/insert for batches* (618) and *Querying too much data* (472). We can also see that *peterhanania/pogy* contains 1,016 occurrences spread across only 3 smell types. A notable part stem from the *Querying too much data* smell (820) then from *Using a document for _id* (118) and *Single update/insert for batches* (78).

Data structure

To characterize the instances reported in the last section, we show here the characteristics relative to collections, fields and indexes from the benchmark projects. On the side of the SMEAGOL run, we also executed queries to extract only the data information without detection smells or antipatterns. Overall, among the 340 projects having instances, we could extract 37506 fields across 1867 collections. From those 37506, the vast majority was from Mongoose Schema declaration (357627, 95%) while a fraction was found in MongoDB queries (1879, 5%). This unbalanced repartition may stem from the verbosity of Mongoose Schema, as those report all the possible fields a document from this collection could have and not the one that are actually used.

Table 5.6 reports different metrics from the 340 projects having at least one instance data structure information. Those metrics (per project) report respectively: total number of fields (Fields #), total number of collection (Collections #), number of attributes per collection (Collection size), maximum embedding per collection, 0 indicates no embedding (Collection depth), index per collection (Collection index) and total number of indexes (Index #). They are expressed under their minimum, first quantile, median, third quantile and maximum.

First, we can see that that the minimum of fields and collection is pretty low. This is due to an example project *actions/example-services*, having only 1 relatively small file performing an insertion without declaring any index. [11] Then, we can see that we did not extract many fields or collections from the projects, this could also indicate that the projects in the benchmark do not deal with a lot of data. Regarding the depth, we can see that a quarter of the projects do not use any embedding and 75% of them have only one level. Concerning the indexes, we can also remark that

[11] https://github.com/actions/example-services/blob/main/mongodb/client.js

they are not extensively used. This can come from how we gather index declaration. Indeed, to avoid any collisions between collections, we decided to consider an index declaration only if we could statically identify the collection on which it was declared, making it likely to be underestimated.

5.5 Conclusion

In this chapter, we presented SMEAGOL, a static analysis tool to detect MongoDB code smells and antipatterns in JavaScript projects using the Native MongoDB Driver or Mongoose. It relies on CodeQL and consists of a set of detection query, one for every code smells, and shared utility files with abstractions to work with MongoDB and Mongoose queries. We first determined which smells from the catalog defined in Chapter 4 could be detected in the application code by assessing their detection level, leaving us with 39 . Then, we estimated the implementation difficulty for each code smell and removed those who were too complex, with 32 smells like results. After that, we described the structure of SMEAGOL, how we abstracted the variety of information sources to collect information about the data structure of the project and how we defined the different detection queries.

Thereafter, we built a benchmark of 704 JavaScript projects using MongoDB or Mongoose. We then ran SMEAGOL on each of them and detected 44,566 instances of 16 different kinds of smells spread across 340 projects. Using those results we were able to outline several frequently occurring code smells and show multiple projects where we detected many code smells. We also showed multiple illustrations of smells we could find.

While SMEAGOL showed its potential through a preliminary evaluation, there are opportunities to further enhance its usefulness for developers. Indeed, we meticulously implemented each query to be as precise as possible on a subset of 10 projects, though a comprehensive evaluation of precision and recall is planned for future work. Nevertheless, SMEAGOL can already assist developers by highlighting potential code smells and antipatterns in their codebase to guide them in their maintenance tasks.

5.5.1 Roadmap

In this chapter, we presented SMEAGOL which relies on the database extraction approach described in Chapter 3 enriched with schema inference techniques to detect instances of code smells depicted in Chapter 4

In the following, we will conclude this thesis by addressing the defined RQ and expanding upon potential areas for future research.

Part II

Postface

CHAPTER 6

CONCLUSION

6.1 Summary of the contributions

_RQ1_: How to automatically and statically extract the database access fragments of application programs using a document-oriented datastore?

We proposed an approach to automatically extract MongoDB database accesses fragment from JavaScript application programs in Chapter 3. It consisted in extracting every method calls matching the name of a MongoDB database access method, then applying filtering heuristics to avoid collision with other libraries.

Following an evaluation that involved blind cross-labeling on a random sample of 828 method calls from 502 open-source JavaScript projects using MongoDB or Mongoose, we obtained a precision of 78%.

We then analyzed the evolution of two systems regarding database accesses. By classifying these accesses based on their effects on the data, we were able to determine whether a program was more intensive in reading, inserting, or updating data. Additionally, we could observe sudden changes in the accesses amount that were linked to major changes in the system as a migration or a new feature implementation.

This approach revealed to be the foundation stone of our static analysis process. Indeed, retrieving database accesses fragments from application code enabled data structure information extraction and further query analysis.

_RQ2_: How to extract implicit data structure information from the database access fragments?

In order to be more precise while detecting code smells and antipatterns, in Chapter 5 we built an approach to detect implicit data structure information from the application code, on top of the one defined in Chapter 3. This approach considered

queries, their surrounding code, index declaration and Mongoose Schema declaration. To achieve this, we extensively used taint tracking analysis between queries and their resulting document variable, so we could track any property access on it and infer fields or access the document object definition between its insertion.

Using this approach, we extracted 37,506 fields spread across 1867 collections from 340 . While we did not gather many fields or collections from using this approach, it is yet to be compared with other approaches relying on the data as the one proposed by Baazizi *et al.* [22].

Still, extracting data information from database access fragments enables further static analysis tasks and we exploited it extensively in our code smell detection tool described in Chapter 5.

RQ3.1: What types of MongoDB code smells and antipatterns have been identified by the community?

In Chapter 4, we built a catalog of MongoDB code smells and antipatterns by integrating grey literature and published paper into an MLM. Then, we classified the obtained smells and antipatterns into different categories using open card sorting.

Overall, we identified 11 smell categories containing 76 different code smells: *Aggregation issues*, *Design oversight* with subcategories `Application code`, `Data/Schema` and `Indexes`, *Performance/Memory issues*, *Sharding issues*, *Backup issues*, *Query issues*, *Security issues*, *Relational design ghosts*, and *Human-oriented decisions*.

Our catalog reflects the diverse findings reported by the community, including various types of artifacts: code smells, antipatterns, and runtime/configuration mistakes. We also noted that these different smells can occur at various stages of development, such as modeling, querying, testing, or deploying.

In conclusion, the community had already identified a wide array of code smells and antipatterns. Our contribution was to consolidate and present these findings in a comprehensive and organized manner.

RQ3.2: Where are discussed those code smells and antipatterns?

To compile the catalog in Chapter 4, we conducted a literature mapping (MLM) that included both white and grey literature. Each source was manually verified for its relevance to MongoDB code smells and antipatterns.

Following this verification, we identified 174 sources that discussed one or more smells. Among these, only a tenth (13) were from the research community. More than half (95) was blog posts or online articles, and approximately 20% were found in online discussions. Books and presentations were equally represented (12 each), while course and project artifacts were minimal (5 and 2, respectively).

During the classification phase, we extracted smells from the sources and rejected those that did not contain relevant smells. This phase highlighted the proportions: 87% of the identified smells originated from blog posts or discussions, while only one smell was found in a published paper. Most smells were accessed via Google search, although they could have been found on the MongoDB website.

We also recorded the publication dates of the sources, observing a significant increase in sources describing smells starting in 2019. This trend indicates growing

interest within the community, further emphasizing the need for a comprehensive catalog.

***RQ4*: How to automatically detect code smells and antipatterns in application programs using MongoDB datastore?**

Chapter 5 saw the introduction of SMEAGOL, a static analysis tool for detecting MongoDB code smells and antipatterns in JavaScript projects. SMEAGOL relies on the approach proposed for *RQ1* and *RQ2*. We identified 39 detectable smells from the catalog in Chapter 4, then narrowed this to 32 based on implementation difficulty. For each of those smells, we designed a dedicated detection rule that we later translated into detection queries. SMEAGOL provides an indicative hint regarding the detected smell without suggesting a mandatory refactoring.

We could evaluate SMEAGOL on a benchmark of 704 JavaScript projects using MongoDB or Mongoose, detecting 44,566 instances of 16 different smells across 340 projects. This allowed us to highlight common code smells and showcase projects with numerous smells. Although some tracked code smells could not be statically detected or were observed with limited frequency, these factors alone are insufficient to discount their classification as code smells.

While SMEAGOL preliminary evaluation showed promising results, further enhancements are planned. Despite this, SMEAGOL can already help developers by identifying potential code smells and antipatterns to aid them in their maintenance tasks.

6.2 Future Directions

Apply database access extraction process to other contexts Chapter 3 presented an approach to extract database accesses from JavaScript application interacting with MongoDB through method calls. The specificity of this approach relies on gathering candidates based on the method call name and then removing the noise using filtering heuristics. This approach was motivated by the complexity of performing sound static analysis of JavaScript programs, resulting of its highly dynamic nature. Other DBMS interacting through method calls in dynamic language could benefit from this approach, like JavaScript systems using Redis or Elasticsearch as databases. It could also be interesting to transpose this approach to extract MongoDB database accesses in Python systems and evaluating how precise it could be. We are also aware of research enquiring the extraction of MongoDB accesses in Node.js microservices applications.

Propose refactorings In Chapter 4, we built a catalog of MongoDB code smells and antipatterns. Currently, this catalog includes only the names and descriptions of each element. However, some antipatterns can also be associated with specific refactoring techniques. For instance, fixing “Inconsistent attribute” would require updating a find query filter to conform to the correct document structure. Finding such refactorings could pave the way for future research on automatic refactoring of MongoDB antipatterns.

Enquire community developers about the catalog Even though we were able to detect many code smells and antipatterns from our catalog into many open-source systems in Chapter 5, we still do not know about the developers perception of them. Surveying practitioners perception about a catalog of smells can add a new dimension to the catalog characterization and help tuning their detection and fixes.We tried to undergo a quantitative online survey aiming to measure developers awareness of an early stage of our catalog but the lack of participants restrained us to draw conclusion about it. Thus, acquiring developer's perception about our catalog is left to future works.

Investigate other NoSQL code smells In Chapter 4, we identified some code smells and antipatterns that can be generalized to broader database modeling principles, even while focusing on MongoDB-specific issues. Examples include "*Manual Backup*" and "*Immortal Cursors*" which are also relevant to other database systems. Although this catalog targets MongoDB, it is the first dedicated to a NoSQL technology.

Similarly, as Redis also uses sharding for horizontal scaling, related community sources might report issues similar to the "*Sharding Issues*" category smells we described. Creating similar catalogs for other notable NoSQL databases could highlight the differences and similarities between these technologies, ultimately aiding the community in comparing them.

Extend SMEAGOL Smells Coverage While we aimed to cover every smell described in Chapter 4, we ultimately tracked only 32 of them. Some smells, such as "*Immortal Cursors*", require tracking a cursor object across an application and checking its iteration count, which adds significant complexity. Developing detection queries for such cases would require substantial time and a larger development team.

We also excluded smells requiring knowledge about the data. However, a maintainer running SMEAGOL would likely have direct access to the database. For these cases, it could be beneficial to allow detection queries to take the data as additional input, identifying more instances. Additionally, access to the data could enhance our data structure information extraction. Schema inference over JSON datasets is well researched, and using their output could reveal more information than static analysis alone.

Finally, we excluded smells requiring knowledge about a project's configuration. Nevertheless, some open-source projects include their configuration files in the repository, which we could leverage for further analysis. However, the variability of these configurations presents a challenge.

Support other programming languages In its current form, SMEAGOL only supports JavaScript projects. However, we know there are other open-source projects that rely on MongoDB as their database, which are typically written in Java or Python, according to an empirical investigation [25]. Fortunately, both of these languages are supported by CodeQL, and we could reuse some of our existing structures and heuristics to detect instances in these languages.

Expanding SMEAGOL to support Java and Python would bring new challenges unique to these languages. For instance, Java's static typing and extensive libraries

and a Python dynamic nature would require to adapt detection strategies accordingly. Some of the core principles and detection queries can be directly translated or slightly modified for use in Java and Python. However, there will be language-specific nuances and idioms that we will need to account for.

By addressing these challenges and expanding SMEAGOL language support, more developers could be assisted in their development.

SMEAGOL user tests In Chapter 5, we ran SMEAGOL on multiple projects and were able to find multiple instances on them. Nonetheless, we do not yet know how these maintainers perceive SMEAGOL's reports on their projects. Prior to writing this thesis, we attempted to contact maintainers about the instances we detected, but we have not received any responses.

Understanding the developers' perceptions of SMEAGOL reports could improve the tool and ensuring it provides more insightful and actionable recommendations. Additionally, this feedback could complement a survey on maintainers' perceptions of the code smell catalog, offering a complete of the community interactions with those smells.

Survey data schema evolution To enable the detection of certain smells, we proposed an approach to extract data structure information from the database accesses fragment in Chapter 5. This approach can gather collections, their related fields and declared index, which could be useful in an analysis of the program schema evolution. Indeed, Loup Meurice and Anthony Cleve showed an approach to extract schema evolution from the application code of Java programs interacting with a MongoDB database [92]. Additionally, Stefanie Scherzinger and Sebastian Sidortschuck proposed another approach relying on ODM notations from 10 Java programs [124].

Surprisingly, no approach has been developed to capture schema evolution from JavaScript source code, despite JavaScript being the most popular programming language using MongoDB. Reusing SMEAGOL's data structure information extraction could enable a potentially larger-scale empirical evaluation, as it is less restrictive than relying solely on ODM annotations and could give us more insights about a project data structure.

BIBLIOGRAPHY

[1] Cassandra. https://cassandra.apache.org/_/index.html. Accessed on 22 March 2024.

[2] Couchbase. https://www.couchbase.com/. Accessed on 20 March 2024.

[3] Hbase. https://hbase.apache.org/. Accessed on 22 March 2024.

[4] jQuery. https://jquery.com/. Accessed on 5 April 2024.

[5] Libraries.io. https://libraries.io/.

[6] Memcached. http://www.memcached.org/. Accessed on 20 March 2024.

[7] Memgraph. https://memgraph.com/. Accessed on 22 March 2024.

[8] Mozilla web docs. https://developer.mozilla.org. Consulted on 5 April 2024.

[9] Neo4j. https://neo4j.com/. Accessed on 22 March 2024.

[10] Redis. https://redis.com/. Accessed on 20 March 2024.

[11] Fatma Abdelhedi, Amal Brahim, Hela Rajhi, Rabah Ferhat, and Gilles Zurfluh. Automatic extraction of a document-oriented NoSQL schema. In **23rd Int. Conf. Enterprise Information Systems**, 2021.

[12] Ariel Afonso, Altigran da Silva, Tayana Conte, Paulo Martins, João Cavalcanti, and Alessandro Garcia. LESSQL: dealing with database schema changes in continuous deployment. In **2020 IEEE 27th Int. Conf. Software Analysis, Evolution and Reengineering (SANER)**, pages 138–148, 2020.

[13] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on linguistic antipatterns affecting apis. In **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. IEEE, September 2018.

[14] Kumar Ajitesh. 10 mongodb best practices for aws prod deploys. https://web.archive.org/web/20230301174730/https://dzone.com/articles/10-mongodb-best-practices-for-aws-prod-deploys, September 2017. Accessed on October 2022.

[15] Ken W. Alger and Daniel Coupal. Building with patterns: The polymorphic pattern. https://www.mongodb.com/developer/how-to/polymorphic-pattern/. Accessed on 10 November 2021.

[16] Esayas Aloto. MongoDB Best Practices: Design, Deployment & More. https://www.datavail.com/blog/mongodb-best-practices/, 2017. Accessed on October 2022.

[17] Amazon. Mongodb on amazon. https://d0.awsstatic.com/whitepapers/AWS_NoSQL_MongoDB.pdf, April 2016. Accessed on October 2022.

[18] Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. **Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA**, pages 17–31, 2014.

[19] Thakur Ankush. What is mongodb sharding and the best practices? https://geekflare.com/mongodb-sharding-best-practices/, July 2020. Accessed on October 2022.

[20] Zameer Ansari. MongoDB schema design. https://medium.com/hackernoon/mongodb-schema-design-86327d8fae83, 2014. Accessed on October 2022.

[21] Hussain Ayaz. Mongodb schema design anti-patterns in a nutshell. MongoDBSchemaDesignAnti-patternsinaNutshell, February 2022. Accessed on October 2022.

[22] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive JSON datasets. **The VLDB Journal**, 28(4):497–521, 2019.

[23] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In **2013 35th International Conference on Software Engineering (ICSE)**, pages 712–721, 2013.

[24] Kyle Banker. **MongoDB in Action**. Manning Publications Co. LLC, New York, 2016. Description based on publisher supplied metadata and other sources.

[25] Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy, and Anthony Cleve. An empirical study of (multi-) database models in open-source projects. In **Conceptual Modeling**, pages 87–101. Springer International Publishing, 2021.

[26] Alex Bevilacqua. The Sights (and Smells) of a Bad Query. https://www.slideshare.net/mongodb/mongodb-world-2019-the-sights-and-smells-of-a-bad-query, 2019. Accessed on October 2022.

[27] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. **MongoDB: the definitive guide: powerful and scalable data storage**. O'Reilly Media, 2019.

[28] Amal Ait Brahim, Rabah Tighilt Ferhat, and Gilles Zurfluh. Model driven extraction of NoSQL databases schema: Case of MongodB. In **11th Int. Joint Conf. on Knowledge Discovery, Knowledge Engineering and Knowledge Management**, pages 145–154, 2019.

[29] William J. Brown, editor. **AntiPatterns**. Wiley computer publishing. Wiley, New York [u.a.], 1998. Literaturverz. S. 285 - 291.

[30] Singh Charanjit. 5 mistakes web developers make when working with mongodb. https://hub.packtpub.com/5-mistakes-web-developers-make-when-working-mongodb/, October 2016. Accessed on October 2022.

[31] Boyuan Chen, Zhen Ming Jiang, Paul Matos, and Michael Lacaria. An industrial experience report on performance-aware refactoring on a database-centric web application. In **Proc. of ASE'19**, pages 653–664, 2019.

[32] Tse Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In **Proc. of ICSE'14**, pages 1001–1012. IEEE, May 2014.

[33] Tse Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. **TSE**, 42(12):1148–1161, December 2016.

[34] Boris Cherry. Online appendix. https://github.com/bocherry/thesis_companion.

[35] Boris Cherry, Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of database accesses in MongoDB applications. In **2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)**. IEEE, mar 2022.

[36] Boris Cherry, Jehan Bernard, Thomas Kintziger, Csaba Nagy, Anthony Cleve, and Michele Lanza. A multivocal mapping study of MongoDB smells. In **2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024)**. IEEE Computer society, 2024.

[37] Boris Cherry, Csaba Nagy, Michele Lanza, and Anthony Cleve. SMEAGOL: A static code smell detector for MongoDB. In **2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024)**. IEEE Computer society, 2024.

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

[39] Chang Chris. 7 best practices new mongodb users should know. 7BestPracticesNewMongoDBUsersShouldKnow, November 2016. Accessed on October 2022.

[40] Rick Copeland. **MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database**. O'Reilly Media, Inc., 2013.

[41] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for msr studies. In **2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)**. IEEE, May 2021.

[42] Ziolkowski David. MongoDB Best Practices: Schema Design, Indexes & More. https://blog.panoply.io/mongodb-best-practices, August 2022. Accessed on October 2022.

[43] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. SQLCheck: automated detection and diagnosis of SQL anti-patterns. In **Proc. of SIGMOD'20**, pages 2331–2345. ACM, 2020.

[44] Shakuntala Gupta Edward and Navin Sabharwal. **Practical MongoDB**. Apress, 2015.

[45] Oren Eini. The relational modeling anti pattern in document databases. https://ayende.com/blog/4465/that-no-sql-thing-the-relational-modeling-anti-pattern-in-document-databases, 2010. Accessed on October 2022.

[46] Murat Erder and Pierre Pureur. **Validating the Architecture**, pages 131–159. Elsevier, 2016.

[47] Phil Factor. **SQL Code Smells**. Red Gate Software Ltd., 2014.

[48] Phil Factor. 14 Things I Wish I'd Known When Starting with MongoDB. https://www.infoq.com/articles/Starting-With-MongoDB/, September 2018. Accessed on October 2022.

[49] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In **2013 Int. Conf. Software Engineering**, page 752–761. IEEE, 2013.

[50] David Frank. Mongodb optimization and scalability. https://www.1cloudhub.com/mongodb_optimization_scalability/, April 2019. Accessed on October 2022.

[51] Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. Schema profiling of document-oriented databases. **Inf. Systems**, 75:13–25, 2018.

[52] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. **Information and Software Technology**, 106:101–121, 2019.

[53] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. **JSS**, 138:52–81, 2018.

[54] Vahid Garousi, Kai Petersen, and Baris Ozkan. Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. **Information and Software Technology**, 79:106–127, 2016.

[55] Gizem Gezici. Case Study: The Impact of Location on Bias in Search Results, 2022.

[56] Alex Giamas. **Mastering MongoDB 6.x**. Packt, Birmingham, third edition edition, 2022. Includes bibliographical references and index.

[57] Paola Gómez, Rubby Casallas, and Claudia Roncancio. Data schema does matter, even in NoSQL systems! In **Proc. of RCIS'16**, 2016.

[58] Paola Gómez, Rubby Casallas, and Claudia Roncancio. Automatic schema generation for document-oriented systems. In **Database and Expert Systems Applications**, pages 152–163. Springer, 2020.

[59] Robin Hecht and Stefan Jablonski. Nosql evaluation: A use case oriented survey. In **2011 International Conference on Cloud and Service Computing**. IEEE, December 2011.

[60] Onyancha Brian Heny. Best practices for mongodb security. https://severalnines.com/blog/best-practices-mongodb-security/, January 2022. Accessed on October 2022.

[61] Boyu Hou, Kai Qian, Lei Li, Yong Shi, Lixin Tao, and Jigang Liu. MongoDB NoSQL injection analysis and detection. In **Proc. of CSCloud'16**, pages 75–78, 2016.

[62] Abdullahi Abubakar Imam, Shuib Basri, Rohiza Ahmad, Junzo Watada, Maria T. Gonzlez-Aparicio, and Malek Ahmad Almomani. Data modeling guidelines for NoSQL document-store databases. **IJACSA**, 9(10):544–555, 2018.

[63] Github Inc. CodeQL documentation. https://codeql.github.com/docs/, 2021. Accessed on 23 Mai 2021.

[64] McCay Jason. Mongodb indexing best practices. https://web.archive.org/web/20230322091314/https://www.compose.com/articles/mongodb-indexing-best-practices/, May 2013. Accessed on October 2022.

[65] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In **Static Analysis**, pages 238–255. Springer, 2009.

[66] Fernando Kamei et al. Grey literature in software engineering: A critical review. **Information and Software Technology**, 138:106609, 2021.

[67] Fernando Kamei, Gustavo Pinto, Igor Wiese, Márcio Ribeiro, and Sérgio Soares. What evidence we would miss if we do not use grey literature? In **Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2021)**. ACM, 2021.

[68] Anuradha Kanade, Arpita Gopal, and Shantanu Kanade. A study of normalization and embedding in MongoDB. In **2014 IEEE International Advance Computing Conference (IACC)**. IEEE, feb 2014.

[69] Joe Karlsson. MongoDB Schema Design Best Practices. https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/, 2022. Accessed on October 2022.

[70] Valeri Karpov. The MEAN Stack: Mistakes You're Probably Making With MongooseJS, And How To Fix Them. https://web.archive.org/web/20210301183252/https://www.mongodb.com/blog/post/the-mean-stack-mistakes-youre-probably-making, 2013. Accessed on 1 March 2021.

[71] Bill Karwin. **SQL Antipatterns: Avoiding the Pitfalls of Database Programming**. Pragmatic Bookshelf, 1st edition, 2010.

[72] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. **ACM SIGSOFT Symposium on the Foundations of Software Engineering**, 16-21-Nove:121–132, 2014.

[73] Karamjit Kaur and Rinkle Rani. Modeling and querying data in NoSQL databases. In **2013 IEEE International Conference on Big Data**, pages 1–7, 2013.

[74] Mat Keep and Henrik Ingo. Performance best practices: Sharding. https://www.mongodb.com/blog/post/performance-best-practices-sharding, 2020. Accessed on October 2022.

[75] Foutse Khomh, Massimiliano Di Penta, Yann Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. **Empirical Software Engineering**, 17(3):243–275, June 2012.

[76] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical report, EBSE Technical Report, School of Computer Science and Mathematics, Keele University, January 2007.

[77] Jitender Kumar and Varsha Garg. Security analysis of unstructured data in NOSQL MongoDB database. In **Proc. of IC3TSN'17**, pages 300–305, 2017.

[78] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. The do's and don'ts of infrastructure code: A systematic gray literature review. **Information and Software Technology**, 137:106593, 2021.

[79] Christian Kvalheim. **The Little Mongo DB Schema Design Book**. CreateSpace Independent Publishing Platform, 2015.

[80] Schaefer Lauren and Coupal Daniel. Separating data that is accessed together. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-separating-data/, May 2022. Accessed on October 2022.

[81] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. A mixed-method empirical study of function-as-a-service software development in industrial practice. **Journal of Systems and Software**, 149:340–359, 2019.

[82] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. **Information and Software Technology**, 88:67–95, August 2017.

[83] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. SAND: A static analysis approach for detecting SQL antipatterns. In **Proc. of ISSTA'21**, pages 270–282. ACM, 2021.

[84] Magnus Madsen and Anders Møller. Sparse dataflow analysis with pointers and reachability. In **Static Analysis**, pages 201–218. Springer, 2014.

[85] Divya Mahajan, Cody Blakeney, and Ziliang Zong. Improving the energy efficiency of relational and NoSQL databases via query optimizations. **Sustainable Computing: Informatics and Systems**, 22:120–133, jun 2019.

[86] Biswajit Maity, Anal Acharya, Takaaki Goto, and Soumya Sen. A framework to convert NoSQL to relational model. In **Proc. of ACIT'18**, pages 1–6, 2018.

[87] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. NL2Type: Inferring JavaScript function types from natural language information. In **Int. Conf. Software Engineering**, pages 304–315. IEEE, 2019.

[88] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In **Proc. of ICSM'03**, pages 381–384, 2003.

[89] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. **EMSE**, 11:395–431, September 2006.

[90] Keep Mat and Ingo Henrik. Performance best practices: Indexing. https://www.mongodb.com/blog/post/performance-best-practices-indexing, February 2020. Accessed on October 2022.

[91] McKnight. NoSQL Evaluator's Guide, 2014.

[92] Loup Meurice and Anthony Cleve. Supporting schema evolution in schema-less NoSQL data stores. In **Proc. of SANER'17**, pages 457–461. IEEE, 2017.

[93] Michael J. Mior. Automated schema design for NoSQL databases. In **2014 SIGMOD PhD Symposium**, page 41–45. ACM, 2014.

[94] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. **IEEE Transactions on Software Engineering**, 36(1):20–36, January 2010.

[95] MongoDB. Mongodb operations best practices.

[96] MongoDB. Mongodb security part ii: 10 mistakes that can compromise your database. http://web.archive.org/web/20230203151245/https://www.mongodb.com/blog/post/mongodb-security-part-ii-10-mistakes-that-can, June 2014. Accessed on October 2022.

[97] MongoDB. Mongodb operations best practices. Technical report, June 2017. Accessed on October 2022.

[98] MongoDB. Atlas. https://www.mongodb.com/atlas, 2018. Accessed on 10 April 2024.

[99] MongoDB. Mongodb acid transactions whitepaper. https://www.mongodb.com/collateral/mongodb-multi-document-acid-transactions, n.d. Accessed on 25 Mai 2021.

[100] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. In **Proc. of MSR'20**, pages 327–338. ACM, 2020.

[101] Csaba Nagy and Anthony Cleve. SQLInspect. In **Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings**. ACM, may 2018.

[102] Csaba Nagy and Anthony Cleve. SQLInspect: A static analyzer to inspect database usage in Java applications. In **40th Int. Conf. Software Engineering: Companion**, ICSE '18, page 93–96. ACM, 2018.

[103] Maric Nedim. Sql injection in mongodb: Examples and prevention. https://brightsec.com/blog/sql-injection-in-mongodb-examples-and-prevention/, September 2021. Accessed on Ocotber 2022.

[104] Big Data Analytics New. Mongodb mistakes to avoid & the best practices to follow. https://bigdataanalyticsnews.com/mongodb-mistakes-to-avoid-the-best-practices-to-follow/, October 2018. Accessed on October 2022.

[105] ObjectRocket. Mongodb security best practices. https://kb.objectrocket.com/mongo-db/mongodb-security-best-practices-699, August 2019. Accessed on October 2022.

[106] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In **Proc. of ESEM 2009**, pages 390–400, 2009.

[107] Arvind Padmanabhan. Data modelling with mongodb. https://devopedia.org/data-modelling-with-mongodb, 2021. Accessed on 26 April 2024.

[108] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. **Information and Software Technology**, 108:65–77, 2019.

[109] Gilberto Recupito, Fabiano Pecorelli, Gemma Catolino, Sergio Moreschini, Dario Di Nucci, Fabio Palomba, and Damian A. Tamburri. A multivocal literature review of MLOps tools and features. In **2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. IEEE, aug 2022.

[110] Filippo Ricca, Alessandro Marchetto, and Andrea Stocco. Ai-based test automation: A grey literature analysis. In **2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**, pages 263–270, 2021.

[111] Filippo Ricca, Alessandro Marchetto, and Andrea Stocco. A retrospective analysis of grey literature for ai-supported test automation. In José Maria Fernandes, Guilherme H. Travassos, Valentina Lenarduzzi, and Xiaozhou Li, editors, **Quality of Information and Communications Technology**, pages 90–105, Cham, 2023. Springer Nature Switzerland.

[112] Aviv Ron, Alexandra Shulman-Peleg, and Emanuel Bronshtein. No SQL, no injection? Examining NoSQL security. In **Proc. of W2SP'15**. arXiv, 2015.

[113] Aviv Ron, Alexandra Shulman-Peleg, and Anton Puzanov. Analysis and mitigation of NoSQL injections. **IEEE Security & Privacy**, 14(2):30–39, 2016.

[114] Vrinda Sachdeva and Sachin Gupta. Basic NOSQL injection analysis and detection on MongoDB. In **Proc. of ICACAT'18**, pages 1–5, 2018.

[115] Charles 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, June 2019. Accessed on October 2022.

[116] Lauren Schaefer. 3 things to know when you switch from sql to mongodb. MongoDB, 2022.

[117] Lauren Schaefer and Daniel Coupal. Case-insensitive queries without case-insensitive indexes. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-case-insensitive-query-index/, 2022. Accessed on October 2022.

[118] Lauren Schaefer and Daniel Coupal. Massive arrays. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-massive-arrays/, 2022. Accessed on October 2022.

[119] Lauren Schaefer and Daniel Coupal. Massive number of collections. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-massive-number-collections/, 2022. Accessed on October 2022.

[120] Lauren Schaefer and Daniel Coupal. Unnecessary indexes. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-unnecessary-indexes/, 2022. Accessed on October 2022.

[121] Coupal Daniel Schaefer Lauren. A summary of schema design anti-patterns and how to spot them. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-summary/, May 2022. Accessed on October 2022.

[122] Doupal Daniel Schaefer Lauren. Bloated documents. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-bloated-documents/, May 2022. Accessed on October 2022.

[123] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing schema evolution in NoSQL data stores. In **Proc. of DBPL'13**, Trento, Italy, 2013.

[124] Stefanie Scherzinger and Sebastian Sidortschuck. An empirical study on the design and evolution of NoSQL database schemas. In **Conceptual Modeling**, pages 441–455. Springer International Publishing, 2020.

[125] Shudi Shao, Zhengyi Qiu, Xiao Yu, Wei Yang, Guoliang Jin, Tao Xie, and Xintao Wu. Database-access performance antipatterns in database-backed web applications. In **Proc. of ICSME'20**, pages 58–69, 2020.

[126] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F. Redmiles. A systematic literature review on the barriers faced by newcomers to open source software projects. **Information and Software Technology**, 59:67–85, 2015.

[127] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In **Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)**, pages 120–131. ACM, 2016.

[128] Uta Störl, Meike Klettke, and Stefanie Scherzinger. NoSQL schema evolution and data migration: State-of-the-art and opportunities. In **23rd Int. Conf. Extending Database Technology, (EDBT 2020)**, pages 655–658, 2020.

[129] Kwangwon Sun and Sukyoung Ryu. Analysis of JavaScript programs: Challenges and research trends. **ACM Comput. Surv.**, 50(4), aug 2017.

[130] Amjed Tahir, Jens Dietrich, Steve Counsell, Sherlock Licorish, and Aiko Yamashita. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. **Information and Software Technology**, 125:106333, sep 2020.

[131] Edith Tom, Aybüke Aurum, and Richard Vidgen. An exploration of technical debt. **Journal of Systems and Software**, 86(6):1498–1516, 2013.

[132] Huib Van Den Brink, Rob Van Der Leek, and Joost Visser. Quality assessment for embedded SQL. In **Proc. of SCAM'07**, pages 163–170, 2007.

[133] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab. In **Proc. of ESEC/FSE'20**, pages 327–337. ACM, 2020.

[134] Yuqing Wang, Mika V. Mäntylä, Zihao Liu, Jouni Markkula, and Päivi Raulamo-jurvanen. Improving test automation maturity: A multivocal literature review. **Software Testing, Verification & Reliability**, February 2022.

[135] Shuo Wen, Yuan Xue, Jing Xu, Hongji Yang, Xiaohong Li, Wenli Song, and Guannan Si. Toward exploiting access control vulnerabilities within MongoDB backend web applications. In **Proc. of COMPSAC'16**, volume 1, pages 143–153, 2016.

[136] Shuo Wen, Yuan Xue, Jing Xu, Li-Ying Yuan, Wen-Li Song, Hong-Ji Yang, and Guan-Nan Si. Lom: Discovering logic flaws within MongoDB-based web applications. **International Journal of Automation and Computing**, 14(1):106–118, February 2017.

[137] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In **Proc. of ICSM'12**, pages 306–315, 2012.

[138] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In **Proc. of CIKM'17**, pages 1299–1308. ACM, 2017.

[139] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In **Proc. of ICSE'18**, pages 800–810. IEEE, May 2018.

[140] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. PowerStation: Automatically detecting and fixing inefficiencies of database-backed web applications in IDE. In **Proc. of ESEC/FSE'18**, pages 884–887. ACM, 2018.

[141] Zhou Yang, Chenyu Wang, Jieke Shi, Thong Hoang, Pavneet Kochhar, Qinghua Lu, Zhenchang Xing, and David Lo. What do users ask in open-source ai repositories? an empirical study of github issues. In **Proceedings of the 20th International Conference on Mining Software Repositories (MSR 2023)**. IEEE, 2023.

[142] Gansen Zhao, Weichai Huang, Shunlin Liang, and Yong Tang. Modeling MongoDB with relational model. In **Proc. of EIDWT'13**, pages 115–121, 2013.