A Multivocal Mapping Study of MongoDB Smells

Boris Cherry*, Jehan Bernard*, Thomas Kintziger*, Csaba Nagy†, Anthony Cleve*, Michele Lanza†
*Namur Digital Institute, University of Namur, Belgium
{boris.cherry|jehan.bernard|thomas.kintziger|anthony.cleve}@unamur.be
†Software Institute, Università della Svizzera italiana, Switzerland
{csaba.nagy|michele.lanza}@usi.ch

***Abstract*—Code smells are symptoms of poor design or bad implementation choices. Their automatic detection is helpful for various reasons. For example, the detected smells can guide developers during code inspection to find the causes of maintenance problems. Many code smells have been proposed for several technologies, including database communication, such as ORM or SQL antipatterns. However, despite its popularity, no research has been conducted on MongoDB smells.**

We present a systematic multivocal literature mapping study, also covering "grey" literature, to build a catalog of MongoDB code smells. After evaluating 1,498 artifacts (e.g., blog posts, online articles, book chapters, scientific papers, presentation slides, and videos) from 12 search engines, we manually reviewed 174 sources and devised a catalog of 76 smells organized into 11 categories. We present the catalog of MongoDB code smells through a series of examples.

***Index Terms*—NoSQL, MongoDB, Code Smells, Multivocal Mapping Study, Static Analysis, JavaScript**

I. Introduction

NoSQL ("Not Only SQL") data stores have become popular backends of database applications [1]–[3]. MongoDB is the most popular NoSQL data store, according to DB-Engines Ranking [4]. 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 [5]. In the worst case, these differences can lead to erroneous constructs, runtime errors, or data loss [6].

Researchers studied approaches to assist MongoDB developers. For example, Kanade *et al.* proposed a normalization and embedding approach to improving DB performance [7]. Mahajan *et al.* evaluated the effectiveness of query optimization on energy efficiency [8]. Zhao *et al.* devised an approach to support the migration of a relational DB to MongoDB [9], and Maity *et al.* investigated this migration scenario in the opposite direction [10]. Security also interests researchers [11]–[13], especially the critical NoSQL injection [14]–[17].

Many problems stem from maintainability or quality issues, often originating from poor design and implementation choices. Code smells, or antipatterns, are well-known indicators of such problems [18]–[24], and they are well-studied in the context of database applications: There are data smells [25], [26], DB schema smells [27], object-relational mapping (ORM) smells [28]–[30], SQL antipatterns or smells [31]–[39].

Despite its popularity, we found a lack of research on antipatterns or code smells for MongoDB. However, there are numerous discussions on forums, blogs, *etc.*

To better understand the state-of-the-art and practice in MongoDB code smells, we present a *multivocal literature mapping* (MLM) study [40]. Similar studies have been recently used in software engineering (*e.g.,* [41]–[48]) 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 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 [42], [43], [49].

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 [50]. 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.

Our paper 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.

The main contributions of this paper are the following:

i) an MLM of 174 sources on MongoDB code smells,
ii) a catalog of 76 MongoDB code smells and their sources in the grey literature,
iii) a discussion of implications in software research and practice,
iv) source code of tools used in our study and relevant data set in an *online appendix* [51].

Paper Structure: Section II introduces the main concepts through an example of a MongoDB code smell. In Section III, we describe the study method. Section IV and Section V present the results of the MLM study. In Section VI, we discuss implications for researchers and practitioners as well as threats to validity. Section VII overviews the related work. Finally, we present our conclusions in Section VIII.

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/SANER60148.2024.00086

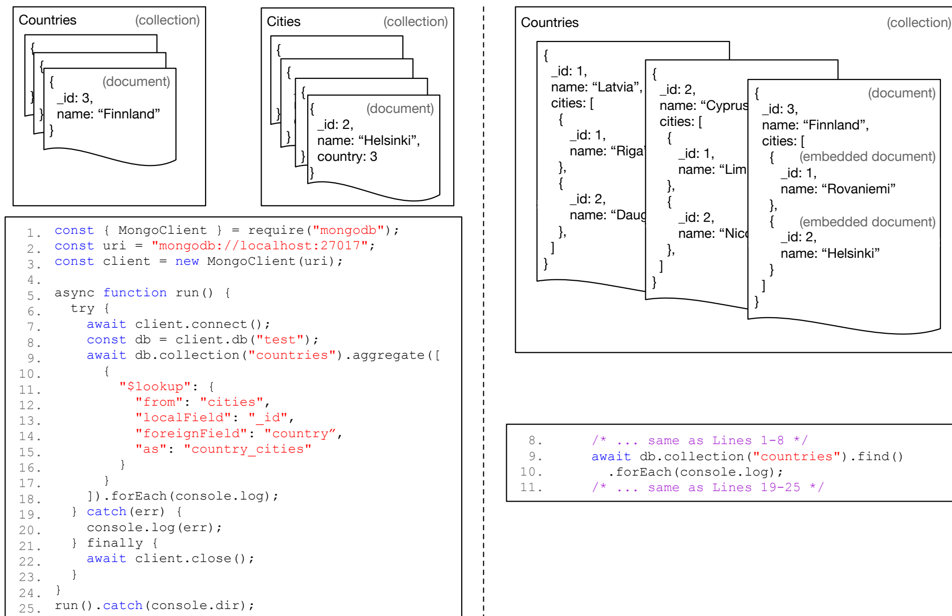

Fig. 1. Example of *Separating data accessed together* smell. `Countries` and `Cities` are two collections frequently queried together (left). The separated documents require slow and resource-intensive join (*i.e.,* `$lookup`) operation. `Countries` could embed their `Cities` to avoid expensive joins (right).

II. Background

MongoDB is a document-oriented datastore that organizes its data in *collection*s, which store a set of BSON (Binary JSON) *document*s. A document has field and value pairs holding relevant information for an object, similar to a table record in a relational DB. Unlike a table, a collection has a flexible structure without a fixed schema, and documents can embed other documents.

MongoDB provides drivers for many languages, including JavaScript. There are also popular libraries built on top of these drivers, *e.g.,* MongoDB Node Driver [52] in Node.js. Some libraries provide more abstract persistence services by mapping objects to documents. They are called ODMs (Object-Document Mapping), similar to ORMs (Object-Relational Mapping) in relational databases, such as Mongoose ODM [53].

The left side of Fig. 1 shows a JavaScript code snippet 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).

The `countries` and `cities` collections hold multiple documents. A country contains `_id` and `name` fields. A city has `_id`, `name`, and `country` fields. The latter references the document of the city's parent country.

There is a MongoDB smell in the example: `$lookup` is slow and resource-intensive compared to operations that do not need to combine data from multiple collections.

The usage of `$lookup` is discouraged in MongoDB, and document embedding is recommended as "*data that is accessed together should be stored together*" [50]. An optimized solution embeds cities in their corresponding countries. The query can then be substituted with a simple `find()` method invocation (see the right side of Fig. 1). The resulting code becomes shorter, requires fewer resources, and runs faster.

III. Method

A. Research Questions

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

RQ$_1$: What types of MongoDB smells have been proposed in the community? (*Mapping based on smell types.*)

RQ$_2$: Where are MongoDB smells discussed by the community? (*Mapping based on types.*)

B. 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 [42], [43], [54], and the approach has become known as *multivocal literature mapping* (MLM) [41], [42].

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. 2 depicts the main steps of the entire MLM approach.

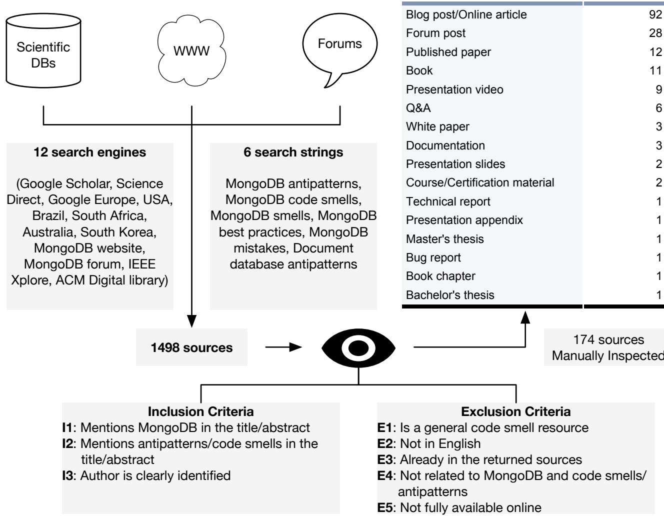

Fig. 2. Overview of the Multivocal Literature Mapping Approach

1) 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. Initially, we did not include MongoDB forums/blogs, but early iterations of the query showed lots of results originating from these sources, which motivated us to incorporate them into our searches.

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 [55]. Finally, we devised the following search string:

("MongoDB" AND ("Smells" OR "Code smells" OR "Antipatterns" OR "Best practices" OR "Mistakes")) OR ("Document database" AND "Antipatterns")

We employed both "Code smells" and "Antipatterns", as these terms are often used interchangeably by practitioners [56]. We added "Mistakes" as we observed that authors would use the term describing code smells (e.g., (*S114*)). We also included "Best practices", as 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* [57]. 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. 2). When deciding whether the source should be included was impossible based on the title and summary, we opened its link and examined it separately.

As suggested by Garousi *et al.* in their guidelines for multivocal literature reviews [40], 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 but extended this limit to 60 for Google Europe, our first search.

We ran 72 queries (6 searches x 12 engines) and examined 1,498 candidate sources. 154 satisfied our quality (source author is clearly identifiable), 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).

2) Smell Classification: We conducted open coding [58] to identify and classify the smells mentioned in the sources. Such approaches are common in systematic literature reviews and software engineering studies [59]–[62] to identify and classify relevant concepts in various sources. Two authors 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 [63], [64]. 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 substantially 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 [51].

IV. RQ_1: MongoDB Code Smell Catalog

Fig. 3 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,[1] are available in our online appendix [51].

a) Aggregation: MongoDB provides *aggregation operations* [65] to process multiple documents in collections. This is an alternative API to the more common `find()` queries. A forum post explicitly asks about common mistakes when using this API (*S125*). The answers highlight that developers with SQL backgrounds are often unaware of aggregations and tend to use CRUD operations or map-reduce processing [66] where aggregations would be more efficient.

Backup: Poor backup strategies can result in permanent data loss; hence, we grouped them into a separate category. Sources highlight the importance of a dedicated backup budget (*S4*) and regular automated backups (*S62*). Most sources (22) emphasized that replicas [67] should not be treated as backups as they offer redundancy to protect against system failure but do not protect from accidents caused by human error, *e.g.,* dropping a collection or database (*S47*).

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. Such issues can stem from the application code, design, data or schema. For example, *Immortal cursors*, *i.e.,* unclosed cursors, can result in memory leaks (*S176*), or *Querying too much data* can generate unnecessary network traffic for the application (*S4*).

On the schema side, *Unbound arrays* was the most often mentioned smell, found in 24 sources.

"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. Various sources noted that caution should be exercised with growing large arrays as they can strain application resources (*S32*). Similarly, *Bloated documents* (*i.e.,* unnecessarily large documents) should be avoided as they can result in working sets unfit for MongoDB's memory allotment (*S30*).

We also separated a subcategory for index usage, as various sources mentioned related smells. *Abusive use of indexes* was the second most frequent, with alternative names in 22 sources.

[1] When we cite sources, we use an *Sx* notation where x is the unique identifier. These references can be found in the online appendix.

For example, common alternative names were "Too many indexes" (*S65*), "Over indexing," or "Index littering" (*S4*). Maintaining unnecessary indexes can lead to degraded performance and excessive storage allocation.

Human-oriented decisions: Fostering human understandability over optimal design choices also produces code smells. For example, the *Human-readable values* (2) smell suggests that inefficient formats result in storage/RAM/network bandwidth consumption overhead, *e.g.,* "*storing a 10-digit int value takes 10 bytes as a string.*" (*S4*). Similarly, *Too long attribute names* (2) can be harmful as BSON requires storing the attribute names as well, *e.g.,* "*Long field names increase the minimum amount of space a database requires.*" (*S66*).

Performance: Most performance issues are due to deficient configurations that harm performance or memory usage. For example, 4 sources pointed out that running MongoDB on a 32-bit system (*S84*) entails significant data constraints (max. 2GB of data). 2 sources stressed unnecessarily high read-ahead (a setting that benefits sequential I/O operations) could be counterproductive because MongoDB disk access patterns are generally random (*S50*).

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 (*S111*). A case-insensitive query should also be covered with a case-insensitive index (7) (*S24*). 2 sources also stressed that leading wildcard searches (*i.e.,* regular expressions that are not left anchored or rooted) should be avoided (*S83*), (*S96*).

Relational design ghosts: Problems often stem from developers trying to apply relational modeling habits in a document database. MongoDB's golden rule is "*Data accessed together should be stored together.*" Joins require the slow and resource-intensive `$lookup` operator, that should be avoided as recommended by the *Separating data accessed together* smell in 19 sources (*S34*). 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. Thus, transactions indicate accessing data that should not be separated (*S154*).

Security: Security practices, if not respected, may lead to system intrusion, data leakage, etc. The official MongoDB Security Checklist [68] suggests enforcing the authentication in the first place. 4 sources pointed out that MongoDB does not enable authentication by default, *e.g.,* (*S126*). Defining a user policy with the least privilege is a common practice to enable access monitoring. In that context, 4 sources stressed that a single user accessing the database is never a good idea as it exposes the application to more vulnerabilities (*S116*). An example of MongoDB user management can also be found in an official MongoDB tutorial [69].

Sharding: Horizontal scaling is achieved in MongoDB via *sharding*, where *shards* (*i.e.,* data subsets) are distributed along a chosen attribute, the *shard key* [70].

Aggregation issues: associated with poor query performance.

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

Design oversights: 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)

Performance/Memory issues: 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)

Sharding issues: 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)

Backup issues: increase the risk of data loss and vulnerabilities.

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

Query issues: 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)

Security issues: 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)

Relational design ghosts: 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)

Human-oriented decisions: 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)

Fig. 3. MongoDB Code Smell Classification

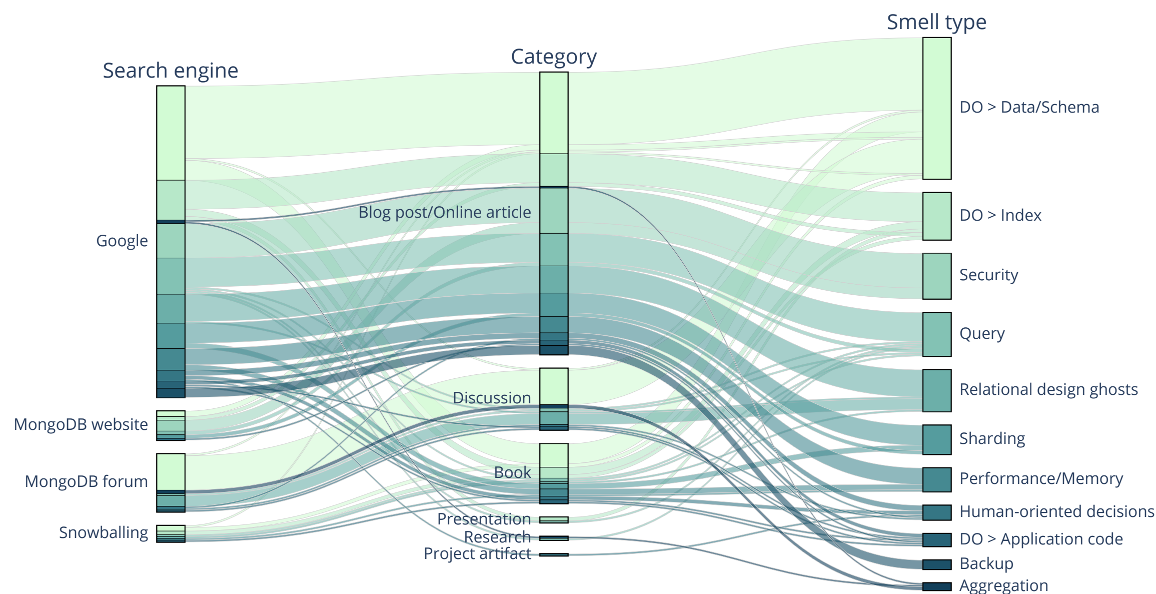

Fig. 4. Parallel categorization of source category and smell type

Selecting a suitable attribute for the sharding key is crucial, as a low-cardinality key will result in poor data distribution across the shards (*S51*).

Sources (3) said that a monotonically increasing shard key is inadequate: "*using an incrementing counter shard key, all documents will be written to the same shard and chunk until MongoDB splits the chunk and attempts to migrate it to a different shard*" (*S19*). The *Working set exceeds memory* also showcases poor sharding, mentioned in 7 sources, *e.g.,* (*S45*). A large working set, *i.e.,* data and indexes accessed during normal operations, indicates the tipping point of the sharding process.

> **RQ$_1$:** *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*.

V. 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.* [40], 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 I breaks down the source types' classification and the distribution of the sources. It shows the number of sources manually reviewed for each 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 highlights 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.* [8] 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 amount of books among the sources. These books are *MongoDB in Action* (*S18*) (with a dedicated chapter for Design patterns in its appendix), *The Little Mongo DB Schema Design Book* (*S19*), *Mastering MongoDB 6.x* (*S20*), *Practical MongoDB* (*S56*), *50 Tips Tricks MongoDB Developers* (*S171*), and *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage* (*S176*).

Fig. 4 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*.

TABLE I
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**

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.

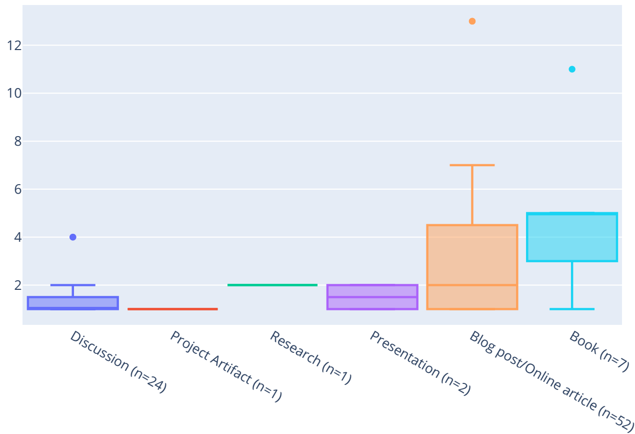

Fig. 5. Induced smell per category

Fig. 5 shows the distribution of the number of induced smells for each source type. As most of the smells are mentioned in blog posts, they are expected to 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.

The median number of induced smells was 1.5 considering all sources.

A blog post on *Big Data Anti-Patterns* by Marc Kenig, published in 2019 (*S4*), induced the highest number of MongoDB smells (13). The next notable source was the *Practical MongoDB* book (*S56*) which induced 11 smells of the catalog.

To have a broader picture of the relationships between sources and code smells, we constructed a graph that can be seen in Fig. 6. 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. 7 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 2009 [71]. However, the first release is 0.0.3 from 2008 in its Git repository [72].

Our earliest sources are from 2010-2011, as 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 (*S47*), (*S50*), (*S53*), (*S65*), (*S66*), and (*S69*). Interesting to highlight is the *14 Things I Wish I'd Known When Starting with MongoDB* InfoQ article by Phil Factor (*S69*), 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* (*S96*) and *Performance Best Practices: Sharding* (*S98*) in 2020, then started a blog series on *MongoDB Schema Design Best Practices* (*S74*) and *Schema Design Anti-Patterns* in 2022. The posts generated related discussions on their forums, e.g., see (*S7*) and (*S11*).

RQ_2: *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%), which we retrieved in the biggest part from Google. 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.

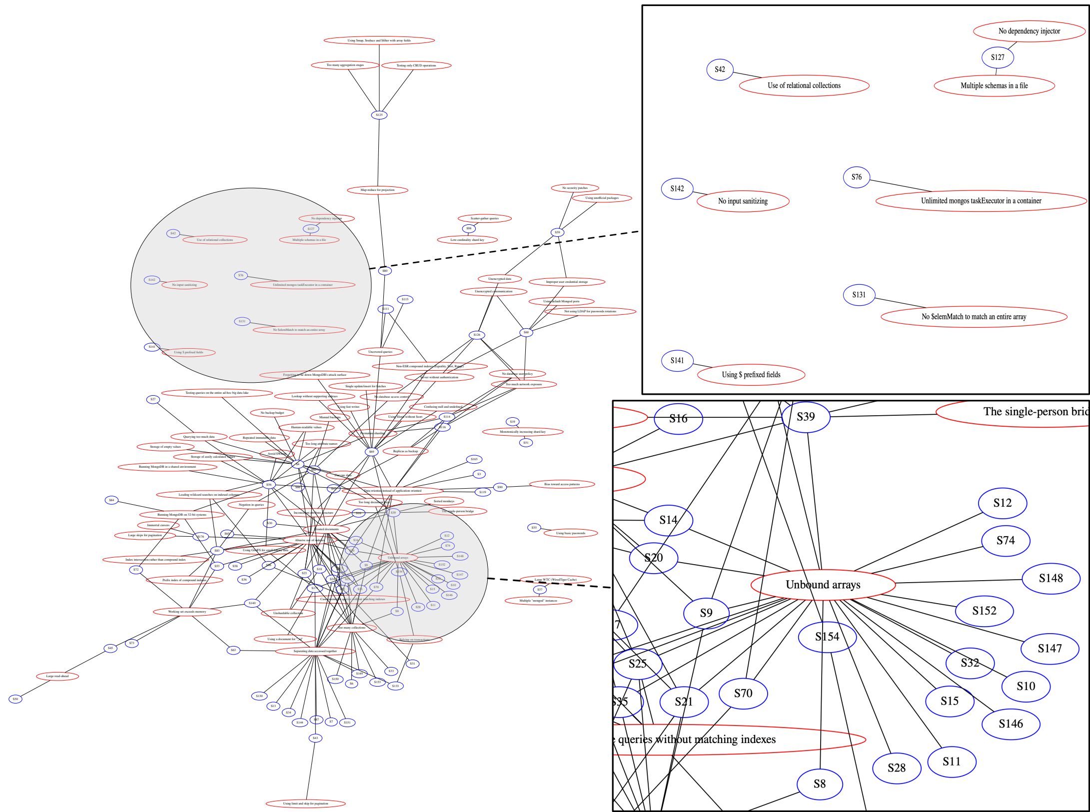

Fig. 6. Network representation of smells and sources

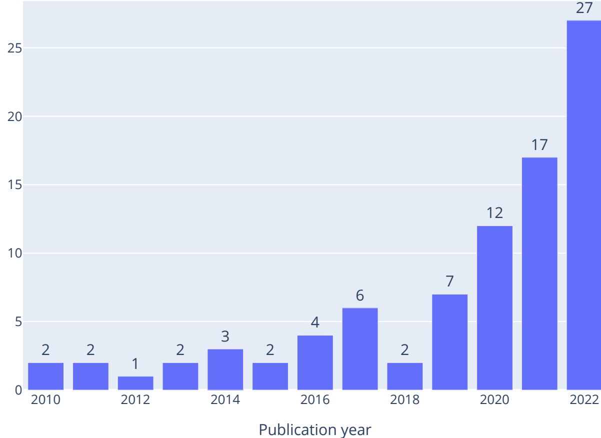

Fig. 7. Distribution of sources according to publication year

VI. Discussion

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

A. 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. They are often made by developers who came to the NoSQL world with a relational database background.

1) Detection tool: We relied on CodeQL [73] as a static analysis framework that supports various analyses (*e.g.,* data flow or taint analysis) for multiple languages, including JavaScript. The language was significant for us as previous studies found that JavaScript is the most widely used language with MongoDB [2].

CodeQL builds a database to store the project's AST (Abstract Syntax Tree) and analysis data. This database can then be queried in QL, a declarative, object-oriented query language with SQL-like syntax. We implemented QL queries for the *Storage 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 [51].

```
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 1. CodeQL query to find *Storage of empty values* instances

Listing 1 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 [51].

2) Smell instances: We analyzed top-starred open-source systems on GitHub (see the online appendix [51]) and found interesting smell instances, which we share as motivational examples for future studies.

a) Use of relational collection: The *voluntarily/vly2* repository hosts the source code of the *Voluntarily* [74] 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.[2] 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 (*S42*).

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.

b) Storage of empty values: The *impronunciable/hackdash* repository is a collaborative dashboard to organize hackathon ideas. The method *setCachedPage*[3] inserts a document into the *pages* collection (see Listing 2).

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 [75]. Consequently, empty values unnecessarily complicate queries. An alternative would be to omit the attribute from the inserted document.

[2]See https://github.com/voluntarily/vly2/blob/master/server/api/statistics/statistics.lib.js/#L8.

[3]See https://github.com/impronunciable/hackdash/blob/master/seo.js/#L85.

```
82  function setCachedPage(url){
83
84    db.collection('pages', function(err, collection) {
85      if (err){ return console.log(err); }
86
87      collection.insert({
88        key: url,
89        value: '',
90        created: new Date(),
91        pending: true
92      }, { w: 0 });
93
94    });
95  }
```

Listing 2. An example query with a *Storage of empty values* smell in *impronunciable/hackdash*

B. 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 POSTGRES, for example, dates back to 1989 [76]. 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 around 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 [42]. 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.* [8] 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.

C. Threats to Validity

Our study is based on a multivocal mapping approach exposed to threats to validity by its nature of relying on "grey" literature. While we considered several academic search engines (Google Scholar, IEEE Xplore, ACM Digital Library), very few were kept from these sources (12). Despite our quality criteria and manual inspection of the sources, one might consider the grey literature origin of our catalog as a threat to validity. In contrast to the sizeable knowledge in sources written by practitioners and enthusiasts, the marginal presence of scientific literature (1 source inducing 2 code smells) justifies the MLM study. When peer-reviewed literature is unavailable, studying grey literature [41]–[43], [54] is common in software engineering. To the best of our knowledge, there are no prior studies on MongoDB smells.

We tried to mitigate potential threats following established guidelines (*e.g.,* [40], [55]) by relying on various search engines, minimizing search bias, selecting diverse sources, theoretical saturation for stop criteria, employing quality criteria, and open coding with multiple participants. Additionally, we share our tools and data in a public repository including detailed smell descriptions with references to all sources [51].

VII. Related Work

Code smells [18] have been studied in various languages and contexts. Many researchers have proposed catalogs or taxonomies for object-oriented code smells [18], [77]–[80] and investigated their effects on software systems [19], [81]–[85].

Smells in Database Communication: There is significant literature about smells related to database communication. Karwin *et al.* [31] published a book on SQL antipatterns. Red Gate Software Ltd. published a booklet of 240 SQL code smells [86]. SQL code smells do not necessarily cause bugs but impact the performance of the database communication, thus, the system itself [29], [32], [87], [88].

Some tools exist to detect smells in database communication. For example, Van Den Brink *et al.* analyze SQL queries in PL/SQL, COBOL, and Visual Basic systems [89]. Nagy *et al.* implemented SQLInspect to find SQL antipatterns defined by Karwin [31] embedded in Java programs [38]. Chen *et al.* detected antipatterns in systems using ORM [28]. In their follow-up work, they investigated the performance impact of redundant data accesses [29] and how web applications can be improved by refactoring performance antipatterns [90].

Yang *et al.* detected ORM performance antipatterns in Ruby on Rails applications with dynamic analysis [88]. Later they presented PowerStation, a RubyMiner IDE plugin to detect ORM inefficiency problems and suggest fixes to developers [91]. Yan *et al.* also proposed static analysis to identify and fix ORM issues in Ruby on Rails applications [92]. Cheung *et al.* proposed a lazy evaluation approach to batch and reduce the number of round trips to the database [93]. Lyu *et al.* studied Android apps and identified potential problems associated with local database usage [94].

They implemented a static approach to optimize inefficient database writes [95] and developed SAND, a static analysis tool for detecting SQL antipatterns in mobile apps [96].

MongoDB and NoSQL: Despite their popularity, there are few studies on code smells or bad practices in NoSQL databases. Gomez *et al.* compared the performance of six MongoDB databases containing the same data set but with different data structuring choices [97]. Imam *et al.* suggested 23 guidelines for designing a document-oriented database [98]. Such guidelines can also be found in the MongoDB Applied Design Patterns book by Copeland *et al.* [99]. Kumar *et al.* performed a security analysis of unstructured data in NoSQL MongoDB databases [11]. Wen *et al.* proposed a tool for discovering access control vulnerabilities in web applications using MongoDB.

Summary: Overall, to the best of our knowledge, our work is the first attempt to organize MongoDB code smells or antipatterns in a catalog.

VIII. 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.[4] 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 highly relevant topics for developers and practitioners, but have received little attention from researchers so far. Furthermore, the historical analysis presented in Section VI shows a sharp increase of interest in the topic in recent years.

The catalog of MongoDB smells presented in this paper constitutes a starting point for future research in the field. First, this catalog could still be refined and further extended. Second, automated detection strategies could be implemented for all the smells of the catalog, which in turn will enable further empirical investigations. For instance, we intend to investigate the *evolution* of the smells, and to understand why developers introduce them and when and how they fix them. Another research objective would be to provide recommendations to (automatically) remove the detected smells. Given the ephemeral nature of database-related source code, it is easy to foresee that removing the smells is a non-trivial endeavor.

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

[4]Due to space reasons we discussed only selected example smells, but we provide a complete and detailed list in our online appendix.

REFERENCES

[1] S. Scherzinger and S. Sidortschuck, "An empirical study on the design and evolution of NoSQL database schemas," in *Proc. of ER'20*. Springer, 2020, pp. 441–455.

[2] P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "An empirical study of (multi-) database models in open-source projects," in *Proc. of ER'21*. Springer, 2021, pp. 87–101.

[3] K. Kaur and R. Rani, "Modeling and querying data in NoSQL databases," in *Proc. of BigData'13*. IEEE, 2013, pp. 1–7.

[4] "DB-Engines Ranking." [Online]. Available: https://db-engines.com/en/ranking

[5] S. Scherzinger, M. Klettke, and U. Störl, "Managing schema evolution in NoSQL data stores," in *Proc. of DBPL'13*, Trento, Italy, 2013.

[6] L. Meurice and A. Cleve, "Supporting schema evolution in schema-less NoSQL data stores," in *Proc. of SANER'17*. IEEE, 2017, pp. 457–461.

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

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

[9] G. Zhao, W. Huang, S. Liang, and Y. Tang, "Modeling MongoDB with relational model," in *Proc. of EIDWT'13*. IEEE, 2013, pp. 115–121.

[10] B. Maity, A. Acharya, T. Goto, and S. Sen, "A framework to convert NoSQL to relational model," in *Proc. of ACIT'18*. ACM, 2018, pp. 1–6.

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

[12] S. Wen, Y. Xue, J. Xu, H. Yang, X. Li, W. Song, and G. Si, "Toward exploiting access control vulnerabilities within MongoDB backend web applications," in *Proc. of COMPSAC'16*. IEEE, 2016, pp. 143–153.

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

[14] A. Ron, A. Shulman-Peleg, and E. Bronshtein, "No SQL, no injection? Examining NoSQL security," in *Proc. of W2SP'15*. arXiv, 2015.

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

[16] B. Hou, K. Qian, L. Li, Y. Shi, L. Tao, and J. Liu, "MongoDB NoSQL injection analysis and detection," in *Proc. of CSCloud'16*. IEEE, 2016, pp. 75–78.

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

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

[19] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proc. of ICSM'12*. IEEE, 2012, pp. 306–315.

[20] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proc. of ICSE'15*. IEEE, 2015, pp. 403–414.

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

[22] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *Proc. of SANER'18*. IEEE, 2018, pp. 612–621.

[23] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[24] R. Tighilt, M. Abdellatif, N. Moha, H. Mili, G. E. Boussaidi, J. Privat, and Y.-G. Guéhéneuc, "On the study of microservices antipatterns," in *Proc. of EuroPLoP'20*. ACM, Jul. 2020.

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

[26] H. Foidl, M. Felderer, and R. Ramler, "Data smells: Categories, causes and consequences, and detection of suspicious data in AI-based systems," in *Proc. of CAIN'22*. ACM, 2022, pp. 229–239.

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

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

[29] ——, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1148–1161, Dec. 2016.

[30] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, "HBSniff: A static analysis tool for Java Hibernate object-relational mapping code smell detection," *Science of Computer Programming*, vol. 217, p. 102778, 2022.

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

[32] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of SQL code smells in data-intensive systems," in *Proc. of MSR'20*. ACM, 2020, pp. 327–338.

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

[34] P. Dintyala, A. Narechania, and J. Arulraj, "SQLCheck: Automated detection and diagnosis of SQL anti-patterns," in *Proc. of SIGMOD'20*. ACM, 2020, pp. 2331–2345.

[35] F. G. De Almeida Filho, A. D. F. Martins, T. Da Silva Vinuto, J. M. Monteiro, I. P. De Sousa, J. De Castro MacHado, and L. S. Rocha, "Prevalence of bad smells in PL/SQL projects," in *Proc. of ICPC'19*. IEEE, 2019, pp. 116–121.

[36] E. Eessaar, "Automating detection of occurrences of PostgreSQL database design problems," in *Proc. of DB&IS'20*. Springer, 2020, pp. 176–189.

[37] R. Gupta and S. Kumar Singh, "A novel metric based detection of temporary field code smell and its empirical analysis," *Journal of King Saud University - Computer and Information Sciences*, 2021.

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

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

[40] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[41] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[42] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[43] F. Kamei, I. Wiese, C. Lima, I. Polato, V. Nepomuceno, W. Ferreira, M. Ribeiro, C. Pena, B. Cartaxo, G. Pinto, and S. Soares, "Grey literature in software engineering: A critical review," *Information and Software Technology*, vol. 138, p. 106609, 2021.

[44] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Information and Software Technology*, vol. 137, p. 106593, 2021.

[45] Y. Wang, M. V. Mäntylä, Z. Liu, J. Markkula, and P. Raulamo-jurvanen, "Improving test automation maturity: A multivocal literature review," *Software Testing, Verification & Reliability*, Feb. 2022.

[46] F. Ricca, A. Marchetto, and A. Stocco, "AI-based test automation: A grey literature analysis," in *Proc. of ICSTW'21*. IEEE, 2021, pp. 263–270.

[47] ——, "A retrospective analysis of grey literature for AI-supported test automation," in *Quality of Information and Communications Technology*, J. M. Fernandes, G. H. Travassos, V. Lenarduzzi, and X. Li, Eds. Springer, 2023, pp. 90–105.

[48] G. Recupito, F. Pecorelli, G. Catolino, S. Moreschini, D. D. Nucci, F. Palomba, and D. A. Tamburri, "A multivocal literature review of MLOps tools and features," in *Proc. of SEAA'22*. IEEE, Aug. 2022.

[49] F. Kamei, G. Pinto, I. Wiese, M. Ribeiro, and S. Soares, "What evidence we would miss if we do not use grey literature?" in *Proc. of ESEM'21*. ACM, 2021.

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

[51] B. Cherry, "Online Appendix," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10517104

[52] "MongoDB Node Driver." [Online]. Available: https://www.mongodb.com/docs/drivers/node/

[53] "Mongoose ODM." [Online]. Available: https://mongoosejs.com/

[54] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta, "Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab," in *Proc. of ESEC/FSE'20*. ACM, 2020, pp. 327–337.

[55] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," EBSE Technical Report, School of Computer Science and Mathematics, Keele University, Tech. Rep., Jan. 2007.

[56] A. Tahir, J. Dietrich, S. Counsell, S. Licorish, and A. Yamashita, "A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites," *Information and Software Technology*, vol. 125, p. 106333, Sep. 2020.

[57] G. Gezici, "Case study: The impact of location on bias in search results," 2022.

[58] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *Proc. of ICSE'16*. ACM, 2016, pp. 120–131.

[59] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019.

[60] V. Garousi, K. Petersen, and B. Ozkan, "Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review," *Information and Software Technology*, vol. 79, pp. 106–127, 2016.

[61] I. Steinmacher, M. A. Graciotto Silva, M. A. Gerosa, and D. F. Redmiles, "A systematic literature review on the barriers faced by newcomers to open source software projects," *Information and Software Technology*, vol. 59, pp. 67–85, 2015.

[62] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, 2019.

[63] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. of ICSE'13*. IEEE, 2013, pp. 712–721.

[64] Z. Yang, C. Wang, J. Shi, T. Hoang, P. Kochhar, Q. Lu, Z. Xing, and D. Lo, "What do users ask in open-source AI repositories? An empirical study of GitHub issues," in *Proc. of MSR'23*. IEEE, 2023.

[65] "MongoDB Aggregation Operations." [Online]. Available: https://www.mongodb.com/docs/manual/aggregation/

[66] "MongoDB Map-Reduce." [Online]. Available: https://www.mongodb.com/docs/manual/core/map-reduce/

[67] "MongoDB Replication." [Online]. Available: https://www.mongodb.com/docs/manual/replication/

[68] "MongoDB Security Checklist." [Online]. Available: https://www.mongodb.com/docs/manual/administration/security-checklist/

[69] "MongoDB Manage Users and Roles." [Online]. Available: https://www.mongodb.com/docs/upcoming/tutorial/manage-users-and-roles/

[70] "MongoDB Sharding." [Online]. Available: https://www.mongodb.com/docs/manual/sharding/

[71] "MongoDB Release Notes." [Online]. Available: https://www.mongodb.com/docs/manual/release-notes/

[72] "MongoDB re-tag for release r0.0.3." [Online]. Available: https://github.com/mongodb/mongo/releases/tag/r0.0.3

[73] "CodeQL." [Online]. Available: https://codeql.github.com/

[74] "Voluntarily platform." [Online]. Available: https://www.voluntarily.nz/

[75] "MongoDB $exists." [Online]. Available: https://www.mongodb.com/docs/manual/reference/operator/query/exists/

[76] "A brief history of PostgreSQL." [Online]. Available: https://www.postgresql.org/docs/current/history.html

[77] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proc. of ICSM'03*. IEEE, 2003, pp. 381–384.

[78] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, pp. 395–431, Sep. 2006.

[79] R. Marticorena, C. López, and Y. Crespo, "Extending a taxonomy of bad code smells with metrics," in *Proc. of WOOR'06*, 2006, p. 6.

[80] G. Rasool and Z. Arshad, "A lightweight approach for detection of code smells," *Arabian Journal for Science and Engineering*, vol. 42, no. 2, pp. 483–506, 2017.

[81] F. Khomh, M. Di Penta, and Y. G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proc. of WCRE'09*. IEEE, 2009, pp. 75–84.

[82] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. of ESEM'09*. IEEE, 2009, pp. 390–400.

[83] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2639–2653, Oct. 2013.

[84] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proc. of ICSE'13*. IEEE, 2013, pp. 682–691.

[85] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of Android code smells," in *Proc. of MOBILESoft'16*. ACM, 2016, pp. 59–69.

[86] P. Factor, *SQL Code Smells*. Red Gate Software Ltd., 2014. [Online]. Available: https://github.com/Phil-Factor/SQLCodeSmells

[87] Y. Lyu, A. Alotaibi, and W. G. J. Halfond, "Quantifying the performance impact of SQL antipatterns on mobile applications," in *Proc. of ICSME'19*. IEEE, Sep. 2019, pp. 53–64.

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

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

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

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

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

[93] A. Cheung, S. Madden, and A. Solar-Lezama, "Sloth: Being lazy is a virtue (When issuing database queries)," *ACM Trans. Database Syst.*, vol. 41, no. 2, Jun. 2016.

[94] Y. Lyu, J. Gui, M. Wan, and W. G. J. Halfond, "An empirical study of local database usage in Android applications," in *Proc. of ICSME'17*, 2017, pp. 444–455.

[95] Y. Lyu, D. Li, and W. G. J. Halfond, "Remove RATs from your code: Automated optimization of resource inefficient database writes for mobile applications," in *Proc. of ISSTA'18*. ACM, 2018, pp. 310–321.

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

[97] P. Gómez, R. Casallas, and C. Roncancio, "Data schema does matter, even in NoSQL systems!" in *Proc. of RCIS'16*. IEEE, 2016.

[98] A. A. Imam, S. Basri, R. Ahmad, J. Watada, M. T. Gonzlez-Aparicio, and M. A. Almomani, "Data modeling guidelines for NoSQL document-store databases," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, pp. 544–555, 2018.

[99] R. Copeland, *MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database*. O'Reilly Media, Inc., 2013.