Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Serializability Checking for MongoDB Clients

Master Thesis

Johannes Baum

May 04, 2017

Advisors: Lucas Brutschy, Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

Replicated data stores have become an important mechanism to achieve scalability and availability of modern distributed systems. While strong consistency is desired, replicated data stores often only provide weak consistency, which can lead to non-serializable behaviors. We formalize the MongoDB consistency model and present a dynamic analysis for MongoDB clients that detects serializability violations applying an existing serializability criterion that only demands eventual consistency. We evaluate the method by analyzing a set of open source applications and detecting previously unknown errors.

Contents

Contents **iii**

1 Introduction **1**
- 1.1 Contributions . . . 2
- 1.2 Overview . . . 2

2 Preliminaries **3**
- 2.1 Serializability Criterion . . . 3
  - 2.1.1 Basic Definitions . . . 3
  - 2.1.2 Relations . . . 4
- 2.2 Related Work . . . 6

3 MongoDB **7**
- 3.1 Document Structure . . . 7
- 3.2 Operations . . . 8
  - 3.2.1 Inserts . . . 9
  - 3.2.2 Queries . . . 9
  - 3.2.3 Updates . . . 9
  - 3.2.4 Deletes . . . 10
- 3.3 Replication . . . 10
  - 3.3.1 Automatic Failover . . . 10
  - 3.3.2 Oplog . . . 11
  - 3.3.3 Read Preference . . . 11
  - 3.3.4 Read Concern . . . 11
  - 3.3.5 Write Concern . . . 12
  - 3.3.6 Consistency Model . . . 12
  - 3.3.7 Consistent Prefix . . . 15
  - 3.3.8 Possible Sequential Consistency Violations . . . 15
  - 3.3.9 Atomicity Violations . . . 18

4 Dynamic Analysis for MongoDB **19**
4.1 Instrumentation . . . 19
4.1.1 Program Order . . . 19
4.1.2 Arbitration Order . . . 20
4.1.3 Visibility . . . 20
4.1.4 Transactions . . . 22
4.1.5 Special Cases . . . 22
4.2 Transformation of Driver Operations . . . 23
4.2.1 updateMany . . . 23
4.2.2 updateOne . . . 25
4.2.3 update . . . 25
4.2.4 insertMany . . . 26
4.2.5 insertOne . . . 27
4.2.6 deleteOne . . . 27
4.2.7 deleteMany . . . 28
4.2.8 findOneAndUpdate . . . 29
4.2.9 findOneAndDelete . . . 29
4.2.10 find . . . 29
4.2.11 findOne . . . 31
4.2.12 count . . . 31
4.2.13 Index operations . . . 31
4.3 Performance . . . 33
4.4 Limitations . . . 34

5 Analysis **37**
5.1 Commutativity Specifications . . . 37
5.1.1 Basic definitions . . . 38
5.1.2 Insert and Query . . . 40
5.1.3 Insert and Insert . . . 40
5.1.4 Query and Query . . . 40
5.1.5 Insert and Update . . . 41
5.1.6 Query and Update . . . 41
5.1.7 Update and Update . . . 43
5.2 Absorption Specifications . . . 45
5.2.1 Basic definitions . . . 45
5.2.2 Insert and Delete . . . 45
5.2.3 Update/Delete and Delete . . . 45
5.2.4 Update and Update . . . 46
5.3 Extension of ECRacer . . . 48
5.3.1 Relation Construction . . . 48
5.4 Runtime Analysis . . . 49
5.5 Pattern-Matching Extension . . . 50
5.5.1 Patterns . . . 51
5.5.2 Pattern Filtering . . . 52

5.5.3 Algorithms . . . 53
5.5.4 Monotonic-Reads . . . 54

6 Stress Testing **57**
6.1 Strategy . . . 57
6.2 Discussion . . . 58

7 Implementation **61**
7.1 MongoRacer . . . 61
7.1.1 Architecture . . . 61
7.1.2 Composer . . . 62
7.1.3 Instrumentation . . . 63
7.1.4 Network Partitions . . . 66
7.2 Extension of ECRacer . . . 66
7.2.1 Commutativity and Absorption . . . 66
7.2.2 Document Matching . . . 66

8 Evaluation **69**
8.1 Experimental Set-Up . . . 69
8.2 Results . . . 71
8.2.1 Session Violations . . . 72
8.2.2 Harmless Data Violations . . . 73
8.2.3 Harmful Register Violations . . . 74
8.3 Fixing Violations . . . 74

9 Conclusion **77**

A Serializability Violations **79**

Bibliography **85**

Chapter 1

Introduction

Replicated data stores have become an important mechanism to achieve scalability and availability of modern distributed systems and internet services ([9], [10] [11], [15], [16]). While strong consistency is desired, consistency, availability and partition-tolerance cannot be provided simultaneously, as stated by the CAP theorem [19]. Therefore, replicated data stores often only provide weak consistency guarantees like eventual consistency. However, this relaxation of consistency can lead to non-serializable [12] behaviors, called serializability violations. An execution is non-serializable if there does not exist a serial schedule of the execution's operations that leads to the same outcome. This complicates application development using these data stores, because parts of the application that require strong consistency are now responsible for providing that guarantee. Serializability allows to establish desired correctness properties easily without considering the effects of weak consistency. Because deciding serializability on concurrent executions is NP-hard in general, many stronger serializability criteria have been explored ([12], [13], [23]).

As a motivating example for a serializability violation consider the following piece of code:

```
1 setSecurityLevel(TOP_SECRET);
2 if (getSecurityLevel() == PUBLIC) {
3   releaseInformation();
4 }
```

A serializability violation can now lead to the situation, that the classification of information as top secret in line 1 is not observed by the security level check in line 2 and therefore top secret information is released. This constructed example shows that serializability violations can cause serious bugs in applications.

Brutschy et al. [13] propose a serializability criterion that generalizes conflict serializability to eventually consistent semantics and further supports high-level operations, allowing precise reasoning about replicated data types. Because the criterion has previously only been evaluated on key-value stores and cloud platforms, this work applies this criterion to the popular distributed database MongoDB and shows that the criterion is feasible to discover serializability violations in real-world applications by performing and evaluating a dynamic analysis on a set of open source Node.js applications that use MongoDB as a replicated data store.

1.1 Contributions

The main contributions of this work are:

- Formalizing the consistency model of MongoDB and giving a complete list of serializability violation classes that can occur
- Providing an instrumentation method to apply the criterion of Brutschy et al. to the replicated database MongoDB
- Demonstrating the usefulness of the approach by providing an implementation that performs a dynamic analysis on a set of open source Node.js applications and discovers previously undetected errors

1.2 Overview

We start by explaining the serializability criterion of Brutschy et al. in Chapter 2. Chapter 3 gives an introduction into MongoDB and formalizes its consistency model. We further present all possible classes of serializability violations that can occur under this model and prove their completeness. We introduce our instrumentation method in Chapter 4 and describe how to instrument a MongoDB client application in order to perform the serializability checking based on the introduced serializability criterion. Chapter 5 covers how to use the data collected by the instrumented clients during the dynamic analysis to apply the proposed algorithm from [13]. We further increase the frequency of occurring serializability violations by triggering network partitions as described in Chapter 6. Chapter 7 covers the implementation of the instrumentation and the analysis. We perform the described dynamic analysis on a set of open source Node.js applications that use MongoDB as a data store and evaluate the detected serializability violations in Chapter 8.

Chapter 2

Preliminaries

This chapter covers basic definitions and the serializability criterion used in this work. We further cover related work in this field of research.

2.1 Serializability Criterion

Brutschy et al. [13] propose a criterion that can be applied to a dynamic execution of database operations of a replicated data store that provides *eventual consistency*, which is defined later in this section.

2.1.1 Basic Definitions

To describe the criterion we need to specify the terminology and give some basic definitions.

Actions and Events: A primitive operation that is performed by the data store is called an *action*. An action is a database operation with concrete arguments and return values. In MongoDB these are insert, delete, update and find. If an action occurs in an execution it is called an *event*. Chapter 3 gives detailed explanations of the possible MongoDB events. If we assume that we have a data store that provides the operations *get* and *set* on a record *r*, examples for actions are r.get():3, where 3 is the return value and r.set(4), where 4 is the value to be set.

Updates and Queries: Each action is assumed to be either an update or a query. Updates modify the data store but return no values while queries do not modify the data store but return values. This assumption is non-restrictive because every action can be converted into a sequence of updates and queries. We refer to the MongoDB operations insert, delete and update as updates and to MongoDB find operations as queries.

Commutativity: Two events e_1 and e_2 are called *commutative* if and only if $e_1e_2 \equiv e_2e_1$, where $\equiv$ denotes equivalence between two sequences of traces and states that the two sequences produce the same final state for any initial state. Intuitively, commutativity means that swapping two neighboring commuting elements in a sequence of actions produces an equivalent one. Consider for example a data store that supports the simple operation *set* on a record and let r_1 and r_2 be two database records. Then the two events r_1.set(1) and r_2.set(4) commute because they operate on different records.

Absorption: An event e_2 *absorbs* another event e_1 if and only if $e_1e_2 \equiv e_2$. Intuitively, e_2 absorbs any effect of e_1. Let r be a database record. Then r.set(3) absorbs r.set(5) because they both affect the same record and the later event simply overwrites the effect of the former one.

A *far-reaching* absorption relation $\blacktriangleright$ is defined by: $u \blacktriangleright v \Leftrightarrow \forall \chi \in U^* : u\chi v \equiv \chi v$, where U^* is the set of update sequences. Intuitively, far-reaching absorption is an absorption that is not influenced by events that occur in between the absorbing and the absorbed event.

A *trace* is a sequence of events that relaxes the total order of a sequence to a partial order by omitting ordering commutative events.

Processes and Transactions: Let E be the set of events. A *process* is a partition of the set E, while a *transaction* is a partition of the processes into transactions $t_1, t_2, ... \subseteq E$.

Eventual Consistency: A data store is strongly eventual consistent if it satisfies the following conditions:

1. Processes observing the same set of events observe the same state
2. Updates are eventually observed by all processes

2.1.2 Relations

The criterion by Brutschy et al. provides an algorithm that works on a set of relations on events and constructs a graph from it. A cycle in that graph represents a serializability violation.

This section introduces the relations that are used by the criterion. Let $E = U \cup Q$ be a set of events, while U contains the update events and Q the query events.

Program Order: The program order relation po $\subseteq (E \times E)$ partitions E into processes. Intuitively, it reflects the order of event executions on a client process.

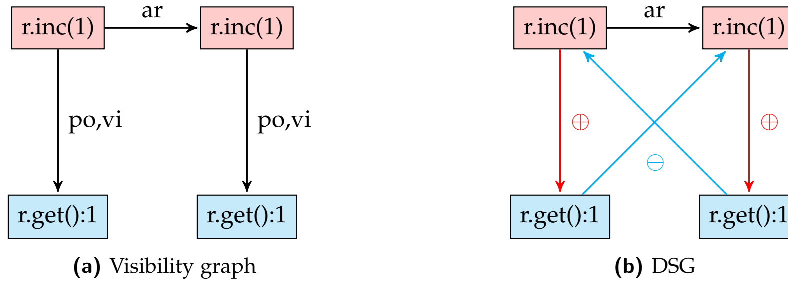


Figure 2.1: Example of a DSG containing a serializability violation, indicated by a cycle in the graph. *inc* is assumed to be an operation that increments a record r of the data store.

Arbitration Order: The arbitration order relation $\text{ar} \subseteq (U \times U)$ orders non-commuting updates and is determined by the semantics of the data store. Intuitively, it gives an order of execution on update events.

Visibility: The visibility relation $\text{vi} \subseteq (U \times Q)$ states whether a query can observe the effects of an update or not. An update event is visible to a query event if it was executed on the queried replica before the execution of the query.

Program order, arbitration order and visibility form a *schedule* of events.

Dependency: The dependency relation $\oplus \subseteq \text{vi} \subseteq (U \times Q)$ reduces the visibility relation by expressing that a query depends on a visible update if the result of the query would change if the update becomes invisible.

Anti-Dependency: The anti-dependency relation $\ominus \subseteq (Q \times U)$ expresses that an update u anti-depends on a query q if $(u, q) \notin \text{vi}$ but making the update visible to the query implies $(u, q) \in \oplus$.

Both, $\oplus$ and $\ominus$ can be derived from the relations ar and vi and corresponding commutativity and absorption specifications for the events as described in [13].

Dependency Serialization Graph: The dependency serialization graph (DSG) is defined as $\text{po} \cup \text{ar} \cup \oplus \cup \ominus$. Each cycle in the DSG of an event trace represents a serializability violation.

Figure 2.1 shows a simple example of a visibility graph ($\text{po} \cup \text{ar} \cup \text{vi}$) and the corresponding DSG.

2.2 Related Work

Burckhardt [14] defines a model for specifying consistency models by deconstructing them into several consistency guarantees and building a hierachy of different consistency models giving the foundation of the model used by [13]. This work also uses the proposed ordering guarantees in a modified shape to specify the consistency model of MongoDB.

There exist models for the detection of consistency anomalies of cloud data stores using cycle detection on a dependency graph [28].

In [29] Zellag et al. further propose an approach for detecting consistency anomalies in multi-tier architectures and automatically reducing their occurrence. Their approach is completely independent of the used data store, which is treated as a black box. Because [28] considered cloud applications, only an approximated graph could be provided, which can lead to false positives during cycle detection. Furthermore, cloud applications do not allow to take isolation levels into account. [29] tackles both shortcomings.

Other works consider Snapshot Isolation (SI) as a consistency guarantee and propose methods to determine whether applications under SI are free of serializability violations by using cycle detection in a dependency graph ([17], [20]).

Bernardi et al. give a criterion to detect serializability violations on causally consistent data stores [11].

The mentioned works differ from the criterion of Brutschy et al. by expecting the guarantee of stronger consistencies than strong eventual consistency from the data store. Further, these approaches use low-level read and write reasoning, while Brutschy et al. use algebraic reasoning, which makes the criterion applicable to a broader class of systems.

Additionally to the already described serializability criterion, Brutschy et al. also provide a polynomial-time algorithm that checks whether a given program execution satisfies the criterion. Further, they provide an implementation of the algorithm for the cloud platform TouchDevelop and the key-value-store Riak and evaluate the usefulness of the criterion on open source projects by reporting previously undetected errors. This work further proves the usefulness of the criterion by applying it to a replicated database and evaluating it on a set of open source applications.

Chapter 3

MongoDB

MongoDB is a replicated multiplatform NoSQL database developed in C++ in 2007 by 10gen. It is a document store database, which groups documents with a structure as described in 3.1 into collections. A collection can keep documents with a differing structure in the same collection, but this leads to a less efficient data management of MongoDB [8]. Furthermore, MongoDB does not support transactions.

MongoDB databases support replication and sharding. Replication means replicating the same data on different MongoDB instances. Usually these instances are located on different physical machines, which increases availability of the data. The set of MongoDB instances that are part of this replication is called *replica set* and each instance is called *replica node*.

Sharded sets consist of MongoDB instances that each keep one part of the whole data, so the data is not replicated but distributed over the nodes. This work focuses on replica sets and will not further discuss sharded sets.

We use the following terminology within this work:

Operation: A MongoDB operation can either be an insert, update, delete or find. We refer to find as query, because this avoids confusion with the find-method of the MongoDB Shell. An operation can contain several operators.

Operator: A MongoDB operator is either a query operator or an update operator and starts with a $ sign.

3.1 Document Structure

A document is a set of key-value pairs. A key-value pair is also called an attribute, where the key is the attribute name and the value is the attribute

```
{
  "_id": "507f191e810c19729de860ea",
  "type": "book",
  "title": "Science for Scientists",
  "authors": [
    {"name": "Clara", "age": 46},
    {"name": "Josh", "age": 23}
  ]
}
```

Figure 3.1: Example of a MongoDB document

value. Values can have different data types as listed in the MongoDB documentation and can also contain other documents.

Figure 3.1 shows an example of a valid MongoDB document.

Each stored document in MongoDB has an *object-ID*, which uniquely identifies a single document within a single collection and is represented by the key *_id*. Relational databases have a similar concept using a so called primary key. The name of the object-ID (_id) cannot be changed and every document needs to have an attribute with the name _id and a unique value. Furthermore, the value of the object-ID is immutable and can therefore not be deleted or changed.

It is also possible to define indexes on MongoDB collections. Like in Relational Database Management Systems (RDMS), indexes speed up searching the table/collection when only indexed attributes are used in the search query. Furthermore, indexes guarantee uniqueness of the corresponding values within a collection. It is possible to define compound indexes, which consist of two or more keys and guarantee uniqueness of the combination of all involved values. As an example, consider we define a compound index on the keys *name* and *age* on a collection *users*. The compound index guarantees that there is only one document in the collection users having name *John* and age 30. But there can exist many documents with name *John* and many documents with age *30*.

3.2 Operations

In this section we cover the different low level operations of MongoDB. There are interfaces like the MongoDB Shell or certain drivers that provide higher level operations. We explain some of these higher level operations in Section 4.1, but they are transformed into low level operations on the

server. We have insert, update, delete and query operations. Each operation is executed on a specific database collection.

3.2.1 Inserts

An insert operation consists of an insert document, which needs to have the MongoDB document structure but can lack the object-ID. When the server receives an insert, it will add the object-ID if it is not contained in the document. If the document contains an object-ID that is already present in the corresponding collection, the server throws an error. We ignore failed updates in our analysis, because the serializability criterion demands updates not to have pre-conditions. An example for an insert document is `{"name": "John", "age": 30, "lastName": "Doe"}`.

3.2.2 Queries

A query operation consists of a selector, which is a MongoDB document that can contain MongoDB query operators or normal key-value pairs. The following examples demonstrate the syntax and semantics of MongoDB selectors:

- `{"name": "Sarah", "age": 26}`: matches all documents with name equaling Sarah and age equaling 26.
- `{"name": "Bob", "age": {"$gt": 40}}`: matches all documents with name equal to Bob and age greater than 40.
- `{"$or": [{"name": "Bob"}, {"name": "Sarah"}]}`: matches all documents with name equaling Bob or Sarah.

We use the query operators $or for disjunction and $gt, which demands that a value has to be greater than the given value.

3.2.3 Updates

An update operation consists of a selector and a modifier. Selectors are described with query operations. A modifier is a MongoDB document that can contain MongoDB update operators or normal key-value pairs. If the update modifier only consists of key-value pairs that do not contain any operators, the modifier is called replacement modifier or replacement document. Examples of modifiers are:

- `{"name": "Sarah", "age": 26}`: replacement modifier that is used to completely replace a document.
- `{"$set": {"title": "newTitle", "price": 200}}`: sets the attribute title to newTitle and price to 200 and leaves all other attributes untouched.

- `{"$inc":{"age":1, "yearsToLive":-1}}`: increments attribute age and decrements attribute yearsToLive by 1 and leaves all other attributes untouched.

3.2.4 Deletes

A delete operations consists of a selector, specifying which documents to delete.

3.3 Replication

Replication can increase redundancy and availability. It can further increase the read capacity as clients can read from more than one server. A replica set in MongoDB is defined as a group of MongoDB instances that maintain the same data set. Each MongoDB instance is called a replica node that can be either a *primary* or a *secondary* node. A replica set can only have one primary node, which receives all write operations and keeps an operations log (see Section 3.3.2), which contains all changes to the primary's data set. Secondary nodes replicate the primary's data by replicating its operations log and applying the contained operations to their local data set.

3.3.1 Automatic Failover

If the primary node becomes unavailable the secondary nodes will hold an election to determine a new primary node. Only if a majority of all nodes (including unreachable ones) of the replica set vote for a specific node it becomes the new primary node. The other way around, a primary steps down as a primary if it loses contact to the majority of nodes. To allow a majority in the elections it is necessary that the replica set contains an odd number of nodes. This method assures that a primary node is always connected to a majority (including itself) of nodes from the set.

It is possible that a replica set transiently contains two primary nodes, as shown in Figure 3.2. This is the case if a primary is not reachable by a majority of nodes and a new primary is elected. The old primary might take some time to realize that it must step down because it cannot reach a majority of nodes anymore.

Arbiter nodes do not replicate data. Their only purpose is to ensure an odd number of replica nodes. This is useful in cases where only an even number of replicating nodes are possible or affordable. The arbiter solves the problem of tied votes in an election, but uses less resources than a data bearing node.

3.3.2 Oplog

The oplog (operations log) is a special MongoDB collection that stores all executed operations that changed the data of the corresponding MongoDB instance. As already mentioned, secondary nodes copy the oplog from the primary and apply the contained operations to replicate the primary's data set. Each secondary has a local copy of the primary's oplog that represents the database state of a secondary node.

The oplog only contains idempotent operations. That means that every insert, update or delete operation is converted into one or many idempotent operations that have an equivalent effect. This has consequences for the atomicity of MongoDB operations. All update operations are only atomic on a per document basis. *Upsert* operations for example are special update operations that insert a document if the corresponding selector does not match any document of the collection. However, because an upsert is not idempotent it is split up into a query and an update or an insert. Multi-updates, which are defined as updates that update multiple documents, are split up into a query and multiple single updates. To guarantee idempotency, the selector of each update in the oplog only contains the object-ID. This guarantees that the document to be updated is uniquely identified. Furthermore, non-idempotent update operators are converted into idempotent ones.

3.3.3 Read Preference

Read preference is a setting for client applications and describes how they route the query to the members of a replica set. The read preference modes relevant for this work are:

primary (default): Query is routed to the primary. If no primary is reachable from the client, the operation fails.

primaryPreferred: Query is routed to the primary. If no primary is reachable the read will be performed on a secondary.

secondary: Query is routed to the secondary. If no secondary is reachable from the client, the operation fails.

secondaryPreferred: Query is routed to the secondary. If no secondary is reachable the read will be performed on the primary.

It is further possible to specify the exact nodes to read from.

3.3.4 Read Concern

Read concern is an option for query operations and determines which data to return from it. There exist the following read concern levels:

local (default): The instance's most recent data is returned by the query.

majority The instance's most recent data that has been acknowledged as having been written to a majority of replica nodes is returned by the query.

linearizable: Data that has been acknowledged as having been written to a majority of replica nodes prior to the start of the query is returned. This level is only allowed for queries to the primary node.

3.3.5 Write Concern

Write concern is an option for write operations (insert/update/delete) and specifies the level of acknowledgement requested from MongoDB. The acknowledgement of the write concern is an acknowledgement to the client, while that of the read concern is an acknowledgement of a replica node to all other replica nodes. A write concern can take the following options:

***number*:** *number* represents the minimum number of nodes that an operation has to propagate to in order to count as acknowledged.

majority: Write operations need to have propagated to a majority of nodes of the replica set to count as acknowledged.

The default value for the write concern is 1. This implies that an operation becomes acknowledged as soon as it has propagated to the primary node.

3.3.6 Consistency Model

This section describes the consistency model of MongoDB that is inferred from the specifications of the previous sections. Write concern and read concern are essential to the level of consistency that is achieved. We will present some configurations and discuss the resulting consistency level. We start by giving very weak configurations and continue by strengthening them. As we need eventual consistency for the serializability criterion to be applicable, we discuss which configurations satisfy this guarantee.

We use the following terminology:

Stale reads describe read operations that do not return the most recent acknowledged and therefore persistent data of the replica set.

Example 3.3.1 Consider the following sequence of events in a replica set with one primary and two secondary nodes:

1. A client sends a write operation to the primary.
2. The client sends a read operation to a secondary that has not yet executed the write from Step 1. This is a stale read, because the update from Step 1 has not yet propagated to the secondary. □

Dirty reads specify reads of data that is going to be rolled back and will therefore not be persistent.

Example 3.3.2 Consider the following sequence of events in a replica set with one primary and two secondary nodes:

1. A network partition occurs that blocks the communication between the primary on one side of the partition and the two secondaries on the other side of the partition as illustrated in Figure 3.2a.

2. The secondary nodes realize that they cannot reach the primary and elect one of the secondaries to be the new primary (Figure 3.2b)

3. A client (that is on the same side of the network partition as the old primary) sends a write operation to the old primary, which has not yet realized that it cannot reach the two other nodes of the replica set and executes the write.

4. The client sends a read operation to the old primary and reads the previously written value.

5. The old primary realizes that it is isolated and steps down as a primary (Figure 3.2c).

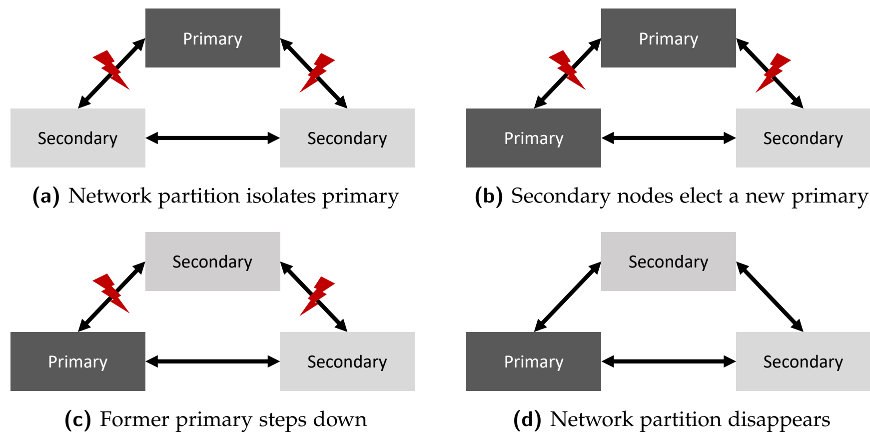
6. Another client sends a write operation to the new primary of the replica set. The new primary executes that write.

7. The network partition disappears and the old primary (which is a secondary now) pulls the oplog from the new primary. There is now a conflict between the oplog of the old primary and the new one because both applied write operations. Therefore, the old primary rolls back the write operation and executes the missing operations from the oplog of the new primary (Figure 3.2d). Step 4 contains a dirty read, because the read value is rolled back in Step 7. □

Dirty writes describe write operations that are acknowledged by MongoDB but will later be rolled back.

Consider Example 3.3.2 again. Step 3 is a dirty write if it is acknowledged to the client, because it is rolled back at Step 7. The acknowledgement of the write depends on the write concern.

In order to satisfy eventual consistency, we cannot have dirty reads or dirty writes, as they both violate the condition of EC that each update has to propagate to all nodes eventually.



(a) Network partition isolates primary **(b)** Secondary nodes elect a new primary

(c) Former primary steps down **(d)** Network partition disappears

Figure 3.2: A possible network partition in a replica set and the subsequent recovery. It shows how there can transiently exist two primary nodes.

Read Concern: local, Write Concern: 1

This configuration can lead to stale reads, dirty reads and dirty writes.

Consider Example 3.3.2. A write concern of 1 leads to Step 3 performing a dirty write. A read concern of local leads to Step 4 performing a dirty read. Therefore, this configuration does not provide eventual consistency.

Read Concern: local, Write Concern: majority

By setting write concern to majority we eliminate dirty writes. However, having a read concern of local still allows stale and dirty reads as in Step 4 of Example 3.3.2. Therefore, this configuration is still not eventually consistent.

Read Concern: majority, Write Concern: majority

By strengthening the read concern we can avoid dirty reads. Step 4 in Example 3.3.2 is not possible with a majority read concern, because the value written in Step 3 cannot be acknowledged by a majority of nodes. Stale reads are still possible in this configuration. However, this configuration provides EC, because stale reads to not violate it.

Read Concern: linearizable, Write Concern: majority

We further strengthen the read concern by setting it to linearizable. This configuration does not even allow stale reads anymore. But this level of consistency comes to the price of read capacity because all reads have to be performed on the primary node. The only advantages of having a replica set instead of a standalone MongoDB instance in this configuration are availability and redundancy.

We conclude that the weakest configuration that still provides EC needs a read and write concern of majority.

3.3.7 Consistent Prefix

As discussed, we need majority read concern and majority write concern in order to provide strong eventual consistency. This configuration further gives an ordering guarantee on events that we call *consistent prefix* (CP). We adapt a slightly modified version of the definition given by [14]: A consistency model has the consistent prefix guarantee if and only if $(\text{ar};\text{vi}) \subseteq \text{vi}$.

More intuitively, it expresses that an update event u being visible to a query q implies that all update events that are arbitrated before u are also visible to q. Therefore, each query always observes a consistent prefix of all existing update events ordered by arbitration.

As an explanation, assume the consistent prefix guarantee does not hold. Then there exists an update event u_2 that is visible to a query event q_1, but there also exists an update event u_1 that is arbitrated before u_2 and is not visible to q_1. For this situation to occur we need q_1 to perform a dirty read, which is not possible with majority read concern.

Note that this guarantee strengthens eventual consistency.

3.3.8 Possible Sequential Consistency Violations

This section introduces violations of the *sequential consistency* (SC) guarantee, that is formally defined in Section 3.3.8 and is a weaker guarantee than serializability but stronger than eventual consistency. The reason for considering only SC in this section is that MongoDB does not support transactions and serializability is a guarantee on transactions. So we focus on violations that are caused by the MongoDB consistency model.

Again, we assume majority read concern and majority write concern. We further assume a read preference that allows reads from secondary nodes, because this is the only way of benefiting from the possible performance advantages of a replica set. We present two classes of serializability violations and then show that these violation types are indeed the only possible violations that can occur in the MongoDB consistency model with the assumed write and read concern.

Read My Writes

Read My Writes (RMW) is an ordering guarantee of events defined as follows: $(\text{po} \cap (U \times Q) \subseteq \text{vi})$.

Violating this property means: $\exists(u,q) \in \text{po} \cap (U \times Q)$. $(u,q) \notin \text{vi}$. Intuitively, it is violated if an update is invisible to a query even though it is ordered before it according to program order.

Monotonic Reads

Monotonic Reads (MR) is defined as the following ordering guarantee: $(\text{vi}; (\text{po} \cap (Q \times Q))) \subseteq \text{vi}$.

Violating this property means: $\exists u \in U,\ q_1,\ q_2 \in Q$. $(u,q_1) \in \text{vi} \wedge (q_1,q_2) \in \text{po} \wedge (u,q_2) \notin \text{vi}$. Intuitively, this property is violated if a query event q_2 that occurs later than another query event q_1 in program order cannot see an update event u but q_1 can.

Proof of Completeness

We prove that violations to the RMW and MR properties are the only ones that can occur if we assume the consistent prefix guarantee.

We can further assume *causal arbitration* (CA) for MongoDB: $\text{po} \cap (U \times U) \subseteq \text{ar}$ [14]. This is guaranteed by the *monotonic writes* guarantee of MongoDB, which expresses that the order of execution of write operations equals the order of arrival at the server. Because program order is only allowed on non-overlapping operations, CA is guaranteed.

non-circular causality (NCC) is defined as $\text{acyclic}((\text{po} \cup \text{vi})^+)$ and directly follows from basic eventual consistency as shown in [14].

We construct our proof by assuming CP, RMW, MR and CA and show that these guarantees imply sequential consistency. The model is *sequentially consistent* (SC) if $(\text{ar} \cup \text{vi} \cup \text{vi}^- \cup \text{po})$ is acyclic, where $\text{vi}^- := ((U \times Q) \setminus \text{vi})^{-1}$. This definition is a general version of the serializability criterion given in [13], where $\text{vi} = \oplus$ and $\text{vi}^- = \ominus$. $(\text{ar} \cup \text{vi} \cup \text{vi}^- \cup \text{po})$ satisfies Axiom 1 and 2 and can therefore be applied to Theorem 1, which implies SC. The axioms and the theorem are defined in [13].

Proof We assume CP, RMW, MR, CA and show that $(\text{ar} \cup \text{vi} \cup \text{vi}^- \cup \text{po})$ is acyclic.

We start by showing that $\text{vi} \cup \text{ar}$ is acyclic and use this to prove $\text{vi} \cup \text{ar} \cup \text{vi}^-$ to be acyclic, too.

Lemma 3.1 $\text{vi} \cup \text{ar}$ *is acyclic.*

Proof Since $\text{vi} \subseteq (U \times Q)$ and $\text{ar} \subseteq (U \times U)$, a cycle must consist only of updates. That is a contradiction to arbitration being acyclic. □

Lemma 3.2 $\text{vi} \cup \text{ar} \cup \text{vi}^-$ *is acyclic*

Proof Assume it contains a cycle. Then it contains a path $(e_1, ..., e_n)$ with $e_1 = e_n$. Because $\mathrm{vi} \cup \mathrm{ar}$ is acyclic, this path must contain a pair $(q, u_1) \in \mathrm{vi}^-$ that is essential to the cycle. In order to form a cycle there must also exist an edge from an update to a query. The only relation defined on that set is vi and therefore there exists a u_2 with $(u_2, q) \in \mathrm{vi}$.

Case 1 ($u_1 = u_2$): We have $u_1 \xrightarrow{\mathrm{vi}} q$ and $q \xrightarrow{\mathrm{vi}^-} u_1$, which contradicts our definition of vi and vi^-.

Case 2 ($u_1 \neq u_2$): Because arbitration is a total order we must either have one of the two cases:

Case 2.1 ($u_1 \xrightarrow{\mathrm{ar}} u_2$): $\overset{\mathrm{CP}}{\Longrightarrow} u_1 \xrightarrow{\mathrm{vi}} q$. This is a contradiction to the definition of vi and vi^-, because we cannot have both, $u_1 \xrightarrow{\mathrm{vi}} q$ and $q \xrightarrow{\mathrm{vi}^-} u_1$.

Case 2.2 ($u_2 \xrightarrow{\mathrm{ar}} u_1$): $\mathrm{vi} \cup \mathrm{ar}$ contains a cycle. This is a contradiction to Lemma 3.1. □

We will now show that $(\mathrm{ar} \cup \mathrm{vi} \cup \mathrm{vi}^- \cup \mathrm{po})$ is acyclic.

Assume it contains a cycle. We perform a case distinction on the number of nodes that are part of the cycle:

Case 1 (Cycle contains 1 event): All relations are acyclic, so must be their union.

Case 2 (Cycle contains 2 events): The cycle contains two events $n_1, n_2 \in E$ with $n_1 \xrightarrow{\mathrm{po}} n_2$. In order to form a cycle, there must be a relation from n_2 to n_1:

Case 1.1 ($n_2 \xrightarrow{\mathrm{ar}} n_1$): $n_1 \xrightarrow{\mathrm{po}} n_2 \overset{\mathrm{CA}}{\Longrightarrow} n_1 \xrightarrow{\mathrm{ar}} n_2 \xRightarrow{\mathrm{ar\ total}} n_2 \overset{\mathrm{ar}}{\not\to} n_1$, which contradicts the assumption of this case.

Case 1.2 ($n_2 \xrightarrow{\mathrm{po}} n_1$): This means that po contains a cycle, which contradicts po being acyclic.

Case 1.3 ($n_2 \xrightarrow{\mathrm{vi}} n_1$): $n_1 \xrightarrow{\mathrm{po}} n_2$ and $n_2 \xrightarrow{\mathrm{vi}} n_1$ contradict NCC.

Case 1.4 ($n_2 \xrightarrow{\mathrm{vi}^-} n_1$): $n_1 \xrightarrow{\mathrm{po}} n_2 \xRightarrow{\mathrm{RMW}} n_1 \xrightarrow{\mathrm{vi}} n_2$ and therefore we cannot have $n_2 \xrightarrow{\mathrm{vi}^-} n_1$ because our definition of vi and vi^- does not allow both $n_1 \xrightarrow{\mathrm{vi}} n_2$ and $n_2 \xrightarrow{\mathrm{vi}^-} n_1$.

Case 3 (Cycle contains more than 2 events): The cycle contains a path $(g_1, ..., g_n)$: $g_1 = g_n$. Because $\mathrm{vi} \cup \mathrm{ar} \cup \mathrm{vi}^-$ is acyclic, there exists a path $(q_1, ..., q_m)$ in po of maximal size that is essential to the cycle. This path can only

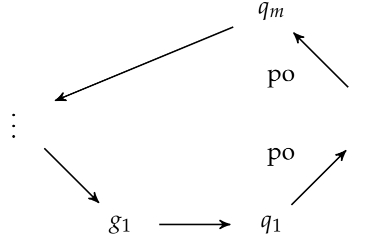


Figure 3.3: Structure of cycle for correctness proof

consist of queries. This follows from CA and RMW. These guarantees ensure that the only edges that program order can contribute to the cycle that are not contained in $\mathrm{vi} \cup \mathrm{ar} \cup \mathrm{vi}^-$ are between two queries.

Assume this cycle has the structure shown in Figure 3.3.

Observe that $g_1 \in U$ because otherwise g_1 must be part of $(q_1, ..., q_m)$. We already observed that $q_1, q_2, ..., q_m \in Q$.

We must either have $q_m \xrightarrow{\mathrm{vi}^-} g_1$ or $g_1 \xrightarrow{\mathrm{vi}} q_m$. $g_1 \xrightarrow{\mathrm{vi}} q_m$ implies that $\mathrm{vi} \cup \mathrm{ar} \cup \mathrm{vi}^-$ already contained a cycle, which is a contradiction. So we must have $q_m \xrightarrow{\mathrm{vi}^-} g_1$. But we also have $g_1 \xrightarrow{\mathrm{vi}} q_1$, because vi is the only relation defined on $U \times Q$ besides po. Together with $q_1 \xrightarrow{\mathrm{po}} q_m$ and MR follows that $g_1 \xrightarrow{\mathrm{vi}} q_m$, a contradiction to $q_m \xrightarrow{\mathrm{vi}^-} g_1$.

Therefore, $(\mathrm{ar} \cup \mathrm{vi} \cup \mathrm{vi}^- \cup \mathrm{po})$ is acyclic.

□

3.3.9 Atomicity Violations

Atomicity violations are one class of violations that are even possible under SC. After focusing only on SC in Section 3.3.8, we now consider serializability again. Even though not supported by MongoDB, our method provides annotating a group of events as a transaction, which signals the analysis that these operations are assumed to be executed atomically. The analysis will then merge the nodes of the corresponding DSG into a single node. Because MongoDB does not support transactions and a transaction violation is based on the assumption of an atomic execution of the corresponding events, we do not consider this as a serializability violation of the MongoDB consistency model, because MongoDB does not make that assumption of atomicity. Nevertheless, transactions can be helpful to detect violations that are based on the fact that the developer assumed an atomic execution of the events that are marked as one transaction. Therefore, we support the notion of transactions as described in Section 4.1.4.

Chapter 4

Dynamic Analysis for MongoDB

We perform a dynamic analysis of MongoDB clients to detect serializability violations. Therefore, we need to record all database events of the application and instrument them in a way that allows to perform the analysis proposed by Brutschy et al. [13]. After performing an execution of the instrumented application to be analyzed, we create the relations program order, arbitration order and visibility, run the proposed algorithm to yield the dependency and anti-dependency relation and finally build the corresponding DSG. We achieve that by extending the analysis tool used in [13]. A cycle detection on the DSG will then uncover the serializability violations that occurred during the instrumented execution run of the application.

This chapter covers the instrumentation of the client application such that we are able to extract the needed relations from them.

4.1 Instrumentation

This section describes how to instrument the database events of a client in a way that allows to yield the relations arbitration order, visibility and program order and further gives the necessary transformations of driver operations in order to perform the proposed instrumentation.

4.1.1 Program Order

Depending on the programming model of the client, the program order needs to be a partial order. This is the case for clients with an asynchronous programming model like Node.js for JavaScript [26] or Play! [6] for Java and Scala. The fact that a database event e_a started before another database event e_b does not imply that e_a also returns before e_b. Therefore we store a start time stamp and a return time stamp with each event. If we have two database events e_1 and e_2 by the same client, we say that $(e_1, e_2) \in \text{po}$ iff

endTime(e_1) $<$ startTime(e_2). The analyzer can then later use these time stamps to construct the program order as explained in Section 5.3.1. The time stamps must be generated by the client because only the client's system clock is relevant for this purpose.

4.1.2 Arbitration Order

As described in Section 3.3.6, all updates go to the primary node of our replica set. Therefore there exists a total arbitration order of all updates, inserts and deletes, which is indicated by a time stamp of execution on the server that is attached to each of those events.

4.1.3 Visibility

In order to construct the dependency and anti-dependency relations with ECRacer in the later analysis, we need to be able to construct the visibility relation from the instrumented records. To achieve this, we attach a unique update identifier (UID) to every update or insert event.

The UID is generated by the client and should satisfy the following conditions:

1. The UID is unique.
2. The UID contains enough information to uniquely identify the corresponding client application.

Both conditions only need to provide uniqueness within a single dynamic analysis run. While the first condition is necessary for the visibility relation, as will be explained in this section, the second condition is essential for creating the program order relation, described in Section 5.3.1.

However, we give more semantic meaning to it by constructing it using the following pattern: *clientId#subClientId#eventCounter*.

clientID uniquely identifies a single client application, which is a Node.js instance in our case.

subClientID uniquely identifies a child process within a single client application. We introduce this ID to handle applications that spawn child processes. This identifier is not necessary for the uniqueness of the UID, because the eventCounter is shared for all child processes of a client application. However, it helps to associate an event with a specific process for debugging purposes and identifying sources of serializability violations.

eventCounter is an increasing integer that uniquely identifies a single event within a client application.

The UID is added as a property to the updated/inserted document and is therefore stored in the database. The visibility relation can be constructed by checking the UIDs of all documents that a query retrieves. For a UID *uid* let $\text{upd}(uid)$ be the corresponding update event. The visible updates/inserts (deletes are covered in Section 4.2.6) of a query q are obtained in the following way:

1. Iterate through all documents retrieved by q and calculate the maximum (by arbitration) UID $maxUID$
2. An update/insert u is visible to q iff $(u, \text{upd}(maxUID)) \in ar$

This follows from the consistent prefix property described in Section 3.3.7.

Example 4.1.1 This example demonstrates how the visible updates/inserts of a query are determined. Let i_1, i_2 be two insert events with d_x being the document inserted by i_x. Let further u_1 and u_2 be two update events with s_x and m_x being the selector and modifier of u_x. We define:

- d_1 := `{"name":"John","age":30, "__uid":"0#0#0"}`
- d_2 := `{"name":"Sarah","age":23, "__uid":"1#0#1"}`
- s_1 := `{"name":"John"}`
- s_2 := `{"name":"Sarah"}`
- m_1 := `{"$set":{"age":31, "__uid":"1#0#0"}}`
- m_2 := `{"$set":{"age":24, "__uid":"0#0#1"}}`

We assume the arbitration $i_1 \xrightarrow{ar} u_1 \xrightarrow{ar} i_2 \xrightarrow{ar} u_2$, where we introduce the notation $op_1 \xrightarrow{ar} op_2 \Leftrightarrow (op_1, op_2) \in ar$.

Let q be a query event with selector $s := \{\}$. An empty selector in MongoDB matches all documents of the collection. Let q further return the set D of documents with $D :=$

```
{
  {"name": "John", "age": 31, "__uid": "1#0#0"},
  {"name": "Sarah", "age": 23, "__uid": "1#0#1"}
}
```

Because each UID uniquely identifies one update event, they are also ordered by arbitration: $\text{upd}(0\#0\#0) \xrightarrow{ar} \text{upd}(1\#0\#0) \xrightarrow{ar} \text{upd}(1\#0\#1) \xrightarrow{ar} \text{upd}(0\#0\#1)$. The maximum UID of all documents contained by D is $maxUID = 1\#0\#1$, because $\text{upd}(1\#0\#0) \xrightarrow{ar} \text{upd}(1\#0\#1)$.

We said that an update/insert u is visible to q iff $(u, \text{upd}(maxUID)) \in ar$. Therefore we construct the following visibility relation: $\{(i_1, q), (i_2, q), (u_1, q)\} \subseteq vi$.

□

The instrumentation implementation should also take care to remove the UID attribute from all documents before returning them to the client application. This avoids a change in the behavior of the application that is caused by an additional property on the retrieved document. This can for instance be the case if the application counts the fields of that document.

But there remain some further extensions to the client in order to obtain the complete visibility relation. These are described in the following sections.

4.1.4 Transactions

MongoDB does not support database transactions, but in the analysis we support them. There are many ways to specify transactions. For instance, they can be annotated in the code by the user. To assign an event to a transaction it is sufficient to attach a transaction identifier to it.

In our instrumentation of Node.js applications we group all events of a single HTTP-request that was sent to the application into one transaction. The reasoning behind this approach is that many developers are used to the situation where each request is handled by a synchronous server using a database transaction of a relational database. This habit leads to programming patterns that fail to be reliable on asynchronous servers without database transactions. If websockets are used instead of HTTP we apply this approach to a websocket request.

4.1.5 Special Cases

Limited Queries

Limited queries are those that limit the result set of a cursor. The limit specifies the maximum number of documents the cursor will return. So query event q with a selector $s :=$ `{"name":"John"}` and a limit of 10 returns all but at most 10 documents with the attribute *name* having the value *John*. Limited queries must be handled carefully to keep our visibility relation correct. These queries are therefore transformed into unlimited queries to retrieve all visible documents. Before returning the result set to the client application later, we limit it to the specified number to avoid changing the application's behavior.

Projections

All projections of non-ID-queries must be removed. Projections reduce each retrieved document of a query to a subset of its attributes. This is usually

used to decrease network traffic and memory usage of the client. However, since we retrieve the whole collection and filter the documents at the client, we need the complete documents in order to perform a correct filtering. After doing the filtering, the projection should be performed as part of the instrumentation to prevent a different client behavior.

GetMore

A MongoDB cursor is able to fetch the matching documents in chunks. This is realized by the *getMore* command of MongoDB. If the set of matching documents exceeds the maximum chunk size, the set will be split up. Each *getMore* has to be handled as an own query event for the analysis, because they can observere different database states, even though they relate to the same cursor. An alternative is to set the maximum chunk size to a value that is higher than the collection size. This prevents the server from splitting up a query at all.

4.2 Transformation of Driver Operations

This section introduces all transformations of driver operations that need to be performed in order to realize the introduced instrumentation. The described transformations relate to the Node.js MongoDB driver, but can be applied to other drivers easily, because the introduced operations are defined by MongoDB. The arguments may slightly differ for other drivers, though.

We made the following simplifications to keep the transformations clean and more general:

- The operations are assumed to be synchronous. This simplification keeps all the callback handling out of the transformations, which does not add any information to the concept to be shown.
- $freshUid()$ creates a new UID, but in case of deletes the created UID will not be recreated when the delete delegates to an update. Therefore the UID and the delete property have the same value.
- The *find* operation of MongoDB actually returns a cursor that can then be used to fetch documents from the database. We assume here that it directly returns a set of documents to simplify the transformation specifications.

4.2.1 updateMany

updateMany updates all documents that match the given selector.

Updates that possibly modify more than one document are called multi-updates. Since multi-updates are not executed atomically, it is unsound to keep them as a single update event in our graph. To cope with that fact and to further increase the precision of our commutativity specification, we perform these steps in our instrumentation and only send single-updates to the server. The exact impact on our commutativity specifications is described in Section 5.1.6.

As explained in Section 3.3.2, the MongoDB server transforms all events into their idempotent equivalents. Therefore, multi-updates are split up into single updates whose selector only contains the object-ID. Upserts are transformed into inserts or updates, depending on the documents matching the selector. Since all updates go to the primary replica node, the splitting process is also performed on the primary.

Replacement Documents: A modifier is a replacement document if it only contains key-value-pairs and no update-operators.

As described, multi-updates without a replacement document are split up into n single updates, where n is the number of documents that match the selector. The MongoDB documentation states that multi-updates with a replacement document are converted into one single update that only updates the first document matching the selector.

As an example, consider the following modifiers:

1. {"_id": 123, "name": "John"}
2. {"name": "John", "age": 32}
3. {"$set": {"age": 33}, "name": "John"}

While modifiers 1 and 2 are replacement documents, modifier 3 is not, because it contains the update-operator $set.

We split up our multi-update *updateMany* by first executing a *find* operation. Let $uid_1, uid_2, ..., uid_n$ be the UIDs of the documents returned by $find(selector)$. If the modifier is a replacement document, all uid_x with $x > 1$ are simply omitted. There are now the following cases:

Case 1: No documents found ($n = 0$)

Case 1.1: *updateMany* is upsert

$$updateMany(selector,\ modifier) \\ \quad \rightsquigarrow insertOne(createInsertDocument(selector, modifier))$$

Case 1.2: *updateMany* is no upsert

$$updateMany(selector,\ modifier) \rightsquigarrow \bot$$

$\bot$ indicates that *updateMany* is omitted.

Case 2: Documents found ($n > 0$)

$$\begin{aligned} &updateMany(selector,\ modifier) \\ &\rightsquigarrow \\ &updateOne(\{''__\text{uid}'' : uid_1\},\ modifier) \\ &updateOne(\{''__\text{uid}'' : uid_2\},\ modifier) \\ &\vdots \\ &updateOne(\{''__\text{uid}'' : uid_n\},\ modifier) \end{aligned}$$

$createInsertDocument(selector, modifier)$ creates an insert document based on the selector and the modifier as described in the MongoDB documentation [3]:

Case 1: Modifier is a replacement document $\Rightarrow$ Return replacement document.

Case 2: Modifier is not a replacement document

1. Create a base document from the equality clauses of the selector, ignoring comparison operators

2. Apply the update expressions from the modifier

4.2.2 updateOne

updateOne is used for single-updates. We delegate it to *update* because it is more flexible as it can take a replacement document as well as a normal modifier, while *updateOne* can not take replacement documents.

$$\begin{aligned} &updateOne(selector,\ modifier) \\ &\quad \rightsquigarrow update(selector,\ modifier) \end{aligned}$$

4.2.3 update

update can be used for single updates or multi updates.

We need to delegate multi-updates to *updateMany*:

Case 1: Multi-update:

$$update(selector,\ modifier) \\ \rightsquigarrow updateMany(selector,\ modifier)$$

Case 2: Single-update: We need to check if the selector matches that of an ID-query, as described in Section 4.2.10:

Case 2.1: Selector only contains object-ID:

$$update(selector,\ modifier) \\ \rightsquigarrow update(selector,\ modifier \cup \{"__uid" : freshUid()\})$$

Case 2.2: Selector does not only contain object-ID:
Execute $findOne(selector)$ with primary read preference.

Case 2.2.1: $findOne(selector)$ found document with UID *uid*

$$updateOne(selector,\ modifier) \\ \rightsquigarrow update(\{"_id" : uid\},\ modifier)$$

Case 2.2.2: $findOne(selector)$ found no document

$$update(selector,\ modifier) \\ \rightsquigarrow insert(createInsertDocument(selector,\ modifier))$$

4.2.4 insertMany

insertMany inserts a set of documents. Multi-inserts are split up and sent as single-inserts to the server for the same reason as multi-updates.

Let $d_1, d_2, ..., d_n$ be the set of documents to be inserted. Then we apply the following transformation:

$$
\begin{aligned}
& insertMany(docs) \\
& \quad \rightsquigarrow \\
& \qquad inserteOne(d_1) \\
& \qquad inserteOne(d_2) \\
& \qquad \vdots \\
& \qquad inserteOne(d_n)
\end{aligned}
$$

4.2.5 insertOne

insertOne takes a single document and performs a single-insert. We need to add the UID to the document.

$$insertOne(doc) \rightsquigarrow insertOne(doc \cup \{"__uid" : freshUid()\})$$

4.2.6 deleteOne

deleteOne deletes one document matching the specified selector. Figure 4.1 gives an example of how we transform a delete event that leads to a wrong visibility relation and therefore to a wrong DSG into an update that resolves the issue. The upper DSG shows that the delete event removes the previously inserted document from the collection, such that the following query does not retrieve the document. This causes an incorrect visibility relation. The lower part shows how the issue is resolved by transforming the delete into an update that sets a delete property as we propose now.

Delete events are problematic for our construction of the visibility relation, because a deleted document is never retrieved by any query. This causes the delete event to be invisible for the query event. Therefore its UID is never taken into account when calculating the maximum UID. To overcome this obstacle we avoid deleting documents at all and mark them as deleted instead. This happens by attaching a property to the document to be deleted, which marks it as deleted and also stores the UID of the corresponding delete event. While the existence of the delete property on a document signals that the document has been deleted, its value describes when in the arbitration order that deletion occurred.

Therefore, delete events are transformed into update events that set the delete property as follows:

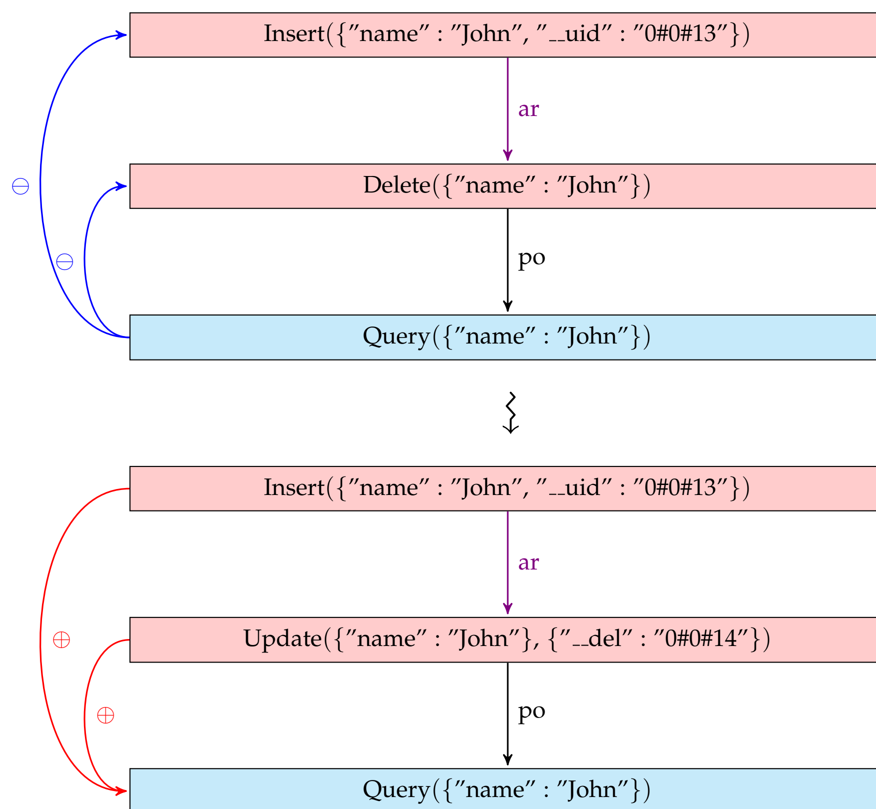


Figure 4.1: Example of a delete event causing a cycle in the DSG, which does not represent a real serializability violation. The lower DSG shows how the modified delete event resolves the cycle.

$$\begin{aligned}&\mathit{deleteOne}(\mathit{selector})\\&\qquad\qquad \rightsquigarrow \mathit{updateOne}(\mathit{selector},\ \{\$\mathit{set} : \{\text{"__del"} : \mathit{freshUid}()\}\})\end{aligned}$$

The proposed method demands a mechanism in the instrumentation that filters out all deleted documents that a query receives. It is not sufficient to just add that filter to the selector of the query, because we need to know which deleted documents the query can see in order to construct our visibility relation.

4.2.7 deleteMany

deleteMany is used for multi-deletes. Since we are transforming deletes into updates, we also have to transform multi-deletes into multi-updates. We achieve that by delegating to *updateMany*.

$$deleteMany(selector) \\ \quad \rightsquigarrow updateMany(selector,\ \{\$set : \{"__del" : freshUid()\}\})$$

4.2.8 findOneAndUpdate

This operation finds one document matching the selector, performs an update on it and returns the original document in the state before the update. There exists an option to return the updated document instead. We delegate this operation to a slightly modified version of our *updateOne* operation. This modified version, which we call *updateOne'*, differs from *updateOne* in terms of the returned document. While the usual *updateOne* operation only returs a status object, *updateOne'* returns the original object or the updated object, based on the given options.

$$findOneAndUpdate(selector,\ update) \\ \quad \rightsquigarrow updateOne'(selector,\ update)$$

4.2.9 findOneAndDelete

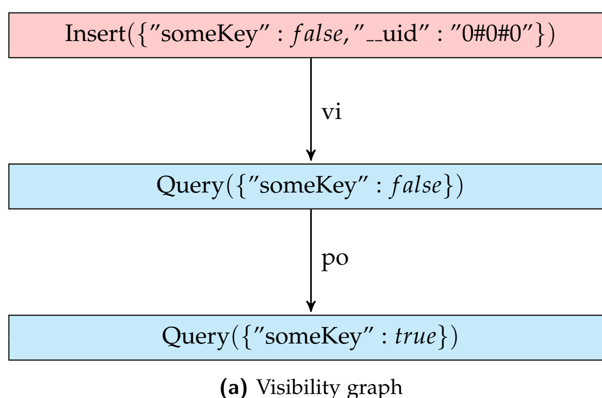
This operation finds one document matching the given selector, deletes it and returns the deleted document. Since we convert delete events to update events we can delegate this operation to *findOneAndUpdate* and add the delete property to the update modifier.

$$findOneAndDelete(selector) \\ \quad \rightsquigarrow findOneAndUpdate(selector,\ \{\$set : \{"__del" : freshUid()\}\})$$

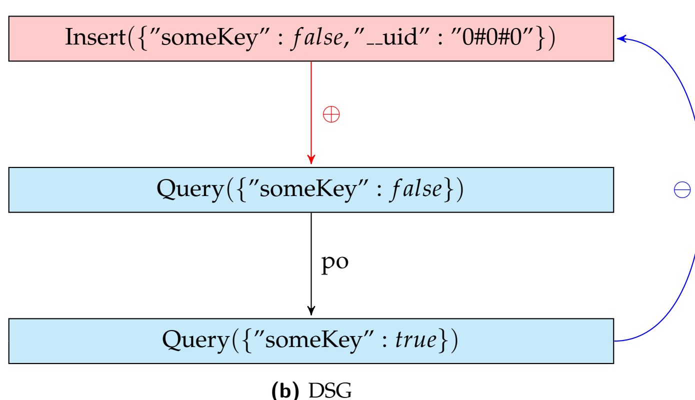
4.2.10 find

To transform the find operation, we need to handle ID-queries properly. By ID-queries we describe queries that only keep the object-ID in their selector. An example selector leading to an ID-query is $\{"_id" : 123\}$, because it only contains the object-ID. $\{\$or : [\{"_id" : 123\}, \{"size" : "XL"\}]\}$ and $\{"_id" : 123, "size" : "XL"\}$ on the other hand are examples for selectors of non-ID-queries.

Figure 4.2 illustrates the problem of non-ID-queries in regard to visibility with an example. In the visibility graph we observe that the second query cannot see the insert, because the corresponding document does not match the query selector and is therefore not amongst the retrieved documents. This leads to a cycle in the DSG. However, that cycle is not a real serializability violation, because by definition of our visibility relation the insert should be visible to both queries.



(a) Visibility graph



(b) DSG

Figure 4.2: Example of a cycle in the DSG caused by a wrong visibility relation that is in turn caused by a non-ID-query.

ID-queries are not problematic because the only possible situation in which they do not retrieve an existing document with the corresponding object-ID is a real serializability violation. That is because their selectors only contain the object-ID, which in turn uniquely identifies a single document. Furthermore the object-ID is immutable in MongoDB.

To handle non-ID-queries and keep our visibility relation correct, we need to adjust the selector of that query to fetch the whole collection. Therefore we need to distinguish:

Case 1: Non-ID-query

$$find(selector) \rightsquigarrow filter(find(\{\}), selector)$$

Case 2: ID-query

$$find(selector) \rightsquigarrow find(selector)$$

filter(documents, selector) filters a set of documents with the given selector. This makes sure that all those documents get filtered out that do not match the initial selector and ensures that the instrumented application does not change its behavior based on a different set of documents returned by a query.

4.2.11 findOne

findOne returns the first document matching the query. For our visibility relation it is not enough to simply return one document. We need to transform the *findOne* operation into a *find* operation and only return the first document to the client application.

$$findOne(selector) \rightsquigarrow first(find(selector))$$

first(documents) returns the first document of an ordered set of documents.

4.2.12 count

This operation counts the number of documents that match the selector. To get a correct visibility relation we need to transform it into a *find* operation and then count the number of retrieved documents.

$$count(selector) \rightsquigarrow |find(selector)|$$

4.2.13 Index operations

Indexes on document attributes are problematic in combination with our approach of transforming deletes into updates. If a unique index is attached to an attribute, there can only exist one document with the same value in the database. But with our deletion attribute we actually keep the document in the database. That leads to an error if a new document with the same value for the indexed attribute is added to the database and also if an update event changes the value of an existing document to the same value of the deleted document. Figure 4.3a gives an example of such an indexing error. The *name* attribute is registered as an index. We can see that the second insert leads to a database error, because there already exists an entry with *name* equal to *John*.

To overcome this obstacle we use compound indexes, which are composed of more than one attribute. We transform every index into a new compound index by adding our delete attribute to it. It is no problem if a document does not have the delete attribute. This is equivalent to the delete attribute

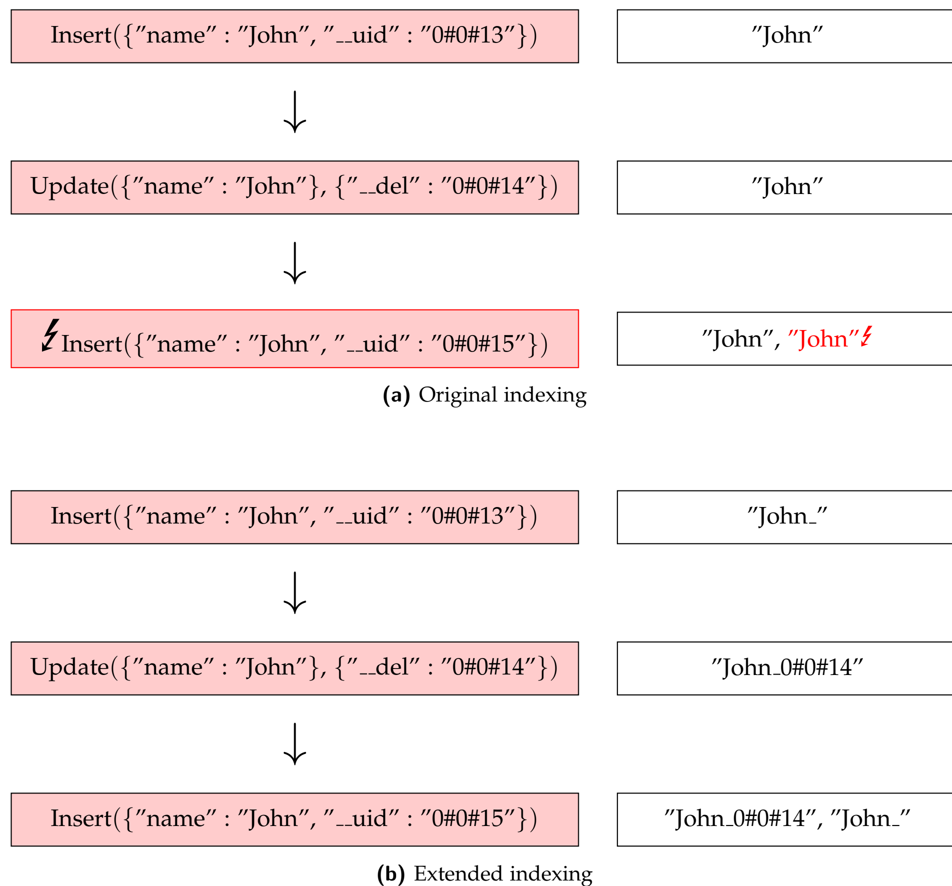


Figure 4.3: Example of a sequence of events and the corresponding indexing state. An index on the *name* attribute is assumed. Events are shown on the left and current index entries on the right. The index prohibits duplicate values.

having the special value *undefined*. There can only be one document in the collection with the same value, *undefined* included. This produces the same behavior as the initial index with real deletes. It is also the main reason why we need to store the UID (or another unique identifier) as the value of the delete attribute. If the delete attribute would only contain a boolean value, the indexing would not work as described, because there can only exist one deleted document with the same value of the indexed attribute.

Figure 4.3b demonstrates the proposed method with an example. Here, the first insert creates an index entry with value *John_*, because the delete attribute *__del* is undefined. The underscore is used to illustrate a compound index. The update event, which is actually a delete, sets the delete attribute with the UID as its value. So the index value changes to *John_0#0#14*. The next insert can now add the index value *John_* without creating a conflict.

createIndex

createIndex creates one new index on the collection. The parameter *fieldOrSpec* contains all fields of that index. The corresponding value can be 1 or -1 and determines the order of the index (ascending or descending). We simply need to add the delete property to the index fields. It does not matter whether the index ordering is ascending or descending because the delete property only needs to make the index unique:

$$createIndex(fieldOrSpec) \rightsquigarrow createIndex(fieldOrSpec \cup \{"__del" : 1\})$$

createIndexes

This operation behaves like *createIndex* but can create more than one index. Let $s_1, s_2, ..., s_n$ be the given index specifications.

$$createIndexes(indexSpecs) \rightsquigarrow \begin{array}{l} createIndex(s_1) \\ createIndex(s_2) \\ \vdots \\ createIndex(s_n) \end{array}$$

dropIndex

dropIndex drops the given index. We need to add the delete property here, too. It is concatenated to the *indexName*. Different components of a compound index are separated with an underscore.

$$dropIndex(indexName) \rightsquigarrow dropIndex(indexName + "___del_1")$$

4.3 Performance

There are several reasons why an instrumented client has a much slower performance than a non-instrumented one. One reason is the fetching of the whole collection for all non-ID-queries. Not only does this create a higher workload for the database itself. It also increases the network traffic immensely. Additionally, the memory consumption on the client grows to the size of the whole collection for each query. Furthermore, the client has to iterate through all elements of the collection and apply a filter. It cannot take advantage of indexing on the database server.

Another performance aspect is the transformation of deletes into updates. This can possibly lead to a much larger size of the database. This larger size further increases the effects of the performance influence of non-ID-queries as described before.

Since multi-updates, multi-inserts and multi-deletes are split up into single updates/inserts/deletes, we have a lower performance based on the Round-Trip-Delay (RTD) that is now added to every insert/update/delete event.

Transforming limited queries into unlimited ones increases the load on the database as well as the network traffic. But this effect is already considered in the effect of non-ID-queries, because limiting an ID-query has no effect and for all non-ID-queries the whole collection is fetched anyway.

More formal, the overhead of the instrumentation can be described asymptotically as follows. Let d be the number of delete events, i the number of insert events, q the number of non-ID-queries and s the maximum document size. Let further o_m be the number of multi-updates and multi-deletes, i_m the number of multi-inserts, u the number of non-ID-updates and l the highest number of single-events per multi-events. We assume an empty database at the beginning of the dynamic analysis run.

Space Overhead

The space overhead on the database is $\mathcal{O}(d * s)$, because every deleted element stays in the database and the object-ID field is immutable in MongoDB.

Network Traffic

The overhead for the network traffic is $\mathcal{O}((q + o_m + u) * i * s)$. This is composed by $\mathcal{O}(q * i * s)$ for the overhead of all non-ID-queries, $\mathcal{O}(o_m * i * s)$ for the overhead of all query operations triggered by the splitting of multi-updates and multi-deletes and $\mathcal{O}(u * i * s)$ for the additional query event of non-ID-updates.

Round-Trip-Delay (RTD)

We assume the RTD to be constant. Then we have an RTD overhead of $\mathcal{O}((o_m + i_m) * l)$. It describes the number of additional requests to the database that our instrumentation creates. Each multi-update or multi-insert creates $\mathcal{O}(l)$ additional requests.

4.4 Limitations

There are some limitations to the proposed instrumentation approach.

One limitation is based on our transformation of deletes into updates. In case that a document d with object-ID o has been deleted, it remains in the database having the delete property. If now an insert or an upsert with a specified object-ID that equals o is sent to the server, an error is triggered, because there cannot be two documents with the same object-ID in the database. We do not see this as a problematic issue, since it is generally very error prone to reuse object-IDs. Usually the object-ID is generated by the driver or by the server. In both cases the generated object-IDs will not be reused. Therefore this case is rarely to occur.

Another limitation is the influence of the instrumentation on the client application. Even though that influence is kept as small as possible there are edge cases that lead to different behavior. One case is that the documents of the client application already contain fields with the same name as the field for the UID or the field for the delete property. Therefore, a name should be chosen that is unlikely to occur in the document expected by the client application. However, these are negligible corner cases and do not pose a real limitation to the approach.

Chapter 5

Analysis

Chapter 4 described how to instrument a client application in order to record all database traces that are necessary in order to perform the analysis proposed by Brutschy et al. This chapter covers how we construct the necessary relations, explained in Section 2.1.2, and use the proposed algorithm to detect serializability violations. We define commutativity and absorption between the database events in Section 5.1, as they are needed by the algorithm. We extend the software ECRacer that was used for the experiments in [13] to also support MongoDB database traces. This extension is described in detail in Section 5.3.

5.1 Commutativity Specifications

MongoDB 3.4 has 22 different update operators [4]. Because commutativity and absorption have to be defined for every combination of two operators, the number of necessary specifications grows quadratically with the number of operators. Therefore, we only support a subset of all update operators. Supporting all operators means to define $22^2 = 484$ different absorption specifications and $22^2/2 = 242$ different commutativity specifications.

We performed a static text search on the source code of our open source projects, which are listed in Section 8.3, to get an impression of the distribution of the different operators in practice. Figure 5.1 shows the results of this text search. As one can see, $set occurs 63 times in the source code files. All other operators only occur 0 to 10 times. This is also reflected by the number of projects that use the different operators. Some projects do not use any operators at all and work either completely with replacement documents, explained in Section 4.2.1, or delegate the operator logic to external libraries. Based on this data we support the operators $set, $inc, $mul, $rename, $unset, $addToSet, $pull and $push. The $each operator is not included in Tables 5.1 and 5.2, because it is part of the supported $push

operator. We support \$mul due to its similar logic to the supported \$inc operator. It is an easy step from specifying one of these operators to specifying the other one. Therefore we support $\sim$ 36% of all MongoDB update operators.

Note that the static text search allows no direct statement about executing the operators at runtime.

Within this section we simply tread delete events as update events, because they were transformed, as discussed in Section 4.2.6.

5.1.1 Basic definitions

We give some basic definitions that can be used to define all of the following commutativity and absorption specifications.

- U: Set of all possible MongoDB update events.
- M: Set of all possible MongoDB modifiers.
- S: Set of all possible MongoDB selectors.
- O: Set of all possible MongoDB update operators.
- A: Set of all possible MongoDB attribute keys.
- V: Set of all possible MongoDB attribute values.
- sel : $U \longrightarrow S$ - Gives the selector of a given update event.
- mod : $U \longrightarrow M$ - Gives the modifier of a given update.
- ops : $U \longrightarrow \mathcal{P}(O)$ - Gives the set of update operators used in the given update.
- $\text{keys}_o : M \longrightarrow \mathcal{P}(A)$ - Gives all attribute keys of operator $o \in O$ of a given modifier.
- keys : $S \longrightarrow \mathcal{P}(A)$ - Gives all attribute keys of any query operator of a selector.
- $\text{vals}_o : M \longrightarrow \mathcal{P}(V)$ - Gives all attribute values of the operator $o \in O$ of a given modifier.
- $\text{val}_{o,m} : A \longrightarrow V$ - Gives the attribute value for the given attribute key of modifier m and operator o.
- $\text{com}_{(o_1,o_2)} \subseteq U \times U$ - Commutativity relation where $o_1, o_2 \in O$. $u_1 \ \text{com}_{o_1,o_2} \ u_2$ iff u_1 and u_2 are commutative considering only the operator o_1 for u_1 and o_2 for u_2.

Example 5.1.1 Assume we have an update that has selector s and modifer m with $s :=$

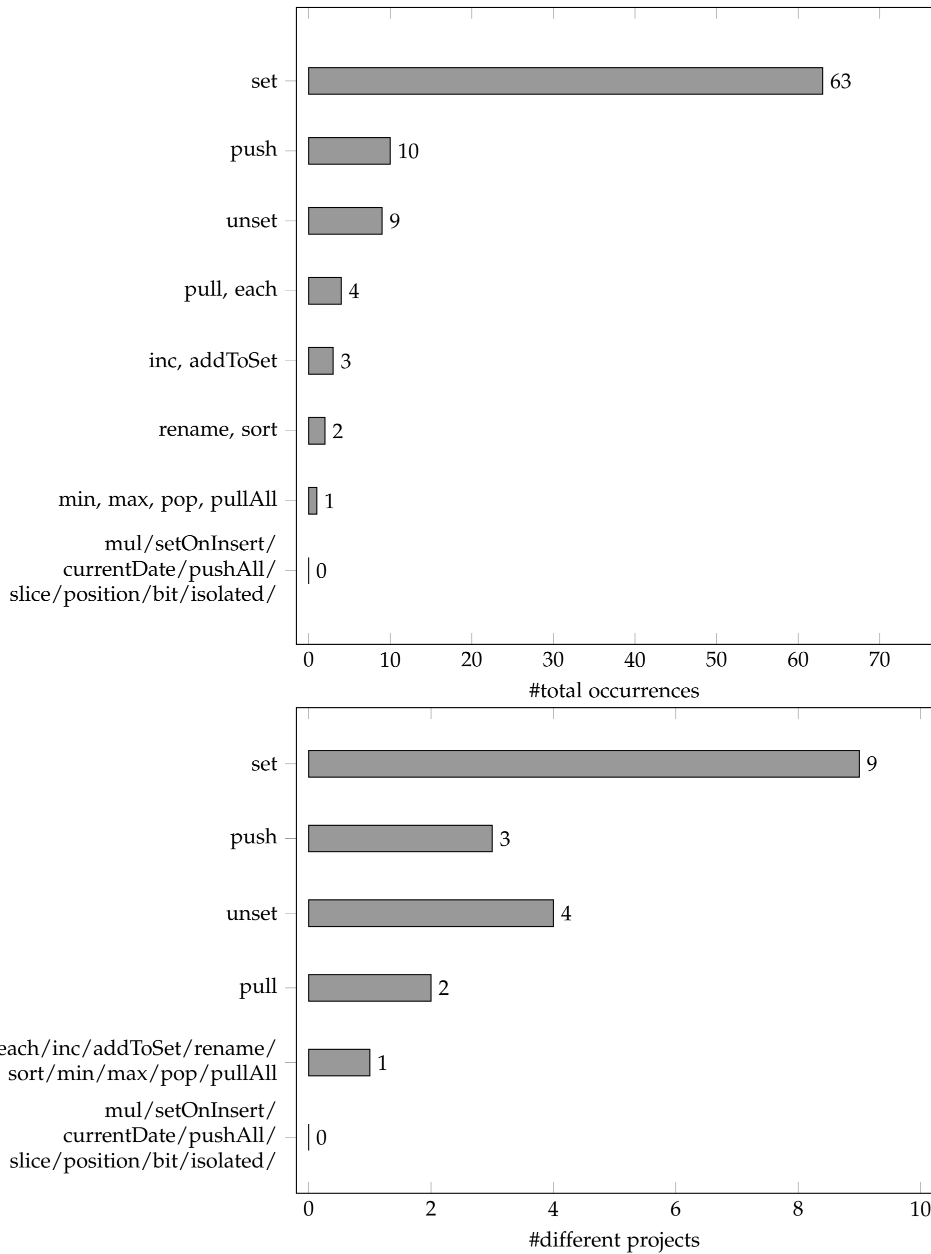


Figure 5.1: Results of the static text search in the source code of 17 open source projects, listed in Section 8.3. The upper graph shows the total occurrence in the source files of all projects. The lower graph shows the number of different projects that contained the corresponding update operator. The results do not contain source files which are part of the external libraries of the projects, because these libraries use a lot of operators that are not used by the projects.

```
{"$or":
  [
    {"name": "John"},
    {"age": {"$gt" : 30}}
  ]
}
```

and $m :=$

```
{
  "$set": {"age": 99, "last_name": "Doe"},
  "$unset": {"address": ""}
}
```

Then we have

- $\text{keys}_{\$set}(m) = \{\text{"age"}, \text{"last_name"}\}$
- $\text{keys}(s) = \{\text{"name"}, \text{"age"}\}$
- $\text{vals}_{\$set}(m) = \{99, \text{"Doe"}\}$
- $\text{val}_{\$set,m}(\text{"age"}) = 99$ □

5.1.2 Insert and Query

In insert is commutative to a query if and only if the selector of the query matches the document to be inserted.

We cannot apply this strategy to updates because an update event does not contain the complete updated document.

5.1.3 Insert and Insert

The commutativity of two insert events i_1 and i_2 is best described by the following case distinction:

Case 1: i_1 or i_2 does not contain an object-ID $\Rightarrow$ commutative

Case 2: i_1 and i_2 both contain an object-ID

> **Case 2.1:** Object-IDs equal $\Rightarrow$ non-commutative
>
> **Case 2.1:** Object-IDs unequal $\Rightarrow$ commutative

5.1.4 Query and Query

Two queries are always commutative, because they only read data without any side effects.

Query	Selector:	`{"name":"John","age":{"$gt" : 30}}`
Update u_1	Selector:	`{ "name":"John" }`
	Modifier:	`{"$set":{"address":"New Street 5"}}`
Update u_2	Selector:	`{"name":"John","age":{"$gt" : 40}}`
	Modifier:	`{"$set":{"name":"Bob"}}`

(a) Events

_id	name	age	address
1	"John"	10	"Old Drive 10"
2	"John"	35	"Old Drive 10"
3	"John"	17	"Old Drive 10"
4	"John"	45	"Old Drive 10"
5	"John"	31	"Old Drive 10"

$\longrightarrow_{u_1}$

_id	name	age	address
1	"John"	10	"New Street 5"
2	"John"	35	"New Street 5"
3	"John"	17	"New Street 5"
4	"John"	45	"New Street 5"
5	"John"	31	"New Street 5"

(b) Type 1: u_1 changes the content of the documents fetched by the query

_id	name	age	address
1	"John"	10	"Old Drive 10"
2	"John"	35	"Old Drive 10"
3	"John"	17	"Old Drive 10"
4	"John"	45	"Old Drive 10"
5	"John"	31	"Old Drive 10"

$\longrightarrow_{u_2}$

_id	name	age	address
1	"John"	10	"Old Drive 10"
2	"John"	35	"Old Drive 10"
3	"John"	17	"Old Drive 10"
4	"Bob"	45	"Old Drive 10"
5	"John"	31	"Old Drive 10"

(c) Type 2: u_2 changes the set of documents retrieved by the query

Figure 5.2: Demonstration of the two types of ways that an update can change a query's result. Documents in the database are represented as tables here. Blue rows indicate membership to the query's result set. Red cells indicate changes by the corresponding update. The left tables show the result of the query without the former update. The right tables assume that the update has been executed before.

5.1.5 Insert and Update

An insert is commutative with an update if the document to be inserted does not match the selector of the update. It is a similar specification to that of the commutativity between inserts and queries.

5.1.6 Query and Update

There are two types of ways an update can change the results of a query.

Type 1: The update modifies the content of the documents retrieved by the query.

Type 2: The update modifies the set of documents retrieved by the query.

Figure 5.2 demonstrates both types. The execution of u_1 changes the *address* fields of all documents. The query will still return the same set of documents, but they will have different *address* values. u_2, on the other hand, changes

the set of documents returned by our query. This happens by changing a value that appears in the query's selector. Therefore, the document will no longer match that selector and will not be contained in the result set. So u_1 is an example for Type 1, while u_2 is one for Type 2.

The absence of Type 1 can be checked using the projection of the query. Let u be our update, q our query and P_q the set of projected fields of q. Then we have no changes of Type 1 if $\forall o \in O : \text{keys}_o(\text{mod}(u)) \cap P_q = \emptyset$. More intuitively, this means that none if the projected fields must be within the keys of the modifier of u.

To check for changes of Type 2 we need to use object-IDs. Because we transformed all updates into ID-updates, we have the object-IDs of every update. We also have the query's result set. Let $\text{oid}(u)$ be the object-ID of u and $\text{oids}(q)$ the set of object-IDs from the result set of q. u is free of Type 2 changes to the result set of q if one of the following conditions holds:

1. q and u are executed on different collections
2. q and u both keep object-IDs in their selectors and the values of these object-IDs are not equal
3. $\forall o \in O : \text{keys}_o(\text{mod}(u)) \cap \text{keys}(\text{sel}(q)) = \emptyset$

Condition 1 is trivial, while condition 2 covers the case that both selectors uniquely identify different documents. Condition 3 ensures that all document fields that are possibly changed by the modifier of u are not part of the query's selector.

We can neither have changes of Type 1 nor Type 2 if the set of documents matching the update selector is disjoint from the set of documents matching the query selector. Because this is depending on the database state during the execution, we cannot directly access that information. However, we can approximate this using the following rule: The documents matching the selector of u are disjoint from those matching q's selector if there is no Type 2 change and $\text{uid}(u) \notin \text{oids}(q)$.

This leads to the following strategy when checking for commutativity:

Case 1: Type 2 change found $\Rightarrow$ not commutative

Case 2: No Type 2 change found

Case 2.1: $\text{uid}(u) \notin \text{oids}(q) \Rightarrow$ commutative

Case 2.2: $\text{uid}(u) \in \text{oids}(q)$

Case 2.2.1: Type 1 change found $\Rightarrow$ not commutative
Case 2.2.2: No Type 1 change found $\Rightarrow$ commutative

Table 5.1: Commutativity specifications. Commutativity is symmetric and so is this table. Therefore, duplicate definitions have been omitted and replaced by –. * Events (o_1 and o_2) need to be swapped on these properties. ** These two events are not commutative because $push guarantees a specific ordering of the array elements while $addToSet does not. Equal definitions are marked with the same shade of gray as a background color.

	$set	$inc	$mul	$rename	$unset	$addToSet	$pull	$push
$set	P.1 ∧ (P.2 ∨ P.3)	P.1 ∧ (P.2 ∨ P.6)	P.1 ∧ (P.2 ∨ P.7)	P.8 ∧ P.9	P.1 ∧ P.2	P.1 ∧ P.2	P.1 ∧ P.2	P.1 ∧ P.2
$inc	–	P.1	P.1 ∧ (P.2 ∨ P.6 ∨ P.7)	P.8 ∧ P.9	P.1 ∧ P.2	P.1 ∧ P.2	P.1 ∧ P.2	P.1 ∧ P.2
$mul	–	–	P.1	P.8 ∧ P.9	P.1 ∧ P.2	P.1 ∧ P.2	P.1 ∧ P.2	P.1 ∧ P.2
$rename	–	–	–	P.3 ∧ (P.4 ∨ P.5)	P.8* ∧ P.9*	P.8* ∧ P.9*	P.8* ∧ P.9*	P.8* ∧ P.9*
$unset	–	–	–	–	P.1	P.1 ∧ P.2	P.1 ∧ P.2	P.1 ∧ P.2
$addToSet	–	–	–	–	–	P.1	P.1 ∧ (P.2 ∨ P.10)	(P.1 ∧ P.2)**
$pull	–	–	–	–	–	–	P.1	P.1 ∧ P.2
$push	–	–	–	–	–	–	–	P.1 ∧ P.2

5.1.7 Update and Update

To decide whether an update u_1 is commutative with another update u_2 we have to check commutativity for every update operator in u_1 and every update operator in u_2. As already mentioned, we only support a subset of update operators, which are listed in table 5.1. The specifications are separated into different properties, which are then used to define the commutativity of the different operator pairs.

Therefore, u_1 and u_2 are commutative if one of the following condition holds:

1. u_1 and u_2 are executed on different collections
2. u_1 and u_2 do both keep object-IDs in their selectors and the values of these object-IDs are not equal
3. $\forall(o_1, o_2) \in (\text{ops}(u_1) \times \text{ops}(u_2)) : u_1 \text{ com}_{(o_1,o_2)}\ u_2$

Commutativity Properties

The commutativity specifications used in Table 5.1 can be broken down into certain properties that are used to define commutativity between update events. As a convention we will write s_i for $\text{sel}(u_i)$ and m_i for $\text{mod}(u_i)$ to keep the definitions more concise.

P.1 $(\text{keys}(s_1) \cup \text{keys}(s_2)) \cap (\text{keys}_{o_1}(m_1) \cup \text{keys}_{o_2}(m_2)) = \emptyset$

P.1 makes sure that the attribute keys used in either of both updates are distinct from any attribute keys used in the queries. This prevents the updates from changing parts of the documents that are essential for the query of the other update to match on that document.

P.2 $\text{keys}_{o_1}(m_1) \cap \text{keys}_{o_2}(m_2) = \emptyset$

P.2 assures that the attribute keys of the updates are distinct.

P.3 $\forall k \in (\text{keys}_{o_1}(u_1) \cap \text{keys}_{o_2}(m_2)) : \text{val}_{o_1,m_1}(k) = \text{val}_{o_2,m_2}(k)$

P.3 covers the case that the attribute keys of both updates are not distinct but the intersecting ones share the same value.

P.4 $(\text{keys}(s_1) \cup \text{keys}(s_2)) \cap (\text{keys}_{\$rename}(m_1) \cup \text{keys}_{\$rename}(m_2) \cup \text{vals}_{\$rename}(m_1) \cup \text{vals}_{\$rename}(m_2)) = \emptyset$

P.4 is only used with the $rename operator and is much like condition P.1 but also takes the attribute values into account, because the operator renames attribute keys.

P.5 $(\text{keys}_{\$rename}(m_1) \cup \text{vals}_{\$rename}(m_1)) \cap (\text{keys}_{\$rename}(m_2) \cup \text{vals}_{\$rename}(m_2)) = \emptyset$

P.5 is also only used with the $rename operator and differs from P.2 by including attribute values.

P.6 $\forall a \in \text{keys}_{\$inc}(m_2) : \text{val}_{\$inc,m_2}(a) = 0$

P.6 expresses that all attribute keys used in the $inc operator have the corresponding attribute value 0, which is the neutral element of $\mathbb{R}$ with respect to addition.

P.7 $\forall a \in \text{keys}_{\$mul}(m_2) : \text{val}_{\$mul,m_2}(a) = 1)$

P.7 states that all attribute keys used in the $mul operator have the attribute value 1. Since this is the neutral Element of $\mathbb{R}$ with respect to multiplication, it does not affect the corresponding value in the document.

P.8 $(\text{keys}(s_1) \cup \text{keys}(s_2)) \cap (\text{keys}_{o_1}(m_1) \cup \text{keys}_{\$rename}(m_2) \cup \text{vals}_{\$rename}(m_2)) = \emptyset$

P.8 behaves like P.4 but is only applicable if exactly one of the operators is the $rename operator.

P.9 $\text{keys}_{o_1}(m_1) \cap (\text{keys}_{o_2}(m_2) \cup \text{vals}_{o_2}(m_2)) = \emptyset$

P.9 behaves like P.5 but is only applicable if exactly one of the operators is the $rename operator.

P.10 Let $C \subseteq V$ be set of values that are possible conditions of the $pull operator and sat : $C \longrightarrow \mathcal{P}(V)$ be a function that gives all values satisfying the given condition. Then P.10 is the following property: $\forall a \in (\text{keys}_{\$addToSet}(m_1) \cap \text{keys}_{\$pull}(m_2)) : \text{val}_{u_1}(a) \notin \text{sat}(\text{val}_{m_2}(a))$

P.10 covers the case that all values of the mutual attributes of m_1 and m_2 in m_1 do not satisfy the condition for the corresponding attribute in m_2. In other words does this express that the values added by $addToSet should not be covered by the condition for removal of $pull. Otherwise the values added by u_1 are being removed by u_2 and this leads to non-commutativity between u_1 and u_2.

5.2 Absorption Specifications

We give a definition of the absorption relation in Section 2.1.1. Please note that we only consider right-absorption here, as it is also needed by the algorithm proposed by Brutschy et al. [13]. With the same argumentation as in Section 5.1, we specify absorption only on a subset of update operators. Trivially, two queries cannot absorb each other. The same holds for two insert events. Inserts and updates can also not absorb each other in any way. It remains to specify under which circumstances a delete absorbs an insert or an update absorbs another update.

5.2.1 Basic definitions

Additionally to the definitions of Section 5.1, let $\text{abs}_{(o_1,o_2)} \subseteq U \times U$ be the absorption relation where $o_1, o_2 \in O$. u_1 abs_{o_1,o_2} u_2 iff u_2 absorbs u_1 considering only the operator o_1 for u_1 and o_2 for u_2.

5.2.2 Insert and Delete

To decide if a delete event absorbs an insert we simply need to check if they operate on the same collection and the selector of the delete matches the document to be inserted. We use the same mechanism as in Section 5.1.5.

5.2.3 Update/Delete and Delete

A delete event absorbs an update/delete if they operate on the same collection and the selector of the potentially absorbing delete matches the selector

Table 5.2: Absorption specifications. × indicates no absorption. * $addToSet does not absorb $pull because $pull is order-preserving and $addToSet is not. ** $pull does not absorb $push because it is not guaranteed that $push adds the element at the same position as $pull removed it. Equal definitions are marked with the same shade of gray as a background color.

	$set	$inc	$mul	$rename	$unset	$addToSet	$pull	$push
$set	AP.1, AP.2, AP.3	×	AP.1, AP.2, AP.3, AP.7	×	AP.1, AP.2, AP.3	×	×	×
$inc	AP.1, AP.2, AP.3	AP.1, AP.2, AP.3, AP.4	AP.1, AP.2, AP.3, AP.8	×	AP.1, AP.2, AP.3	×	×	×
$mul	AP.1, AP.2, AP.3	AP.1, AP.2, AP.3, AP.9	AP.1, AP.2, AP.3, AP.5	×	AP.1, AP.2, AP.3	×	×	×
$rename	×	×	×	×	×	×	×	×
$unset	AP.1, AP.2, AP.3	AP.1, AP.2, AP.3	AP.1, AP.2, AP.3	×	AP.1, AP.2, AP.3	AP.1, AP.2, AP.3	×	AP.1, AP.2, AP.3
$addToSet	AP.1, AP.2, AP.3	×	×	×	AP.1, AP.2, AP.3	AP.1, AP.2, AP.3, AP.6	AP.1, AP.2, AP.3, AP.10	×
$pull	AP.1, AP.2, AP.3	×	×	×	AP.1, AP.2, AP.3	×*	AP.1, AP.2, AP.3, AP.6	×**
$push	AP.1, AP.2, AP.3	×	×	×	AP.1, AP.2, AP.3	×	AP.1, AP.2, AP.3, AP.10	×

of the update/delete to be absorbed. This ensures that both events operate on the same set of documents.

5.2.4 Update and Update

We specify absorption in our analysis in the way that update u_2 absorbs update u_1 if both updates need to operate on the same collection and $\forall(o_1, o_2) \in (\text{ops}(u_1) \times \text{ops}(u_2)) : u_1 \text{ abs}_{(o_1,o_2)}\ u_2$. The absorption is decided based on table 5.2, which gives the specifications for the absorption between any two update operators that are supported by our analysis.

Absorption Properties

The properties used to specify absorption between two updates are explained in this section.

AP.1 $s_1 = s_2$

AP.1 ensures that the queries of both updates equal. If they do not, it cannot be guaranteed that they both operate on the same set of documents and therefore absorption cannot be guaranteed either.

AP.2 $(\text{keys}_{o_1}(m_1) \cap \text{keys}(s_2)) = \emptyset$

AP.2 expresses that the attributes used in m_1 should not occur in s_2. Without this condition it cannot be guaranteed that u_1 changes the set of documents matching s_2.

AP.3 $\text{keys}_{o_1}(m_1) \subseteq \text{keys}_{o_2}(m_2)$

AP.3 states that m_2 only absorbs m_1 if all attributes used in m_1 in operator o_1 are also used in m_2 in operator o_2.

AP.4 $\forall k \in \text{keys}_{\$inc}(m_1) : \text{val}_{\$inc,m_1}(k) = 0$

AP.4 covers the case that all increments of u_1 are of value 0. So they have no effect besides creating the attribute if not existent and get absorbed by the increments done in u_2.

AP.5 $\forall k \in \text{keys}_{\$mul}(m_1) : \text{val}_{\$mul,m_1}(k) = 1$

AP.5 is similar to AP.4, but demands the values to be equal to 1 to have no effect on multiplication.

AP.6 $\forall k \in \text{keys}_{o_1}(m_1) : \text{val}_{o_1,m_1}(k) = \text{val}_{o_2,m_2}(k)$

AP.6 allows absorption only if for every attribute in m_1 the corresponding value equals the one in m_2 for that same attribute.

AP.7 $\forall a \in \text{keys}_{o_1}(m_1) : \text{val}_{\$mul,m_2}(a) = 0$

AP.7 expresses that a multiplication with 0 sets the value of the attribute to 0, independent of its former value.

AP.8 $\forall k \in \text{keys}_{\$inc}(m_1) : \text{val}_{\$inc,m_1}(k) = 0 \vee \text{val}_{\$mul,m_2}(k) = 0$

AP.8 covers the case that either the $inc operator does not change the value by adding 0 or the $mul operator absorbs any changes by multiplying with 0.

AP.9 $\forall k \in \text{keys}_{\$mul}(m_1) : \text{val}_{\$mul,m_1}(k) = 1$

AP.9 assures that the $mul operator does not change the value because it is multiplying by 1.

AP.10 $\forall a \in \text{keys}_{o_1}(m_1) : \text{val}_{m_1}(a) \in \text{sat}(\text{val}_{m_2}(a))$

AP.10 makes sure that each value of $addToSet has a corresponding constraint in $pull that satisfies that value.

5.3 Extension of ECRacer

The analysis of the recorded traces, as proposed by Brutschy et al. [13], has been accomplished by extending the software ECRacer. It has been developed by Brutschy et al. to perform their proposed analysis. This section covers the conceptional description of the analysis. The concrete implementation is explained in Section 7.2.1. ECRacer performs the proposed algorithm based on the relations that are introduced in Section 2.1.2. Therefore, we need to convert the traces from our instrumentation into the required relations and implement the commutativity and absorption specifications as extensions to ECRacer.

5.3.1 Relation Construction

This section covers the construction of the required relations from the database records created by the instrumented client.

Arbitration Order

In section 3.3.6 we explained that all write events go to the primary replica node of MongoDB. Therefore, there exists a total order of execution of all write events, which is our arbitration order. Each write event contains a time stamp from the server, stating the time of execution. That time stamp is given in milliseconds and also contains a counter covering the case that more than one event has been executed during the same millisecond. We simply use these server time stamps to create our arbitration order.

Program Order

As explained in Section 4.1.1, we need the program order to be a partial relation, but the proposed method is easily adaptable to fit a total program order, if needed. In Section 4.1.1 is also explained that we have a start time stamp and a return time stamp for every event and that for two events e_1 and e_2 by the same client we say that $(e_1, e_2) \in \text{po}$ iff $\text{endTime}(o_1) < \text{startTime}(o_2)$. It is sufficient for the analysis if we give the transitive reduction of the program order.

Algorithm 1 shows the construction of the partial program order's transitive reduction. The algorithm's output is the program order relation, initialized as an empty set in line 1. Intuitively, the algorithm runs over the sorted

events of a client (lines 4 to 15) and for each of these events it checks for all remaining events (lines 7 to 14) if they happen after it and includes the pair into the relation if this is the case. Lines 8 and 9, using the minimum return time stamp of the outer loop, ensure that no unnecessary program order is included, since we aim for the transitive reduction. Retrieving the maximum return time can be achieved while sorting without changing the asymptotic runtime complexity.

Let c be the number of clients and e the maximum number of events per client. Then the asymptotic runtime complexity of the algorithm is $\mathcal{O}(c * e^2)$. In most cases we have $c \ll e$, which lets the complexity collapse to $\mathcal{O}(e^2)$.

The asymptotic space complexity is $\mathcal{O}(e)$, because we need to keep all events in memory. Sorting does also not exceed this upper bound.

Visibility

How the visibility is determined for a given query and update is described in Section 4.1.3. In our extension of ECRacer we perform that check for every possible pair consisting of a query and an update/insert.

```
1:  po ← ∅
2:  for all client ← clients do
3:      sortedOps ← sortByStartTimeAsc(getOps(client))
4:      for all op ← sortedOps do
5:          minRetTime ← maxRetTime
6:          sortedOps.remove(op)
7:          for all op2 ← sortedOps do
8:              if minRetTime < op2.startTime then
9:                  break
10:             else if op2.startTime ≥ op.returnTime then
11:                 po.add((op, op2))
12:                 minRetTime ← min(minRetTime, op2.returnTime)
13:             end if
14:         end for
15:     end for
16: end for
```

Algorithm 1: Algorithm for the construction of the partial program order relation, given all database events including their start and return time.

5.4 Runtime Analysis

Brutschy et al. propose two algorithms for the analysis. One generic algorithm with a higher runtime complexity, and one optimized algorithm

with a lower runtime complexity. The optimized algorithm demands a *far-reaching* absorption, as defined in 2.1.1. If that far-reaching absorption is weaker than common absorption, the method will be less precise. Our absorption specifications are also far-reaching. Therefore, we use the optimized algorithm without a loss in precision.

Let u be the number of update/insert events, q the number of queries and $m = |\text{ar}|$ in our dynamic analysis run. The asymptotic runtime complexity of the optimized algorithm is then $\mathcal{O}(\max((u+q)m, (u+q)^2))$ [13].

The creation of the program order has a time complexity of $\mathcal{O}((u+q)^2)$, as shown in Section 5.3.1 where we used e as the number of all events. Because all events are either updates (including deletes) or queries, we have $e = u + q$. This does not change the asymptotic runtime complexity of the analysis.

The arbitration order is generated by iterating over all pairs of update events and decide if the one event is arbitrated before the other. It takes $\mathcal{O}(1)$ to decide whether an update/insert is arbitrated before another one. This leads to an asymptotic runtime complexity of $\mathcal{O}(u^2)$ and does therefore not change the complexity of the algorithm.

Creating the visibility relation has a more involved complexity. Visibility is created by checking every pair of updates/inserts and queries. For each of those pairs we need to calculate the $maxUID$ of the query and then check if it is arbitrated after the UID of the update/insert. Because in the worst case there can be the UIDs of all documents in the query's visible UIDs, we conclude a time complexity of $\mathcal{O}(q * i * u)$, where i is the number of insert events. This does also not increase the time complexity of ECRacer.

5.5 Pattern-Matching Extension

Because analysis runs have shown that serializability violations are hard to catch during dynamic analysis, we further extend ECRacer with a pattern-matching mechanism. First, we define classes of violation patterns that are possible to occur in the DSG under the consistency model of MongoDB. We then discover patterns that do not contain violations but can be transformed into a violation pattern by changing visibility between one update and query. That change in visibility must be valid in the MongoDB consistency model, such that we do not create unsound executions. We call these patterns *pre-violation patterns*. The intuition behind that process is to discover all locations in a specific dynamic analysis run that could also have lead to a violation. This tackles the issue that these violations occur only with a small probability and are therefore not likely to occur in a dynamic run.

After changing the visibility we need to discard all parts of the graph that follow the pre-violation pattern by arbitration or program order. This is

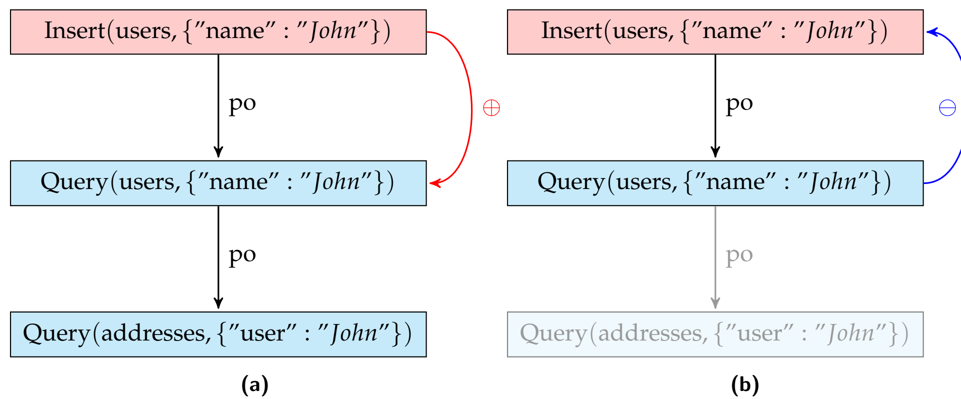


Figure 5.3: Demonstrates invalidity of parts of the DSG after changing visibility to match a violation pattern. Invalid parts of the DSG are marked with a lower opacity. (a) shows the original DSG and (b) the version with a manipulated visibility.

necessary, because it cannot be guaranteed that this part of the graph is still valid. The application could have behaved differently based on the changed visibility.

The following example demonstrates this issue.

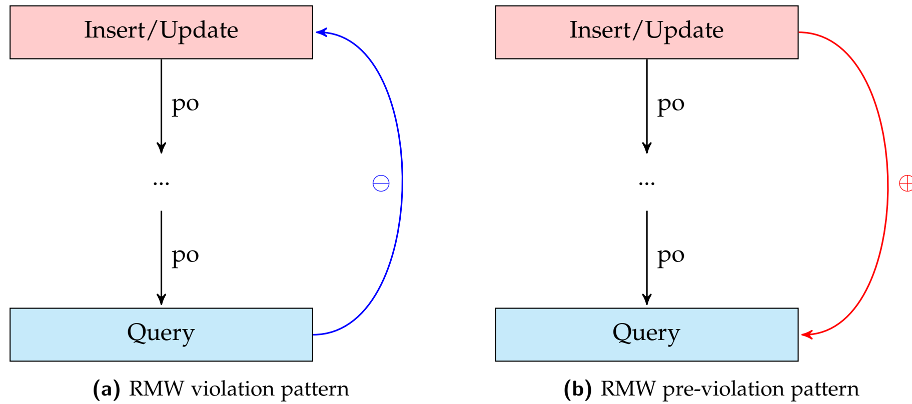
Example 5.5.1 Consider we have the following simplified piece of code, which does assume synchronous database operations for the sake of readability.

```
var userColl = db.collection("users");
userColl.insert({"name": "John", ...});
var user = userColl.findOne({"name": "John"});
if (user) {
  // perform code depending on existence of user "John"
  var address = db.collection("addresses").findOne({
    "user" : "John"
  });
  ...
}
```

The DSG produced by the previous code is shown in Figure 5.3a. By making the insert invisible to the first query, we obtain graph 5.3b. The second query of the graph is now invalid, since the application would never create this graph. If the first query cannot see the insert, then the branch of the code responsible for executing the second query is never reached.

5.5.1 Patterns

We describe one pattern for each possible violation pattern that is possible within the MongoDB consistency model, as discussed in ection 3.3.8.



(a) RMW violation pattern

(b) RMW pre-violation pattern

Figure 5.4: RMW patterns in the DSG

Read-My-Writes

We give a formal specification of a Read-My-Writes (RMW) violations in Section 3.3.8. A RMW violation occurs whenever an insert/update is not visible to a query, even though it precedes it in program order. The violation pattern in the DSG that is created by RMW violations is shown in Figure 5.4a. It shows an update/insert followed by arbitrary many events connected by program order and finally a query that does not see the initial update/insert. The corresponding pre-violation pattern with the changed visibility is illustrated in Figure 5.4b.

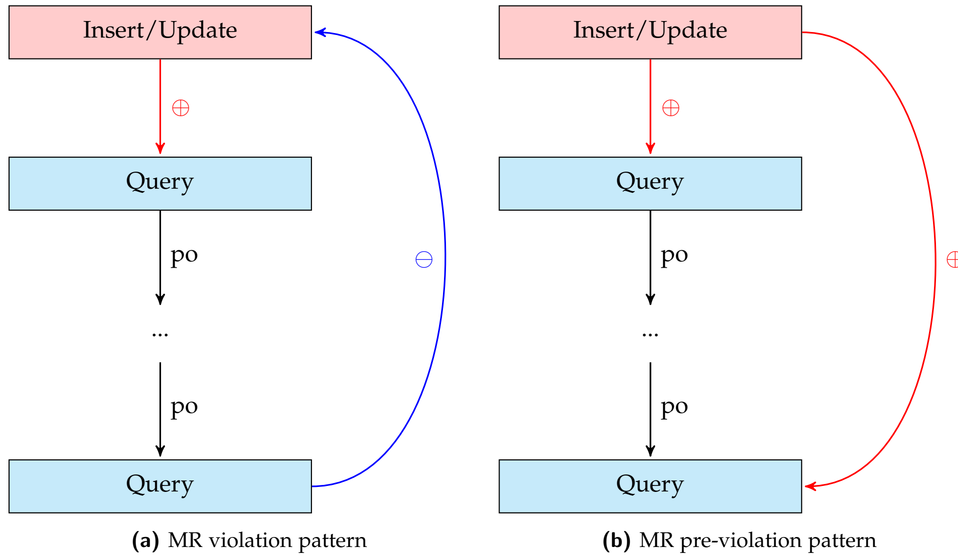
Monotonic-Reads

As described in Section 3.3.8, Monotonic-Reads (MR) violations occur if an insert/update is visible to a query, but not to another query that follows the first one in program order. This is expressed by the pattern shown in Figure 5.5a. Figure 5.5b shows the corresponding pre-violation pattern.

5.5.2 Pattern Filtering

In the case of RMW violations for each update there exist as many different RMW violations as there exist non-commutative queries following by program order. In the case of MR violations this argument applies to all updates with two queries following. Therefore, we only look for the smallest possible match to the pattern.

In practice, most inserts/updates are followed by one or more non-commutative queries at any point of the execution. This leads to approximately as many matches to the patterns as we have inserts/updates. This is usually not very helpful for manual inspection. Therefore, we limit the number of reported patterns to 5. As a heuristic for identifying the 5 most relevant patterns,



(a) MR violation pattern

(b) MR pre-violation pattern

Figure 5.5: MR patterns in the DSG

we rank them in ascending order by the real-time interval that the pattern includes. More precisely, we measure the timespan between the first and the last event included in the pattern. Note that this method is as imprecise as the system clocks of the clients differ from each other. However, in our implementation all clients are run on the same machine.

5.5.3 Algorithms

While pattern matching on a graph is NP-complete in general [27], we can exploit the structure of our graph to extract the matching patterns in $\mathcal{O}(|\oplus|)$ time.

Read-My-Writes

Algorithm 2 shows how the RMW patterns are extracted in our extension of ECRacer. We filter all event pairs of the dependency relation by only considering those pairs that belong to the same client (line 2). We then sort the dependency pairs (line 3) and group them by their update events (line 4). Afterwards, we iterate over all these update events (lines 5 to 12) and for each update event over the corresponding dependency pairs (lines 6 to 11). If there exists a dependency whose query has a read-concern distinct from *linearizable*, we add both events of the dependency to the set of patterns and continue with the next update. The reason we ignore queries with read-concern *linearizable* is that these queries make RMW violations impossible by definition, which is described in Section 3.3.6.

The algorithm has an asymptotic runtime complexity of $\mathcal{O}(|\oplus|)$.

1: $P \leftarrow \emptyset$
2: $depList \leftarrow$ toList(sameClient($\oplus$))
3: $depList \leftarrow$ sortByStartTime($depList$)
4: $depMap \leftarrow$ groupBySource($depList$)
5: **for all** $update \leftarrow depMap$.keys() **do**
6: **for all** $dependency \leftarrow depMap[update]$ **do**
7: **if** $dependency.query.readConcern \neq$ "linearizable" **then**
8: $P \leftarrow P \cup \{update, dependency.target\}$
9: break
10: **end if**
11: **end for**
12: **end for**

Algorithm 2: Extraction of RMW pre-violation patterns, where P is the returned set of patterns

5.5.4 Monotonic-Reads

Algorithm 4 describes our method of extracting the MR pre-violation patterns in ECRacer. We sort and group the dependency pairs like in Algorithm 2. Then we create an empty mapping from clients to queries (line 5) and iterate over all grouped updates (line 6 to 19). For each of these updates we iterate over the corresponding dependency pairs (line 7 to 18) and check if the query has read-concern *linearizable*. If not, we check if the current client of the dependency is already contained in our map (line 10) and add it with the query if necessary (line 11). If it was already contained we check if our map only contains one query for that client (line 12). In this case we can finish the pattern with the current query (line 14) and add the query to the map (line 15) to avoid multiple entries for the same pattern. The reason for ignoring linearizable queries is the same as in Section 5.5.3.

The algorithm has an asymptotic runtime complexity of $\mathcal{O}(|\oplus|)$.

```
1:  P ← ∅
2:  depList ← toList(⊕)
3:  depList ← sortByStartTime(depList)
4:  depMap ← groupBySource(depList)
5:  clientToQueries ←Map()
6:  for all update ← depMap.keys() do
7:      for all dependency ← depMap[update] do
8:          targetClient ← dependency.darget.clientId
9:          if dependency.readConcern ≠ "linearizable" then
10:             if targetClient ∉ clientToQueries then
11:                 clientToQueries.put(targetClient, {dependency.target})
12:             else if |clientToQueries[targetClient]| == 1 then
13:                 firstQuery ← clientToQueries[targetClient].head
14:                 P ← P ∪ {update, firstQuery, dependency.target}
15:                 clientToQueries[targetClient].add(dependency.target)
16:             end if
17:         end if
18:     end for
19: end for
```

Algorithm 3: Extraction of MR pre-violation patterns, where P is the returned set of patterns

Chapter 6

Stress Testing

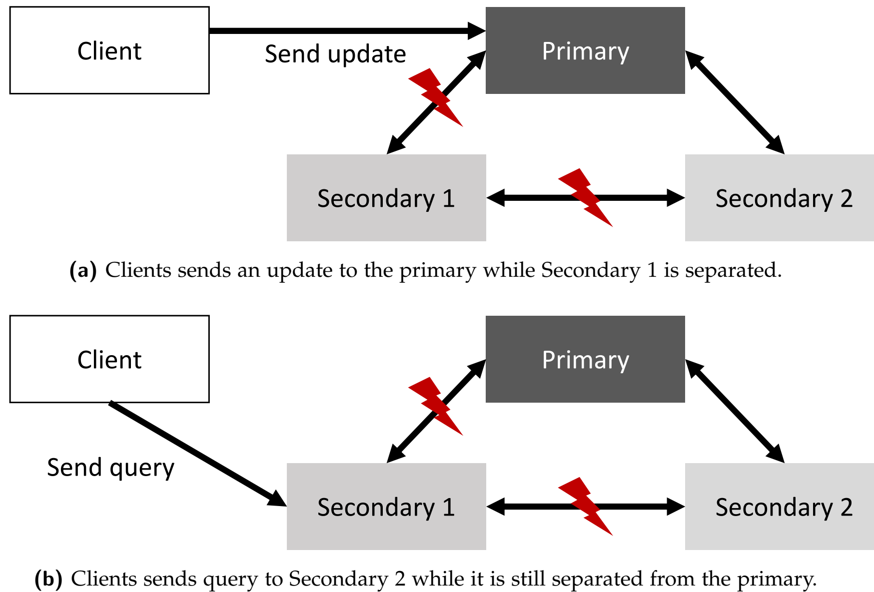
Our experimental results, presented in Chapter 8, demonstrate that the frequency of occurring serializability violations during an instrumented execution are generally low. To increase that frequency and therefore make the dynamic analysis more useful, we propose a strategy that is based on the triggering of network partitions between the different nodes of a replica set. Our approach is designed to cause stale reads, which lead to RMW and MR violations as explained in Section 3.3.8.

6.1 Strategy

We know that stale reads are caused by read operations from a secondary node, that has not yet applied the latest oplog entries of the primary. In an experimental run, where all nodes of the replica set are located on the same machine, the secondary nodes apply the newest changes to the primary's data set with a very small delay in time. We enforce the secondary to keep stale data by blocking the network traffic between the secondary and the other nodes. We control the triggering of network partitions from the instrumented client application. Example 6.1.1 demonstrates how we enforce a stale read using our strategy.

Example 6.1.1 Assume we have a replica set consisting of one primary and two secondary nodes. Figure 6.1 illustrates the following events:

1. The client initiates an update to the primary node.
2. A network partition is triggered that separates Secondary 1 from the remaining replica set.
3. The primary receives and applies the update.



(a) Clients sends an update to the primary while Secondary 1 is separated.

(b) Clients sends query to Secondary 2 while it is still separated from the primary.

Figure 6.1: Stale read forced by our partitioning strategy.

4. The client sends a query to the replica set. We enforce it to be sent to Secondary 1 via special read preference settings. This prevents it from observing the update event from Step 1.
5. We finally resolve the network partition. □

The exact strategy is defined by Algorithm 4. We have two listener functions which are called when a client initiates a query or an update. On each update we trigger a network partition if currently no partition exists and perform the update after the partition has been created. For each initiated query we enforce it to read from the separated secondary if there currently exists a network partition. After performing the query we resolve the partition. Note that this approach can easily be adapted to trigger partitions on every nth update instead of each one.

6.2 Discussion

This strategy is not designed to produce stale reads for every query. It solves the purpose to significantly increase the amount of stale reads during a dynamic analysis run. We heuristically exhaust the pattern that an update event commonly has a non-commutative query event following. Our approach does only work for executions that have a non-commutative query following on an update. Consider an application that always sends several commutative queries after an update. This leads to the resolving of

```
1: partitionTriggered ← FALSE
2: function onUpdateInitiated(update)
3:     if partitionTriggered == FALSE then
4:         triggerNetworkPartition()
5:     end if
6:     update.execute()
7: end function
8: function onQueryInitiated(query)
9:     if partitionTriggered == TRUE then
10:        query.setReadPref(SECONDARY_1)
11:    end if
12:    query.execute()
13:    resolveNetworkPartition()
14: end function
```

Algorithm 4: Extraction of MR pre-violation patterns, where P is the returned set of patterns

the network partition before the non-commutative query is executed and can cause the stale read. There are several approaches for these situations. One approach is to sustain the partition for not only one query but m queries. Another approach is to perform an approximation of commutativity of the query and the update that triggered the partition and wait for a non-commutative query before resolving the partition. However, there is no guarantee of such a query ever being initiated. Since the approach is not designed to guarantee a stale read in every case, but as a technique to increase the rate of stale reads, it suffices for our experiments. The results in Chapter 8 show that the approach is able to significantly increase the occurring serializability violations.

Chapter 7

Implementation

To evaluate the proposed dynamic analysis, we implemented most of the instrumentation concepts that are described in Chapter 4 for Node.js clients. Furthermore, we extended the ECRacer software as explained in Chapter 5. This chapter presents our implementation that was used to run the experiments of Chapter 8.

7.1 MongoRacer

We developed a Node.js application called *MongoRacer* that allows to instrument other Node.js applications that use MongoDB as a database. Our application can be used to run one or more instances of the application to be instrumented and covers setting up and shutting down the replica set as well as instrumenting the application and recording the instrumented database traces. Additionally it performs the analysis of Chapter 5 on the collected data by running our extended version of ECRacer. We refer to the application to be instrumented as the target application.

7.1.1 Architecture

The architecture of our tool is illustrated in Figure 7.1. Each node of the replica set is contained in its own Docker-container. Using Docker simplifies controlling the communication between the nodes, used to trigger network-partitions, but it also increases modularity and system independence. Further, MongoRacer starts another MongoDB instance that is used to store the instrumented database operations and is called *Records* in Figure 8.1. The Composer component has the following responsibilities:

- Starting/stopping Docker [21] containers
- Setting up/shutting down MongoDB replica set

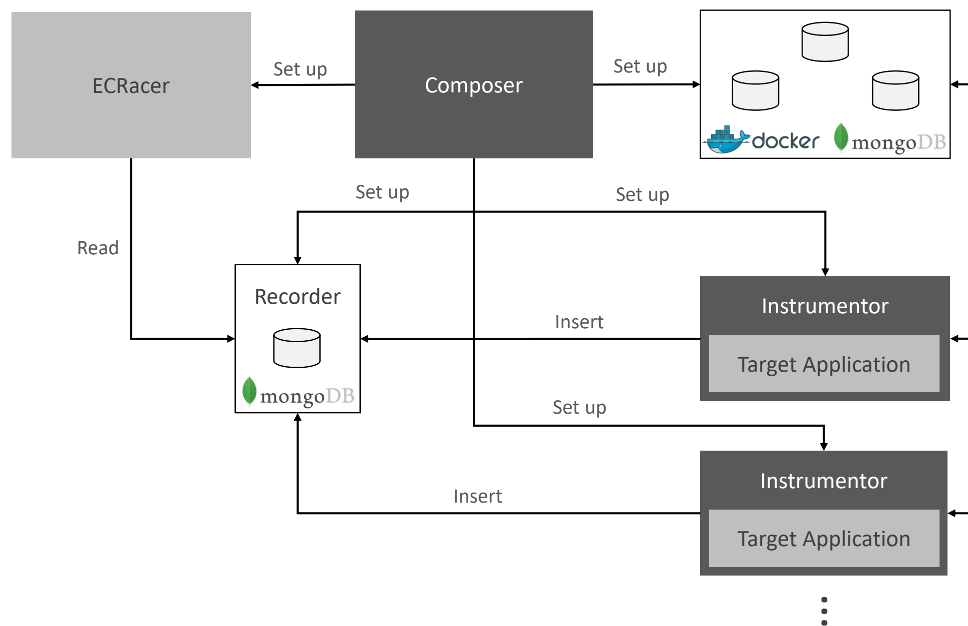


Figure 7.1: Architecture of MongoRacer

- Setting up/shutting down MongoDB instance for storing instrumented traces
- Starting/stopping target applications with injected instrumentation
- Instrumenting and recording oplog entries of primary node
- Running ECRacer on the recorded traces

The Instrumentor component takes care of:

- Instrumentation of the Node.js driver
- Overwriting the target application's database configurations, such that they point to the replica set that is set up by the Composer component
- Overwriting the target application's port, such that it listens on the port specified in the configuration file of our tool

7.1.2 Composer

When MongoRacer is started in the terminal, the Composer parses the command line arguments and loads the corresponding configuration file, which contains the path to the target application. It then starts the MongoDB recorder instance for storing the instrumented database traces and the replica set. Further, it configures the replica set and waits until the nodes have elected a primary and the replica set is ready to operate. The Composer

then runs an optional initialization script, which can be used to perform some custom configurations before the target application is started. This can be inserting sample data to the database, creating a user account or creating files in the file system. After the initialization script has terminated, the Composer starts the specified number of injected target applications. The injection is explained in Section 7.1.3. Because Node.js does not support threads, each instance of the target application is run in its own process. The target applications are then executed until they terminate or the MongoRacer is signaled to stop the analysis. The Composer will then stop the target applications, shut down the replica set, run ECRacer on the collected data and finally shut down the MongoDB recorder instance.

7.1.3 Instrumentation

We extend the official MongoDB driver for Node.js [5] by implementing most of the instrumentation concepts of Chapter 4.

We choose an approach that is widely called monkey patching and describes changing the behavior of an application at runtime without changing the original code. We use this approach to wrap the functions of the Node.js MongoDB driver in order to implement our instrumentation. In most cases this means replacing the function of the prototype of the driver by a modified function. This has the following advantages over providing a modified version of the driver as a Node.js package:

- Since we are usually wrapping the original driver functions into additional functionality or only change the arguments, our approach is less sensitive to changes to the driver and can therefore operate with other versions as long as they provide the same interface. Providing an own version of the driver means to change the driver on every update of the official one. However, monkey patching does not guarantee compatibility to all other driver versions either.
- Monkey patching decouples the instrumentation from the target application. It allows to leave the code of the target application completely untouched.

The arguments for monkey patching are also the reason why we perform our instrumentation directly on the functions that are called by the user of the driver and not on a deeper level in the driver. Manipulating code that is at a very deep level increases the possibility of code changes with newer driver versions, while the functions that represent the user interface are less likely to change in order to maintain backward compatibility.

An example for the replacement of a prototype driver function is the following:

```
Collection.prototype.replaceOne = function () {
  return Collection.prototype.updateOne.apply(this, arguments);
};
```

This code simply delegates a call of the replaceOne function to updateOne by overriding the prototype of Collection.

We also override the driver function to connect to MongoDB and give it a different MongoDB connect string that points to our replica set. This avoids having to change the configuration of the target application. For the same reason we override functions of the *http* module to override the port that the target application listens on.

The Instrumentor is called in an own process by the Composer, performs the instrumentations via monkey patching and runs the code of the target application by importing it. This can not be achieved by calling the Node.js function *require*. The Node.js runtime needs to know which module is the main module (the module which was initially loaded by the Node.js process). We want this to be the target application after we performed the instrumentation. Therefore we call the *_load* function of the Node.js native Module package. The require function internally calls _load as well. The main difference between both functions is that _load offers a parameter that states if the loaded module is to be loaded as the main module or not.

Document Filtering

We have two cases that need a filtering mechanism for queries:

1. Deleted documents
2. Documents fetched by non-ID-queries

As described, deleted documents only contain a delete property and are still fetched by the query in order to construct our visibility relation and need to be filtered out before returned to the target application. The result set of non-ID-queries needs to be filtered, because we removed the filter in order to fetch the whole collection.

As described in Chapter 4, we finally delegate all queries to the find function, which returns a cursor. Therefore, we implement the filtering by overriding the corresponding function of the cursor object. We filter out all documents that contain the delete property and in case of a non-ID-query all documents that do not match the query selector. We achieve the latter by using the Node.js package Mingo [2], which is an implementation of the MongoDB query language in JavaScript and allows filtering JavaScript objects. We store all documents that we filtered out in a data structure in order to attach them to the traces that we record.

Identifiers

- The *client identifiers* that we use to create the UID are generated by the Composer and given to the Instrumentor when creating the corresponding process.
- The *sub-client identifier* is generated by the Instrumentor. Whenever a child process is created, the Instrumentor wraps the application that is to be run in the process into another Instrumentor instance. Therefore, we cannot have uninstrumented child processes.
- The *UID* is generated using the client identifier, the sub-client identifier and an event counter that is incremented on each initiated database operation.
- Each Instrumentor has its own *transaction identifier*. It starts with zero and is incremented on each HTTP-request or SOCKET.IO-request, where SOCKET.IO [25] is a library for handling websockets.

Recording of Traces

To record the database traces of the target application we use the Application Performance Monitoring (APM) API of the driver. It offers listener functions that allow logging all database events, containing additional information like the IP address of the replica node. We add additional information like time stamps to these events and insert them into the MongoDB recorder database.

Oplog Instrumentation

To construct the arbitration order relation, we need to know the time of execution of a database event on the server. Unfortunately there exists a bug in the driver's APM API that removes this time stamp from the events. We were able to fix this bug and contribute the fix to the driver, such that it will not occur in new versions. However, for other versions we attach a listener to the oplog of the primary node, because each oplog entry contains a server time stamp and our UID. This listener function is called on every new entry in the oplog such that we can insert it into the record database.

Limitations

We do not support bulk operations. One reason for that is that they are not mapped to the driver interface functions that are used for non-bulk operations. Instead, they are based on functions deeper in the driver. This means a stronger dependency to the driver version and the same implementation effort as for all the non-bulk operations. Another reason is that bulk operations are only used in one of the projects that we used to evaluate our

approach. However, the implementation can be extended to also support bulk operations. All the concepts of Chapter 4 also apply for bulk operations, because they are not executed atomically. Therefore, it is a limitation of our concrete implementation and not of the instrumentation method.

Another limitation is that we cannot directly map a violation to a line of code in the target application. The MongoDB APM API uses an event listener, which is called asynchronously and therefore loses the code location that triggered the database operation in its stack trace. An instrumentation that avoids using the driver's APM API can solve this issue.

7.1.4 Network Partitions

We implement the triggering of network partitions, as described in Chapter 6, by configuring the Linux kernel firewall using the user-space application iptables [24]. Because each replica node is executed in its own Docker container we can precisely control the allowed communication between the different containers via iptables. This has the consequence that the network partitioning only works on Linux operating systems.

7.2 Extension of ECRacer

As mentioned in Chapter 5, we extend ECRacer, which is written in Scala, to support our recorded MongoDB traces. As already explained, ECRacer expects the relations program order, arbitration order and visibility as well as commutativity and absorption specifications for the events in order to run the analysis algorithm. Therefore, we extend it to transform the recorded traces from our MongoDB record database into the internal data structures representing the relations. We iterate over all events of a trace and create an ECRacer specific data structure for each event. In this step we add the missing time stamp of execution from the oplog records.

The creation of arbitration order, program order and visibility have already been covered in Chapter 5.

7.2.1 Commutativity and Absorption

Commutativity and absorption are implemented in Scala code by using the specifications from Sections 5.1 and 5.2 and applying them to the internal data structures of ECRacer.

7.2.2 Document Matching

To specify commutativity between an insert and a query or a delete event, we need to be able to determine if a given document matches a given selector. We over-approximate the document matching of MongoDB in ECRacer

and only support a subset of query operators, because there exist 34 query operators in MongoDB 3.4 and the effort of supporting each operator is high in comparison to the gain of precision for the analysis. We replicate the behavior of the document matching method of the MongoDB server using the specifications of the different query operators. If our method detects an unknown or unsupported operator in the selector it simply reports that the document matches the selector. This assures that the analysis does not become unsound. The supported operators are $or, $and, $not, $nor, $elemMatch, $eq, $gt, $gte, $lt, $lte, $ne, $in, $exists, $mod, $all, $size. Therefore, we support $\sim$ 47% of all query operators. Examples for not supported operators are regular expressions and geospatial operators. We do not support regular expressions because it is hard to guarantee the soundness of the analysis if we are not sure to exactly replicate or over-approximate the regular expression matching of MongoDB. Furthermore, only one of the projects that we analyzed uses regular expressions.

Chapter 8

Evaluation

As a proof of concept, we use our instrumentation tool *MongoRacer* from section 7 together with our extended version of ECRacer to analyze 17 open source Node.js projects that use MongoDB as their database.

8.1 Experimental Set-Up

The experimental set-up is shown in Figure 8.1. Each experiment consists of 4 phases.

Phase 1 (Initialization): Phase 1 is responsible for setting up the environment. MongoRacer sets up a new MongoDB replica set consisting of one primary and two secondary nodes. MongoRacer finishes phase 1 by starting two instances of the client application to be analyzed.

Phase 2 (Recording): The recording of instrumented database operations performed by the client applications happens in phase 2. One browser window is opened for each of the two client instances. We automatically perform actions in these browser windows using a technique called monkey testing [22], which randomly simulates user actions on the website. This technique has the advantage of being very flexible in terms of a large variety of client applications. We perform the testing by using the browser extension Tampermonkey [7] with a custom test script. All instrumented database operations are stored in the MongoDB record database.

Phase 3 (Shutdown): Phase 2 is terminated by manually signaling MongoRacer to finish recording. It will then enter phase 3 and therefore terminate the client applications and shut down the replica set. This phase makes sure that the replica set and the client applications are stopped in a con-

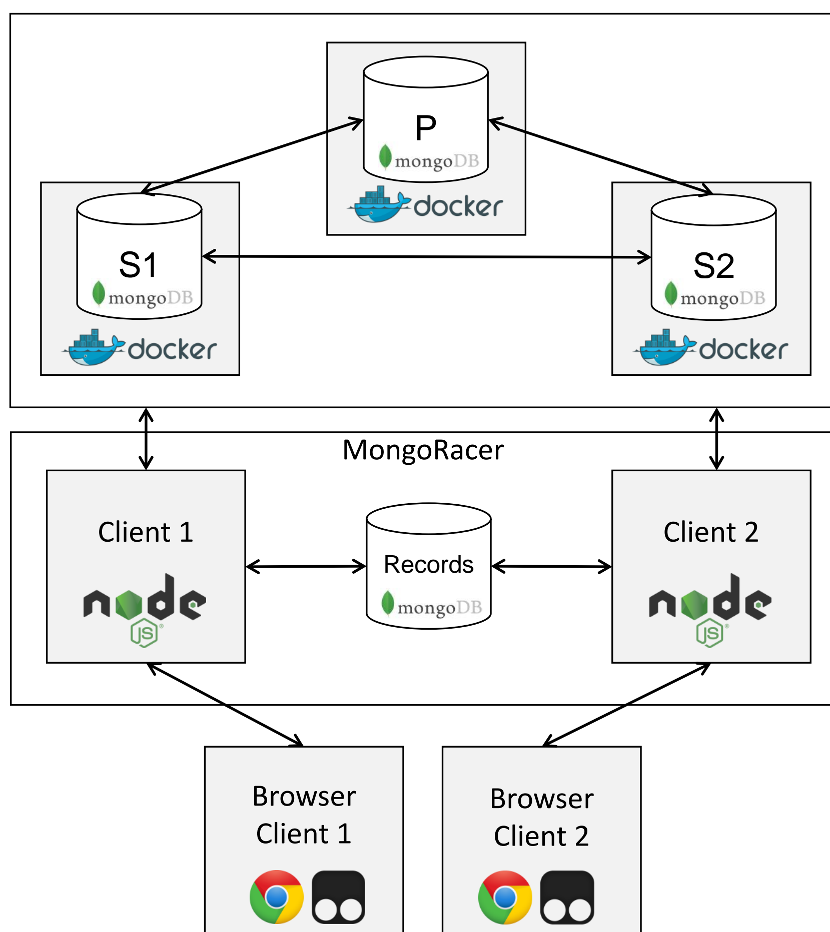


Figure 8.1: Experimental set-up; P: Primary replica node, S1: Secondary replica node #1, S2: Secondary replica node #2

trolled manner. The MongoDB recorder instance stays online for the next phase.

Phase 4 (Analysis): Phase 4 handles the analysis of the recorded database operations from phase 2. It uses our extended version of ECRacer, which fetches the records from the MongoDB recorder instance. After the analysis is finished, MongoRacer shuts down the recorder instance and exits.

We perform two dynamic analysis runs for each project. One normal run and one with stress testing, explained in chapter 6. Therefore we collect two sets of database records for each project. Each of these record sets is analyzed with ECRacer twice, while one run considers transactions and the other one does not, because enabling transactions can hide violations that occur within a single transaction.

Table 8.1: Experiment results; S: Number of Strongly Connected Components (SCCs) found; V: Number of different serializability violations; t: Transactions enabled; p: Network partitioning enabled

Project	S	V	S (t)	V (t)	S (p)	V (p)	S (p,t)	V (p,t)
lets-chat	0	0	18	2	12	3	17	3
NodeBB	0	0	3	2	29	4	21	3
relax	0	0	6	1	8	2	9	2
strider	0	0	3	1	0	0	2	1
node-login	0	0	0	0	7	2	7	2
aqua	0	0	-	-	1	1	-	-
tvshow-tracker	0	0	0	1	1	1	1	1
node-todo	1	1	0	0	15	3	15	3
node-stripe-membership-app	0	0	4	1	3	2	3	1
chat.io	1	1	3	3	3	3	3	3
Nodejs-MongoDb-TodoMVC	9	2	9	2	2	2	2	2
mean-stack-registration-login-example	0	0	1	1	0	0	1	1
mean_mytasklist	1	1	1	1	4	2	4	2
event-backend	0	0	0	0	18	1	18	1
timetracker	0	0	0	0	1	1	1	1
game-of-life-javascript	0	0	2	1	2	1	5	1

8.2 Results

As explained in Section 2.1, a serializability violation is represented by a cycle in the DSG. ECRacer gives a set of Strongly Connected Components (SCCs) as its output, where each SCC contains one ore more cycles. Table 8.1 shows the number of SCCs for each project. Because SCCs indicate the frequency of violations and not their diversity, we also give the number of violations that are not all caused by the same database operation using different parameters.

One can observe that a partition-free analysis run with disabled transactions does generally not yield many violations. Enabling transactions turns out to cause a significant increase of the amount of violations. But transaction violations are only one class of violations and can also occur in strongly

consistent data stores.

Triggering network partitions drastically increases the amount of serializability violations. Enabling transactions on these runs adds new violations only in the analysis of Mean-stack-registration-login-example.

While a short description of each project together with the corresponding violations can be found in A, the next sections describe the most important detected violations.

8.2.1 Session Violations

156 of 277 discovered SCCs and 34 of 76 discovered violations in all analyses are caused by session handling.

Sessions are used to equip the stateless HTTP protocol with state. It is mainly used for authentication. Typically a user fills out a login web form with his login credentials. After a successful authentication, a session object is created on the server and a cookie with the corresponding session-ID is stored at the user's browser, which keeps it from authenticating again with every request. The client only sends the session-ID and the server checks its validity.

Atomicity Violations

A common DSG pattern of session violations is shown in Figure 8.2. Because a session object usually has a limited period of validity, the session is updated with every request to the server. This strategy achieves that the validity of the session is always dependent on the last action while logged in and not on the time of login itself. After updating the session, it is fetched from the database in a separate query. In our transaction model this is done within the same transaction. This can lead to an overlapping of these 2 operations on two different clients if they use the same session. This is the case if they log in with the same account, like in our experiments. That explains the large amount of transaction related session violations. However, these violations are not harmful, because in the worst case they only lead to a minimum deviation in the duration of validity of the session. In practice this period of validity does not need to have an accuracy that high.

RMW/MR Violations

RMW violations occur on session objects, which are usually fetched to check the authentication of a user. If the query does not fetch the session object, access is denied to that user. The user logs in with his credentials but is denied access. However, if the violation does not occur on the creation of a session object but on the update of one, the impact on the application is

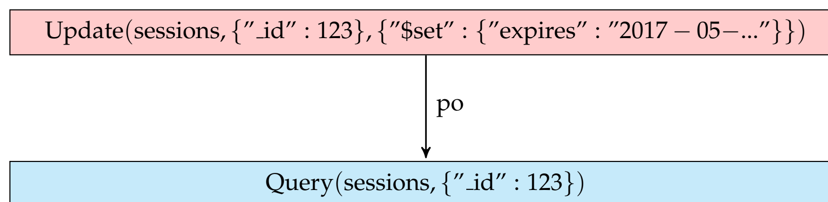


Figure 8.2: Common pattern used in session management

very small, because only the update to the period of validity of the session is not observed. Another situation occurs if the session object is being deleted, can not be observed and the user is therefore still authenticated. Developers have to be careful here not to introduce security violations.

8.2.2 Harmless Data Violations

40 of 76 discovered violations are violations that we could not directly link to a harmful influence on the application. However, that does not prove the absence of such a harmful influence. Note that some violations are counted multiple times in the above statements, since they were found in multiple analysis runs.

RMW violations always consist of one update event that inserts/updates/deletes a document and a following non-commutative query event that does not observe the update. In general this is not a harmful effect. However, if there exists code that relies on the visibility of the update to the query it can lead to errors, even though we did not observe any during the monkey testing. We found RMW violations containing the following update events:

1. (Lets-chat) Insertion of user
2. (Lets-chat) Insertion of chat room
3. (NodeBB) Update of forum category
4. (NodeBB) Update of forum topic
5. (Tvshow-tracker) Insertion of user
6. (Node-todo) Insertion/update/deletion of todo entry
7. (Chat.io) Insertion of user
8. (Chat.io) Insertion/update of chat room
9. (Nodejs-MongoDb-TodoMVC) Insertion/update of todo entry
10. (Mean-stack-registration-example) Insertion of user
11. (Mean_mytasklist) Insertion/update of todo entry

All MR violations depend on an insert, update or delete and two following queries. As with RMW violations, the effects of a MR is generally harmless, but can lead to errors in specific situations that we could not observe during our tests. We found MR violations for the following update events:

1. (Node-login) Insertion of user
2. (Node-todo) Insertion/update/deletion of todo entry
3. (Node-stripe-membership-app) Insertion of user
4. (Timetracker) Insertion of user

We found the following harmless violations related to transactions:

1. (Lets-chat) Both clients add a message to a chat room. Lets-chat then checks if there exists only one chat room with the corresponding name, adds the message object, fetches the corresponding chat room and updates that room setting a new *lastActive* time stamp. These two transactions overlap in such a way that the queries that are looking for similar rooms with the same name, cannot see the update of the other client.
2. (NodeBB) While client 1 is performing several queries on the user collection, client 2 updates a user. This leads to some queries of client 1 seeing the update and some not.

8.2.3 Harmful Register Violations

We found a harmful atomicity violation in the projects Node-login and Mean-stack-registration-example, which occurs when two users register at the same time. It is a common pattern in applications that is demonstrated in a general way in Figure 8.3. Both clients register a user by first checking if a user with that user name already exists in the database and then adding that user. If these transactions overlap in a way that both queries cannot observe the insert of the other client, two inserts happen. If the same user name was picked, there exist two users with the same user name in the database. A state that is usually unwanted. In the violation that we observed, two user accounts with the same e-mail address but with different user names are created. This is especially problematic in a project that serves as a foundation to build other applications upon. All applications that use this project for account management also inherit this bug.

8.3 Fixing Violations

For RMW/MR violations there exists a simple fix for MongoDB 3.4 or higher. Using the linearizable read concern, described in Section 3.3.6, guarantees the absence of RMW and MR violations. But it also enforces to read from

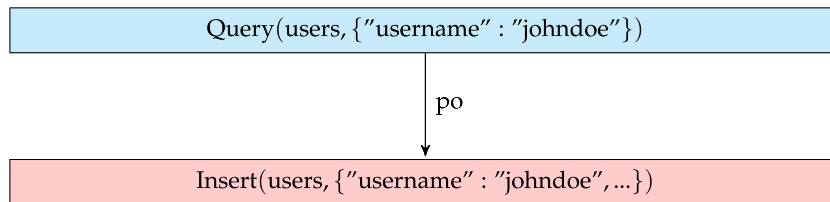


Figure 8.3: Common pattern used in user registration

the primary replica node. Therefore, it is not an optimal solution to just set a linearizable read concern for every operation. That would destroy all performance benefits of a replica set. But it is highly recommended to ensure linearizable read concern on parts critical for security or error prevention. For the sake of performance and scalability, violations can be ignored if they are guaranteed to only have harmless effects on the application. An example for a harmless effect is an outdated information displayed to the user in an environment where it is not critical to have information in real time. If a user gets displayed an outdated post counter of a topic in NodeBB, that is a harmless violation. But in the end it has to be decided by the developer if a violation can have a harmful effect on the application or not.

It is more complicated to fix transaction violations, because there are no transactions in MongoDB. Therefore, one has to make sure that all operations that need to be atomically executed are within the same single update/insert/upsert operation.

As an example, we show how to fix the common user registration violation shown in Figure 8.3. One can solve this problem by adding an index on the fields to be unique. If for instance the username and the e-mail address should be unique for each user account then a compound index involving username and e-mail address solves the problem. The application only has to handle the MongoDB error thrown at a unique index violation.

Chapter 9

Conclusion

We showed that a MongoDB replica set that is configured to provide eventual consistency additionally guarantees the consistent prefix property. This reduces the possible serializability violations to read-my-writes and monotonic-reads violations.

We explained how to instrument a MongoDB client application in order to collect enough data to apply the serializability criterion of Brutschy et al. and the corresponding algorithm.

Furthermore, we introduced a software that implements the proposed instrumentation and extended the analysis tool by Brutschy et al. to run on these instrumented recorded executions.

We evaluated our implementation by performing the proposed dynamic analysis on a set of Node.js open source applications that use MongoDB as their data store and conclude that the criterion proposed by Brutschy et al. is suitable to serve as a foundation for tools that detect serializability violations in dynamic executions of clients of replicated data stores.

Appendix A

Serializability Violations

Lets-chat

Lets-chat is a chat application that allows registered users to send text messages into custom chat rooms.

Violations

- Session violation
- RMW violation on insertion of user
- RMW violation on insertion of chat room
- Atomicity violation on insertion of message to chat room

Manual testing with enabled network partitioning shows that it takes several approaches to log into the application. This behavior is caused by session violations.

NodeBB

NodeBB is a commonly used bulletin board software. It is the most complex of the analyzed projects.

Violations

- Session violation
- RMW violation on update of forum category
- RMW violation on update of forum topic
- Atomicity violation on update of user

Manual testing with network partitioning enabled shows some effects of the violations. Users are not logged in after registration and logging in takes several approaches. Both issues are caused by session violations.

Relax

Relax is a content management system on top of Node.js.

All violations are caused by session management.

More involved testing methods than monkey testing are likely to unravel more violations when used in network partition mode.

Strider

Strider is an open source continuous integration and deployment server.

All violations are transaction violations in session management.

Node-login

Node-login is a basic account management system whose purpose is to be used in other applications to handle user registration and login.

Violations

- Harmful registration atomicity violation
- MR violation on insertion of user

Aqua

Aqua describes itself as a website and user system starter. That means that it is supposed to be used as a starting point for creating a website or application. It gives basic features like account management and a contact page.

Transaction violations: Transactions like we implemented them, do not work here. This is because our implementation of the instrumentation does not cover the library that is used in Aqua to create requests.

RMW/MR violations: We only found one violation and it is caused by session management.

The monkey testing approach is not able to deeply cover the functionality of Aqua. We are therefore convinced that a more specific test method will detect more violations.

Tvshow-tracker

This is a project that offers a platform for tv-shows. It allows to subscribe to tv-shows and get informed about new episodes. Monkey testing could not discover the application properly. Therefore we were forced to perform manual testing here. We could only find one RMW violation during user registration.

Node-todo

Node-todo is a simple todo-list application for demonstration purposes. It has no user registration or authentication but offers only basic CRUD (Create Read Update Delete) functionality on todo entries.

All violations are RMW or MR violations on either inserting, updating or deleting todo entries. Because Node-todo is an application for demonstration purposes and does not offer a lot of functionality there is no harmful effect of the violation. However, it demonstrates that violations occur even in minimalistic applications.

Node-stripe-membership-app

This project is a boilerplate application for creating membership/subscription sites with Stripe, an online payment service.

All transaction violations are related to sessions. While most RMW/MR violations are related to sessions, one violation turns out to be a MR violation during user registration. A newly registered user is fetched by a query but not by a following one. We explained similar violations in relation to other projects already.

Chat.io

Chat.io is another real time chat application.

We found three RMW violations. One occurs during the registration of a user. The other two violations occur while inserting/updating a chat room.

Nodejs-MongoDb-TodoMVC

This is again a simple todo application for demonstration purposes. We used another monkey testing approach that did not involve our Tampermonkey script but a JavaScript library called gremlins.js [1]. Because it does not rely on the Document Object Model (DOM) of the website, but randomly applies user interactions on the browser screen, it allows much more interactions

per second. This leads to many RMW violations that occur even without network partitioning.

All violations are RMW violations while inserting/updating todo entries. Other violations are not possible in this application because of its simple structure and small functionality.

Mean-stack-registration-example

This project was created for a tutorial on the MEAN (MongoDB, Express, Angular.js, Node.js) stack and provides a simple registration/login example functionality.

We found the common but dangerous user registration violation, already described in the results of Node-login. Furthermore, we found a RMW violation on user registration.

Even though it is a simple example application this reflects the way many applications are developed. It is also to expect that readers of the corresponding tutorial use this project as a starting point, inheriting this issue.

Mean_mytasklist

This is another minimalistic todo-list application.

Like for the other todo-list projects, we found RMW violations occurring during CRUD operations on todo-entries.

Event-backend

Event-backend is a simple content-management-system specialized on event managing.

All violations that occurred are session related. The monkey testing approach was not able to discover the functionality in depth.

Timetracker

This application is a management system for time tracking. Users can register and add tasks that contain a duration.

The only violation that we found is a MR violation during user registration. But the monkey testing was not able to cover the functionality of the application appropriately.

Game-of-life-javascript

This is a JavaScript version of Conway's Game of Life [18].

We only discovered session violations in this project.

Bibliography

[1] gremlins.js. https://github.com/marmelab/gremlins.js/blob/master/README.md. Accessed: 2017-04-25.

[2] Mingo - javascript implementation of mongodb query language, howpublished = https://www.npmjs.com/package/mingo, note = Accessed: 2017-04-29.

[3] MongoDB documentation. https://docs.mongodb.com/v3.4/reference/method/db.collection.update/. Accessed: 2017-04-20.

[4] MongoDB documentation - update operators. https://docs.mongodb.com/v3.4/reference/operator/update/. Accessed: 2017-04-21.

[5] Mongodb node.js driver. https://mongodb.github.io/node-mongodb-native/. Accessed: 2017-04-29.

[6] Play! the high velocity web framework for java and scala. https://www.playframework.com/. Accessed: 2017-04-20.

[7] Tampermonkey. https://tampermonkey.net/. Accessed: 2017-04-24.

[8] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering*, pages 14–22. ACM, 2013.

[9] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 163–172. IEEE, 2013.

[10] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[11] Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 59. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[12] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison- Wesley, 1987.

[13] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. Serializability for eventual consistency: Criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 458–472, New York, NY, USA, 2017. ACM.

[14] Sebastian Burckhardt. *Principles of Eventual Consistency*, volume 1. now publishers, October 2014.

[15] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 568–590, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[17] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[18] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[19] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[20] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1263–1274. VLDB Endowment, 2007.

[21] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[22] Noel Nyman. Using monkey test tools. *Software Testing & Quality Engineering Magazine*, pages 18–21, 2000.

[23] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[24] Gregor N Purdy. *Linux iptables Pocket Reference: Firewalls, NAT & Accounting*. " O'Reilly Media, Inc.", 2004.

[25] Rohit Rai. *Socket. IO Real-time Web Application Development*. Packt Publishing Ltd, 2013.

[26] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, Nov 2010.

[27] Gabriel Valiente and Conrado Martínez. An algorithm for graph pattern-matching. In *In Proc. 4th South American Workshop on String Processing, volume 8 of Int. Informatics Series*, pages 180–197. Carleton University Press, 1997.

[28] Kamal Zellag and Bettina Kemme. How consistent is your cloud application? In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 6:1–6:14, New York, NY, USA, 2012. ACM.

[29] Kamal Zellag and Bettina Kemme. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *The VLDB Journal*, 23(1):147–172, 2014.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Serializability Checking for MongoDB Clients

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Baum	Johannes

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Zürich, 04.05.2017	JBaum

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.