The Journal of Systems and Software 159 (2020) 110451

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

In Practice

Using Orthogonal Defect Classification to characterize NoSQL database defects

João Agnelo, Nuno Laranjeiro*, Jorge Bernardino[1]

CISUC, Department of Informatics Engineering, University of Coimbra, Portugal

ARTICLE INFO

Article history:
Received 4 March 2019
Revised 31 July 2019
Accepted 21 October 2019
Available online 23 October 2019

Keywords:
Defect analysis
NoSQL
Orthogonal Defect Classification
Software defect
Software fault

ABSTRACT

NoSQL databases are increasingly used for storing and managing data in business-critical Big Data systems. The presence of software defects (i.e., bugs) in these databases can bring in severe consequences to the NoSQL services being offered, such as data loss or service unavailability. Thus, it is essential to understand the types of defects that frequently affect these databases, allowing developers take action in an informed manner (e.g., redirect testing efforts). In this paper, we use Orthogonal Defect Classification (ODC) to classify a total of 4096 software defects from three of the most popular NoSQL databases: MongoDB, Cassandra, and HBase. The results show great similarity for the defects across the three different NoSQL systems and, at the same time, show the differences and heterogeneity regarding research carried out in other domains and types of applications, emphasizing the need for possessing such information. Our results expose the defect distributions in NoSQL databases, provide a foundation for selecting representative defects for NoSQL systems, and, overall, can be useful for developers for verifying and building more reliable NoSQL database systems.

1. Introduction

In recent years, we observed a noticeable growth in the use of NoSQL Database Management Systems that nowadays compete with the older and much more mature, Relational Databases. This evolution derives from the needs of the Big Data era, namely the need for supporting high data volumes, high velocity, and also variety (i.e., different forms of data), along with easier horizontal scalability (Leavitt, 2010). In these systems, losing data or being unable to use the storage service has negative consequences not only for the user, but also for the provider, that sees its reputation affected. Such events are many times the result of the presence of a defect in the software (Marks, 2014; Hulme, 2017).

Software defects (i.e., bugs) can range from simple spelling or grammar mistakes in user messages, to security vulnerabilities which, once exploited, may result in disclosure of private information or in infrastructure damage (Avizienis et al., 2004), for example. There are many cases of software defects that have led to disastrous consequences, from misguided space rockets, to crashing military airplanes, and ultimately, human deaths (Mellor, 1994). For the sake of dependability, any software defect found should be fixed in due time. Otherwise, and this is especially true for high priority defects (e.g., defects that for instance impair the core capabilities of the system) the product reputation may collapse to an irrecoverable level (this may also apply to the company behind the product, or the one using it). In the case of the most popular NoSQL databases, the identification and fixing of bugs is largely supported by their large communities, which actively find software defects, report them in an issue-tracking platform, and trigger their correction.

Understanding the nature of software defects (e.g., by using structured methods for defect classification Chillarege, 1996; IEEE Standard Classification for Software Anomalies 2010; Grady, 1992) can provide extremely useful information for developers, who can better focus their software verification efforts or, for instance, concentrate their development efforts in certain components or code areas known to be prone to a particular software defect type (Chillarege, 1996). This kind of effort has been undertaken for many and very different types of software that operate in distinct domains, including browsers, games, operating systems, satellite software, business-critical applications, build systems, or machine learning software, just to name a few (Durães and Madeira, 2006; Thung et al., 2012; Xia et al., 2013; Silva et al., 2017). Many times, the general hypothesis placed is that the specificities involving the system (e.g., programming language, project nature and requirements, type and experience of the application development team) make it necessary for such specific analysis to take place. In this

* Corresponding author.
E-mail addresses: jagnelo@student.dei.uc.pt (J. Agnelo), cnl@dei.uc.pt (N. Laranjeiro), jorge@isec.pt (J. Bernardino).
[1] ISEC – Polytechnic of Coimbra, Portugal.

https://doi.org/10.1016/j.jss.2019.110451
0164-1212/

context, the analysis of defects affecting storage intensive systems (i.e., NoSQL databases, in the case of this work) has been disregarded so far.

In this paper, we analyze the defects of three of the most popular NoSQL databases: MongoDB (2019), Cassandra (Apache Software Foundation, 2019), and HBase (Apache Software Foundation 2019b). We begin by training one researcher with the application of Orthogonal Defect Classification (ODC) (Chillarege, 1996) against a total of 300 defects affecting these databases. We discard the training dataset (to avoid using a potentially low-quality annotated dataset, as the quality of the ODC labelling tends to increase with the experience of the annotator) and then analyze a set of 4096 defects extracted from the databases' issue-tracking systems and manually classify them with ODC using the previously trained ODC researcher and according to six of the eight ODC attributes (the age and source ODC attributes were not considered due to the general lack of information in the defect reports regarding these two cases). We internally double-check the classification result for 20% (i.e., 820 bugs) of the defects and then ask for two external and ODC-knowledgeable researchers to produce independent ODC classifications for an additional 20% of the total defects (10% per researcher). The outcome of the verification carried out by the external researchers revealed the overall good quality of the dataset, with *almost perfect* Cohen's Kappa agreements (Cohen, 1960) found for the majority of the ODC attributes involved.

We then analyze the resulting dataset of ODC-classified defects according to the following four perspectives: (i) we analyze the distribution of values concerning six of the eight existing ODC attributes (e.g., type of defect, conditions that trigger the defect, impact of the defect); (ii) we form pairs of attributes and analyze the value distributions, as in related work (e.g., Chillarege, 1996; Durães and Madeira, 2006; Zhi-bo et al., 2011); (iii) we analyze the defect type distribution in the top 3 most affected components per each database; and (iv) we analyze our results in perspective with previous work. We also describe a practical use case to show how this kind of analysis may be beneficial for developers by generating tests to target a database component showing high prevalence of *checking* defects.

We observed a huge variation in the distribution of defect types found in related work, which goes through the defects of various types of applications from other domains, such as browsers, operating systems, satellite onboard software, machine learning tools, or build tools. Our results have confirmed this heterogeneity, which in practice means that we cannot assume, *a priori,* the existence of a certain defect distribution. This implies that this kind of study must be carried out whenever it is important to know which are the representative defects for a certain type of system, built in specific conditions. In this work, we actually observed a unique distribution of values (i.e., in terms of the relative popularity of the types of defects found), that is not present in any of the related work analyzed. The results suggest that the overall nature of the system may be a factor that influences the distribution of defects.

We also observed great similarity across the three databases, when inspecting the results for the individual ODC attributes. Sometimes, a single value dominated the distribution. When crossing pairs of attributes, three of the main observations include: (i) the fact that testing activities are more than two times more frequently associated with reliability defects than with capability defects; (ii) checking defects (e.g., input validation) are more frequent when the impact is reliability; and (iii) checking defects are more likely to be associated with the "missing" qualifier. We also noticed disparities in the distribution of defect types in different system components, which may be justified with the nature of the component (e.g., a replication component holding Timing/Serialization defects), and, finally, we found certain types of defects being consistently associated with longer times to fix across all three databases (e.g., Function/Class/Object).

Overall, our results (available in detail, along with supporting code, at Agnelo et al., 2019) confirm the need for understanding the defect trends in NoSQL systems and bring insight regarding their defect distribution. The main contributions of this paper are the following:

- We use Orthogonal Defect Classification to provide a detailed view over a relatively large dataset of reported bugs in three popular NoSQL databases, including a view on the most affected database components;
- We show our results in perspective with the very heterogeneous related work, signaling differences and also similarities found;
- We provide an open dataset holding 4096 bug reports classified using ODC and freely available for future research (Agnelo et al., 2019).

The kind of analysis carried out in this paper can serve for process improvement in a variety of ways (e.g., performing root cause analysis Silva et al., 2017), as the analysis of field data using ODC, in general allows for better defect prevention and detection (IBM 2013) (e.g., a known tendency for having a certain type of defect in a certain component of the system represents essential information for practitioners). Thus, the data allow practitioners to obtain knowledge regarding the overall reliability – or lack thereof – of their systems, but it could also help to improve the quality of the development processes, for instance by directing verification efforts to the appropriate areas (e.g., design, algorithm) or selecting certain verification techniques (e.g., testing, code inspections). Researchers may benefit from this type of results and use them as basis for creating new verification techniques for this kind of systems (e.g., fault injection based techniques using the defect information obtained from this work, as in Durães and Madeira, 2006); or even for tailoring development processes (Børretzen and Dyre-Hansen, 2007) to specifically consider the specificities of this kind of systems and the nature of the defects that typically affects NoSQL databases.

The remainder of this paper is organized as follows. Section II presents background on ODC and discusses the related work. Section III describes our approach for classifying the extracted defects and Section IV presents and analyses the results. Section V discusses the main findings of this work and Section VI presents the threats to the validity of the work. Finally, Section VII concludes the paper.

2. Background and related work

In this section, we briefly go through the main software defect classification schemes used nowadays, with focus on the main concepts regarding the Orthogonal Defect Classification, and we then discuss the related work.

2.1. Background concepts

NoSQL databases are known to typically provide features like easy horizontal scaling, replication and data partitioning across servers, weak transactional properties (when compared to the typical ACID properties found in most relational databases), efficient distributed indexes or in-memory storage, or the ability to dynamically add new attributes to data (Cattell, 2011). Different storage models have been developed in recent years, namely: (i) key-value, in which data are represented as key-value pairs stored in key-based lookup structures (e.g., Redis, Membase); (ii) wide column, where data are represented using rows and a fixed number

Table 1
Activity-Trigger mapping.

Activity	Design review Code inspection	Unit test	Function test	System test
Trigger	• Design Conformance • Logic/Flow • Backward Compatibility • Lateral Compatibility • Concurrency • Internal Document • Language Dependency • Side Effects • Rare Situation	• Simple Path • Complex Path	• Test Coverage • Test Variation • Test Sequencing • Test Interaction	• Workload/Stress • Recovery/Exception • Startup/Restart • Hardware Configuration • Software Configuration • Blocked Test

of columns which can be easily partitioned vertically and horizontally across nodes (e.g., HBase, Cassandra); (iii) document, which are essentially key-value stores where the value is represented as a document encoded in standard semi-structured format; (iv) graph stores which are optimized towards efficient entity relationship traversals (Cattell, 2011). Research on these databases touches various areas but tends to focus on the typical needs brought by the advances in web technology, namely horizontal scalability, high availability and fault tolerance, database schema maintainability, and also reliability (Davoudian et al., 2018). Topics like efficient and reliable query and transaction processing, elasticity, or dependability and security have also been identified in the literature as open research areas (Davoudian et al., 2018).

Software Defect Classification Schemes are powerful tools which developers and researchers can use for different purposes such as development process improvement, product quality enhancement or empirical analysis (Chillarege, 1996). A typical application is within defect causal analysis processes, which aim at identifying the root cause of software engineering problems so that it is possible to take action and prevent them from occurring again (Kalinowski et al., 2012). The most popular software defect classification schemes are Hewlett-Packard's Defect Origins, Types and Modes (DOTM) (Grady, 1992), IEEE's standard 1044–2009 (IEEE-1044) (IEEE Standard Classification for Software Anomalies 2010), and Orthogonal Defect Classification (Chillarege, 1996).

Hewlett-Packard's Defect Origins, Types and Modes (DOTM) (Grady, 1992) contains three base categories to classify defects (i) *Origin* – the first activity in the defect's lifecycle in which it have been detected, (ii) *Type* – the area, of a particular Origin, that is responsible for the defect (e.g., "Logic" or "Standards" for Origin "Code", or "Software Interface" for Origin "Design"), and (iii) *Mode* – classification of why the defect occurred. IEEE 1044–2009 - IEEE Standard Classification for Software Anomalies (2010) is a relatively complex standard, mostly due to its large number of attributes. Still, this may allow for less subjectivity in its application, as the large number of attributes covers a wider variety of possibilities.

Orthogonal Defect Classification is a set of analytical methods used for software development and test process analysis to characterize software defects that consists of eight orthogonal attributes (Chillarege, 1996). ODC was created with the intention of bridging two methods commonly used for analyzing defects, namely Statistical Defect Modelling (e.g., software reliability statistical models) (Wood, 1996) and Root Cause Analysis (Wilson et al., 1993). It allows the defect classification process to be faster (an advantage of the former method) and have better accuracy in categorizing issues (similarly to the latter method) (Chillarege, 1996). ODC is widely regarded as very popular means to characterize defects and, as discussed in this section, has been used by researchers and practitioners in several contexts. Its documentation (Chillarege, 1996; IBM 2013) is easily accessible and detailed, leaving little space for doubts regarding its use. ODC is based on the definition of eight attributes grouped into two sections: opener section and closer section (IBM 2013). The *opener section* refers exclusively to the attributes that can be classified when the defect is found (i.e., independently of a later possible correction of that particular defect) and corresponds to the following attributes:

- *Activity*: indicates the activity being performed at the time the defect was disclosed (e.g., system testing);
- *Trigger*: describes what caused the defect to surface, i.e., the required condition that allowed the defect to manifest itself;
- *Impact*: refers to either the impact a user experienced when the defect surfaced (in case of a user-reported defect), or the impact a user would have hypothetically suffered, had the defect surfaced (when a developer finds a dormant defect, which is yet to be triggered).

The *closer section* refers to the attributes that can be classified when a defect has been corrected and the correction information becomes available. This section's attributes are the following:

- Target: the entity that was corrected (e.g., source code);
- Defect Type: the nature of the change that was performed to fix the defect (e.g., algorithm);
- Qualifier: this attribute complements the defect type and describes the state of the implementation prior to having been corrected (e.g., whether it was missing, incorrect, or present but unnecessary);
- Age: refers to the point in time in which the defect was introduced (e.g., introduced during the correction of another defect);
- Source: whether the defect was introduced by an external component (i.e., outsourced), or is something developed by the team itself (i.e., in-house).

At the time of writing, the most recent ODC documentation is at version 5.2 (IBM 2013) and it describes the ODC attribute set, their possible values and example cases. This ODC documentation should be used as a guideline and software development teams are allowed to create their own set of values for each attribute, in case there is a better fit for their specific software system. In the context of this work, we opted to use the sets of attribute values presented in IBM (2013), without further customization, as we are not involved with the communities developing the systems analyzed in this work (and, thus, are not able to define new attribute values with adequate criteria).

It is worth mentioning that the defect type and qualifier attributes are only applicable when a defect's target is either "Design" or "Code", which means that for all other target values, both attributes must be suppressed (IBM 2013). Additionally, there is a mapping between the activity and trigger attributes, where certain activities directly map to certain triggers (IBM 2013), this mapping is shown in Table 1.

The ODC documentation does not specify a particular procedure to apply the classification scheme, thus when classifying a defect, we go through the attributes in the order they are discussed in IBM (2013). Obviously, when a defect is found, the first step of the ODC classification is to categorize the occurrence in

the opener section attributes. When the defect is corrected, the remaining ODC attributes – the ones in the closer section – are classified. Notice that the scale is orthogonal, i.e., there are no overlaps and the order in which attributes are classified, in general, does not impact the results. The only special case is the case of Activity, where an incorrectly classified Activity will lead, in some cases, to an incorrect Trigger. So, in the case of this work, the researchers involved were made aware of this, and were instructed to pay special attention to this characteristic.

The next section summarizes related work, especially works in which the numeric outcome of the application of ODC is clearly reported by the authors. Works that do not clearly mention the outcome of bug analysis, such as the exact distribution of defect types, are excluded from the analysis.

2.2. Related work

We conducted a search through the major indexing engines, namely Google Scholar, Scopus, IEEE Xplore, ACM Portal, and Springer Link search engines. Main keywords used for searching included ODC, Orthogonal Defect Classification, Software defect, Software fault. After examining the individual search results, analyzing the references of each paper found during the search, and also examining the papers that cited each of the papers found, we were able to find 17 papers of various types (e.g., published in peer-reviewed conference proceedings and journals) that use ODC to classify software defects and that, in total, analyze 60 different applications, going through a total of 13,171 bugs. The next paragraphs describe the related work found, but, note that we also summarize the results obtained by other authors in Section V, in perspective with our own results (and, therefore, do not present them in depth in this section).

The related work that uses ODC fits in essentially two groups of work. In *Group I* authors carry out *empirical analyses* using ODC to characterize a given set of bugs (e.g., due to the importance of the system or absence of prior data) within a certain system context (e.g., machine learning tools, build systems, safety-critical software). The works belonging to *Group II* also perform empirical analyses, but the work has an additional goal, a second application, which is, most of the times, to use the ODC data for *process improvement* (in *lato sensu*). In this latter group of works, it is frequent to see the authors using the field data to perform root-cause analyses (Silva et al., 2017) or apply it within the scope of another proposal (e.g., fault injection) (Durães and Madeira, 2006). Our work fits directly in the former group.

Regarding the works belonging to *Group I*, which concentrate on performing empirical analyses, Basso et al. (2009) use field data of several Java-based systems in production and apply ODC, with the goal of understanding fault representativeness, including security vulnerabilities. The authors put their results in perspective with previous work (Durães and Madeira, 2006), which targeted systems written in C. Thung et al. (2012) perform an empirical study of bugs in machine learning systems. The authors aim to understand how often bugs appear in these systems, which types of bugs surface, how long does it take to fix different types of bugs, and how many files are impacted by the defects. Xia et al. (2013) analyze bugs affecting four different build systems (e.g., Ant, Maven), considering that the reliability of these tools could affect the build process reliability. The authors analyze a random sample of 800 bug reports and their corresponding fixes and present the different relative frequency types of bugs. Also discussed is the relationship between types of bugs and their corresponding severities.

Fonseca and Vieira (2008) used ODC to manually characterize 655 software bugs related to security using six widely used web applications written in PHP. The results show that, in that context, just a part of the types of software bugs is related to security. The authors also note that web application vulnerabilities result from software bugs that tend to affect a reduced set of statements. The work allows future definition of realistic fault models that originate security vulnerabilities in web applications. Based on the fact that it is rare to find analysis of software defects for safety-critical systems, Silva and Vieira (2014) applied ODC to a set of 243 defects obtained from systems that operate in the aerospace and space domains. In addition to the ODC results, the authors have highlighted the challenges of performing the classification, in particular in applying the broad ODC approach to certain specific types of issues. Xuan et al. (2015) carry out an empirical evaluation of 300 software bugs in industrial financial systems. These systems tend to be heavily complex, holding many parts with intricate business logic, which makes them different from systems studied in related work. The authors have analyzed bug density, detection time, the distribution of bug categories, and the relationship between categories and bug severity.

Morrison et al. (2018) use ODC to characterize and understand the differences between the discovery and resolution of non-security related defects compared to vulnerabilities. The analysis of the differences allows for security-specific software development process improvements, which are lightly mentioned by the authors, but the work's research questions aim at: i) understanding if detection and discovery of vulnerabilities occur in the same manner as for plain software defects, and ii) are the different types of vulnerabilities discovered and resolved in the same way. The work goes through 1166 bugs in three open-source projects: FireFox, PPHMyAdmin, and Google Chrome and applies ODC+*V* to better capture vulnerability data. Rahman et al. (2018) classify software defects in Puppet scripts belonging to the open software repositories of Mirantis, Mozilla, Openstack, and Wikimedia Commons. The motivation is that, besides the growing importance of this kind of scripts, the nature of the defects in Puppet scripts has not yet been categorized. Using 2 raters per defect (89 raters in total), the authors apply ODC to 3187 defects and observe that configuration-related defects are prevalent.

Regarding the works belonging to *Group II*, Lyu et al. (2003) studied the nature, type, detectability, and effect of software defects in program versions built by 34 programming teams for a critical flight application. ODC was used to classify the defects. The authors applied mutation testing techniques to create mutants, using real faults. Among other aspects, the results showed that coverage testing is an effective way for detecting faults, but mutation testing can be a more trustworthy indicator of test quality. Lutz et al. (2004) used ODC to characterize bugs and discover defect patterns in the spacecraft domain. The results obtained allow the authors to produce a short list of recommendations to avoid undesirable defect patterns and, more generally, for process improvement.

Christmansson and Chillarege (1996) use ODC to present a set of errors which emulate software faults. The proposed approach uses field data for generating a set of injectable errors (an error is an effect of a fault Avizienis et al. (2004)), each of which being defined by (i) error type, (ii) error location, and (iii) injection condition. The authors resorted to ODC, as a way of categorizing the required field data, and classified 408 real software defects with just one of the many attributes ODC offers – *Defect type*. The work answers questions related with fault forecasting and has established a solid basis for work in the field of dependable systems.

Durães and Madeira (2006) analyzed how software faults can be injected in a source code-independent manner. The authors analyzed 668 real software defects and classified them using ODC. The resulting dataset was analyzed for patterns which were then used in the definition of a set of software fault emulation operators. These operators allow for software faults to be injected in real software, which is particularly important in cases where field

data is scarce, regarding the typical faults that affect the software. The emulation operators derived from the field study allowed the authors to propose a fault injection technique named G-SWIFT.

Børretzen and Dyre-Hansen (2007) characterized faults in five business-critical industrial projects with the goal of determining possible areas of process improvement. Findings include the fact that a certain fault type (Function/Class/Object) generally dominates the bug reports from all projects and that there is a tendency for some fault types to be marked as more severe as others (e.g., relationship, timing/serialization). The work resulted in a set of recommendations to the organization fault reporting process, in particular the need for enriching the information contained in the fault reports.

The work by Gupta et al. (2009) aims at characterize bugs present in reusable software to understand the reasons for the typical lower defect densities in this kind of software. The authors conduct a case study in an industrial setting and apply ODC to 1310 bugs and then perform a qualitative root cause analysis. Results indicate that several factors must be taken into account when analyzing the cause-effect relationship between software reuse and lower defect density. The findings should also encourage practitioners to implement further software reuse policies.

Silva and Vieira (2016) applied ODC in the context of space critical systems, discovering difficulties in classifying about 32% of the bugs analyzed due to context specificities. Thus, the authors propose an adaptation of the taxonomy that is particularly suitable for critical systems, allowing for a more effective classification. Silva et al. (2017) carried out a root cause analysis on 1070 software defects belonging to the subsystems of four safety-critical space systems. The authors applied a version of ODC particularly tailored for the context and examined types of defects, impacts, and triggers. The authors propose modifications to the way software is developed and also to the verification and validation activities. The idea is that the analysis of the field data using ODC allows better defect prevention and detection. Finally, Sotiropoulos et al. (2017) characterize bugs in navigation software for outdoor robots. The authors use ODC and the analysis of the triggers and effects of the bugs shows that a large part can be disclosed using low-fidelity simulation. Also, the work provides insight into navigation scenarios that could serve as basis for testing.

As we have seen, previous work either fits in purely empirical studies (which is also the case for this work) or in empirical studies accompanied of a second goal (e.g., fault injection). In the related work, defect classification schemes have been used in a very large variety of contexts and tend to produce heterogeneous results (as discussed in Section IV). This strongly suggests that there is the need for understanding the nature of the software defects involved in the particular class of systems studied in the present work. To the best of our knowledge and at the time of writing, we produced the largest ODC dataset among those analyzed in related work, which we make available at (Agnelo et al., 2019) for future research. We also use 6 out the 8 ODC attributes (in practice, all attributes for which we had sufficient information – the exception is 'age' and 'source') in the analysis which is something rare to observe in related work (mostly due to the amount of human effort involved).

3. Study design

This section describes the design of our study, which has the general goal of understanding the overall nature of software defects present in NoSQL databases. With the resulting defect classification, we will be mostly aiming at:

(i) Understanding if the types of defects (across the different ODC attributes) that tend affect this class of systems are consistently the same (or not) across different NoSQL databases. In practice, this is evidence of some shared nature between the three systems.
(ii) Detecting coupling relations between ODC attribute values (e.g., a checking defect strongly associated with reliability problems). This allows for process improvement, as in the first goal, but in a more fine-grained manner, e.g., by allowing the definition of testing activities known to perform better with certain types of defects or in certain scenarios, such as robustness testing, which tends to detect checking defects that impair the robustness of the system, hence its reliability in more general terms).
(iii) Signaling variations in the types of defects within the different components of each database, which may further help finetuning the development process by selecting, extending, or directing verification activities to problematic components (e.g., using static analysis to detect command injection bugs in a certain component that is exposed to user commands).

In practice, and to reach a reliable defect classification that allows reaching the abovementioned goals, we go through the following five steps that are detailed in the next paragraphs:

(1) Selection of NoSQL databases;
(2) Training a researcher, named *Researcher1*, to learn the classification process, using real defect descriptions (selected and extracted from the databases' public defect-tracking platforms);
(3) Selection and extraction of a dataset composed of defect descriptions from the databases' public defect-tracking platforms;
(4) Manual classification of each software defect according to the ODC scheme, carried out in five batches for each database;
(5) Manual verification of the defects classified in *step 4)* (with possible reclassification of some of the defects):
    a. *Internal Verification* carried out by *Researcher1* to signal errors and improve the overall quality of the dataset. As described in the next paragraphs, in an effort to gradually increase the quality of the classification, this was a task incrementally carried out along with the classification procedure (and not only after the complete classification was finished);
    b. *External Verification* using two new independent classifications, each of them carried out by external researchers (*Researcher2* and *Researcher3*).

We started by *selecting NoSQL databases*, so that we could gather defect data that would allow us to characterize the defects present in these kinds of systems. We aimed for popular open-source NoSQL databases, as we are interested in analyzing systems that are important for users. As these are supported by larger communities, we also had the expectation of finding large quantities of defect descriptions in their public defect-tracking systems, thus providing sufficient data for analysis.

We started by examining the database popularity rankings in *db-engines.com* (DB-Engines Ranking 2017), *stackoverkill.com* (SQL, and NoSQL Ranking - StackOverkill 2017), and *kdnuggets.com* (Top NoSQL Database Engines 2017) and found a general agreement in the following ranking (from the most popular to the least): (1) MongoDB; (2) Redis; (3) Cassandra; (4) HBase; and (5) Neo4j. Our intention was to analyze the defects of the top three most popular databases, in a semi-automated manner (e.g., although the analysis is manual, supporting processes, such as defect selection and defect description extraction, are automatic). Of the five, three (MongoDB, Cassandra, and HBase) use the popular defect-tracking system JIRA, and the remaining two (Redis and Neo4j) use GitHub.

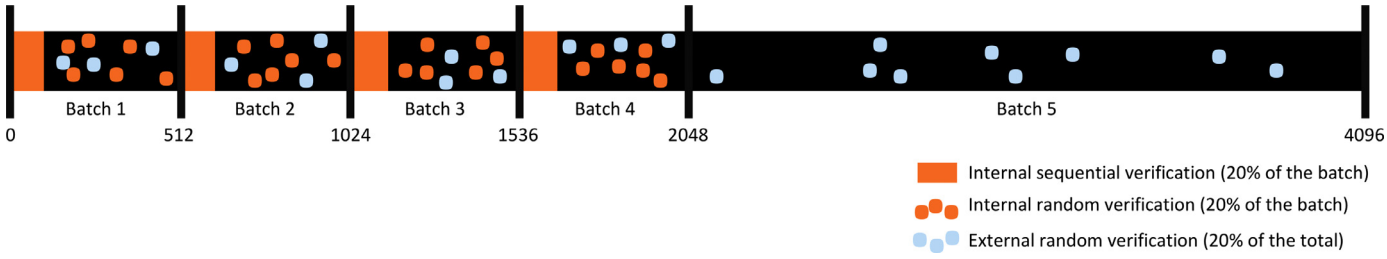

Fig. 1. Diagram of the internal (in orange) and external (in blue) verification of the classification process for each batch. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The latter were excluded from the analysis in this work, as their defect-tracking platforms severely lacked the necessary information for a proper ODC classification. Additionally, while each JIRA deployment presented a lot more than 1000 defects, Redis and Neo4j's GitHub platforms barely reached half of that. The number of defects at our disposal was an important factor, as the analysis of more defects can allow for more accurate analysis.

The three selected databases (MongoDB, Cassandra, and HBase) serve the same general purpose but obviously also have technical differences. At the time of writing and, as reported by OpenHub (https://openhub.net), MongoDB is mostly written in *C++* (42% of the code) and Go (33% of the code). The whole codebase is close to 2 million lines, created by 492 contributors (135 active contributors in the last year). Cassandra's codebase is around 400 K lines of code, with 96% of the codebase being in Java that has been created by 372 contributors (80 active contributors in the last year). Finally, HBase's codebase is around 920 K lines of code, of which 86% are written in Java that have been committed by 337 contributors (135 active contributors in the last year).

As a way of overcoming the learning curve inherent to applying the ODC scheme and also with the goal of obtaining more reliable results, we chose to use an initial set of 300 defect descriptions (100 defects for each of the three databases) to *train the classification process*. This also allowed for a better understanding of how these databases' JIRA platforms are structured, and the procedures used by the communities and development teams when registering and describing defects. This training effort was done by randomly selecting defect reports from the whole range of defects present in each database's JIRA platform, and classifying them individually. The results obtained from this process were discarded as they were merely used for training and learning.

The *selection and extraction of the set of defects* for applying the ODC scheme was carried out as follows. We started by defining the period of analysis, for which we opted to set no particular limit – we considered all defects present in each of the databases' JIRA platforms as potential candidates for classification. Obviously, for our analysis we use only defects tagged with *closed* and *resolved* (i.e., defects whose existence has been confirmed by the developers and for which a correction already exists), so that our analysis would not involve any false positive (i.e., a report of a defect that actually does not exist). Also, this allows us to perform the full ODC classification, which requires a defect to have passed through both the opener section and closer section stages.

We first identified all closed and resolved defects, registered until May 1st 2017, which accounts for 7456 for MongoDB, 4576 for Cassandra, and 4905 for HBase (respectively 8, 8, and 10 year periods of defect reporting). As the classification procedure is manual and we must reduce the human effort involved (also as a way to reduce the risk of erroneous classification), we selected a subset of the defects, which we aimed to be around one fourth of the total closed and resolved defects, for a total of *4096 defects*. This resulted in a total of 1618, 1095 and 1383 defects for MongoDB, Cassandra and HBase.

The next step was to *classify each defect* with the ODC scheme, according to the following steps:

i. Interpretation of the defect's written description;
ii. Classification of the opener section ODC attributes;
iii. Analysis and interpretation of the source code changes;
iv. Classification of the closer section ODC attributes.

The information required to perform the abovementioned steps is held in each defect description. This is essentially a textual description that typically includes the conditions that allowed the defect to surface (e.g., a memory leak in a routine), the environment in which it occurred (e.g., the defect surfaced during a resource-intensive procedure), and the corrective measures that were applied in the respective fix (e.g., the defect was fixed by adding the missing functionality to release unused resources), and obviously, a description of the defect itself. Note also that these steps are the direct application of the ODC methodology, without any particular adaptation. The only relevant aspect to mention is that we did not use the age and source attributes because, in general, the defect reports did not include sufficient or clear information on these attributes.

Due to the human intervention involved in the classification process, the final result of classifying a given set of defects may hold errors. This is due to the fact that the application of a given defect classification scheme may involve some subjectivity (i.e., in some cases, two users of the same scheme could classify a given defect in a different way), which is aggravated when the defect is not described in a complete or clear way. Obviously, there is also human error in the process, even when the description is completely accurate. This is a common issue in works that use defect classification schemes, as mentioned in Henningsson and Wohlin (2004) and El Emam and Wieczorek (1998), which is usually mitigated by the use of more than just one rater (i.e., more than one human classifying each bug). Due to the very large size of our dataset and associated huge manual effort, we verify part of the bugs and involve external researchers in the process, as discussed in the next paragraph.

To *verify the classification* of the bugs and, most of all to have some insight regarding the reliability of the annotated dataset, we performed two verification activities against 40% of the bugs (1640 out of 4096 bugs), which we name *Internal* and *External* and that are depicted in Fig. 1. During the *Internal Verification* activity, the researcher responsible for originally classifying the bugs (named *Researcher1*) revisited 20% of the bugs (820 of the 4096 bugs) to check for errors and correct any misclassification found. In practice, the whole set of bugs was divided in two, and this internal verification was performed against the first half. This first half (i.e., 2048 bugs) was divided in four batches (512 bugs per batch) that were verified in the following manner. By the end of each batch, the researcher double-checked a total of 40% of the already classified defects in that batch, according to the following distribution: (i) the first 20% of the bugs in the batch; (ii) a random selection of another 20% from the remaining defects in the batch. The *Exter-*

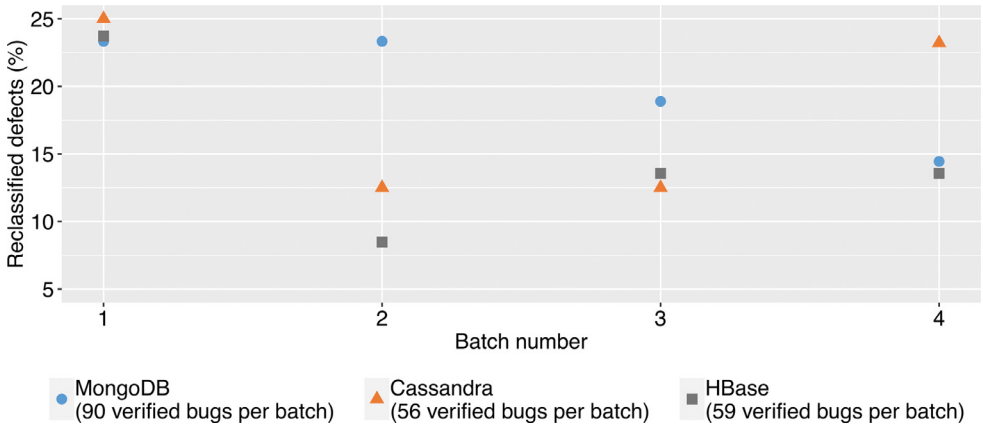

Fig. 2. Reclassified defects in respect to the total number of bugs per batch and for each of the four initial batches, for each database.

Table 2
Confusion matrixes for the verification procedures.

	Defect Type	A/I	C	A/M	F/C/O	T/S	I/OOM	R
Internal (746 bugs)	Assignment/Initialization (A/I)	34		4				
	Checking (C)		71	6				
	Algorithm/Method (A/M)	4	4	394	13	1	4	
	Function/Class/Object (F/C/O)			6	126		1	1
	Timing/Serialization (T/S)			1		4		
	Interface/O-O Messages (I/OOM)	1		3	1		65	1
	Relationship (R)							1
External 1 (379 bugs)	Assignment/Initialization	20		2				
	Checking		41	3				
	Algorithm/Method		2	216	5			
	Function/Class/Object			3	44			
	Timing/Serialization			1		6		
	Interface/O-O Messages		2	1	3		29	
	Relationship							1
External 2 (382 bugs)	Assignment/Initialization	14		8			9	
	Checking		34	8				
	Algorithm/Method	5		219				
	Function/Class/Object			8	51			
	Timing/Serialization					2		
	Interface/O-O Messages			2			22	
	Relationship							

nal Verification activity involved independent classification by two external researchers (named *Researcher2* and *Researcher3*) of 20% randomly selected, but non-overlapping bugs (i.e., 820 bugs in total, 410 bugs per each of the two researchers) across the whole set of 4096 bugs.

In Fig. 2, we show the error results of the *Internal Verification*, detailing the number of defects that had their classification corrected per batch (i.e., where at least one of the ODC attributes changed its value) and including an error trendline for each of the three databases. In terms of the overall accuracy of the *internal classification*, we obtained 82% of correctness, with the researcher maintaining its classification in 672 bugs of the 820 verified bugs. Notice that this is a pessimistic view of the process in the sense that we mark a bug as incorrectly classified, even if the error refers to just one of the six attributes (with the remaining five being correct). Obviously, there is always some error associated with the application of ODC, which is the result of the human intervention during the process, but also due to the limited or ambiguous information found in some defect reports, which introduces some uncertainty about which value to apply to a given ODC attribute. Note also that among the three databases, there are a few differences in the way defects are reported, which explains the higher error values for the first batches in each database.

Table 2 shows three confusion matrixes that detail the outcome of the verification procedure, which is composed of three verification tasks, named *Internal* (carried out by *Researcher1*), *External 1* (carried out by *Researcher2*) and *External 2* (carried out by *Researcher3*), regarding the *Defect Type* attribute. We chose to present the detailed results for this case, due to the fact that it is the most widely used ODC attribute in related work (Durães and Madeira, 2006; Thung et al., 2012; Basso et al., 2009). Notice that the count of bugs for *Internal* is 746, which is slightly below the previously mentioned 40% (820 bugs) as we are considering only the bugs where the *Target* attribute had Code as value (thus, qualify for being marked with a defect type) and were verified and kept as Code by *Researcher1*. Thus, 69 bugs were confirmed to not be code defects and the value of the *Target* attribute was changed by *Researcher1* for 5 bugs (i.e., 2 bugs left *Target* Code and 3 bugs became Code problems). This rationale applies, in the same manner to the numbers presented for *External 1* and *External 2* (i.e., 379 and 382 bugs respectively). Note also that, in each matrix, each cell holds the total number of bugs mapped to a certain attribute value, the values read in the columns represent the outcome of the verification procedure. In light blue, we highlight the true positives (the bugs in which there is agreement between the original classification and the respective verification task).

We also analyzed the results obtained by *Researcher2* with Cohen's Kappa, as it is able to measure the agreement between two raters (i.e., *Researcher1* and *Researcher2*) that classify items in mu-

Table 3
Accuracy and Cohen's kappa Agreement for researcher2 external verification.

ODC Attribute	Accuracy	*Kappa*	Agreement
Activity	0.71	0.64	*substantial*
Trigger	0.60	0.58	*moderate*
Impact	0.83	0.82	*almost perfect*
Target	0.98	0.98	*almost perfect*
Defect Type	0.94	0.93	*almost perfect*
Qualifier	0.93	0.89	*almost perfect*

tually exclusive categories (Cohen, 1960). The definition of k is:

$$\kappa = \frac{p_0 - p_c}{1 - p_c} \quad (1)$$

where p_0 is the relative observed agreement between raters (i.e., accuracy) and p_c is the hypothetical probability of chance agreement. If the raters fully agree, then $\kappa = 1$, if there is no agreement beyond what is expected by chance, then $\kappa = 0$, and, finally, a negative value reflects the cases where agreement is worse than random choice. Overall, the following terms apply for the value of κ: *poor* when less than 0, *slight* between 0.01 and 0.2, *fair* between 0.21 and 0.4, *moderate* between 0.41 and 0.6, *substantial* between 0.61 and 0.8, and finally *almost perfect* between 0.81 and 0.99 (Landis and Koch, 1977). The accuracy results of *Researcher2* are presented in Table 3, along with the Kappa value. Due to time restrictions, *Researcher3* classified only the defect type attribute, thus the corresponding results are only presented afterwards, by the end of the next paragraph.

As we can see in Table 3, we have obtained an *almost perfect* agreement for most of the ODC attributes. The exception is *Activity* with *substantial agreement* and *Trigger* with *moderate agreement*. Regarding *Activity*, the most notable case of disagreement between the researchers is the case where code inspection bugs are marked by *Researcher2* with one of the three possible *Testing* cases (i.e., unit testing, function testing, system testing). In fact, *Researcher2* classified about one third of the 229 bugs marked with code inspection by *Researcher1* as one of the *Testing* cases (15 bugs marked with unit testing, 30 with function testing, and 32 with system testing). This has impact in the *Trigger* classification, as a wrong activity will lead to a wrong *Trigger* (due to the Activity-Trigger mapping presented in Table 1, and with exception of bugs that pass from *design review* to *code inspection,* and vice-versa, which does not occur in our study). Therefore, it is expectable that the accuracy values for Trigger are lower (due to the accumulated error), which is actually the case. As mentioned, *Researcher3* classified only the defect type attribute, for which it reached an accuracy of 0.91 which corresponds to a Kappa value of 0.9, also achieving an *almost perfect* agreement. These results and analysis reflect the overall quality of the dataset, providing us with the quality assurances that should be present in this type of work. Detailed results are available at (Agnelo et al., 2019).

It is also worthwhile mentioning that, besides classification accuracy, a significant threat to the quality of the final classification is related with the size of the sample used during the process. In real scenarios, this is always a cost-benefit situation, where the size of the sample is very much related with the resources available to allocate to the classification task. So, after classifying all bugs we also tried to understand what would be the loss associated with using smaller sample sizes, in particular by understanding the degree of similarity between subsets of our sample and the actual sample of 4096 bugs. Small differences found in the prevalence of the different ODC attribute values indicate that smaller sets of defects may still be adequate for the analysis (depending on the available classification resources).

Table 4
Results of the One Sample Sign test.

Sample size	Reject H_0
16	46.15
32	30.77
64	28.85
128	19.23
256	15.38
512	9.62
1024	5.77
2048	3.85

We randomly generated subsets of the 4096 population of defects (from 16–2048, following the powers of 2) and repeated the process 30 times per each sample size. We then calculated the prevalence (i.e., the rate of occurrence) of each value of each ODC attribute for each of the resulting samples. We analyzed the data using the Kolmogorov-Sminorv, Shapiro-Wilk, and Levene tests (Öner and Kocakoç, 2017) to conclude that it did not follow a normal distribution and therefore we used the *one sample sign test* (Wackerly et al., 2008) with a 95% confidence level to verify the null hypothesis H_0 that the prevalence of the different attribute values, for each of the ODC attributes, is the same between a particular sample and the whole population. Table 4 shows the average results for 30 runs per each sample size.

In practice, Table 4 shows how increasing dataset sizes tend to resemble the population of 4096 bugs. Obviously, a larger dataset is beneficial as it better represents the whole population. This is helpful information for practitioners or researchers, which are able to understand the exact cost involved in analyzing smaller sets. For instance, selecting 2048 bugs shows that, in average, in less than 5% of the cases (3.85%) the null hypothesis is rejected, which is actually a small difference, when compared to the huge effort necessary to classify another set of 2048 bugs.

4. Results

In this section, we describe the results obtained from applying ODC to the 4096 defects collected from the issue-tracking platforms of the three previously mentioned NoSQL databases. We first analyze the distribution of values obtained for each individual ODC attribute, we then analyze the results for pairs of ODC attributes and conclude with a detailed view on the most affected components per database. All detailed results and also supporting code are available at (Agnelo et al., 2019).

4.1. Value distribution across ODC attributes

We begin by overviewing the results obtained for each individual ODC attribute (i.e., activity, trigger, impact, target, defect type, and qualifier). This allows us to understand trends regarding common types of defects or corrective measures, for instance, but is particularly important for dependability researchers to identify representative software defects in the domain of NoSQL databases (e.g., to use in fault injection campaigns). The values obtained for the *activity attribute* for each of the three databases, are shown in Fig. 3.

As we can see in Fig. 3, the values for the activity attribute are distributed in a very similar fashion among the three databases, with the most common value being "Code Inspection" appearing in around half of the cases. Notice that, in our case, with no special adaptation of ODC, "Code Inspection" refers to human manual inspection but also automatic inspection carried out by tools (e.g., static code analyzers). The fact that this attribute value is associated with around half of the defects essentially shows the importance of this activity in disclosing defects in these database sys-

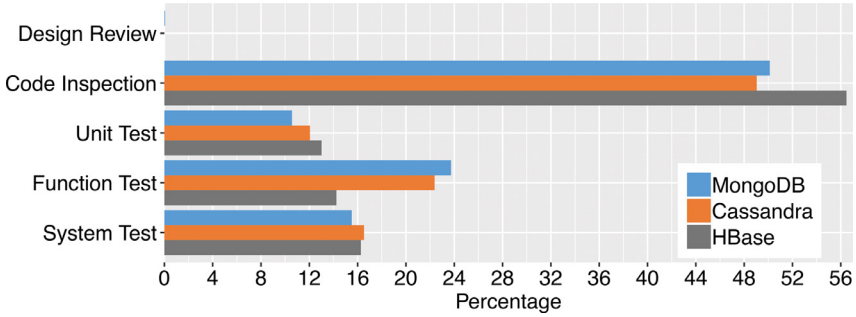

Fig. 3. Activity attribute distribution.

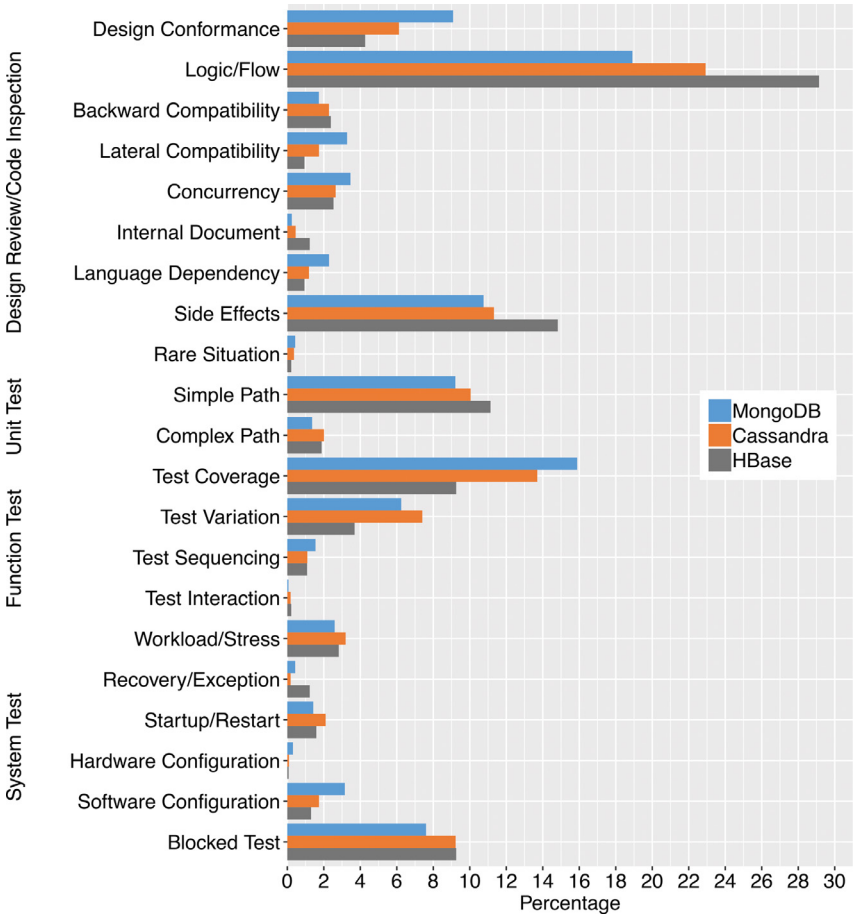

Fig. 4. Trigger attribute distribution.

tems. Testing related activities (i.e., "Unit Test", "Function Test" and "System Test") show somewhat similar values between each other. It was often difficult to classify defects in terms of this attribute, mostly due to the lack of information, in the defect description, regarding which activity was being carried out at the time the defect was found. So, whenever the information was not sufficiently clear, *Researcher1* classified the trigger first, which allowed to return and classify the activity, based on the activity-trigger mapping previously mentioned.

Fig. 4 shows the distribution of the values obtained for the *trigger attribute*. The different trigger values are grouped by the activity–trigger mapping shown previously. "Logic/Flow" is the most frequent trigger and has its higher count in HBase. "Side Effects" is also a frequent HBase issue (and also in the remaining databases, although with a lower expression), which may suggest a stronger coupling in the way the different parts of the system are written or built. This suggests that the design of the system and the process used for development may require a reflection and improvement. Finally, "Simple Path", "Test Coverage", and "Blocked Test" are expectable cases (in the sense that they represent basic testing cases), arising from unit, function, and system testing, respectively. Overall, this emphasizes the *importance of carrying out different testing activities for defect disclosure*. The remaining trigger types show lower occurrence rates, with most of them below 4%. Across databases, there is high variation in the frequency of triggers observed, with no clear pattern. The only visible pattern is the fact that Cassandra occupies the middle position in 16 of the 21 triggers. So, there is a strong variation in the triggers across databases and the different Activities they are associated with, which mostly reflects the way the different communities operate and the way the databases are being developed and verified.

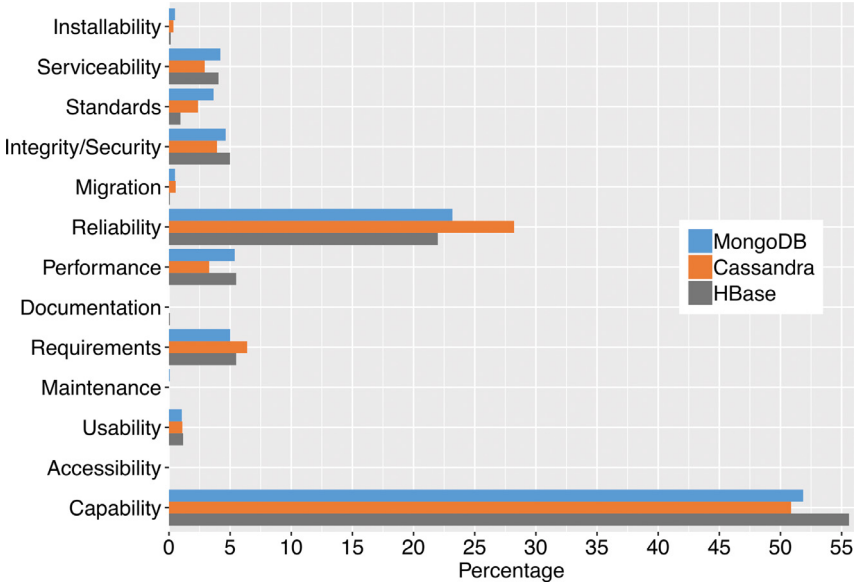

Fig. 5. Impact attribute distribution.

The *impact attribute* represents the hypothetical effect a defect would have had on a user, if it had been triggered on the field. Fig. 5 shows this attribute's value distribution across the three databases. As we can see in Fig. 5, "Capability" and "Reliability" jointly account for about three quarters of the types of impact in each of the three databases. "Capability" is, as defined in (IBM 2013), the ability of the system to perform its intended and required functions (the customer is not impacted in any of the remaining impact values). Capability is, by definition, a generic attribute value, and it is often used when none of the remaining impact types seem to fit the defect being analyzed. This may justify its higher values in all three databases, but, more importantly, it reflects the fact that a faultload holding representative bugs should impact, in half of the cases, the intended function of the system. In the same manner, in one fourth of the cases, bugs should impact "Reliability". In this context, reliability represents critical situations in which a system would halt or crash in an unplanned behavior (IBM 2013) and Cassandra is slightly more prone to such issues which may be important information for providers that want a long running, reliable, installation of this database.

The remaining one fourth of this distribution is covered by all the other impact values. Impacts such as "Installability", "Maintenance", "Migration", "Documentation" or even "Accessibility" were found to be the rarest in the distribution, with some cases showing no occurrences whatsoever. Apart from these, the other impact types appear with some variation, with values ranging between around 5% and 2%. Cassandra is slightly less prone to performance issues, which is important information when the main need is to support high performance business critical systems. Overall, the three databases see their defects roughly distributed in the same manner (although with some variations).

Table 5 presents the *target attribute*'s distribution (i.e., the entity that was corrected) regarding the three database systems. In what concerns the target attribute, most of the reported defects refer to source code problems (every nine out of ten defects are a code bug, in all three databases). Much less defects were found under "Build/Package" (build or packaging scripts), with the remaining types showing up either very infrequently or never.

Fig. 6 shows the value distribution for the *defect type attribute*. Overall, the three databases share very similar distributions, with the defect types repeatedly following the same order of prevalence across databases. The distribution of values is dominated by "Algorithm/Method", which is a relatively generic type of defect that fits many different cases (IBM 2013). As an example, a defect that consists of multiple "Assignment/Initialization" corrections, may correspond to an "Algorithm/Method" defect type as opposed to the otherwise assumed "Assignment/Initialization". Furthermore, cases which contained corrections of both "Assignment/Initialization" and "Checking" types were often classified as "Algorithm/Method", as there is no option to place multiple tags on a given defect.

Table 5
Target attribute distribution.

Target	MongoDB	Cassandra	HBase
Requirements	–	–	–
Design	–	–	–
Code	93.1%	94.1%	94.7%
Build/Package	6.8%	5.4%	4.4%
Information Development	0.1%	0.5%	0.9%
National Language Support	–	–	–

The second most frequent type of defect – "Function/Class/Object" refers to large changes in the design of a software system, i.e., a complete change in the way the system performs a certain function, or even the addition or removal of such functions. The need for redesign could either be a consequence of a poor initial design of these databases before the first implementation steps, or simply the need for redesigning large functions in order to bring these databases up-to-date with modern standards, or to fit new requirements, for example. MongoDB seems slightly more prone to interface issues than the remaining, which may be relevant information for developers, as interface issues tend to modify the developer's perception about a given system's reliability.

In addition to this analysis, and considering the importance of the defect type attribute in the ODC process, we also drilled down to analyze the distribution of defect per major version of each of the three databases. We did not observe any relevant differences across versions and therefore do not include results here. For further information on this, please refer to our detailed results at (Agnelo et al., 2019).

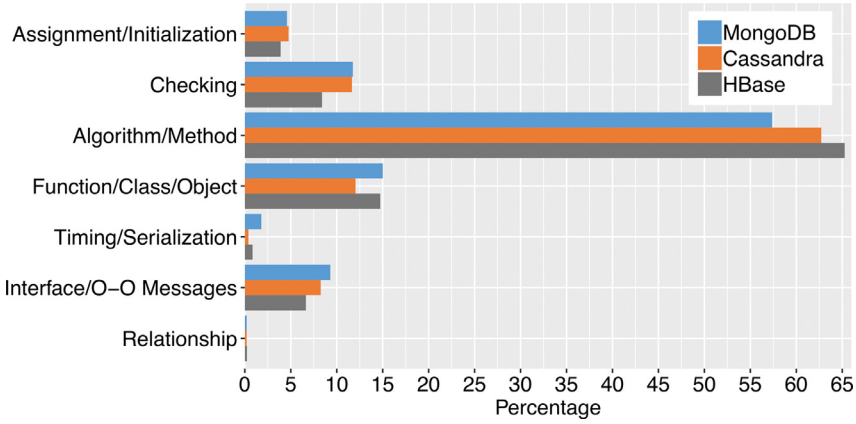

Fig. 6. Defect Type attribute distribution.

Table 6
Qualifier attribute distribution.

Qualifier	MongoDB	Cassandra	HBase
Missing	33.0%	29.9%	27.0%
Incorrect	63.1%	65.6%	68.9%
Extraneous	3.9%	4.5%	4.1%

Finally, Table 6 shows how the defects analyzed fit in the *qualifier attribute*. As we can see in Table 6, more than half of the defects were found to be of the type "Incorrect". This means that more than half of the defects were corrected by directly changing (i.e., re-implementing) the affected source code. The second most common qualifier is "Missing", which occurs when the correction of a defect is done by adding code that was otherwise absent. The least common type of qualifier is "Extraneous" (around 5% in the three databases), where the corrective measure consists of removing unnecessary code.

Overall, we observed great similarity in values (and thus, in defect trends) across the three databases, although there are a few exceptions. The exact root cause of all exceptional cases is difficult to determine (and is actually out of the scope of this paper), but, in the case of this single attribute view, the relevant part is the overall distribution found. Quite often we found cases where one single value dominated the distribution by occurring more than half of the remaining values together (e.g., in activity, defect type and qualifier). As we will see in Section V, the dominant value for defect type (i.e., Algorithm/Method) is different from what was observed in all (except one) other works analyzed in this paper, which emphasizes the specificity of the distribution of types of defects for NoSQL databases. Dependability researchers may be able to use this data to select one or a set of representative defects for NoSQL databases (e.g., to use in fault injection campaigns, or to generate code mutations, as carried out respectively in the work by Durães and Madeira 2006 and Lyu et al., 2003). The dataset is open (available at Agnelo et al., 2019), and selection of certain subsets of bugs is obviously also possible, if the goal is, for instance, to explore certain properties of the system (e.g., timing).

4.2. Value distribution across pairs of attributes

Analyzing the ODC results according to pairs of attributes is a common practice in this kind of study (Chillarege, 1996; Durães and Madeira, 2006; Zhi-bo et al., 2011). These two-way attribute relationships allow us to identify further defect trends. Of the six ODC attributes used in this work, we have excluded activity and target from this phase of the analysis, as activity can be extrapolated from the value of a trigger, and because target is dominated by a single value – "Code" (over 90%).

We begin by analyzing the *impact–trigger* pair. We have previously observed that "Capability" and "Reliability" together cover around three quarters of the impact attribute's distribution (see Fig. 5). Due to this, we limit the analysis to these two values against the different types of triggers, which we present in Fig. 7. Note that, to further improve the readability of this figure, we excluded triggers showing up with a frequency lower than 1% for at least one database in the trigger attribute's distribution. Fig. 7 shows that the distribution of the types of triggers tend to be different depending on the type of impact. The inspection-related triggers (i.e., from "Design Conformance" to "Side effects") show a much stronger link to capability defects (i.e., associated with about 2 thirds of the bugs), whereas in "Reliability" such triggers appear in about one fifth of the bugs. Such link is very visible, especially in namely "Design Conformance" and "Logic/Flow" and this is expectable as, by definition, these kinds of triggers can easily impact the ability of the system to perform its intended functions.

The testing-related triggers tend to occur more often together with the "Reliability" impact, representing around two thirds of the reliability issues. Indeed, a defect that impacts reliability is something that, in general tends to be detected at runtime, rather than by inspecting source code in which some kinds of issues are hard to catch (e.g., concurrency problems). A particular example can be found with "Blocked Test", which occurs when a certain system test cannot be concluded or cannot even run due to basic problems, we can see that these three databases struggled, at some point, to run against the developers' system tests. This does not mean that the system cannot be deployed and run in normal operating conditions, it just means that a certain system test fails to run (e.g., a stress test failing to run due to a misconfiguration of the server). On one hand this may indicate quality of the developers' system test suites, on the other hand, may also point out low quality development. In either case, what is relevant is to understand the distribution across impacts so that properly configured verification activities may take place (e.g., selecting the right verification activities to try to trigger certain types of problems).

Fig. 8 shows the results of the *impact–defect type* pair. As before, we limit impact to the most frequent types "Capability" and "Reliability" and we can see that there are no major differences between the relative distribution of types of defects across both impacts. The only minor differences are related with a relatively higher presence of "Checking" defects in the "Reliability" impact, which is understandable as these types of issues are, many times, associated with robustness problems (Laranjeiro et al., 2012). Also, "Capability" tends to gather more presence of "Function/Class/Object"

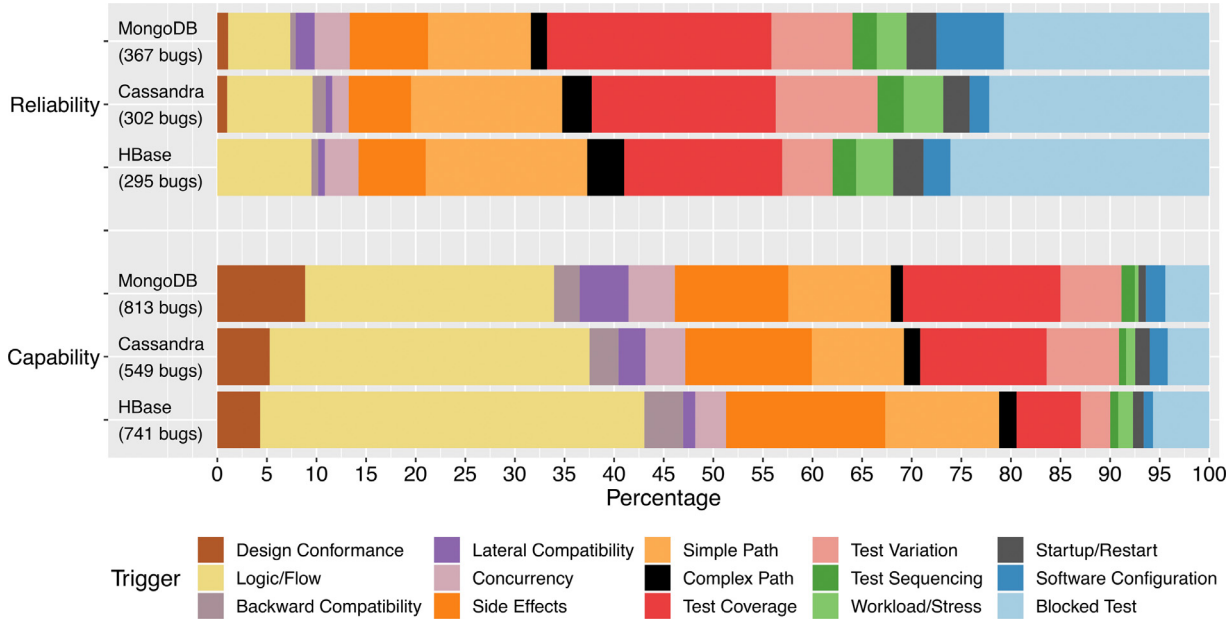

Fig. 7. Impact-Trigger Type pair distribution.

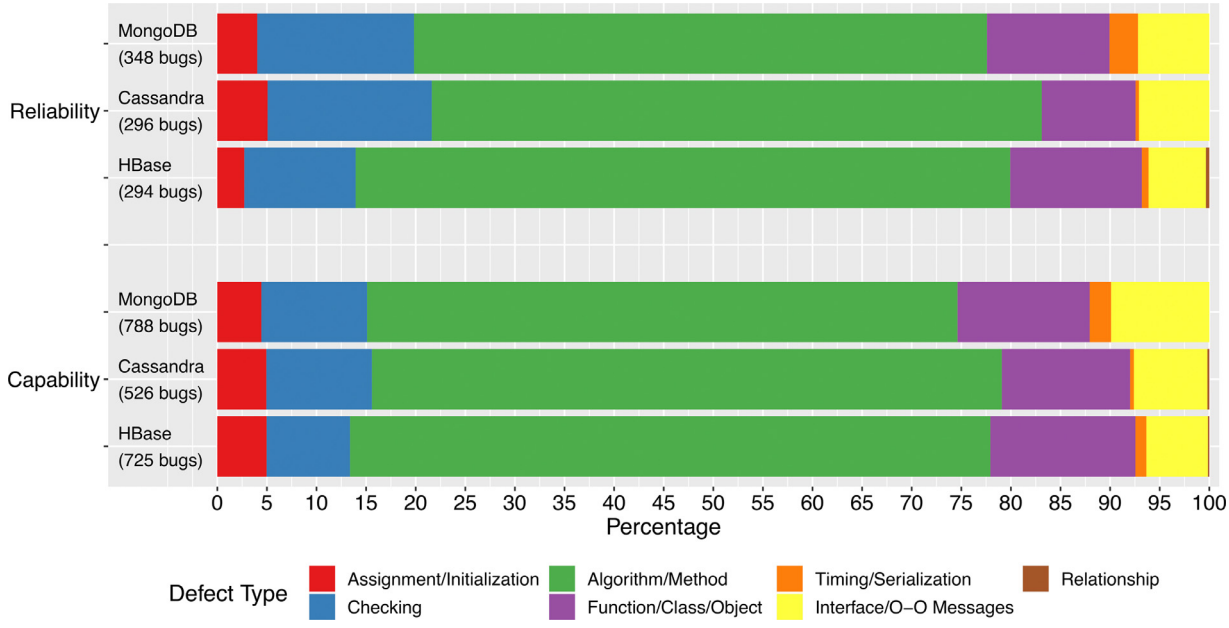

Fig. 8. Impact-Defect Type pair distribution.

defects, which, in agreement with ODC, is in fact a defect that significantly affects "Capability" and this highlights the importance of design in the overall system. So, if the goal is to prioritize capability versus reliability, for instance, tuning the design-oriented activities of the project, or the overall development process, could help diminishing the frequency of "Function/Class/Object" defects in the final product.

The last pair involving the impact attribute is the *impact-qualifier* pair. Fig. 9 displays the distribution of the "Capability" and "Reliability" impacts for each of the three existing types of qualifier.

Despite there are a few variations between the three databases, in general the distribution of values is quite similar, with "Incorrect", "Missing" and "Extraneous" being the most, second and least common qualifiers, respectively. Note that this order has also been previously observed in the qualifier's individual distribution. It is worth mentioning that this order of the qualifiers is also kept in the remaining types of impact (not included in Fig. 9).

Figs. 10 and 11 show the results involving the *trigger attribute* against *defect type* and *qualifier*, respectively. In both cases, we excluded all types of trigger values that, for all three databases, occurred less than ¼ of the value for the most frequent trigger type. For instance, given a top trigger value with 20% occurrence rate, the threshold for accepting other triggers would require any of their values to exceed one fourth of this (i.e., 5%). The goal was to focus the analysis on the more relevant pairs between the trigger attribute and other attributes, as the former contains a huge array of values, which would render the figures unreadable.

Fig. 10 presents the selected *trigger-defect type* pair and their respective value distribution. We can see that there is some varia-

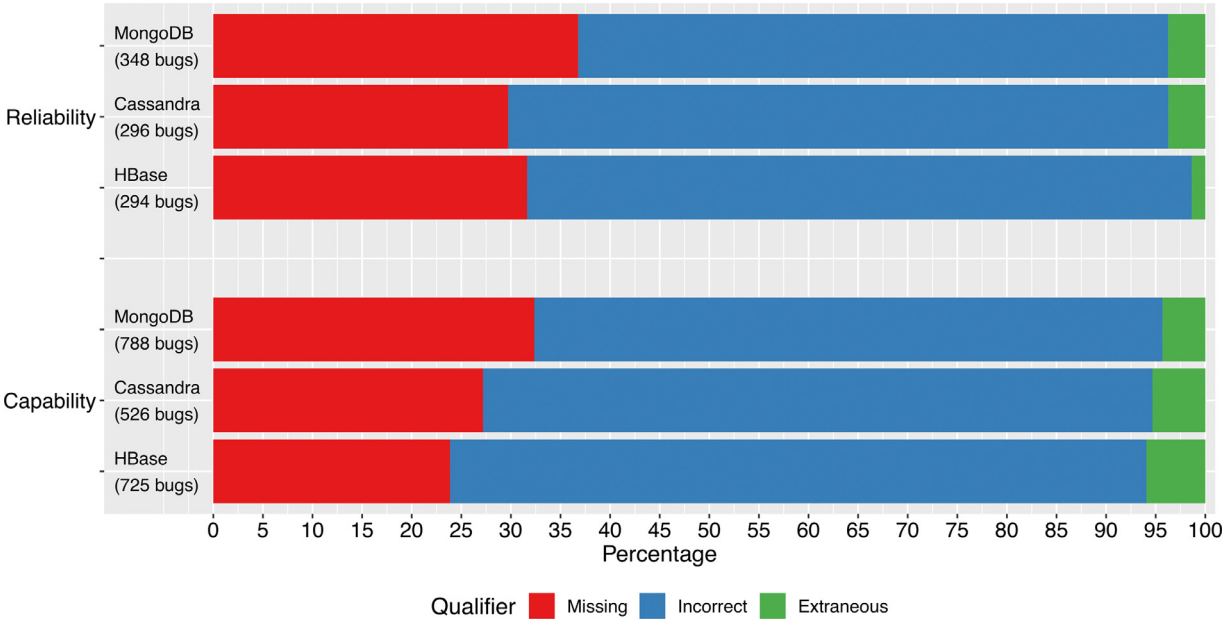

Fig. 9. Impact-Qualifier pair distribution.

tion of the relative popularity of each defect type, including some differences regarding the overall distribution of defect types. The most notable cases include HBase/Design Conformance (a bug detected when comparing a certain element against its specification), where the top defect is, by far, "Function/Class/Object" (a defect that should formally require a design change). This association suggests either a weaker design of the database or simply a need for frequent design changes. In either case, developers may be able to reduce the presence of this defect if more time is invested in the design of the system. Another visible case is the fact that "Checking" defects are the second most frequent defect types in the testing-related triggers – Simple Path and Test Coverage (in the overall distribution Function/Class/Object stands at second place). Indeed, this kind of issues is better captured with testing activities than with code inspection or design reviews and it would be abnormal to see defects like Function/Class/Object defects being captured by this kind of triggers more often than Checking defects. Actually, such case happens for HBase under the Test Coverage trigger, which suggests that the related testing activities (i.e., Function Test) may be in need for improvements. Fig. 11 shows the distribution of values for the *trigger-qualifier* pair.

In a similar way to the previous figure, here we also refer to the fact that these results generally match the ones presented by these attribute's individual distributions. In terms of the distribution of the qualifier values, and although there are some variations, we observe the same order of frequency for all trigger types, with "Incorrect" being the most frequent value. This essentially means that, in most cases, bugs are fixed by correcting code and not by adding (or removing, which is a very rare case) code.

Fig. 12 shows the value distribution of the *defect type–qualifier* pair, which is a very common view of this kind of data (Chillarege, 1996; Durães and Madeira, 2006; Zhi-bo et al., 2011). It essentially characterizes the correction that was applied – defect type – and the state in which the code was, prior to being corrected – qualifier.

We can see in Fig. 12 that "Checking" is the type of defects that has a higher relative probability of being "Missing" than "Incorrect" or "Extraneous" (e.g., a missing value verification in the code). This also occurs in "Timing/Serialization", but in this case the bug count is rather low. The case of "Missing" "Checking" defects becomes particularly problematic, in the sense that this type of defects is many times related to instructions that are able to control the flow of a program based on specific conditions – hence "Checking". Their absence can lead to many kinds of problems, typically robustness problems (if checking is missing at the boundaries of the system) but also security problems (e.g., missing validation for malicious user input) (Laranjeiro et al., 2012).

4.3. Internal view and time to fix

In this section, we provide an internal view of the types of bugs affecting the top 3 components that had the highest number of closed and resolved bugs (in each of the databases). We then close the section showing the average time to fix per type of defect and per database. Table 7 overviews each of the most affected components, the descriptions were adapted from each database's technical documentation.

Fig. 13 shows the distribution of defect types found in each of the top 3 most affected components per database. Notice that the overall sum adds up to 3846 (instead of 4096) as some defects are not code defects and, thus, do not qualify for being classified with a 'defect type'.

We computed the Relative Change (*RC*) (Using and Understanding Mathematics 2019) values for the different types of defect for each component in respect to the overall values, as follows:

$$RC\left(x, x_{reference}\right) \frac{x - x_{reference}}{x_{reference}} \quad (2)$$

where $x_{reference}$ represents the overall value for a certain type of defect in a certain database and x represents the value for that same type of defect type but observed in a specific database component. Table 8 holds the finals values, where the cases marked in blue represent *RC* values greater or equal to |0.5| and where $x_{reference}$ has a value of at least 2%.

In perspective with the overall values, there are a few tendencies in specific components that are worthwhile mentioning within each database. Regarding *MongoDB*, Timing/Serialization defects tend to have more weight in the *Replication* component, which emphasizes the component's sensitivity to timing issues and, at the same time, suggests that stronger verification activities, namely further extensive runtime testing, that particularly takes in

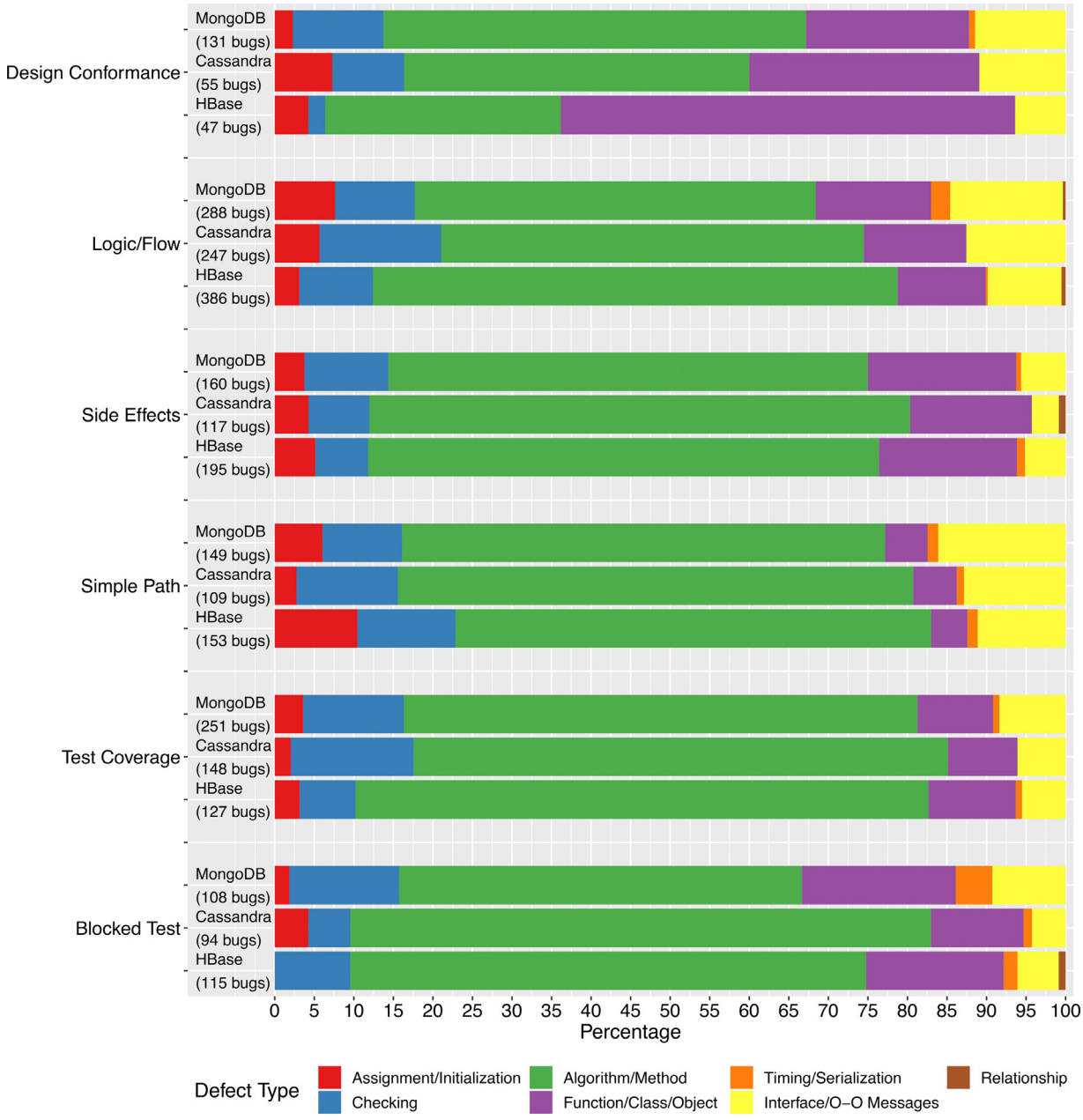

Fig. 10. Trigger-Defect Type pair distribution.

consideration timing aspects, are important and a relevant case to consider in the context of this component. Another visible aspect is related with the *Querying* component, where the Interface/O–O Messages defects are relatively lower. The component is very user-centric and possible problems should be captured and quickly resolved during informal testing activities. We have observed that bugs in this highly exposed component actually tend to be fixed very quickly. The average time to fix of Interface/O–O Messages defects in the Querying component of MongoDB is 1.36 days, while the overall time to fix of this kind of defects considering the whole system is 25.8 days. A more detailed analysis of the time to fix per defect type is presented later in this section (please refer to Table 9).

In the case of *Cassandra*, the *Tools* component presents relatively less Function/Class/Object defects and more Interface/O–O Messages defects. This component includes a number of highly diverse tools, so it is difficult to reason about possible causes, as there is no single nature for the whole component. Regardless of the reason, this data highlights weak spots that could benefit from different verification strategies. In the case of CQL, the component that supports Cassandra's query language, the highest variations considering the overall values come from Function/Class/Object (with nearly the double of defects of the overall) and also, at a smaller scale, from Assignment defects, which reduce to about half the overall value. The former case, according to ODC (IBM 2013), reflects the need for a formal design change and this higher count of Function/Class/Object defects just reflects the fact that the component is frequently experiencing design changes. Depending on the precise reasons behind the changes, a different development process may be in need. The whole overall design of the component may be in need for a larger major design change. The lower presence of assignment defects may the result of various factors, but it suggests that tests under the presence of limit conditions, which are known to effectively disclose such bugs, are being car-

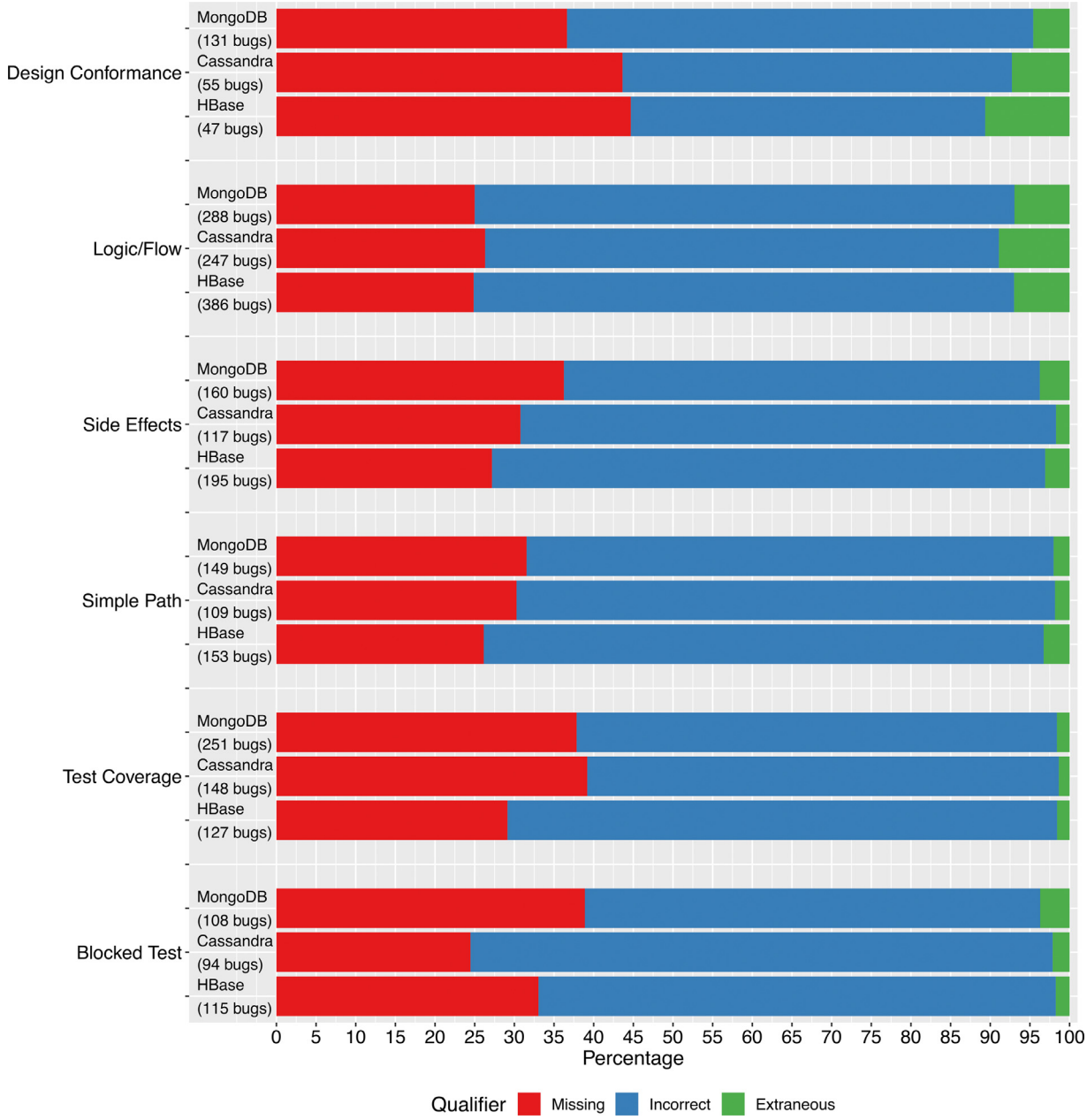

Fig. 11. Trigger-Qualifier pair distribution.

ried out by developers (which are, in fact, present at the respective testing code repository Apache Software Foundation 2019). In the Local Write-Read Paths component, Algorithm/Method defects have the highest relative weight across components, even considering components from other databases. Also, Function/Class/Object and Interface/O–O Messages show large decreases in this component, but, as the number of bugs is also relatively low, no further conclusions can be made.

Regarding *HBase*, regionServer has little variation in comparison with the overall results. The *Client* component has relatively less presence of Checking and Interface/O–O Messages defects. As this component has high exposure, we believe that existing bugs of this type (interface/message problems and checking issues, that are usually performed at the entry point of the API) should be disclosed easily. Finally, in the Master component, Checking bugs also decrease, while Function/Class/Object and Timing/Serialization increase. Again, this is a time sensitive component for which the current verification activities are apparently not sufficient. The higher presence of Function/Class/Object defects shows that this component frequently experiences design changes. Depending on the precise reasons for such changes, a higher investment in the design aspects may help reducing such cases.

Table 9 shows the average time the developers take to fix each type of defect for each of the databases, including a view of the times per different bug priority. The table also includes the standard deviation and respective count of bugs. It is worthwhile mentioning that previous work has been carried out in this area, e.g., on the analysis and definition of methods for predicting the time it will take to correct a certain type of bug (Weiss et al., 2007; Zhang et al., 2013; Giger et al., 2010). At this point of the paper we aim to understand any clear differences (in terms of time to fix) between the different types of bugs across databases.

One of the first visible aspects in Table 9 is that, most of the reported times are associated with quite high standard deviation val-

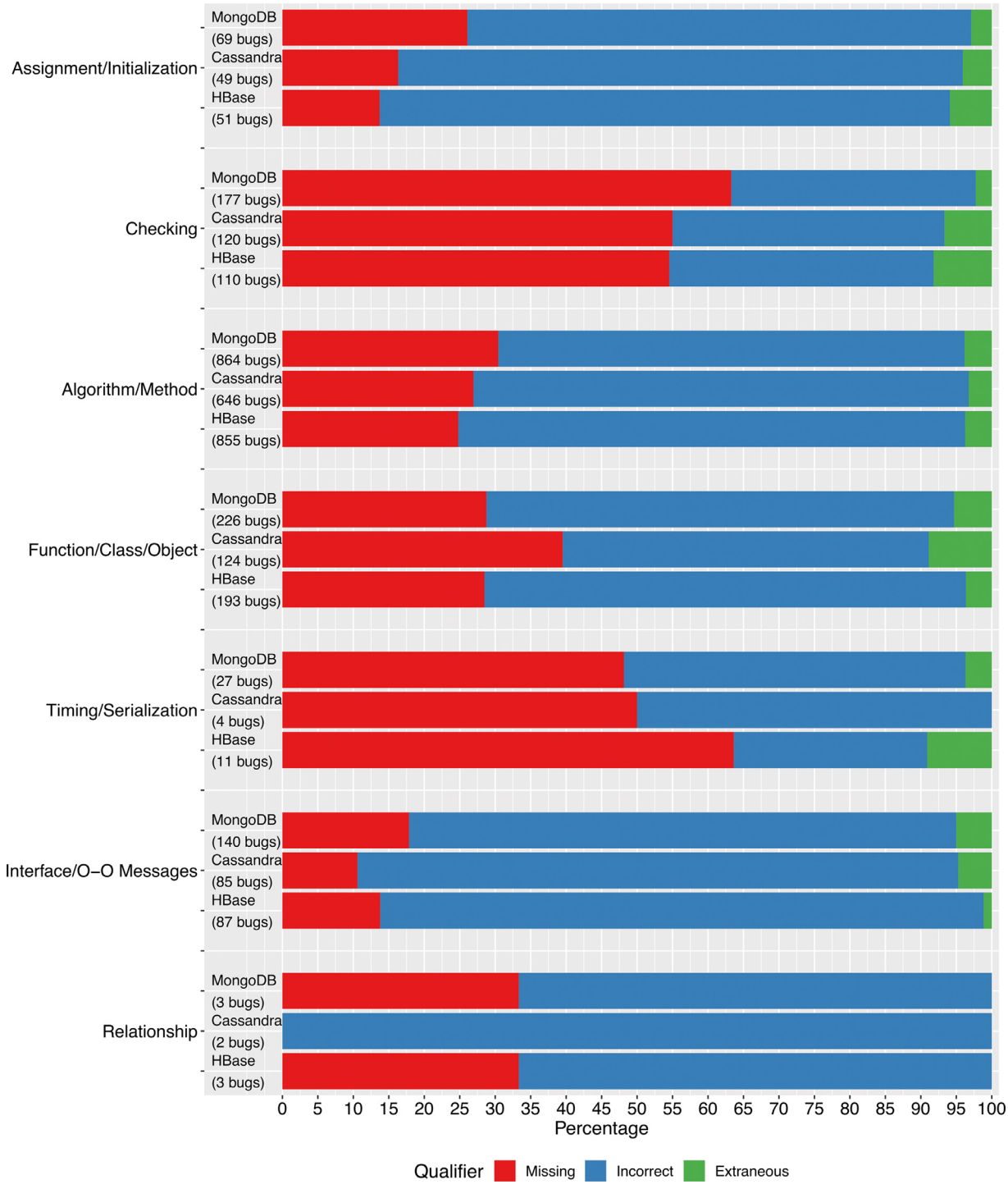

Fig. 12. Defect Type-Qualifier pair distribution.

ues, which means that it is hard to predict how much time it will take after a certain type of bug is reported and before a fix is produced. Considering the overall results, Function/Class/Object bugs take the longest to fix. This type of defect should require a formal design change and implies significant capability is affected, so it is interesting to observe that this kind of defects is the one that takes longer to fix, and this applies to all three databases. At the other extreme, we find the rare cases of Relationship bugs, where the few cases observed are corrected either in the same day, or in just two or three days.

Among the fast resolution bugs (and besides Relationship), we usually find Timing/Serialization defects. Their absolute number is higher for MongoDB, where 9 out of 10 of these bugs tend to be marked with the highest priority (in general, these should receive immediate attention from developers). The remaining databases had Timing/Serialization issues with a clearly better distribution across priorities, although the total number of bugs is relatively small. With exception of these two cases of long and short time to fix, the results across databases are quite heterogeneous (especially considering the different priorities involved), which is possibly the

Table 7
Description of the three database components with higher defect counts.

Database	Component	Description
MongoDB	Sharding	Sharding is the process of distributing data across multiple machines (shards) and is handled and balanced automatically by MongoDB, depending on the amount of data and cluster size. Sharding effectively enables easy horizontal scalability and ensures good performance in high throughput environments using commodity hardware.
	Replication	Replication consists in replicating the same data and storing it in multiple servers within a replica set (i.e., a group of processes solely dedicated to this task). It provides redundancy (it is a fault-tolerant mechanism) and increases availability. In some cases, this may even increase read performance, as read requests for the same data may be routed to any of the available machines in a given replica set.
	Querying	Refers to the MongoDB querying API. As opposed to classical SQL used in relational Databases, MongoDB's query model is function-based, as opposed to string-based.
Cassandra	Tools	This client-side component refers to tools that ship with Cassandra such as the Stress Tool used in stress tests, the Node Tool used for managing clusters, the CQL shell (CQLSH) and other tools to handle SSTables.
	CQL	Cassandra Query Language (CQL) is the query language to be used by clients and it highly resembles the well-known Structured Query Language (SQL) used for querying in relational databases.
	Local Write-Read Paths	Bugs labelled with this component affect one or more of the elements related to writing and reading to and from Cassandra. This may include Memtables (in-memory tables for hot writes and reads), the commit log (a log of commits ensuring redundancy and fault tolerance), SSTables (Sorted Strings Tables, persistent data tables stored in disk), Cache-related logic or even low-level disk I/O issues.
HBase	RegionServer	In HBase, tables are divided into regions, which are themselves managed by Region Servers. These act as nodes that make up the distributed database logic behind HBase (i.e., sharding).
	Client	Bugs marked with this component affect the HBase Java Client API. This is the interface through which one is able to perform all CRUD operations on HBase tables by using the Data Manipulation Language (DML).
	Master	This is known as the Master Server in HBase. It is responsible for managing all the subsystems such as assigning regions to each server and balancing the load between Region Servers. In short, it orchestrates an entire HBase cluster.

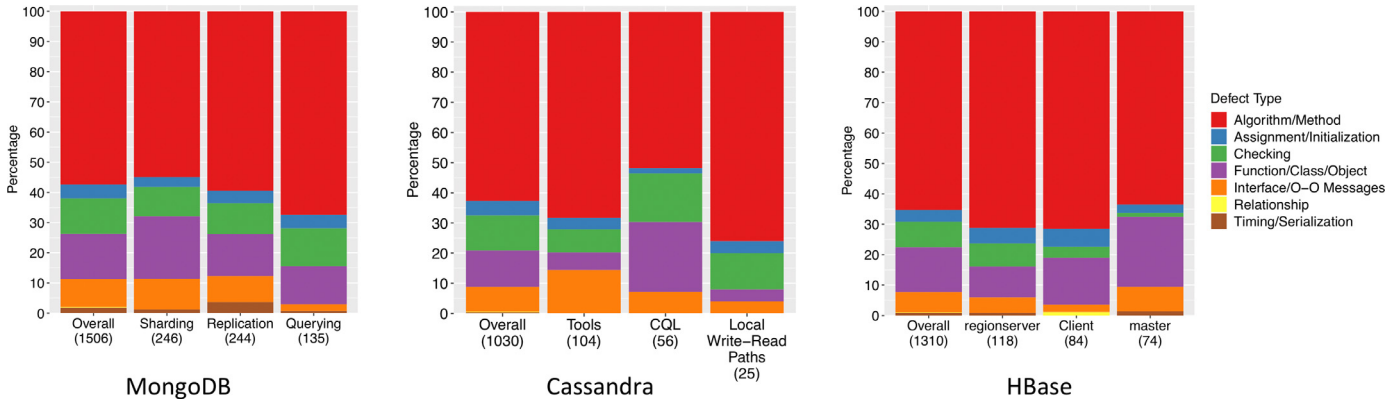

Fig. 13. Types of defects found in the three components with higher defect counts in each database.

result of the intervention of different communities, along with the specificities of each system involved.

4.4. Practical use case

In this section, we describe an experimental use case designed to illustrate how the analysis performed in this paper can benefit developers, by helping them tune and target the software development process, namely the software verification activities. Note that the experimental use case is of anecdotal nature, as its goal is simply to show that improvement is possible and made easier by the kind of analysis performed in this paper.

We selected a database, a defect type, and a database component particularly affected by that defect type to be at the center of our experiments. So, we selected *MongoDB* out of the three databases, as its bug reports tend to be much richer than those found in the other two databases. This is helpful as we are external to the MongoDB community and need to understand the causes and examples for reproducing issues. We then selected *checking* as the defect type to be the aim of our experiments, as these defect types tend to be easy to detect, their correction is relatively simple and are often related with inputs, which tends to make them easy to trigger. We finally selected the *querying* component as it

Table 8
Relative change values of the defect types for the most affected components in respect to the overall values.

	MongoDB			Cassandra			Hbase		
Defect type	***Sharding***	***Replication***	***Querying***	***Tools***	***CQL***	***Local W-R P***	***regionServer***	***Client***	***Master***
Assignment/Initialization	-0.3	-0.1	0.0	-0.2	-0.6	-0.2	0.3	0.5	-0.3
Checking	-0.2	-0.1	0.1	-0.3	0.4	0.0	-0.1	-0.6	-0.8
Algorithm/Method	0.0	0.0	0.2	0.1	-0.2	0.2	0.1	0.1	0.0
Function/Class/Object	0.4	-0.1	-0.2	-0.5	0.9	-0.7	-0.3	0.1	0.6
Timing/Serialization	-0.3	1.1	-0.6	-1.0	-1.0	-1.0	0.0	-1.0	0.6
Interface/O-O Messages	0.1	-0.1	-0.8	0.7	-0.1	-0.5	-0.2	-0.6	0.2
Relationship	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	4.2	-1.0

Table 9
Time to fix per type of defect.

		Overall time to fix			Priority													
					Blocker - P1			Critical - P2			Major - P3			Minor - P4			Trivial - P5		
	Defect Type	*avg*	*stdev*	*count*	*avg*	*stdev*	*count*	*avg*	*stdev*	*count*	*avg*	*Stdev*	*count*	*avg*	*stdev*	*count*	*avg*	*stdev*	*count*
MongoDB	Assignment/ Initialization	29.3	100.0	69	5.9	10.8	55	108.1	179.3	11	170.1	239.4	3	–	–	–	–	–	–
	Checking	48.2	156.2	177	33.3	30.5	2	47.5	159.0	17	43.5	158.9	130	77.5	153.7	24	34.7	56.6	4
	Algorithm/ Method	42.7	114.4	864	7.5	14.2	11	28.5	71.2	62	37.0	96.4	696	90.3	192.5	82	146.1	292.3	13
	Function/Class/ Object	110.9	277.4	226	9.0	1.8	2	179.1	400.7	23	71.5	156.5	186	544.9	658.5	14	2.7	0.0	1
	Timing/ Serialization	21.2	44.8	27	10.5	23.7	23	82.7	76.8	4	–	–	–	–	–	–	–	–	–
	Interface/O–O Messages	25.8	80.9	140	2.1	1.7	3	3.8	2.6	7	23.0	75.0	101	55.8	123.4	21	10.8	18.4	8
	Relationship	0.1	0.1	3	0.1	0.1	3	–	–	–	–	–	–	–	–	–	–	–	–
Cassandra	Assignment/ Initialization	28.3	82.7	49	0.9	0.0	1	19.6	49.0	24	21.5	56.8	16	71.6	160.9	8	–	–	–
	Checking	17.8	48.3	120	0.1	0.0	1	9.3	13.1	3	16.5	50.5	53	17.8	43.7	54	30.5	65.7	9
	Algorithm/ Method	24.0	69.3	646	11.8	16.1	8	17.4	55.6	23	25.1	79.0	349	25.8	60.1	226	10.4	26.1	40
	Function/Class/ Object	63.9	183.9	124	49.6	71.6	4	49.9	122.6	76	92.3	266.2	42	26.4	25.7	2	–	–	–
	Timing/ Serialization	1.7	1.4	4	0.1	0.0	1	1.6	0.0	1	1.0	0.0	1	3.9	0.0	1	–	–	–
	Interface/O–O Messages	12.7	31.0	85	0.2	0.0	1	39.7	45.8	4	16.7	38.2	39	7.3	16.9	28	5.1	4.7	13
	Relationship	3.2	1.9	2	5.1	0.0	1	1.3	0.0	1	–	–	–	–	–	–	–	–	–
Hbase	Assignment/ Initialization	22.4	67.9	51	13.1	11.9	2	51.4	97.5	7	24.5	77.9	25	11.2	22.5	13	0.2	0.2	4
	Checking	16.3	65.1	110	3.1	6.4	8	5.7	8.3	6	14.3	65.0	62	30.4	85.0	27	3.7	5.4	7
	Algorithm/ Method	22.0	89.2	855	16.1	51.9	68	36.9	99.4	92	21.9	98.8	501	19.1	69.6	158	9.0	24.4	36
	Function/Class/ Object	35.3	95.4	193	87.3	215.7	22	29.3	45.3	23	28.6	66.3	117	20.9	40.3	29	133.6	128.4	2
	Timing/ Serialization	3.9	7.7	11	1.0	0.0	1	0.2	0.0	1	6.4	9.8	6	1.0	0.9	2	0.8	0.0	1
	Interface/O–O Messages	9.8	24.5	87	0.5	0.2	2	12.8	28.7	7	8.9	23.2	44	12.6	28.6	26	5.4	10.0	8
	Relationship	0.4	0.2	3	0.5	0.0	1	0.0	0.0	1	0.6	0.0	1	–	–	–	–	–	–

Table 10
The 9 bugs Present in the Querying Component of MongoDB.

Bug ID	Version(s) Fix version(s)		Description
	Affected	Fixed	
SERVER-12,056	2.5.4	2.5.5	A request to use a custom index while scanning a database is ignored
SERVER-12,935	2.6.0-rc0	2.6.0-rc1	Sorting query results when they are already ordered generates an issue
SERVER-1296	1.5.3	1.5.5	A specific query triggers an assertion, even though it is syntactically valid
SERVER-14,625	2.6.3	2.6.4, 2.7.4	A specific query over an array field (i.e., column) of a database produces erroneous results
SERVER-15,688	2.6.0	3.3.3	An invalid combination of query operators does not return an error
SERVER-17,158	3.0.0-rc7	3.1.1	Different sorting requests are cached in memory as being part of the same request
SERVER-21,376	3.2.0-rc2	3.2.0-rc3	Bug is about two verbosity levels not being respected when obtaining metadata about a query.
SERVER-468	1.1.4, 1.2.0, 1.3.0	1.3.4	A request for metadata on the query returns the wrong information
SERVER-9035	2.2.3, 3.0.0-rc6	3.1.0	MongoDB fails to optimize a regular expression query (regex) under certain conditions

has a relatively high presence of *checking* defect types (please refer to Fig. 13). These have been reported in the field and had passed undetected through the original MongoDB battery of tests.

We carefully analyzed the 17 bugs marked with the *checking* defect type in the *querying* component of MongoDB and ended up by excluding 8 bugs from the experiments as their root causes and respective code fixes were actually exterior to this component. Table 10 summarizes the 9 remaining bugs targeted by these experiments, including their unique IDs, the affected MongoDB versions, and the versions where they were corrected.

We then carried out a set of experiments where the goal was to show that awareness of the high presence of *checking* defects in the *Querying* component (which is centered around validating and processing commands written according to the MongoDB query language, similar to SQL in relational databases) would be sufficient to generate basic functionality tests that could trigger, at least, part of the *checking* bugs present (or disclose new ones). Considering this goal, the type of defects to be caught, and the component under test, we automatically generated a test suite consisting of different queries to a database provided by MongoDB, named US Zip Codes database (João MongoDB 2019a). The set of tests was generated based on the combination of popular operators with the goal of potentially triggering the *checking* bugs known to be present in the target system selected. To do so, from all available operators (João MongoDB 2019b), we selected the following to be combined in different queries:

(i) All 8 *comparison query operators* (i.e., typical relational operators such as greater than, less than);
(ii) All 4 *logical operators* (i.e., and, or, nor, not);
(iii) Two *query modifier operators* (which are now functions in the latest version of MongoDB), known as *sort and hint*.

We did not select all query modifier operators as our main goal was not to detect unknown or very specific bugs (e.g. SERVER-9035), but simply showing that a relatively general test suite, designed to trigger checking bugs and aiming at the right component, could have been helpful in detecting, at least part of, the reported problems. Thus, for the third category of operators we selected a pair known to be involved in one of the reported bugs (also as a way of reducing the resulting high number of generated test cases). We then generated syntactically valid combinations of the abovementioned operators into queries, as described next.

(1) For each of the 8 comparison query operators we generate 100 boolean expressions. We randomly select a database column to place at the left-hand side of the expression and set a random value for the right-hand side. If the selected database column is a String we randomly select a String present in that column, else if it is numeric we generate a random value between the minimum and maximum numeric value present in that database column. This process results in 800 generated boolean expressions.
(2) For the binary logical operators (*or, and, nor*) we repeat 300 times (i.e., 100 times the number of binary operators):
  (i) We generate a new boolean expression composed of *N* (a random number between two to five) expressions randomly selected from those previously generated in *1)*.
  (ii) For each of the *N-1* available places for placing a binary logical operator we randomly select one of the three available operators (*or, and, nor*).
(3) For the unary logical operator (i.e., the *not* operator) we duplicate the expressions generated in *2)* and for each expression we randomly select a number of operands that will be negated.
(4) The resulting 1500 queries (that target at all columns) are run *without* and *with* up to two query modifier operands, as follows:
  (i) The queries are run without further query modifier operands;
  (ii) The queries are run with the *sort* query modifier (the sort order is randomly selected, such as the number of columns to be sorted);
  (iii) The queries are run with the *hint* query modifier, which uses either a randomly generated index (over the selected columns) or the default index.
  (iv) The queries are run with the *sort* and *hint* query modifiers.

This process generates a total of 5600 $((8 \times 100 + 3 \times 100 + 3 \times 100)^{*}4)$ test cases. All the test cases generated are syntactically valid, with the exception of a few test cases which would be signaled as erroneous by some earlier versions of MongoDB wherein a given operator still did not exist. Despite this, some may represent meaningless queries (e.g., a city with a population simultaneously lower than 1000 and greater than 1000) due to the random generation of the query conditions. In any case, it is the responsibility of the database to provide a robust answer. The test cases were then ran against the latest versions of MongoDB known to carry such bugs (i.e., 9 versions of MongoDB) and we added the latest stable version 4.0.10, which assumed the role of test oracle for validating test results. We executed the tests, analyzed the output and the query plan and identified differences. Any existing differences between the results obtained with the latest MongoDB and the remaining were manually analyzed.

This process lead us to identify the presence of 4 existing bugs (SERVER-12,056, SERVER-1296, SERVER-468 and SERVER-12,935) and showed evidence of one bug quite similar to SERVER-12,935, but on an earlier MongoDB version other than the one identified in that particular report, suggesting this bug actually survived multiple versions before being identified. We found evidences of *1 new bug not yet reported* (to the best of our knowledge), for which we filled in bug report SERVER-42,275 (it is to be confirmed by MongoDB developers, at the time of writing). This issue was detected

in version 2.5.4 using only a single test (i.e., test number 70 in our detailed results available at Agnelo et al., 2019), wherein the combination of the hint and sort query modifiers in one of the generated queries cause the MongoDB server (i.e., the *mongod* process) to unexpectedly crash, only to be recovered with a fresh process start. We also observed that this behavior is consistent across repetitions of that same test, thus the issue is easily reproducible.

Overall, were able detect nearly half of the 9 identified bugs plus evidences of an additional bug, using a relatively basic set of runtime tests. This, in practice, shows that the awareness about the high incidence of these types of bugs in a certain component is valuable information for developers towards the definition of more effective test suites. Obviously, this is merely an example and serves as anecdotal evidence that improvement is possible, and made easier, after obtaining the necessary data via the Orthogonal Defect Classification.

5. Discussion

In this section, we highlight and further discuss the main results presented in the previous section and put them in perspective with the results analyzed in related work.

The results obtained in this work, in particular for the *single attribute* analysis presented earlier, show that, regardless of the attribute being considered, *defects extracted from different systems tend to concentrate around just a few attribute values* (e.g., among the thirteen existing impact values, "Capability" and "Reliability" are by far the most common ones). In some cases (e.g., impact, defect type), the distribution is generally around attribute values that are, by definition, somewhat generic and fit a larger number of scenarios. Also, we observed cases where a single value dominated the distribution, by occurring more than half the times than the remaining attributes together. However, in certain attributes, such as the trigger attribute, the results tend to be more scattered, possibly due to the absence of generic trigger types. In certain attributes, we observed several cases of values presenting results ranging from non-existent, to a small fraction of the most common value in the distribution. Additionally, we have observed that in the vast majority of cases, the relative values remain quite similar across the three NoSQL databases.

The results of the analyzed *attribute pairs* in many cases show the same value order observed for the individual distributions. However, there are notable cases worth highlighting. *Testing activities are more than two times more frequent in reliability defects than in capability defects*, which signals the importance of these verification activities to disclose defects associated with this kind of impact. So, a system where reliability is a priority may benefit of a higher investment in testing activities. A similar case was observed with *checking defects, which are more frequent when impact is reliability than when it is capability* which again provides important information for the selection of verification activities.

Contrary to the remaining cases, *in HBase/Design Conformance, the top defect type is, by far, "Function/Class/Object"*, which is a type of defect that is associated with the need of a formal design change. We must also note that *Function/Class/Object defects appear more frequently than Checking defects under the Test Coverage trigger for the HBase database.* This may be due to many reasons, but essentially shows that such defects are not being caught through code inspection activities. suggests that *Function Testing strategies (*i.e., *the activity that is associated with the Test Coverage trigger) in HBase may be in need for improvement*. Finally, we observed no major differences when crossing qualifier with other ODC attributes. The only notable case is the fact that *Checking is a defect type that has a higher probability of being "Missing" than "Incorrect" or "Extraneous", as opposed to all other types of defects* (at least those with sufficiently high bug counts).

Despite the fact that these databases were built in different programming languages (i.e., MongoDB mostly in *C++* and Cassandra and HBase in Java), by different development teams, potentially following different methodologies, results show that the defect distribution, share many similarities. This suggests that *the nature of the systems may be linked to the types of defects* affecting such systems. In fact, when we drill down to the component level, and especially when we go through the most affected components per database, we see that *the nature of the different database components appears to be connected to the distribution of defects*, which is quite visible in certain components. As an example, the Replication component in MongoDB, a time-sensitive component, shows a stronger presence of timing/serialization defects. Obviously, there is no rule of thumb and these observations are mere observed facts, that may not apply to other NoSQL databases with similar characteristics.

We now analyze *our results in perspective with previous work* on defect classification using ODC. In this final analysis, we focus on the defect type attribute, as in the abovementioned previous works this is the only attribute for which we found comparable data (rarely, some works also use trigger and impact, but with just a small portion of the wide array of values we use in this paper). As a summary, Christmansson and Chillarege (1996) presented a set of errors which emulate software faults and Lyu et al. (2003) aimed at evaluating the effectiveness of software testing and software fault tolerance. Lutz and Morgan (Lutz et al., 2004) characterized bugs with to discover defect patterns and Durães and Madeira (2006) analyzed how software faults can be injected in a source code-independent manner. Børretzen and Dyre-Hansen (2007) characterized faults in industrial projects for process improvement. Fonseca and Vieira (2008) used ODC to characterize security patches of widely used web applications and Basso et al. (2009) characterize Java faults to understand their representativeness, including security vulnerabilities. Gupta et al. (2009) characterize defects in reusable software to understand the causes for lower defect densities in this kind of software. Thung et al. (2012) characterized bugs in machine learning tools and Xia et al. (2013) characterize bugs in software build systems. Silva and Vieira (2014) assessed ODC's adaptability to critical software environments and Xuan et al. (2015) study bugs in industrial financial systems. Silva and Vieira (2016) use ODC as basis to classify critical systems engineering issues and propose an adaptation of the taxonomy. Silva et al. (2017) performed a root cause analysis on defects from safety-critical software systems. Morrison et al. (2018) use ODC to characterize and understand the differences between the discovery and resolution of defects compared to vulnerabilities and Sotiropoulos et al. (2017) characterize bugs in robot middleware to understand which can be detected through simulation. Rahman et al. (2018) purely target defect classification, highlighting the fact that the nature of defects in the nowadays very popular Puppet scripts had not been yet categorized.

Table 11 summarizes the main results obtained in these previous works. The columns in the table (from the left to the right) identify the author and year of the work being analyzed, the reference for the work used in this paper, the names of the applications analyzed, the relative percentage found per each value of defect type, the total number of comparable defects (i.e., code or design defects classified with one of the 7 defect types in the ODC specification IBM, 2013) and their corresponding percentage regarding the total number of defects analyzed per application, and also the global number of defects per work. Additionally, we show the number of raters used (those marked with an asterisk correspond to information not present, or not clear, in the paper and that was provided by the paper authors), the programming language, and the type of application being analyzed. The following should be noted: (i) defect types not used in a specific work are represented with dashes (–); (ii) works which used custom ODC

Table 11
Orthogonal Defect Classification results found in related work.

Work	Software	Order	Assign / Init	Order	Check	Order	Alg / Met	Order	Func / Class /	Order	T/S	Order	Interf / OOM	Order	R	Comparable defects (C)	C / T	Total Defects (T)	Total defects per work	Raters per defect	Language	Domain
This work	MongoDB	5	4,3	3	10,9	1	53,4	2	14,0	6	1,7	4	8,7	7	0,2	1506	93%	1618	4096	1+1	C++	NoSQL database
	Cassandra	5	4,5	3	11,0	1	59,0	2	11,3	6	0,4	4	7,8	7	0,2	1030	94%	1095			Java	NoSQL database
	Hbase	5	3,7	3	8,0	1	61,8	2	14,0	6	0,8	4	6,3	7	0,2	1310	95%	1383			Java	NoSQL database
Christmansson and Chillarege, 1996	Tandem Guardian90 and IBM OS	2	19,1	3	15,2	1	37,8	5	7,6	4	13,2	6	7,1		–	408	100%	408	408	–	Operating System	
Lyu et al. 2003	RSDIMU	2	31,9	1	33,8	3	19	4	14,1		–	5	1,2		–	426	100%	426	426	–	C	Spacecraft navigation system
Lutz and Morgan, 2004	Deep space antena controller	3	2,7		–		–	1	78,2		–	2	4,8		–	126	75%	167	167	–	–	Space
Durães and Madeira 2004	Linux	3	22,6	2	25,8	1	33,3	4	12,9		–	5	5,4		–	93	100%	93	668	1*	C	Operating System
	Vim	2	21,3	3	18,2	4	9,1	5	0		–	1	54,5		–	249	100%	249			C	Text Editor
	FCIV	4	11,3	3	13,2	1	52,8	2	15,1		–	5	7,5		–	53	100%	53			C	Game
	Joe	2	25,6	1	44,9	3	15,4	5	0		–	4	14,1		–	78	100%	78			C	Text Editor
	MingW	3	10	2	38,3	1	46,6	5	0		–	4	5		–	60	100%	60			C	Programming toolset
	ScummVM	2	24,3	3	8,1	1	56,8	4	6,8		–	5	4,1		–	74	100%	74			C++	Game emulator
	CDEX	2	18,2	2	18,2	3	9,1		0		–	1	54,5		–	11	100%	11			C++	CD Audio Extractor
	Pdf2html	1	55	3	5	2	40		0		–		0		–	20	100%	20			C++	PDF converter
	Gaim	4	4,3	1	52,2	2	26,1	3	13		–	4	4,3		–	23	100%	23			C	Instant messaging client
	ZSNES	1	66,7		0	2	33,3		0		–		0		–	3	100%	3			Assembly, C, C++	Game emulator
	Bash	1	100		0		0		0		–		0		–	2	100%	2			C	Comand interpreter
	Firebird	1	50	1	50		0		0		–		0		–	2	100%	2			C++	Browser
Børretzen and Dyre-Hansen, 2007	Anonymized P1	2	9,5	3	6,3	7	1,1	1	25,3	4	1,4	5	0,3	5	0,3	217	44%	490	901	–	Java	Business critical - Data registration
	Anonymized P2	4	7,4	2	15,4	3	12	1	24	5	2,3	7	1,1	6	1,7	135	64%	212			Java	Business critical - Administration tool
	Anonymized P3	2	14,6	4	2,4	3	4,9	1	53,7	4	2,4		0		0	33	78%	42			Java	Business critical - Merging of applications
	Anonymized P4	2	26,7		0	4	6,7	1	36,7		0	3	10		3,3	28	83%	34			Java	Business critical - Transaction tool
	Anonymized P5	2	14	4	7,5	3	8,6	1	24,7	7	1,1	5	5,4	6	4,3	81	66%	123			Java	Business critical - Transaction tool
Fonseca and Vieira, 2008	PHP-Nuke, -Fusion, -myAdmin, -BB, Drupal, WordPress (v)	2	6	4	2,4	1	86		–		–	3	5,6		–	679	100%	679	679	1*	PHP	Web apps
Tânia et al., 2009	Vuze BitTorrent client	3	15,2	4	11,2	1	36,8	2	28,8		–	5	8,0		–	125	100%	125	574	3*	Java	P2P BitTorrent client
	Freemind	3	11,1	4	2,2	1	46,7	2	28,9		–	5	11,1		–	90	100%	90		3*	Java	Diagram application
	Jedit	3	14,1	5	11,3	1	36,6	2	25,4		–	4	12,7		–	71	100%	71		3*	Java	Text Editor
	Phex	2	90,0		5,0	1	60,0	3	10,0		–	3	10,0			20	100%	20		3*	Java	P2P Bnutella client
	Struts	3	18,2	4	10,0	1	48,5	5	4,0		–	2	20,2		–	99	100%	99		3*	Java	Library
	Tomcat	4	12,4	2	23,2	1	58,0	5	2,4		–	3	13,6		–	169	100%	169		3*	Java	Web Server
Gupta et al., 2009	Java Enterprise Framework	3	16,7	4	3,1	4	3,1	1	21,2		0,0	2	19,9			143	64%	223	1310	–	Java	Reusable Java Framework
	Digital Cargo Files	2	12,4	3	10,4	5	3,3	1	31,3	6	1,2	4	9,9			300	69%	438			Java	Document storage
	Shipment and Allocation	5	6,7	3	10,9	2	22,8	1	29,8	6	1,6	4	9,3			526	81%	649			Java	Business process management
Thung et al., 2012	Mahout, Lucene, OpenNLP	3	13	4	11,4	1	22,6		–		4,8	2	15,2		–	335	67%	500	500	–	Java	Machine learning
Xia et al., 2013	Ant, Maven, Cmake, Qmake	2	8,36	3	7,62	4	4,49		–	5	1,62	1	30,59		–	428	54%	800	800		Java, C++	Build Systems
Silva and Vieira, 2014	Satellite on-board SW start-up	6	0,9	2	6,5	3	2,8	1	14,0	4	0,9	4	0,9		–	28	26%	107	243	2*	C, Ada, Assembly	Satellite on-board software start-up
	Satellite on-board SW start-up		0,0	3	7,0	4	1,2	1	20,9	2	8,1		0,0		–	32	37%	86		2*		Satellite on-board software start-up
	Satellite software		0,0	2	15,0		0,0	1	55,0		0,0		0,0		–	14	70%	20		2*		Satellite software
	Airborne critical component		0,0		10,0	3	6,7	1	10,0	5	3,3	3	6,7		–	11	37%	30		2*		Airborne critical component
Xuan et al., 2015	PMS, β- Analyzer and OrderPro	3	12,33	4	8,33	2	17,67		–	5	2,33	1	27,33		–	169	56%	300	300	–	Java, Python, C	Industrial financial systems
Silva and Vieira, 2016	Satellite subsystems		–	3	6,6	2	11,4	1	21,3		–	3	6,6		–	606	82%	739	739	2*	C, Ada, Assembly	Space critical Embedded Systems
Silva et al., 2017	Satellite subsystems	5	4,3	3	6,4	2	9	1	19,1	6	3,1	4	5,2		–	503	47%	1070	1070	2*	C, Ada, Assembly	Satellite subsystems
Morrison et al., 2017	Chrome	3	13,7	4	10,9	2	29,1	1	36,0	6,0	2,3	7	1,7	5	6,3	175	100%	175	1166	2+1	C++	Browser
	Chrome (v)	3	12,1	1	39,3	2	30,6	4	8,7	7	1,2	6	2,3	5	5,8	173	100%	173			C++	Browser
	Firefox	3	25,2	2	25,6	1	35,3	4	11	7	0	5	2,2	6	0,6	317	100%	317			C++	Browser
	Firefox (v)	3	18,1	1	40,3	2	32,7	4	7	7	0	5	1,6	6	0,3	315	100%	315			C++	Browser
	PHPMyAdmin	2	33,3	1	38,7	3	19,4	4	5,4	5	1,1	5	1,1	5	1,1	93	100%	93			PHP	Web app
	PHPMyAdmin (v)	2	32,3	1	48,4	3	17,2	4	2,2	5	0	5	0	5	0	93	100%	93			PHP	Web app
Sotiropoulos et al., 2017	OpenRobots (pom, p3d, dtm)	2	21,1	4	6,1	1	21,2	5	3	5	3	3	15,2		–	23	70%	33	33	4*	C	Robot platform
Rahman et al., 2018	Mirantis	1	49,3	6	1,9	3	6,5	4	6,4	5	2,6	2	12,5		–	344	100%	344	3187	2	Puppet	Infrastructure as Code
	Mozilla	1	36,5	2	17	5	7,7	3	10	4	8,4	6	1,9		–	558	100%	558		2	Puppet	Infrastructure as Code
	OpenStack	1	57,5	2	6,7	4	5,9	6	2,4	3	6,5	5	2,9		–	1987	100%	1987		2	Puppet	Infrastructure as Code
	Wikimedia	1	62,7	5	12	2	3,3	3	4,3	4	5,1	6	2,6		–	298	100%	298		2	Puppet	Infrastructure as Code

values (e.g., Silva et al., 2017; Silva and Vieira, 2014) had any additional defect types excluded from the comparison; and (iii) as a visual aid, we present the number of order of each defect type for each work, from 1 (the most popular defect type found for the particular application being analyzed) to 7 (the least frequent defect type). These numbers also correspond, in order, to the colors red, orange, yellow, green, blue, light gray, dark gray.

As we can see in Table 11, the diversity of applications and contexts studied is quite large, including browsers, operating systems, text editors, satellite software, game emulators, machine learning tools, or build tools. In fact, ODC has been applied to a large number of domains, with the exclusion of storage intensive systems – which have now been handled in this work. The closest works (only from a domain perspective) may be the work by Børretzen and Dyre-Hansen (2007) and Gupta et al. (2009) which mention transaction tools and document storage, respectively, but do not analyze NoSQL databases, or even databases in general. Alongside with the highly diverse domains, we also observe applications built in different programming languages, with Java being often present.

The number of defects studied in related work is usually around a few hundred per work. This is related with the huge effort required to perform the ODC classification. Note that not all of the defects used in each work can be used for this analysis, as in certain cases the authors customized ODC to their specific context and in the remaining cases some defects were not code or design defects, thus, are not classifiable with a 'defect type'. We observed it is a common practice to use one or two raters (i.e., a human that takes care of all the classification process), however the vast majority of the works does not even specify how many raters were used. We were able to determine the number of human raters used for a few works, after contacting the authors. This count is marked with an asterisk under the column raters per defect. We also observed that when multiple raters are used, it is very rare that the authors discuss the inter-rater agreement.

Regarding the results obtained in previous works there are just a few observable trends, which is very likely the result of the high diversity of applications and contexts analyzed, aggravated by the general non-uniformity of each research work (e.g., dataset size, procedure applied, number of raters used). The results observed in related work strongly suggest that *there are no a priori expectable results for a certain system*, as the large number of variables involved seem to play an important role and, in some cases, are difficult to measure, namely in what concerns the whole development process used (unless, in such cases, the system is being developed in a very confined or controlled manner, such as in some mission-critical systems). This calls in for the need of having new studies that specifically target new systems, such as NoSQL databases, that are being built in relatively uncontrolled environments, as it is usual in open source projects.

Despite the huge variability observed, there is a clear separation between "Timing/Serialization" and "Relationship" defect types and the remaining, as these to occupy the bottom places. "Interface/O–O Messages" and "Checking" occupy middle positions more often. "Assignment" defects are found more often occupying the second position and "Algorithm/Method" seems to be most frequently at the first position, although "Function/Class/Object" is also a very frequent defect. Other than this, *the specificity of the systems and the way they are built seems to have an overwhelming weight in the overall distribution of defect data*. This means that, *for activities where representative software faults need to be selected, it is*

necessary to previously analyze defect data for the particular context, so that activities such as fault injection campaigns can be effectively carried out.

The data obtained here could be useful for software developers working in this kind of systems, allowing them to direct more the inspection and testing efforts on certain areas of the software systems being developed. This could not only improve development process and product quality for these databases, but also allow time to be saved in otherwise extensive defect finding/fixing activities.

6. Threats to validity

In this section, we present threats to the validity of this work and discuss mitigation strategies. We start by mentioning the fact that, in this work, we have analyzed 4096 bug reports, which is *a subset of all reported bugs* affecting the studied NoSQL databases (about one fourth of the total number of bug reports). Thus, our results may not be representative of the whole population of bugs affecting these systems. This option for a subset of bugs was due to the huge amount of human effort involved, but still we must mention that, to the best of our knowledge, it is the largest dataset found among related work.

Each *bug report was, at a first stage, classified manually* by a single person, which may raise questions regarding the reliability of the dataset. This may introduce some error in the results either due to the lengthy human intervention required, or due to some subjectivity present in the process which may be due to the intrinsic characteristics of the ODC method itself, to the lack/erroneous information present in the bug report, and finally due to the technical expertise of the human that is applying the ODC method. To mitigate such issues, we took the following measures: (i) the researcher involved in the classification process was trained using a set of 300 bugs that were discarded from the results; (ii) we added a verification procedure carried out by the same person (also to have some understanding about the number of errors potentially present and as a means to improve the researcher classification skills); (iii) we added an additional verification procedure carried out by two external researchers. Due to the huge amount of effort involved *both verification steps used a subset of the 4096 bugs* (a total of 40% of the bugs were reanalyzed), in which we found relatively low counts of errors and, in most of the cases, almost perfect inter-rater agreements.

In this work, we have *studied three NoSQL databases*, and thus the results, namely the link between the nature of NoSQL systems and the distribution of bugs, cannot be generalized to all NoSQL database systems. Still, we have selected among the most popular ones, so that our results are meaningful to a potentially larger community of practitioners and researchers. Also, we must mention that, since the related work found does not explore other types of database systems, our findings that relate to NoSQL systems may actually also fit in a higher class of systems (e.g., database, storage systems).

7. Conclusion

In this paper, we analyzed software defects reported for three of the most popular NoSQL databases using Orthogonal Defect Classification. The application of the ODC procedure was entirely manual, and we internally double checked the results and also asked two independent early stage researchers to perform an external verification. The outcome of both verification procedures indicate the overall good quality of the dataset, which we make publicly available at Agnelo et al. (2019).

We have discussed the *main findings* of this work, of which we highlight: (i) First of all, the large variation in the distribution of defect types found in related work (which analyzes different types of systems and domains), which in practice means that we cannot assume, *a priori,* the existence of a certain defect distribution, which implies that this kind of study is carried out whenever it is important to know which are representative defects; (ii) The results for NoSQL databases revealed, in terms of defect type prevalence, unique distributions for the types of defects (not visible in any of the related works analyzed); (iii) We observed similarity in the single-attribute view of all three databases, also with, sometimes, a single value dominating the distribution. This, in conjunction with the previous finding and with the fact that all three databases are being built and verified in a relatively uncontrolled open-source environment and by open and dynamic communities, suggests that the overall nature of the system appears to play an important role in the distribution of defects; (iv) Testing activities are more than two times more frequent in reliability defects than in capability defects; checking defects are more frequent when the impact is reliability and are more likely to be associated with the "missing" qualifier; (v) There are clear disparities in the distribution of defect types in different system components, which, in several cases, relate to the nature of the component (e.g., a replication component holding Timing/Serialization defects); (vi) Certain types of defects are consistently associated with longer times to fix across all three databases (e.g., Function/Class/Object). We have also set up a practical use case to show how this kind of analysis may be beneficial for developers by generating tests to target a component showing high prevalence of *checking* defects.

There were also a few *lessons learned* throughout the classification process and analysis of the results. It became obvious that effective training of the researcher that is responsible for performing the classification is crucial to have a good quality outcome in this kind of process. The contribution of additional raters is crucial, as it allow us to, on one hand, improve the final quality of the dataset, and, on the other hand, provides us with information regarding the possible presence of errors in the dataset. Also important to mention is that, there is a connection between two ODC attributes (Activity and Trigger) where certain triggers map to certain activities. This means that an incorrect activity may lead to an incorrectly classified trigger. The user applying ODC must take special attention when marking the values for these attributes, which we found to be a source of error. A difficult aspect is certainly related with understanding the exact causes of the results. Although we point out some possible reasons for the main issues raised by the results, following up with structured techniques like root cause analysis is out of the scope of this particular work, and is left for future work.

Overall, the resulting data represents vital information for NoSQL database developers, who could mostly benefit from it in defect prevention and removal efforts, thus contributing for more reliable systems. Additionally,researchers can use this data as a starting point in other works (e.g., fault injection experiments on systems of similar nature). As future work, we intend to explore the possibility of automating the classification procedure using machine learning algorithms.

Acknowledgments

This work has been partially supported by the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 823788 (project ADVANCE); by the project METRICS, funded by the Portuguese Foundation for Science and Technology (FCT)–agreement no POCI-01-0145-FEDER-032504; and by the project MobiWise: From mobile sensing to mobility advising (P2020 SAICTPAC/0011/2015), co-financed by COMPETE 2020, Portugal 2020 - Operational Program for Competitiveness and Internationalization (POCI), European Union's ERDF

(European Regional Development Fund), and the Portuguese Foundation for Science and Technology (FCT). We also would like to thank Henrique Marques and Gonçalo Carvalho, for their voluntary and fundamental help in using ODC to classify 820 software defects. It would not have been possible to conclude this work without their valuable collaboration.

References

Agnelo, J., Laranjeiro, N., Bernardino, J. "NoSQL odc dataset, results, and support code," Mar- 2019. [Online]. Available: https://eden.dei.uc.pt/~cnl/papers/2019-jss.zip.

Apache Software Foundation, "Apache Cassandra," Apache Cassandra. [Online]. Available: http://cassandra.apache.org/, 2019

Apache Software Foundation, "Apache HBase," *Apache HBase*. [Online]. Available: https://hbase.apache.org/, 2019

Apache Software Foundation, 2019. Apache Cassandra Unit Testing Code. The Apache Software Foundation.

Avizienis, A., Laprie, J.-.C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1 (1), 11–33.

Basso, T., Moraes, R., Sanches, B.P., Jino, M., 2009. An investigation of java faults operators derived from a field data study on java software faults. In: Proceedings of the Workshop de Testes e Tolerância a Falhas, pp. 1–13.

Børretzen, J.A., Dyre-Hansen, J., 2007. Investigating the software fault profile of industrial projects to determine process improvement areas: an empirical study. In: Proceedings of the Software Process Improvement, pp. 212–223.

Cattell, R., 2011. Scalable sql and nosql data stores. SIGMOD Rec 39 (4), 12–27 May.

Chillarege, R., 1996. Orthogonal Defect Classification. Handbook of Software Reliability Engineering, pp. 359–400 Chapter 9.

Christmansson, J., Chillarege, R., 1996. Generation of an error set that emulates software faults based on field data. In: Proceedings of the Annual Symposium on Fault Tolerant Computing, pp. 304–313.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 20 (1), 37–46 Apr..

Davoudian, A., Chen, L., Liu, M., 2018. A survey on nosql stores. ACM Comput. Surv 51 (2) pp. 40:1–40:43, Apr..

"DB-Engines ranking - popularity ranking of database management systems." [Online]. Available: https://db-engines.com/en/ranking. Accessed: 01-May, 2017.

Durães, J., Madeira, H., 2006. Emulation of software faults: a field data study and a practical approach. IEEE Trans. Softw. Eng. 32 (11), 849–867 Nov..

El Emam, K., Wieczorek, I., 1998. The repeatability of code defect classifications. In: Proceedings of the Ninth International Symposium on Software Reliability Engineering., pp. 322–333.

Fonseca, J., Vieira, M., 2008. Mapping software faults with web security vulnerabilities. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), pp. 257–266.

Giger, E., Pinzger, M., Gall, H.C., 2010. Predicting the fix time of bugs. In: Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE), pp. 52–56.

Grady, R.B., 1992. Practical Software Metrics For Project Management and Process Improvement. Prentice-Hall.

Gupta, A., Li, J., Conradi, R., Rønneberg, H., Landre, E., 2009. A case study comparing defect profiles of a reused framework and of applications reusing it. Empir. Softw Engg 14 (2), 227–255 Apr..

Henningsson, K., Wohlin, C., 2004. Assuring fault classification agreement-an empirical evaluation. In: Proceedings of the International Symposium on Empirical Software Engineering, ISESE'04, pp. 95–104.

Hulme, G.V., 2017. Amazon web services ddos attack and the cloud. Dark Read.. [Online]. Available: https://www.informationweek.com/news/229204417. Accessed: 08-Nov.

IBM, "Orthogonal defect classification v 5.2 for software design and code." Technical Report. 12-Sep-2013. Available: https://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf

"IEEE Standard Classification for Software Anomalies," *IEEE Std 1044-2009 Revis. IEEE Std 1044-1993*, 2010.

Kalinowski, M., Card, D.N., Travassos, G.H., 2012. Evidence-Based guidelines to defect causal analysis. IEEE Softw. 29 (4), 16–18 Jul..

Landis, J.R., Koch, G.G., 1977. The measurement of observer agreement for categorical data. Biometrics 33 (1), 159–174.

Laranjeiro, N., Vieira, M., Madeira, H., 2012. A robustness testing approach for soap web services. J. Internet Serv. Appl. JISA 3 (2), 215–232 Sep..

Leavitt, N., 2010. Will nosql databases live up to their promise? Comput. (Long Beach Calif) 43 (2).

Lutz, R. et al., "Research infusion collaboration : finding defect patterns in reused code," Technical Report, Dec. 2004.

Lyu, M.R., Huang, Z., Sze, S.K.S., Cai, X., 2003. An empirical study on testing and fault tolerance for software reliability engineering. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, ISSRE 2003., pp. 119–130.

Marks, H., 2014. Code spaces: a lesson in cloud backup. Netw. Comput.. 03-Jul- [Online]. Available: https://www.networkcomputing.com/cloud-infrastructure/code-spaces-lesson-cloud-backup/314805651. [Accessed: 08-Nov-2017].

Mellor, P., 1994. CAD: Computer Aided Disasters 1 (2), 101–156.

MongoDB, "US zip codes," 2019. [Online]. Available: http://media.mongodb.org/zips.json.

MongoDB, "Operators — mongodb manual," 2019. [Online]. Available: https://docs.mongodb.com/manual/reference/operator. Accessed: 19-Jul.

MongoDB Inc., "MongoDB," *MongoDB*. [Online]. Available: https://www.mongodb.com/index, 2019

Morrison, P.J., Pandita, R., Xiao, X., Chillarege, R., Williams, L., 2018. Are vulnerabilities discovered and resolved like other defects? Empir. Softw. Eng. 23 (3), 1383–1421 Jun..

Öner, M., Kocakoç, İ.D., 2017. JMASM 49: a compilation of some popular goodness of fit tests for normal distribution: their algorithms and matlab codes (MATLAB). J. Mod. Appl. Stat. Methods 16 (2) Dec..

Rahman, A., Elder, S., Shezan, F.H., Frost, V., Stallings, J., Williams, L. "Categorizing defects in infrastructure as code," *ArXiv180907937 Cs*, Sep. 2018.

Silva, N., Cunha, J.C., Vieira, M., 2017. A field study on root cause analysis of defects in space software. Reliab. Eng. Syst. Saf. 158, 213–229 Feb..

Silva, N., Vieira, M., 2014. Experience report: orthogonal classification of safety critical issues. In: Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering, pp. 156–166.

Silva, N., Vieira, M., 2016. Software for embedded systems: a quality assessment based on improved odc taxonomy. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, New York, NY, USA, pp. 1780–1783.

Sotiropoulos, T., Waeselynck, H., Guiochet, J., Ingrand, F., 2017. Can robot navigation bugs be found in simulation? an exploratory study. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 150–159.

"SQL & Nosql ranking - StackOverkill." [Online]. Available: http://www.stackoverkill.com/ranking/sql-nosql. [Accessed: 01-May, 2017.

"Top Nosql database engines." [Online]. Available: http://www.kdnuggets.com/2016/06/top-nosql-database-engines.html. Accessed: 01-May, 2017.

Thung, F., Wang, S., Lo, D., Jiang, L., 2012. An empirical study of bugs in machine learning systems. In: Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering, pp. 271–280.

"Using and understanding mathematics: a quantitative reasoning approach." [Online]. Available: https://www.pearson.com/us/higher-education/product/Bennett-Using-and-Understanding-Mathematics-A-Quantitative-Reasoning-Approach-5th-Edition/9780321652799.html. Accessed: 14-Feb, 2019.

Wackerly, D., Mendenhall, W., Scheaffer, R.L., 2008. Mathematical Statistics with Applications, 7 ed. Thomson Brooks/Cole, Belmont, CA.

Weiss, C., Premraj, R., Zimmermann, T., Zeller, A., 2007. How long will it take to fix this bug? In: Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops), p. 1 –1.

Wilson, P., Dell, L., Anderson, G., 1993. Root Cause Analysis: A Tool For Total Quality Management. ASQ Quality Press, Milwaukee, Wisconsin.

Wood, A., 1996. Software reliability growth models. Tandem Tech. Rep. 96 (130056).

Xia, X., Zhou, X., Lo, D., Zhao, X., 2013. An empirical study of bugs in software build systems. In: Proceedings of the 13th International Conference on Quality Software, pp. 200–203.

Xuan, X., Zhao, X., Wang, Y., Li, S., 2015. An empirical study of bugs in industrial financial systems. IEICE Trans. Inf. Syst. E98.D, 2322–2327 Dec..

Zhang, H., Gong, L., Versteeg, S., 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In: Proceedings of the 35th International Conference on Software Engineering (ICSE), pp. 1042–1051.

Zhi-bo, L., Xue-mei, H., Lei, Y., Zhu-ping, D., Bing, X., 2011. Analysis of software process effectiveness based on orthogonal defect classification. Proced. Environ. Sci. 10, 765–770.

João Agnelo is an MSc student in Informatic Engineering at the Faculty of Sciences and Technology of the University of Coimbra. He concluded his BSc in Informatics Engineering at ISEC - Polytechnic Institute of Coimbra, in 2018. Alongside his studies, he is also currently working in a few research projects at CISUC - Centre for Informatics and Systems of the University of Coimbra. His-interests include dependability, artificial intelligence and computer graphics.

Nuno Laranjeiro received the Ph.D degree in 2012 from the University of Coimbra, Portugal, where he currently is an assistant professor. His research focuses on robust software services as well as experimental dependability evaluation, web services interoperability, services security, and enterprise application integration. He has authored more than 50 papers in refereed conferences and journals in the dependability and services computing areas.

Jorge Bernardino is a Coordinator Professor at the Polytechnic of Coimbra - ISEC, Portugal. He received the PhD degree from the University of Coimbra in 2002. His main research interests are Big Data, NoSQL, Data Warehousing, Software Engineering, and experimental databases evaluation. He has authored over 200 publications in refereed journals, book chapters, and conferences. He was President of ISEC from 2005 to 2010 and President of ISEC Scientific Council from 2017 to 2019. During 2014 he was Visiting Professor at CMU - Carnegie Mellon University, USA. He participated in several national and international projects and is an ACM and IEEE member. Currently, he is Director of Applied Research Institute of the Polytechnic of Coimbra, Portugal.