Master Thesis

Analyzing Serializability of Cassandra Applications

Arthur Kurath

Supervised by Lucas Brutschy

Prof. Dr. Peter Müller
Chair of Programming Methodology
ETH Zurich

March 31, 2017

Abstract

Programming an application that uses an eventual consistent data-store is challenging due to the presence of unexpected behaviors that are not possible on data-stores with a stronger consistency model, e.g. a traditional relational databases.

In this thesis we implemented a static analysis that checks a given program for possible serializability violations that may occur at runtime. In a first phase, a set of graphs, which abstracts all possible interactions of the program with the data-store, is built using abstract interpretation. This graphical abstractions are then used to find concrete serializability violations that may occur when the program is executed on multiple clients.

We evaluated the static analysis on several real world programs. The analysis reported multiple serious serializability violations and only few false positives. The violations are easily understandable due to their graphical representation. Hence, the analysis is helpful for the development of applications that use an eventual consistent data-store.

Contents

1 Introduction

Large distributed systems often rely on replicated data-stores that offer high availability, scalability and partition tolerance. As it is stated in the CAP theorem [13], it is impossible to provide also consistency for such a data-store. So in exchange, these systems provide weaker consistency guarantees than for example traditional relational databases, usually some variants of eventual consistency.

Programming an application that uses an eventual consistent data-store is challenging as the programmer has to ensure strong consistency himself whenever it is required. Serializability of a program intuitively means that given a schedule of the transactions of the program, there exists a sequence in which the transactions of the schedule can be executed such that the outcome is equal to the outcome of the original schedule. This helps to reason about the correctness of the program: If a program is serializable on a data-store with weak consistency guarantees, one can reason about the program without taking the effects of the weak consistency model into account. Serializability violations on the other hand point to program parts where strong consistency guarantees do not hold. Therefore, these can help the programmer in identifying where additional synchronization mechanisms are required in the program to make it work correctly.

Recently, Brutschy et al. [2] proposed a criterion that helps to decide whether an execution of a program is serializable. In their work they evaluate it using a dynamic analysis, so the criterion is applied to traces that are collected from running programs. While the dynamic analysis is already promising, a static analysis would help to check serializability effectively during development. There is no need for the overhead of running the program multiple times and collecting the traces. Additionally, the results of a static analysis are valid for all executions and not only the observed ones.

In this project we implemented such a static analysis for Java programs that use Apache Cassandra[1] as a data-store. In a first phase, all possible interactions with Cassandra are abstracted in a set of graphs using abstract interpretation. These graphs over-approximate all the operations that may be executed on the database in the given program. In a first approach, we check for serializability violations of the program by modeling dependencies between operations in the collected graphs and search for cycles. In a second approach we model all possible executions of the program on two clients and apply an extension of the criterion defined in [2] on these executions to find serializability violations.

[1] `https://cassandra.apache.org/`

The remainder of this report is structured as follows: The next chapter provides background on the criterion used for dynamic serializability checking and some information about Cassandra. In Chapter 3, the static analysis that creates the graphs that over-approximate the possible interactions with Cassandra is described. Afterwards follows a chapter about checking serializability on these graphs. In Chapter 5 we evaluate the analysis on example projects. The report ends with a conclusion and possible future work.

2 Background

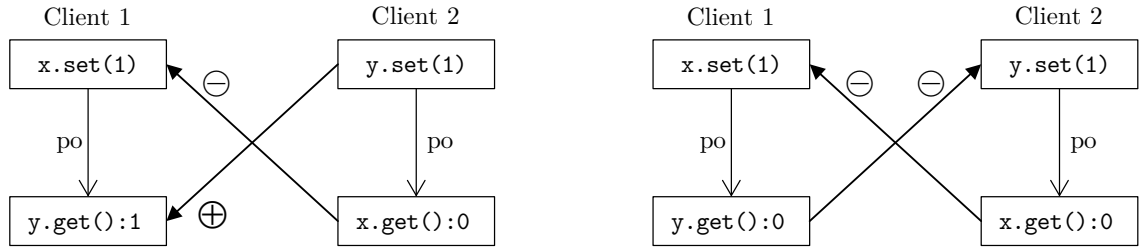
In this chapter, the first section describes dynamically checking serializability of an execution of a program using dependency serialization graphs. Some information about Apache Cassandra can be found in the following section. We assume that the reader is familiar with abstract interpretation [5, 16].

2.1 Dynamic Serializability Checking

Brutschy et al. proposed a criterion for checking if a given schedule is serializable [2]. A schedule has events of two types of operations: An *update* modifies the state of the data-store whereas a *query* retrieves the state (or part of it) of the data-store. An eventually consistent schedule is characterized by visibility and arbitration. Informally, an update is *visible* to a query if it is included in the evaluation of the result of the query. The order in which update conflicts are eventually resolved by the system is called *arbitration*. Given an eventually consistent schedule, one can find dependencies and anti-dependencies between these updates and queries. A query *depends* on an update if the returned data of the query would change if a visible update would not be visible. An update has an *anti-dependency* to a query if the result of the query would change if an invisible update would become visible to the query (i.e. making an update visible to a query would introduce a new dependency).

Given a schedule, one can determine dependencies and anti-dependencies taking commutativity and absorption into account. Two operations (updates / queries) *commute*, if the order in which the operations are executed does not change their result. An operation u *absorbs* v, if the result of executing v and then u is the same as executing only u. Using these notions, a visible update u is a dependency of a query q unless commutativity and absorption can be applied as rewrite rules repeatedly to obtain a schedule where u is not visible to q. An invisible update u anti-depends on a query q if q would depend on u when u would be visible.

All existing dependencies and anti-dependencies together with arbitration and program order can be combined in a directed graph, known as the *dependency serialization graph* (DSG). If this graph does not contain a cycle, the given execution is serializable. One can construct a serial schedule of the events by ordering them so that the partial orderings of the DSG are preserved. If the DSG contains at least one cycle, it is possible that the execution is not serializable.



(a) serializable (b) not serializable

Figure 2.1: Dependency serialization graph for two different schedules. The right DSG contains a cycle, so the schedule is not serializable.

Figure 2.1 shows the dependency serialization graphs for two schedules. Such a schedule could result from two clients that first set some flag and then check the flag of the other client. The left schedule is serializable: The events of client 2 can be executed before the events of client 1. The right schedule is not serializable, as both clients do not see the update of the other client.

2.2 Apache Cassandra

Apache Cassandra is a scalable open source database. Data is distributed and replicated across many nodes which can be placed in multiple data centers. Cassandra extends the concept of eventual consistency by offering tunable consistency, which means that the client application decides how consistent the requested data must be. For example, when writing or reading data, one can specify the number of replicas that need to respond to the request before the result of the operation is returned. Additionally, Cassandra offers a linearizable compare-and-set operation. This mechanism can be used if a value should only be written if a certain condition is met, for example to implement that a new user should only be inserted in the database if no other user with the same username is existing already. In addition to basic types like numbers, strings or dates, Cassandra also offers sets, maps, lists and counters as conflict-free replicated data types [17].

Data is organized in a column-family model, i.e. in rows of tables where each table has a fixed set of columns. The primary key of each row consists of a partition key and an optional clustering key. Both of them can include multiple columns. On which nodes a row is stored is determined by the hash of the partition key. The clustering columns are used to order the rows within a partition on a single node. When data is written to Cassandra, at least the partition key has to be specified. None of the primary-key-columns can be updated.

Data is queried and updated using an SQL-like language named CQL. Statements are ordered on the database using a timestamp, which is generated on the server per default, but can also be specified on the client when executing a statement. The following types of statements exist for querying and updating data:

Query: A `SELECT`-statement is used to execute a query on Cassandra, e.g.
`SELECT * FROM users WHERE username = 'Bob';`.

Upsert: A row is inserted or updated using either an `INSERT` or an `UPDATE` statement. Despite that the syntax is different, the semantics on the database is the same for both statements. If an inserted row already exists, the `INSERT` updates the column values. If a row is not existing, the `UPDATE` statement does create it. So both of the following statements have the same semantics on the database:
`INSERT INTO users (username, password) VALUES ('Bob', 'asdf');`
`UPDATE users SET password = 'asdf' WHERE username = 'Bob';`.

Delete: Some columns or an entire row are deleted using a `DELETE` statement, e.g.
`DELETE playlist_names FROM users WHERE username = 'Alice';`
deletes the column `playlist_name` for the user named "Alice" whereas the following statement deletes the entire row
`DELETE FROM playlists WHERE username = 'Alice' AND playlist_name = 'favourites';`.

Batch: Multiple upsert and delete statements can be combined into a single logical operation using a batch. All statements of the batch will be executed using the same timestamp. Here is an example of a batch:
`BEGIN BATCH`
`UPDATE users SET chat_rooms = chat_rooms + {'cassandra'} WHERE username = 'Bob';`
`UPDATE chat_rooms SET users = users + {'Bob'} WHERE room_name = 'cassandra';`
`APPLY BATCH;`.

A compare-and-set operation (aka "Lightweight-Transaction" (LWT) in Cassandra terms) can be executed by adding an `IF`-clause to an upsert or delete statement. One can either check for the existence of a row or one can update a value based on a condition. So in the following examples, "Alice" is inserted only if no other user with the username "Alice" is already existing and a chat-room is deleted only if it was created by a user named "Bob":
`INSERT INTO users (username, password) VALUES ('Alice', 'asdf') IF NOT EXISTS;`
`DELETE FROM chat_rooms WHERE room_name = 'Cassandra' IF creator = 'Bob'.`
Lightweight-Transactions are linearizable and are implemented using a consensus protocol, which makes them less efficient. [7, 8, 10, 12]

3 Building the Transaction Graphs

In this chapter we describe how we obtain a representation of a program that we can use to check for serializability violations. This representation is a set of graphs where each graph abstracts one transaction. We call such a graph a *transaction graph*. A transaction is a set of events for which a user wants that they are executed atomically and in isolation on the data-store. The data-store itself has no support for transactions. A transaction is defined in terms of a method: All events that are executed between the first and the last statement of this method are grouped into one transaction. The methods that span a transaction are specified by the user either in form of `@Transaction` annotations in the program or as an argument. Transactions of a program are unrelated, i.e. each transaction can be considered as a single program with one method as an entry point and no dependencies to other transactions.

We define the transaction graphs in the next section. Afterwards follows a description of the static analysis that is used to build the transaction graphs for real Java programs.

3.1 Transaction Graphs

Definition: A transaction graph is a pair (E, P) where

E is a set of events on the data-store
$P \subseteq \{(u, v, res) : u, v \in E \cup \{entry, exit\}, res \in \{\emptyset, +, \top\}\}$ is an order on the events

An *operation* is a specific function on the data-store that can be invoked in some way by providing a list of arguments. An *event* is the execution of an operation on the data-store, so it consists of an operation and a list of arguments. Hence, a transaction graph is a directed graph whose nodes are events. An edge from e_1 to e_2 means that e_2 can happen directly after e_1 if either $res = \top$ or else if the result of e_1 matches res. So the edge $(e_1, e_2, \emptyset)$ denotes that e_2 can happen directly after e_1 only if e_1 returns an empty result (respectively non-empty for $(e_1, e_2, +)$). The event *entry* is the entry- and the event *exit* is the exit-node of the transaction. Both do not execute an operation on the database. Each event in the transaction graph has to be on a path from *entry* to *exit*.

For our work with Cassandra as a data-store, an operation is a synonym for a CQL-query that may contain bind-markers. A *bind-marker* in a CQL-query is either a question mark or a text of the form :<id>. So the query

```
SELECT * FROM chat_rooms WHERE name = ?  AND creator = :username
```

has two bind-markers: An unnamed bind-marker at index 0 and a named bind-marker at index 1 with the name "username". Before a CQL-query is executed, all bind-markers have to be replaced with concrete values, i.e. primitives or objects, which is called *binding*. So an event in our work consists of an operation, i.e. a CQL-query as a string, and a list of mappings from bind-markers to a set of possible values.

3.1.1 Examples

In this section, we show code-examples of single transactions in combination with the desired transaction graph. The examples also motivate the design choices we have made for the static analysis. The only way to execute an event on Cassandra so that is recognized in the analysis is by calling the `execute` function on an object of type `Session`. The first argument is the CQL-query, the remaining arguments are the replacements for the bind markers in the CQL-query.

```
1 public class Twitter extends SessionHolder {
2   private final static String TABLE_USERS = "users";
3
4   @Transaction
5   private void register(String username, String password){
6     if (isNewUser(username))
7       addUser(username, password);
8   }
9
10  private boolean isNewUser(String username){
11    ResultSet res = session.execute("SELECT username FROM " + TABLE_USERS
12                                    + " WHERE username = '" + username + "';");
13    return res.isExhausted();
14  }
15
16  private void registerUser(String username, String password) {
17    String query = "INSERT INTO " + TABLE_USERS + " (username, password) "
18                 + "VALUES ('" + username + "', '" + password + "');";
19    session.execute(query);
20  }
21 }
```

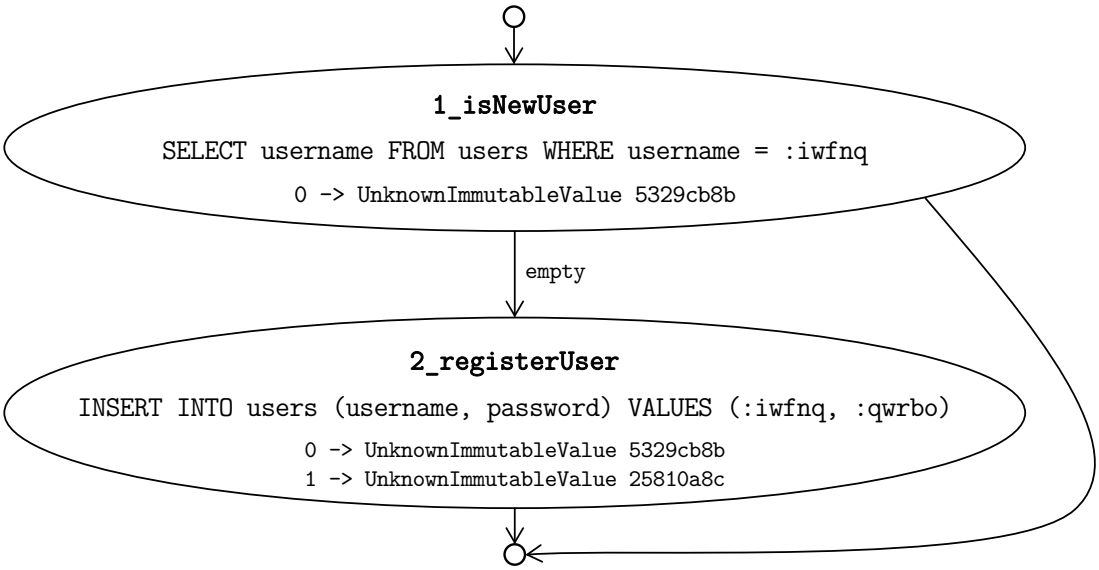


Figure 3.1: This figure shows the transaction graph for a `register` transaction. The CQL-query is created using string concatenation. A static field on line 2 is used to resolve the name of the table. In the expected transaction graph, the concatenated CQL-queries from the program are transformed into operations with bind-markers. The `username` and `password` parameters are abstracted to an unknown value representation. A constraint on the edge from `1_isNewUser` to `2_registerUser` reflects that the second event only happens if the first event returned an empty result.

String concatenation is transformed to the bytecode by creating an object of class `StringBuilder` on which the concatenated values are appended. Therefore to support the provided code, the analysis has to handle objects of class `StringBuilder` precisely. Additionally, string constants stored in fields have to be recognized. Also some sort of inter-procedural analysis is necessary to include all events that are executed inside a called method. Also the inter-procedural analysis has to be precise enough to capture facts like equality of the usernames in the first and the second event and the fact that the second event is only executed if the first returned an empty result.

```
1  @Transaction
2  public void removeUserFromRoom(String roomName, String username) {
3    if (roomName != null && username != null){
4      final BatchStatement batch = new BatchStatement();
5
6      batch.add(QueryBuilder.update("chat_rooms").
7                  with(QueryBuilder.remove("users", username)).
8                  where(QueryBuilder.eq("name", roomName)));
9
10     batch.add(QueryBuilder.update("users").
11                 with(QueryBuilder.remove("rooms", roomName)).
12                 where(QueryBuilder.eq("login", username)));
13
14     session.execute(batch);
15   }
16 }
```

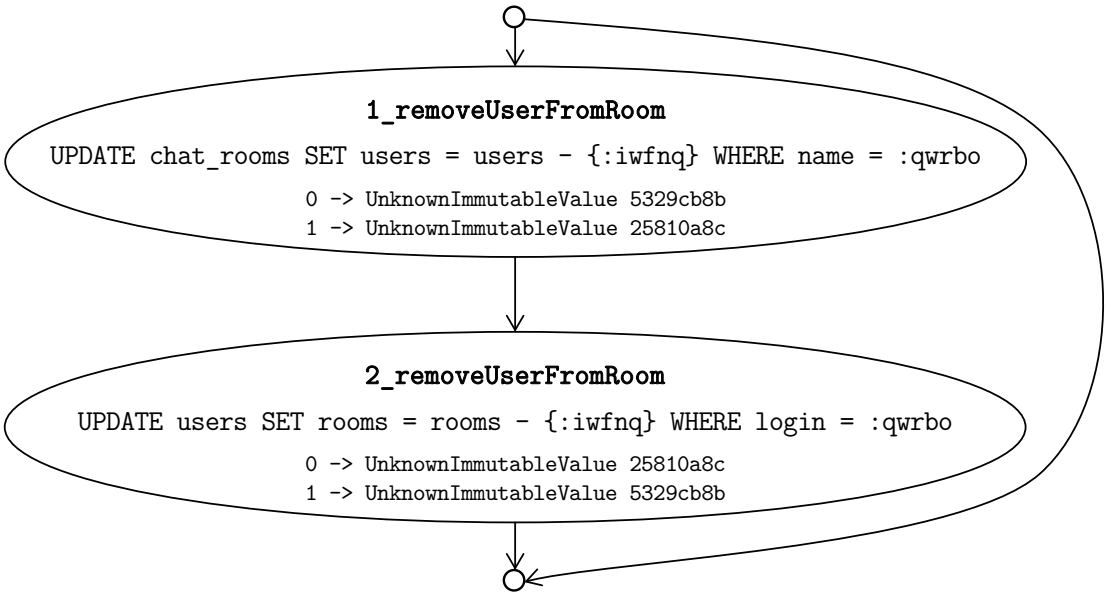


Figure 3.2: In this example, the QueryBuilder-API is used for building CQL-queries. The QueryBuilder-API exposes a set of functions that can be used to create valid CQL-queries.

In the example, a batch statement is used that executes both `UPDATE`s simultaneously. In the transaction graph, the batch statement is split into two distinct events, which makes checking for serializability violations easier. As there is an `if` statement at the beginning of the transaction, it is possible that no event is executed at all in this transaction. Therefore, there is an edge from the entry-node to the exit-node.

The analysis has to support most of the methods that are provided by the QueryBuilder-API. Also, a `BatchStatement` should be abstracted precisely enough such that the two statements added on line 6 and 10 can be split into two events in the transaction graph. The control flow of the program should be represented in the transaction graph too.

```
1 @Transaction
2 public List<String> listArtist(String firstLetter, boolean desc) {
3   String queryText = "SELECT * FROM artists_by_first_letter"
4                      + " WHERE first_letter = ?";
5   if (desc)
6     queryText = queryText + " ORDER BY artist DESC";
7
8   ResultSet results = session.execute(queryText, firstLetter);
9
10  List<String> artists = new ArrayList<>();
11  for (Row row : results)
12    artists.add(row.getString("artist"));
13
14  return artists;
15 }
```

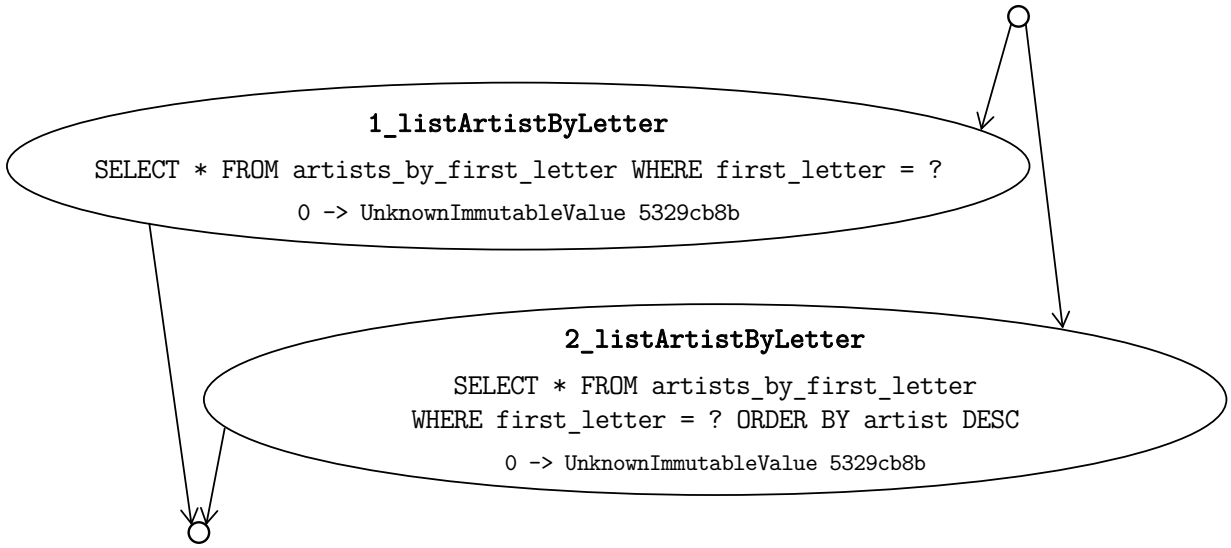


Figure 3.3: This example shows a transaction graph with multiple nodes following the entry-node. Even though there is only one invocation of the `execute` function, the transaction graph has two nodes as the CQL-query is different for the possible values of the parameter `desc`.

Therefore, one node in the transaction graph does not correspond to a point in the program where the `execute` function is invoked. This has to be considered in the design of the analysis.

```
1  public class TimelineDaoImpl {
2    private Session session;
3    private PreparedStatement getTimelineStmt;
4    private PreparedStatement getTweetStmt;
5
6    public TimelineDaoImpl(Session session){
7      this.session = session;
8      this.getTimelineStmt = session.
9            prepare("SELECT tweet_id FROM timeline WHERE username = ?");
10     this.getTweetStmt = session.
11           prepare("SELECT body FROM tweets WHERE tweet_id = ?");
12   }
13
14   @Transaction
15   public List<String> getTimeline(String username) {
16     ResultSet res = session.execute(getTimelineStmt.bind(username));
17     List<String> tweets = new ArrayList<>();
18     for (Row row : res){
19         UUID tweetId = row.getUUID("tweet_id");
20         Row tweet = session.execute(getTweetStmt.bind(tweetId)).one();
21         tweets.add(tweet.getString(body));
22     }
23     return tweets;
24   }
25 }
```

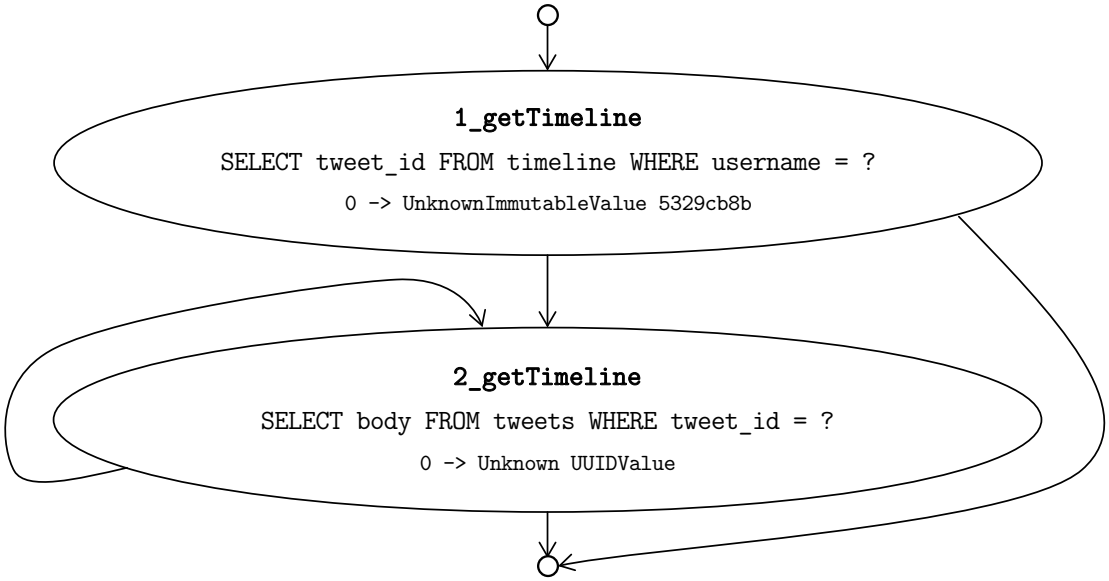


Figure 3.4: This example demonstrates the usage of prepared statements [9]. A prepared statement can be used to increase the performance if the same CQL-query is executed multiple times. It is created by calling the function `session.prepare` with the CQL-query as the argument. In the preparation phase, the CQL-query is sent to Cassandra where it is parsed and cached, which means that parsing can be skipped when an event that uses this query is executed. Prepared statements should be reused and therefore are usually created in a constructor or initialization method and stored in some field.

To handle this example precisely, the analysis needs to know what the possible values for the fields in line 3 and 4 are. Inside the transaction code, these fields are never assigned, so it is also necessary to run an analysis on the code that is never used in a transaction to collect constant field values.

```
1  public class TrackServlet {
2
3    @Transaction
4    protected void doPost(ServletRequest request, ServletResponse response){
5      String artist =  request.getParameter("artist");
6      String track_name = request.getParameter("track_name");
7      TracksDAO newTrack = new TracksDAO(artist, track_name);
8      newTrack.addTrack();
9    }
10 }
11
12 public class TracksDAO {
13   private final UUID track_id;
14   private final String artist;
15   private final String track;
16
17   public TracksDAO(String artist, String track){
18     this.track_id = UUID.randomUUID();
19     this.artist = artist;
20     this.track = track;
21   }
22
23   public void addTrack(){
24     SessionSingleton.get().execute(
25       "INSERT INTO tracks (track_id, artist, track) VALUES (?, ?, ?)",
26       track_id, artist, track);
27   }
28 }
```

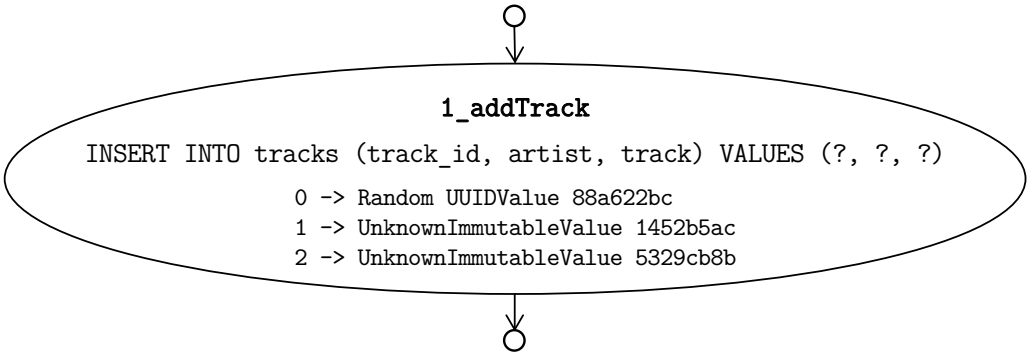


Figure 3.5: This figure shows the transaction graph for a servlet that offers a transaction to add a new track. On line 7, a new track object is created. A random universally unique identifier (UUID) [14] is created inside the constructor as the id of the track. Random UUIDs have the property that they are globally unique, so in this example a track created in one transaction has never the same id as a track created in another transaction.

Whether a UUID is globally unique should be tracked by the analysis, as inequality is useful when defining commutativity. The analysis also has to implement some abstraction for general objects like the `newTrack` object on line 7 to handle the method call on line 8 precisely.

The `session.execute` function takes an arbitrary number of bind arguments. In the bytecode, this is transformed to an array of type `Object` with fixed length. So in the bytecode, the `execute` function has only two parameters where the second parameter in the call on line 24 is an array of length 3 containing the `track_id`, `artist` and `track`. Therefore, there is a need for an abstraction for fixed length arrays in the analysis.

3.2 Analysis

Using abstract interpretation [5], we have implemented a static analysis to build transaction graphs for programs written in Java. The idea behind abstract interpretation is to compute a *flow* (aka abstract context or abstract state) for each point in the program which over-approximates the real execution state at that point, i.e. the flow at a program point must include all possible states that can occur at runtime at the given point. The flow for each program point is computed iteratively on the control flow graph (CFG) by merging all flows from predecessor nodes and then applying the effect of the current statement. Merging is done by applying a *join function* ($\sqcup$). If $f_{res} = f_1 \sqcup f_2$, f_{res} must over-approximate f_1 and f_2. For a node *stmt* in the CFG, we define the *in-flow* as the merged flows from all predecessors, the *transformer* as the function that applies the effects of *stmt* on the in-flow and the *out-flow* as the resulting flow after applying the transformer. The *entry-flow* is the flow that is used as the in-flow for the entry-nodes (i.e. nodes without predecessors) in the CFG. An over-approximation for each program point is calculated by iterating over all program points and calculate the out-flow by applying the transformer on the in-flow. This is done until a fixed point is reached, i.e. for each program point, the out-flow is equal to a new transformation of the in-flow.

The next section describes the flow and the corresponding transformers that we used for the analysis. Afterwards we describe the four steps of the static analysis which consist of transforming the CFGs, the collection of possible field values and finally building and parsing the transaction graphs. We finish the section with an overview of the cases where the resulting transaction graphs may be unsound.

3.2.1 Flow and Transformers

Flow: We use a flow that consists of three parts: A transaction graph, a path constraint and an environment. The transaction graph in the out-flow of a program point p represents the graph if the current transaction spans from the entry-node to p. The path constraint consists of two sets where each set contains all events from the transaction graph that must have returned an empty respectively a non-empty result so that the current program-point is feasible. The environment maps each static field or local variable that may be in scope at the current program point to references. We use allocation-site based abstraction for references [4], i.e. a reference abstracts the program point at which an object is created. Therefore if two variables may point to the same reference, we may have aliasing. The environment additionally maps references to a set of symbolic values. The symbolic values are divided into four categories. Each category has its own definition of a top value.

Immutable values: Strings, Integers and UUIDs cannot be modified in the program after they are created. Therefore, we do not have to care about escaping and can also copy these values around if necessary. Additionally to the constants mentioned before, there exists also an unknown immutable value which is used for example for user inputs or results from unknown functions. This unknown immutable value contains the program point where it was created,

which is used to determine if two unknown immutable values are equal. The top value is an unknown immutable value for which we cannot assume equality if it is used in different statements.

Mutable values: To this category belong mutable values that are specifically handled by the analysis, e.g. objects of type `StringBuilder` or all the objects that have a subtype of `Statement` (e.g. `PreparedStatement`, `BoundStatement`). The top value is an unknown value with no further properties. Also all the symbolic values for the objects created from the static methods of the `QueryBuilder` class belong to this category.

Objects: Each object that does not belong to the other categories and has the type of an analyzed class is represented as a map of field descriptors to references. For each type there exists a distinct top value which contains all the fields that can only have a finite number of possible values at runtime. Theses possible field value sets are calculated in the second step of the static analysis (see Section 3.2.3).

Arrays: An array is represented as a map from integers to references or immutable values. An entry in this map with key i means that a reference or an immutable value was directly assigned to index i, where i is a constant. All remaining indexes point therefore to the initial value depending on the type of the element (0 or `null`). Therefore, the map is bounded by the number of integer constants that appear in the program. The top value is a representation of an array where each index points to the top value of the category the elements belong to.

Widening: The transaction graph and the path constraint in the flows are bounded by the number of `execute` statements in the program, so there is no need for widening. The mapping from static fields or variables to references is also bounded: Static fields and variables are bounded by their occurrences in the program and references are bounded due to the fact that there is only a finite number of allocation sites for a program. Widening is therefore only defined for the mapping from references to symbolic values. Widening for a mapping from a single reference r to a set of values v_1, v_2 is defined as follows:

$$(r, v_1)\nabla(r, v_2) = \begin{cases} v_2, & \text{if } v_1 = v_2 \\ \top, & \text{otherwise} \end{cases}$$

The map that contains the mappings from references to symbolic values is widened by applying the widening defined above on all entries.

Transformers: For a lot of the transformers, the implementation is straight-forward. Thus, we only describe some parts of the transformers that are more special. Whenever we say that we set a variable `var` to top, this means that all references `var` may point to are set to top. Setting a reference `ref` to top means that for all references that are reachable from `ref` and point to a non-immutable value, the values they point to are replaced with the top value. This means that if for example a reference pointing to array `a` is set to top, also all the references of the elements of `a` are set to top.

Default Transformer: This transformer is sound to use whenever we do not have a better transformer for a statement. All local variables that are used in the statement that is abstracted (e.g. parameters and the base object in a method call) are set to top. If a new variable is defined, it is initialized with top.

Field-Assignments: Assignments to a field always appear in the form `o.f = v` in the representation of the CFG where `v` is either a variable or a constant. If `o` points to a symbolic object, the mapping for field `f` is updated to `v`. If `o` points to a top value and `v` is a variable, we check if `v` is in fact constant, i.e. either `v` points only to immutable values or we know from the second phase of the analysis that `v` will never be modified (cf. Section 3.2.3). If `v` is not constant, we set it to top, otherwise we do nothing.

Variable-Assignment from Field: If the value of a field is assigned to a variable, i.e. `v = o.f` and `o` points to a symbolic object, `v` is initialized with the current value of `f` in `o`. If `o` points to a top value and `f` is in the possible field values map (cf. Section 3.2.3), `v` is initialized with the references of the possible field values map. If `f` is not in this map, `v` is initialized with top.

Array-Element-Assignments: Assignments of an array element appear in the representation of the CFG in the form `a[i] = v` where `a` is a variable and `i` and `v` are either a variable or a constant. If `a` points to a symbolic array and `i` is an integer constant, we set the element with index `i` to `v` in `a`. If `i` is not an integer constant, we do not know which element is written in `a`. Therefore, we set `a` and `v` to top. If `a` points to a top value, we set `v` to top.

Casts: As we have categories for different types, it is possible that a value changes the category if it is casted. For example in the statement `String str = (String) o`, `o` may point to a top mutable value. As `str` belongs to the immutable value category, the references `o` points to cannot be assigned to `str`, so `str` is initialized with top and `o` is also set to top.

Execution of Events on the Database: An execution of an event on the database is transformed in three steps:

1. The arguments of the `session.execute` function call have to be transformed to a symbolic value that represents the event, which consists of the CQL-query string and a list of symbolic values that replace the bind markers. If the event was created using a `PreparedStatement` or the `QueryBuilder`, there is a single argument to the `execute` function that already points to a symbolic value with the right form. However, if string concatenation was used for creating the CQL-query, it is likely that the symbolic value for the CQL-query contains non-string-parts (e.g. integers or top values). In this case, the analysis generates a random bind marker for each non-string part, adds the part as a bind marker replacement to the event and uses the bind marker in the CQL-query instead.
2. Afterwards, the event is added to the transaction graph.

3. Finally, if the result from the execution is assigned to a variable `v`, the analysis creates a symbolic value for the result that refers to the newly created event and assigns it to `v`.

Apply-Condition: As the in-flow of each statement is transformed to a single out-flow, conditions of an if-statement cannot be added to the path constraint from the transformer of the if-statement, as this would result in multiple out-flows. Instead, two transformers that apply a condition to the path constraint are added in both branches after the if-statements when transforming the CFG (see Section 3.2.2). For conditions that check whether an event returned an empty (respectively non-empty) result, the flow is transformed by altering the current path constraint.

Invoke-Analyzed-Method and Return-Analyzed-Method: If an analyzed method is called in the inter-procedural analysis, these two transformers enclose the transformers of the called method. These two transformers transform the flow in the following way: When the method is invoked, the base object is assigned to the `this` pointer and the function arguments to the parameter variables. When the method returns, the old `this` pointer is restored and the return value is assigned to the left variable of the method call.

3.2.2 Transformation of the CFG

The first step of the static analysis is to transform the CFG into an *abstract control flow graph* (ACFG). Whereas the CFG consists of the statements that transform the real program state, the ACFG consists of functions that transform the flow we have defined before with respect to the semantics of the abstracted statement. This also means that statements that do not modify the flow can be excluded from the ACFG. Edges in the ACFG also represent possible control flow. We can use the ACFG of a method m to directly run a data-flow analysis on m that calculates the flow defined above at each program point. The in-flow of a node is the flow resulting from merging all out-flows of the predecessors in the ACFG. The out-flow of a node is obtained by applying the transformer of the node on the in-flow.

We use the Soot framework [18] for transforming the byte-code of the program into a CFG. Soot provides different representations of the CFG. Our analysis is built on the Shimple representation[1], which is a typed representation of the byte-code in SSA-form [6]. The `IfStmt` is the only node in the CFG that has more then one successor. It has the form `if <condition> goto <stmt$_2$>` with the two successors `stmt`$_1$ and `stmt`$_2$. So if the condition evaluates to `true`, control flow goes to `stmt`$_2$, otherwise it goes to `stmt`$_1$. To simplify the analysis, our ACFG does not contain such conditional branches. Instead, each `IfStmt` is transformed to an `If-Transformer` with two `Apply-Condition-Transformers` as successors. The `If-Transformer` does not modify the flow whereas the `Apply-Condition-Transformer` applies the condition of the `IfStmt` to the flow.

The CFG of a method is transformed to the ACFG by running an intra-procedural data-flow analysis. The flow is the set of transformers that reach a program point. For the entry-flow, a

[1] https://github.com/Sable/soot/wiki/A-brief-overview-of-Shimple

special transformer t_{entry} is created that marks the entry-point of the ACFG. At each statement *stmt*, the in-flow is transformed to the out-flow according to the following steps:

1. Replace each `If-Transformer` t_1 in the in-flow with an `Apply-Condition-Transformer` t_2 and add an edge $t_1 \rightarrow t_2$ to the ACFG.

2. Check if an analyzed method is called in the statement and no predefined transformer exists. If this is the case, create an `Invoke-Analyzed-Method-Transformer` t_{invoke} and a `Return-Analyzed-Method-Transformer` t_{return}, add t_{return} to the out-flow and add an edge $t_{\text{in}} \rightarrow t_{\text{invoke}}$ for each transformer t_{in} in the in-flow. We create such a transformer pair for each method that may be called from a call-site. The set of possible methods is calculated using a class-hierarchy-analysis [11] and is provided by Soot.
   Otherwise, if a transformer t is needed for the current statement, add an edge $t_{\text{in}} \rightarrow t$ to the execute graph for each transformer t_{in} in the in-flow and set the out-flow to t. If no transformer is needed (e.g. for a `GotoStmt`), copy the in-flow to the out-flow.

When the fixpoint is reached, we have the ACFG for the analyzed method. Note that the graph created in this step may be disconnected. For each invocation of an analyzed method, the created `Invoke-Analyzed-Method-Transformer` has no successors and the corresponding `Return-Analyzed-Method-Transformer` has no predecessors in the graph. Figure 3.6 shows the Shimple representation and the ACFG for a register method.

```
1 private void register(String username, String password){
2   if (dbAccess.isNewUser(username))
3     dbAccess.addUser(username, password);
4 }
```

```
1 private void register(String, String){
2     r0 := @this: SimpleTwitter;
3     r1 := @parameter0: String;
4     r2 := @parameter1: String;
5     $r3 = r0.<SimpleTwitter: DBAccess dbAccess>;
6     $z0 = virtualinvoke $r3.<DBAccess: boolean isNewUser(String)>(r1);
7     if $z0 == 0 goto label1;
8     $r4 = r0.<SimpleTwitter: DBAccess dbAccess>;
9     virtualinvoke $r4.<DBAccess: void addUser(String,String)>(r1, r2);
10  label1:
11    return;
12 }
```

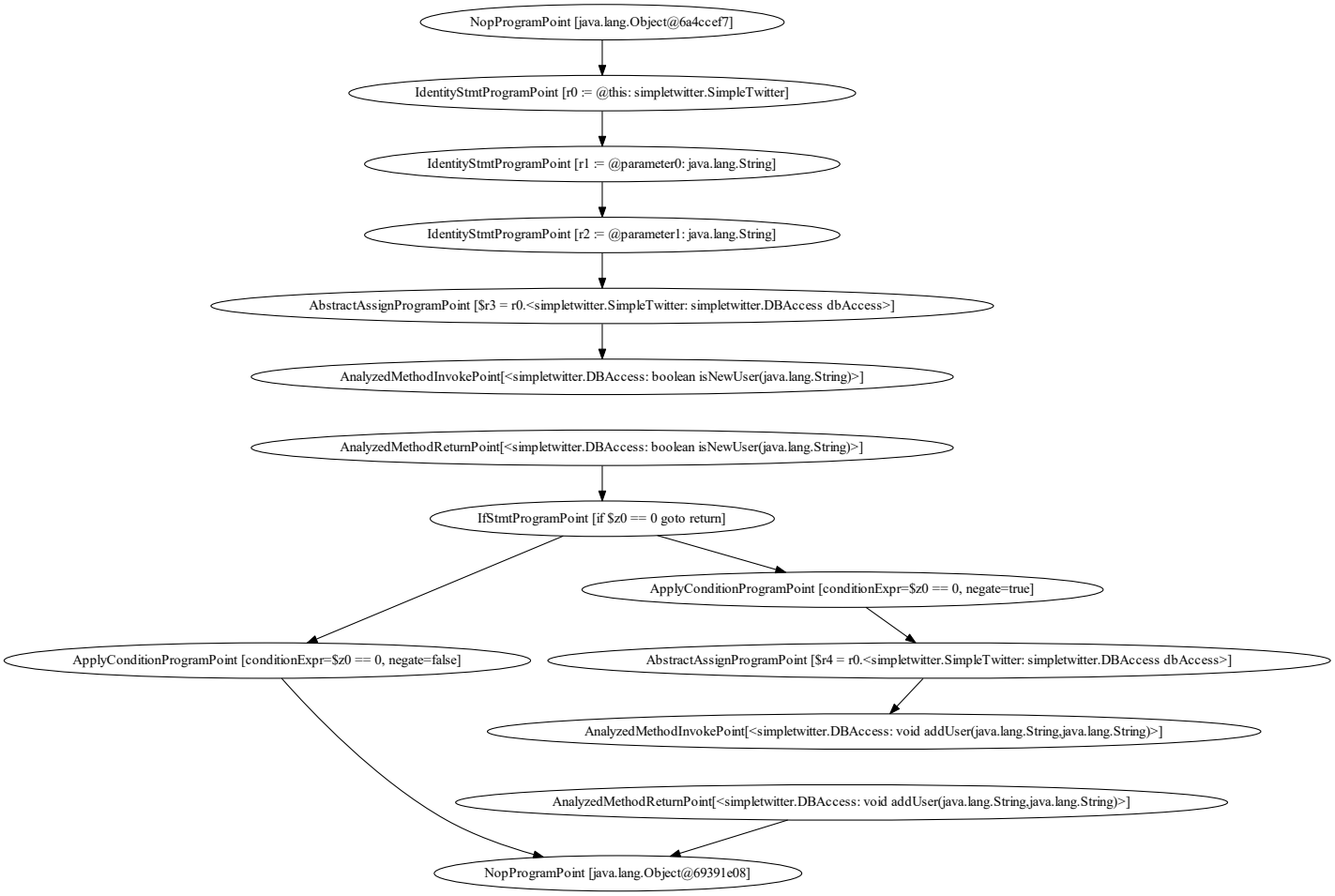


Figure 3.6: Java and Shimple representation of a `register` method. The graph on the bottom shows the abstract control flow graph of the method. The `IfStmt` from the method is represented by three nodes in the ACFG. There are disconnected transformers that abstract the method calls `isNewUser` and `addUser`.

3.2.3 Collection of Possible Field Values

In the second step of the analysis, the ACFGs are used for collecting sets of possible values for each field in the analyzed classes. It is quite common in our examples that prepared statements, table names or CQL-queries are stored in fields (see for example Figure 3.4), so it is crucial for a precise analysis to have some knowledge about possible field values. The analysis is restricted to fields per type, so more formally the goal is to build a map `possibleFieldValuesMap` := `FieldDesc` $\rightarrow$ $\mathcal{P}$(`Reference`) for all fields to which only a specific set of values are assigned. For static fields, the map should be restricted to immutable values. Annotated fields are excluded from this analysis, as the annotation could be used to inject values at runtime (e.g. the `@Autowired` annotation from the Spring framework[2]).

The `possibleFieldValuesMap` maps fields to a set containing references. The values the references point to are stored in the environment part of the flow. Therefore only references that are never created in a transformer can be stored in the `possibleFieldValuesMap`. We call such a reference that is never created in a transformer a *global-reference*.

A field contains a mapping in the `possibleFieldValuesMap` only if the values pointed by the references that are assigned to the field are never modified after the initial assignment. This means that all fields that have a mapping in the `possibleFieldValuesMap` point via references to a set of symbolic values that are in fact constant. Therefore if the value of a field `f` of an unknown object `o` is assigned to a variable `v` (i.e. `v = o.f`), `v` can be directly initialized with the global-references stored in the `possibleFieldValuesMap`.

We iterate over all methods and run an intra-procedural data-flow-analysis on its ACFGs as long as the `possibleFieldValuesMap` changes. The algorithm works as follows:

1. Initialize the `possibleFieldValuesMap` with a reference to the default value for each field, e.g. 0 for an int field or null for a field that contains an object.
2. Loop over all methods as long as the `possibleFieldValuesMap` changes. For each method, execute the following steps:
   (a) Build a representation of the ACFG in which each `Invoke-Analyzed-Method-Transformer` and its corresponding `Return-Analyzed-Method-Transformer` is replaced with a `Default-Transformer` to get a connected graph.
   (b) Create the entry-flow, which means that for the `this` pointer and for each parameter a top value is created. Additionally, the symbolic values for all the references in the `possibleFieldValuesMap` are copied to the entry-flow.
   (c) Run the data-flow analysis until a fixpoint is reached. After each transformation of the in-flow to the out-flow, check if a symbolic value referenced from the `possibleFieldValuesMap` has changed and if this is the case, add the field to a `changedFieldsSet`. Also add a

[2] `http://docs.spring.io/spring-boot/docs/current/reference/html`

static field to the `changedFieldsSet` if a reference that points to a non-immutable value is assigned to the static field.

(d) Remove all fields in the `changedFieldsSet` from the `possibleFieldValuesMap`. Also remove all fields from the `possibleFieldValuesMap` that point directly or indirectly via an array element or another field of an object to a top value.

(e) For all fields that are in the `possibleFieldValuesMap`, add all new references that were assigned to the field in the last analysis to the `possibleFieldValuesMap`. Also copy the corresponding symbolic values from the environment.

(f) Transform all references in the `possibleFieldValuesMap` to global-references. As the same method may be analyzed multiple times, this is needed to ensure that the values assigned to fields in previous data-flow analyses do not interfere with another analysis on the same method.

The `possibleFieldValuesMap` is used if a field of a top value is read. This is especially important for the `this` pointer and the parameters of the methods that span a transaction.

3.2.4 Building the Transaction Graph

In this step, the analysis will build a first version of the transaction graphs. The entry-point of each transaction has to be specified either by using an annotation in the program or by suppling the method-signature as an argument. For each specified method, the transaction graph is created by running an inter-procedural context-sensitive data-flow analysis on the ACFGs. We use call strings for the context as described in [16, 2.5.4], but bound the length of each call string by including each label at most once.

For each transaction, we build a *super-graph* of the ACFGs. The super-graph encodes the context-sensitive inter-procedural analysis in a graph by including the ACFGs of all methods that may be called transitively from the entry-method. Each transformer is instantiated with a stack of method invokes that lead to the execution of the transformer. So for the first method m_1 that is called in a transaction, the stack is empty. If m_1 calls another method m_2 at statement $stmt_1$, $stmt_1$ is pushed on the stack and the transformers of m_2 are instantiated using the new stack. Now, if method m_3 is called from statement $stmt_2$ in m_2, the transformers of m_3 are instantiated with the stack $stmt_1 stmt_2$. If the stack already contains the statement where a new method is invoked, the statement is not added again. This ensures termination even for recursive function calls. Local variables and references are always allocated in the context of the stack at a given transformer, so we still have SSA-form in the super-graph. The super-graph for a transaction is built using a work-list-based approach:

1. Create an empty graph and add all nodes from the ACFG of the method that starts the transaction. Add all `Invoke-Analyzed-Method-Transformers` contained in the ACFG to the work-list.

2. Take the first `Invoke-Analyzed-Method-Transformer` of the work-list. If it was not processed yet, create a new stack containing the invoke statement and add all transformers of the invoked method instantiated with the new stack to the super graph. Add all `Invoke-Analyzed-Method-Transformer` to the work-list.

3. Connect the source nodes of the inserted method with the invoke-node and the sinks with the return-node. Also connect each transformer with all `Caught-Exception-Transformers` that enclose the transformer. If the work-list is not empty, proceed with step 2.

4. Post-process the graph by adding an edge from each `Throw-Exception-Transformer` to the sink node of the graph. Also remove all edges from other transformers than `Throw-Exception-Transformer` to a `Caught-Exception-Transformer` if the sound throw analysis is not enabled (see Section 3.2.6 for a discussion).

Finally, the entry-flow, which consists of possible values for all static fields, the `this` object and all parameters, is built using the results from the possible field values collection step. The data-flow analysis is run on the super-graph until a fixpoint is reached. The transaction graph is then obtained by merging all out-flows of the sink-nodes.

3.2.5 Transformation of the Transaction Graph

The transaction graph obtained from the data-flow analysis on the super-graph is in a form which makes it difficult to analyze it. One node in the transaction graph can contain events with different CQL-queries, e.g. if the CQL-query looks different depending on the value of some parameter (see Figure 3.3). Also, the statements are not parsed at this stage and a bind marker can point to multiple symbolic values. In this step, the transaction graph is transformed into a graph in which each node represents exactly one event. This means that each node is split into possibly multiple nodes in this step. CQL-queries are parsed using the parser from the Cassandra source code[3].

Events are transformed into three structures: Queries (`SELECT`), upserts (`INSERT`, `UPDATE` and `DELETE <col> FROM` (i.e. deletions of single columns)) and deletes (`DELETE FROM` (i.e. deletion of full rows)). All three types consist of the table name and a set of constraints that restrict the rows that are relevant for the event. Constraints are extracted from the `WHERE` part of the CQL-queries. Additionally, queries also have a set of columns that are selected in the event and upserts contain a map from columns to symbolic values that reflects the changes that are applied in the event. So all the facts that we have collected in the static analysis are transformed to a simple data-structure that facilitates the checking afterwards.

3.2.6 Unsoundness

The analysis may be unsound for the following cases:

Lambdas: A lambda is represented as a dynamic invoke call in the bytecode. Thus, we cannot analyze the body of the lambda. Objects that are used inside a lambda are set to top when

[3] `https://github.com/apache/cassandra`

the lambda is created. However, we cannot capture whether a statement is executed on the database from inside the lambda.

Reflection: If a method is called using reflection, this call is not represented in the call graph and thus, the method is not included in the super-graph. This may lead to unsoundness if e.g. an execution of a statement is not included.

Unknown Methods: Calls of method whose source is not available can lead to unsoundness if the method contains assignments to fields of analyzed classes or executes statements on the database.

Single-Threaded: The analysis assumes that the program is executed in a single thread. Concurrent modification of static fields or shared objects may lead to an unsound result.

Exception Handling: Per default the only predecessors of catch statements in the super-graph are throw statements. If another statement throws an exception at runtime, this state is missing after the catch statement. If a sound exception handling is required, each statement that may throw an exception should have an edge to all enclosing catch statements, which may introduce a lot of edges in the super-graph. Sound handling can be enabled using the options, but may have an impact on the performance.

Communication with Cassandra: All events on Cassandra have to be executed using the `execute` function of the `com.datastax.driver.core.Session` interface of the Datastax driver[4]. If for example the object mapper[5] of the Java driver is also used, some events might be missing in the transaction graph.

Asynchronous Execution of Events: The analysis does not differ between synchronous and asynchronous executions of events, i.e. the program order in the transaction graph reflects the program order in the super-graph. This may not be true for an asynchronous execution of an event.

For most applications these restrictions should be acceptable. Some minor modifications were necessary to analyze the evaluated examples soundly, which are discussed in Section 5.1.

[4] `https://github.com/datastax/java-driver`
[5] `https://github.com/datastax/java-driver/tree/3.1.x/manual/object_mapper`

4 Serializability Checking

We check whether all executions of a program are serializable in two ways. The first approach is to over-approximate all possible dependency serialization graphs in a summary graph and search it for specific cycles that indicate a serializability violation. In the second approach, we check all possible executions involving up to two transactions on two clients for serializability violations using an SMT-solver. In the next section, we present the consistency model that we used. The sections following describe the two approaches.

From here on, the term *event* refers to the parsed version of the events (as described in Section 3.2.5). A *query* refers to a query event and an *update* refers to either an upsert or delete event. For readability reasons, we still use the CQL-queries with bind markers or concrete values in illustrations, despite that the parsed events do not have this form anymore.

4.1 Consistency Model

For serializability checking we assume that Cassandra provides causal-consistency [3]. An event e_1 *happens before* another event e_2, if it is ordered by the transitive closure of program order and visibility. A data-store that provides causal-consistency ensures that if an update u happened before a query q, than u is visible to q. Also if an update u_1 happened before update u_2, u_1 is arbitrated before u_2.

Additionally we assume *atomic visibility*, i.e. that all or none of the updates executed in a transaction are visible to queries in other transactions. As transactions are only defined in programs, this has to be ensured somehow by the programmer rather than by the system.

There exists a causal-consistent data-store called Eiger that was implemented by Lloyd et al on top of Cassandra [15]. Eiger provides two operations for reading and writing data on multiple rows, namely read-only and write-only transactions. A read-only transaction does only read data from completed write-only transactions. Using these two types of transactions, a programmer can implement a transaction that is causal-consistent and atomic visible to other transactions by first reading all the values required in a read-only transaction and then finally write back the updates using a write-only transaction.

Using causal-consistency, we can define a *critical cycle* for a static over-approximation of the depen-

dency serialization graph, i.e. edges a cycle has to contain at minimum such that a serializability violation is possible at runtime. Bernardi et al. showed in [1] that a critical cycle consists of at least two anti-dependencies or one anti-dependency and one arbitration edge.

4.2 Over-Approximation of the Dependency Serialization Graph

The dependency serialization graph of an execution can be used to check if serializability violations occurred as described in Section 2.1. One can therefore over-approximate all possible dependency serialization graphs of a program and if there is no critical cycle in the over-approximation, no such cycle can occur at runtime and therefore, all executions of the program are serializable. If critical cycles exist, these can point to transactions that may lead to a serializability violation if executed concurrently.

Given the transaction graphs for a program, the over-approximation of the dependency serialization graph is created by first adding all transactions as nodes. Afterwards, for all pairs of transactions (t_1, t_2), the following edges are inserted:

- If a query q from t_1 and an update u from t_2 exist that may not commute, a dependency edge from t_2 to t_1 and an anti-dependency edge from t_1 to t_2 are added.
- If there is an update u_1 from t_1 and an update u_2 from t_2 that do not always commute, an arbitration edge from t_1 to t_2 and another from t_2 to t_1 are inserted.

Figure 4.1a shows an over-approximation of a program that has two transactions `addTweet` and `viewTweet`. For each transaction t, the checker reports a minimal number of transactions such that a critical cycle exists and t is part of it. One critical cycle for the twitter program can be found in Figure 4.1b. However, the critical cycle reported is a false positive: If there is arbitration between the `addTweet` transactions, there is also absorption, which resolves the cycle.

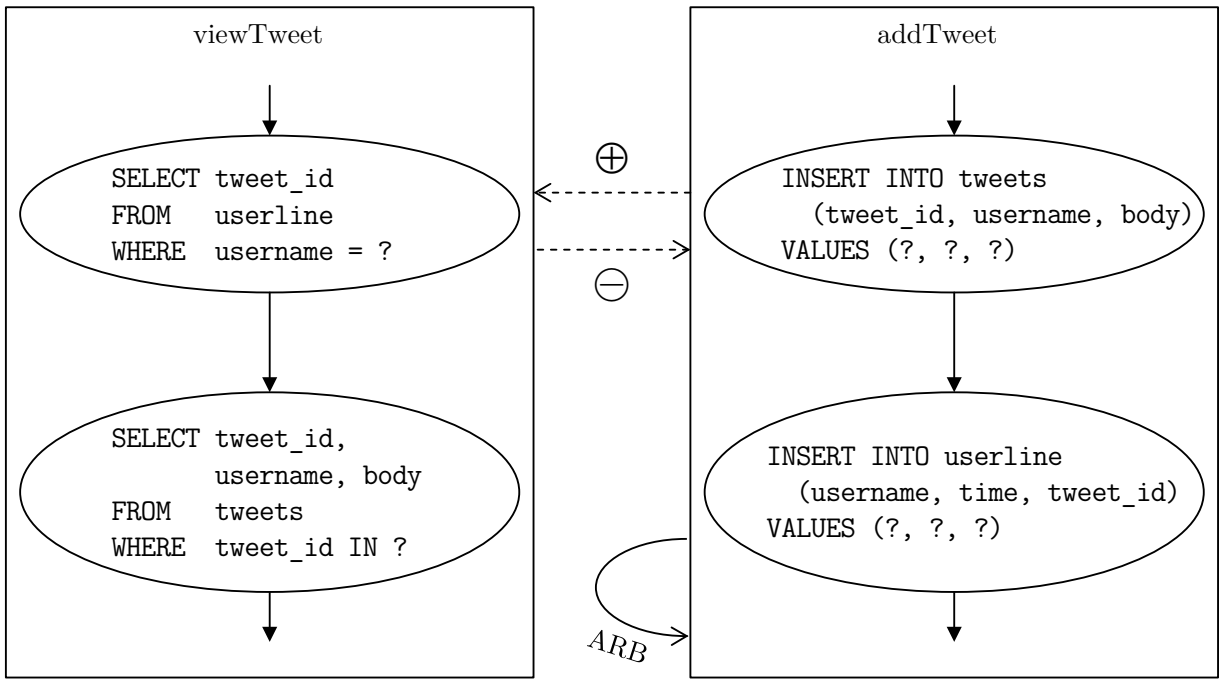
If the checker does not report a critical cycle for a program, all executions of the program are serializable.

4.3 Checking Serializability for Two Clients

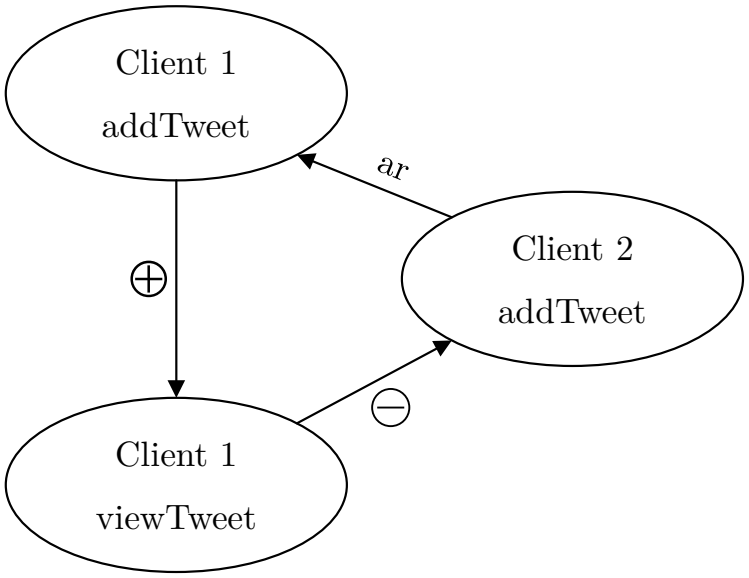
Whether a program is serializable for two clients is checked using a tool called ECChecker. The input to the tool consists of a specification of the operations that are offered by the data-store and all the events that may be executed on a client. It consists of the following parts:

System Specification: Specifies the semantics of the data-store, i.e. the set of the offered operations together with specifications for commutativity, absorption, asymmetric commutativity, synchronization and legality between these operations.

Transaction Graphs: A set of transactions where each transaction is a set of events combined with a program order between these. An event consists of an operation that is part of the system specification and an optional constraint.



(a) Over-approximation of the possible dependency serialization graphs for a program that has a transaction `addTweet` and a transaction `viewTweet`. The dotted lines indicate that there is either a dependency from `addTweet` to `viewTweet` or an anti-dependency from `viewTweet` to `addTweet`.



(b) A minimal critical cycle for the over-approximation of the possible dependency serialization graphs. However, this violation is a false positive: If there is arbitration between the `addTweet` transactions, there is also absorption, which resolves the cycle.

Figure 4.1: Over-approximation of the DSG and a minimal critical cycle.

Transaction Order: A set of edges. An edge (t_1, t_2) means that transaction t_2 can be executed directly after transaction t_1 on a client.

Global Constraint: The last part of the input is a constraint that must hold on both clients.

ECChecker checks for two clients all combinations of up to two transactions per client for serializability violations, which is enough in order to find all possible serializability violations between two clients. For each combination, the transactions are instantiated to the clients and the program order is unrolled such that in the end there is an acyclic graph representing the combination. This graph is transformed into a logical formula that encodes serializability violations for the specified data-store under the assumptions of causality and atomic visibility (cf. Section 4.1). The logical formula is checked for satisfiability using Z3[1], which is an SMT-solver. If the formula is unsatisfiable, no serializability violation exists for the combination. Otherwise, the model is used to build a graph that represents the violation.

The remainder of this section consists of the following subsections: In the next part, the transformation from the transaction graphs to the ECChecker input is described. Afterwards, we motivate and describe the enhancements and annotations that we introduced for reducing false positives.

4.3.1 Encoding of the ECChecker Input

All constraints are encoded using a simple expression language that supports three types: Integers, strings and booleans. Each expression consists of variables, constants, comparisons (equal, not equal) and boolean operations (not, and, or, implies). Variables are typed and are partitioned into global variables, client local variables and argument variables. A global variable has global scope, i.e. has the same value on all clients, a client local variable has client scope, i.e. has the same value on a single client, but may have another value on the other client, and an argument variable has event scope, i.e. an argument variable with the same name can have different values in different events.

Each event of the transaction graphs is encoded as a single operation in the system specification. The argument variables are numbered as follows: The integer argument variable with index 0 is used to refer to the client identifier, a client-unique number. The constraints and column updates are numbered for each event, so the remaining integer argument variables are used to relate to values used in constraints or updates. Additionally for queries, the boolean argument variable with index 0 reflects whether the query returned an empty result. For updates, the boolean argument variable with index 1 reflects whether at least one new row was inserted and the variable with index 2 whether at least one row was updated. We now describe the different parts of the system specification and roughly what we encode for the operations in each part:

Commutativity: Commutativity specifies for pair of operations o_1 and o_2 what constraint implies that executing $o_1; o_2$ leaves the data-store in the same state as executing $o_2; o_1$ and

[1] `https://github.com/Z3Prover`

vice versa. Therefore, commutativity is symmetric. Trivially, if both operations are queries or both operate on different tables, the operations always commute, so `true` is specified for these pairs.

For a query q and an update u, we specify `true` if u updates other columns than the ones that are selected from q. Otherwise we encode that u updates other rows than the ones that are selected by q.

For two updates u_1 and u_2, we encode that u_1 updates other rows than u_2 or that columns that are updated from both updates are set to the same value.

Absorption: For absorption we specify for two updates u_1 and u_2 which constraint implies that u_2 masks the effects of u_1, i.e. that executing $u_1; u_2$ leaves the data-store in the same state as executing only u_2. Clearly, this specification is not symmetric. If both updates operate on different tables, we specify `false`, otherwise we specify that u_2 updates a superset of the columns and a superset of the rows updated by u_1.

Asymmetric Commutativity: For two operations o_1 and o_2, asymmetric commutativity specifies the constraint that implies that executing $o_1; o_2$ leaves the database in the same state as executing $o_2; o_1$, but it does not imply that also the execution of $o_2; o_1$ leaves the database in the same state as $o_1; o_2$. We use asymmetric commutativity for some corner cases, e.g. if we have a delete event d and an update event u where u is a strict update, i.e. it does not insert new rows, then we can specify `true` for asymmetric commutativity of d and u. If u also includes rows that are deleted by d, moving u before d only means that these rows are updated before they are deleted, so the final state of the data-store is the same.

Another example is that if we have an upsert event i and a query event q, than we can specify that i, q commutes if i does not update any rows and q does not return any rows. So if q does not return any rows after new rows are added by i, q will also not return a result if it is executed before i. But note that if i also updates rows, this would not hold, as i could update a value that is used to constrain the rows in q.

Synchronization: Here we can specify for two operations o_1 and o_2 under what conditions the data-store guarantees that o_1 is causally after o_2 or o_2 is causally after o_1. Cassandra imposes an order on two lightweight transactions that operate on the same row, so for synchronization we specify the constraints under which two updates u_1, u_2 update the same row if both are executed in a lightweight transaction. This is not entirely sound, as all events ordered after o_1 and o_2 are also ordered by this synchronization specification, which is not the case in the real system.

Legality: Legality lets us specify constraints for two operations o_1, o_2 that must hold if o_2 is causally ordered after o_1. We use legality for specifications of operations that operate on tables where no deletes happen. If we have for example a query q and an update u that operate on a table with no deletes, we can specify that u cannot insert a new row if the same row was previously returned by q. Another example is that a query q_2 that is executed after q_1 cannot return an empty result if q_1 returned a non empty result and both queries selected

the same rows.

For the input of the transaction order, we specify that each transaction can happen after each other transaction. The transformation of the transaction graphs for the ECChecker input is also straightforward. As we have created an operation in the system specification for each event, we transform each event in the transaction graph to a node that refers to the created operation. Additionally, we encode for each column update and constraint of each event equalities and inequalities between argument variables. For example for client local variables that are annotated by the user in the program, we can assign the argument variables to a client local variable. This means that if we have for example the two events `SELECT * FROM users WHERE username = :u1` and `INSERT INTO users (username) VALUES (:u2)` and we know from the static analysis that the replacements for the binds u1 and u2 are equal on a client, we assign the argument variable representing u1 to the same client local variable as the argument variable representing u2. This ensures that the analysis assumes that the usernames are equal per client.

The edges in the transaction graph are transformed by specifying the expected result from the source event of the edge in the expression language used by ECChecker.

4.3.2 Annotations and Enhancements

In this section we motivate and describe the annotations that a programmer can use to reduce false positives. Also we show what kind of false positives can be excluded when enabling the implemented enhancements and additional specification parts other than commutativity. All the examples provided in the following sections are minimized versions of the transactions of the real examples. For the illustrations of the violations, we use CQL-queries with all bind markers replaced by concrete values, which should help understanding what the violation is.

4.3.2.1 Absorption and Schema Information

The example shown in Figure 4.2 consists of two transactions: One adds a new user to the database and the other retrieves a user from the database. As the `INSERT` events of client 0 and client 1 do not commute, we have arbitration between them. Also the query on client 0 might not see the password inserted in client 1, so there is an anti-dependency from `0_SELECT_2` to `0_INSERT_1`, which results in a potential violation.

The user of the analysis can specify the path to a file that contains the CQL-statements that are needed for creating the schema on the database. This file is parsed in the analysis so that the primary key columns of each table are known, which improves precision when defining the system specification. As `UPDATE` and `DELETE` statements must specify the partition key in the `WHERE` part, the analysis can deduce a set of primary key columns even in the absence of the file, but usually not enough to enable a precise analysis.

This false positive is resolved by taking absorption into account. We know from the schema infor-

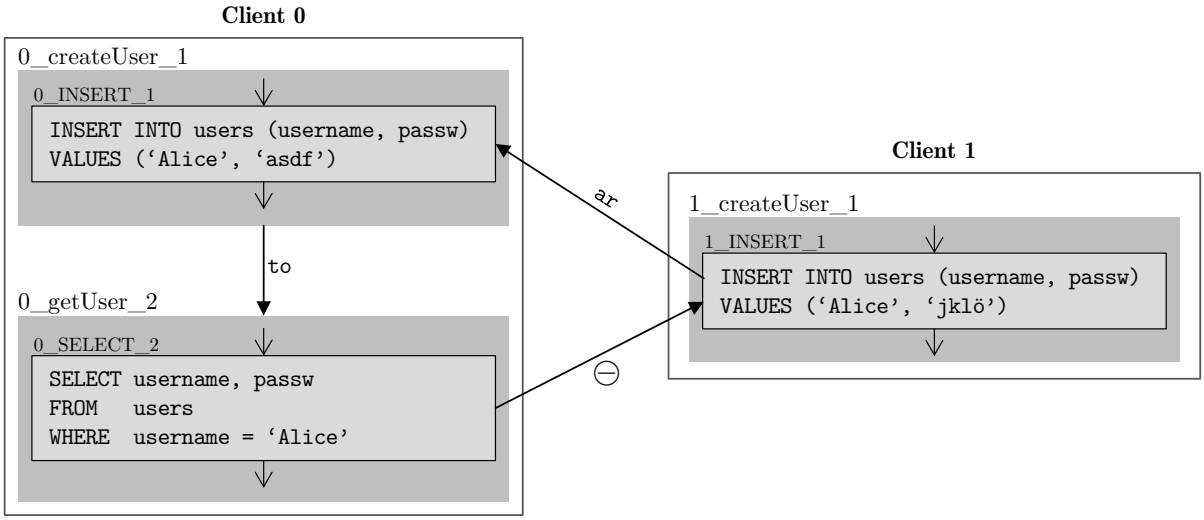


Figure 4.2: Using schema information and absorption removes this false positive.

mation that the primary key of the `users` table is the `username` column, so the event `0_INSERT_1` absorbs the event `1_INSERT_1`. Therefore, we can order the transaction of client 1 before the transactions of client 0 to obtain a serial schedule. Note that the cycle is also resolved if the `username` of 0_INSERT_1 is not equal to the one of `1_INSERT_1`: When the `username`s are unequal, the updates are commutative instead, so in this case there is no arbitration edge.

4.3.2.2 Program Order

In Figure 4.3 we have a transaction that is used to either create a new user or check the credentials. The transaction consists of two events that are both entry- and exit-nodes of the graph. If

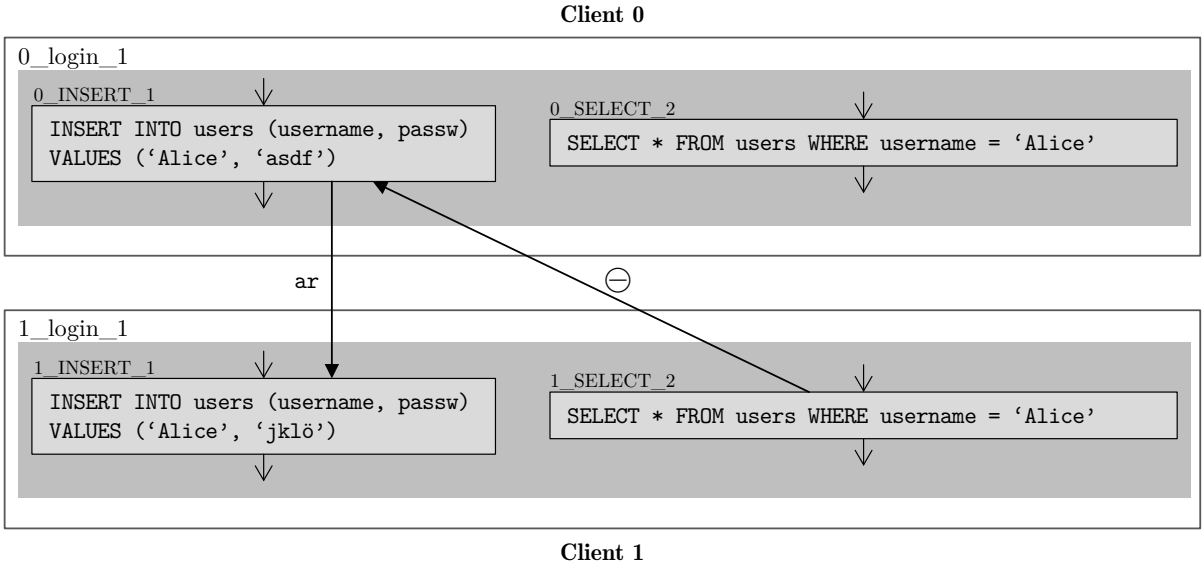


Figure 4.3: Encoding program order removes this false positive.

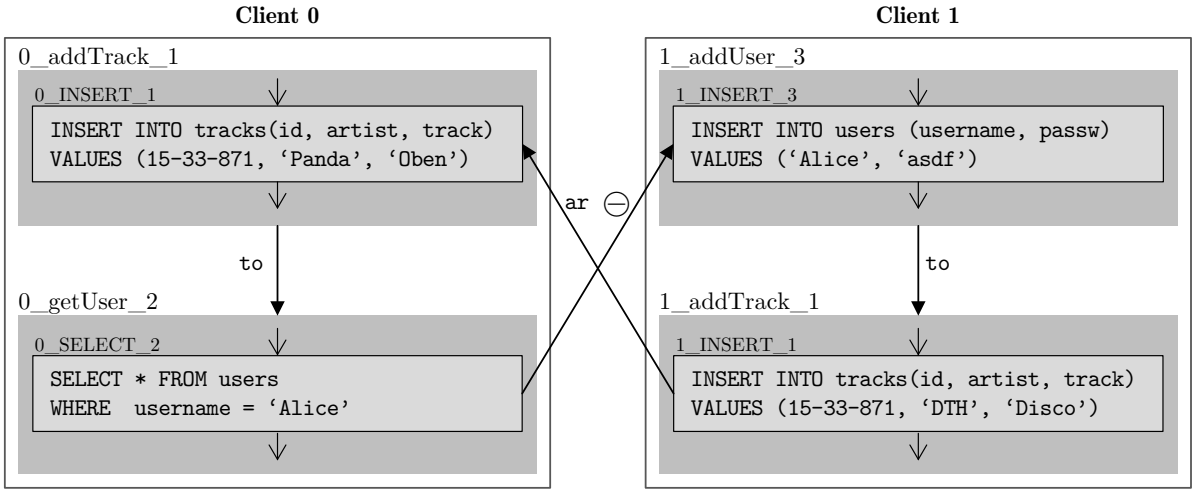


Figure 4.4: Encoding the uniqueness of the track identifiers removes this false positive.

the `1_SELECT_2` event does not observe the password "asdf", there is an anti-dependency between `1_SELECT_2` and `0_INSERT_1`. Also there is arbitration between `0_INSERT_1` and `1_INSERT_1`, so we have a possible violation.

When the program order is included in the analysis, the logical formula specifies additionally that either only the update or only the query event is executed on each client. As at most one edge exist between these transactions when program order is included, these transactions cannot form a critical cycle.

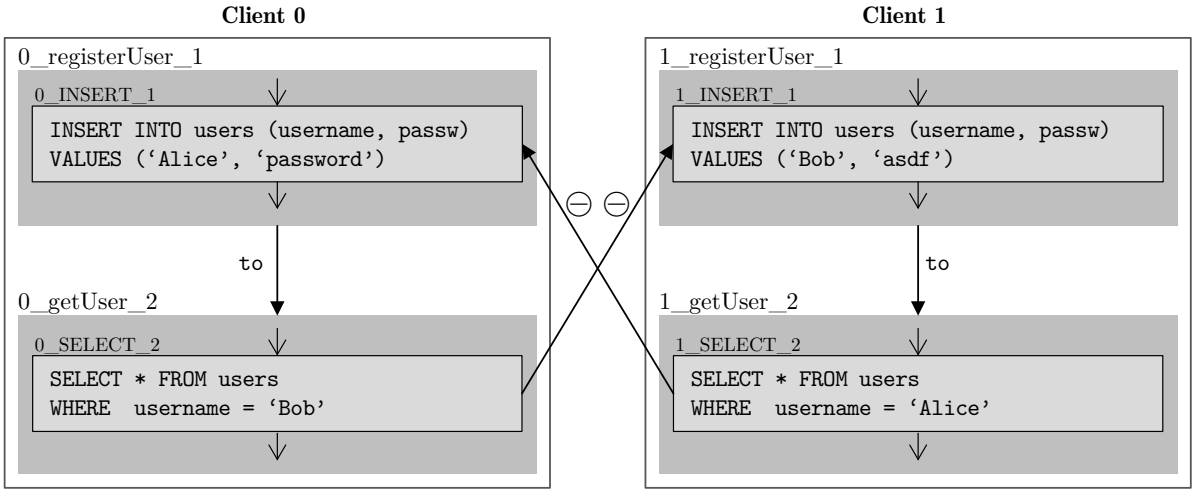
4.3.2.3 Value Analysis

In Figure 4.4, we have two transactions where one adds a new user and the other queries a user. There is an anti-dependency from the event `0_SELECT_2` to `1_INSERT_3`. The violation is completed by the other two transactions where in each a new track is added using the same identifiers, so there is an arbitration edge.

The value that is used as a track id in the `addTrack` transactions is a newly generated random universally unique identifier (UUID) [14]. A new random UUID is never equal to another new random UUID. Therefore if the value analysis is enabled, the static analysis deduces that the UUID used as the id in the event `0_INSERT_1` is unequal to the id from `1_INSERT_1`. When encoding this fact in the specification, the two `addTrack` transactions commute with each other, so there is no arbitration.

4.3.2.4 Client Local Variables

The analysis does not know any facts about the transaction arguments (e.g. the username of a transaction that registers a new user). In the code, the transaction arguments are the parameters of the method that spans the transaction and therefore, the arguments are set to top by the static




```
1  @Transaction
2  public void registerUser(String username, String password){
3    ClientLocalValues.set("username", username);
4    session.execute("INSERT INTO users (username, passw) VALUES (?, ?)",
5                    username, password);
6  }
7
8  @Transaction
9  public ResultSet getUser(String username){
10   ClientLocalValues.set("username", username);
11   return session.execute("SELECT * FROM users WHERE username = ?", username);
12 }
```

Figure 4.5: Using client local variables removes the false positive on top. In the code on the bottom, the parameter `username` is annotated on line 3 and line 10. Therefore, the analysis assumes for this code that the usernames are equal in both transactions.

analysis. In Figure 4.5, we have a `register` and a `getUser` transaction. The violation occurs, as on both clients a new user is registered first and afterwards, the user that was created on the other client is queried. So there is a possible anti-dependency from the `getUser` to the `register` transaction which results in a critical cycle.

When inspecting the program manually, one can see that the username for the `getUser` is extracted from a session and must therefore be equal to the username that was registered in the `registerUser` transaction. This resolves the false positive, as if the usernames are not equal on both clients, all the transactions from client 0 commute with all transactions of client 1 and if the usernames are equal, we again have absorption (cf. Section 4.3.2.1).

Such equalities can be annotated in the program by calling the `set` method of the `ClientLocalValues` class. Hence, the analysis can deduce for the program in Figure 4.5 that the usernames are the same and does not report the false positive.

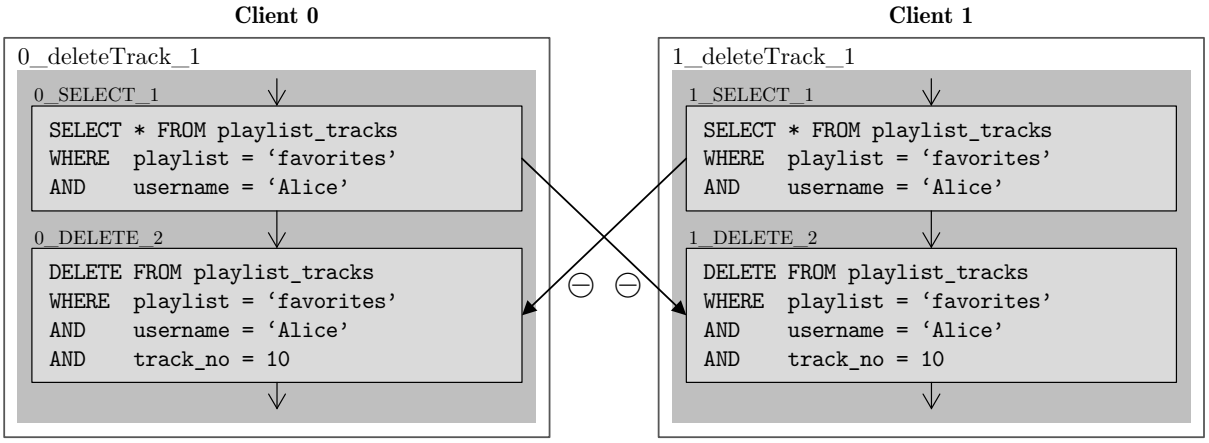


Figure 4.6: Declaring client local variables as unique per client, this violation can be removed.

4.3.2.5 Unique Client Local Variables

In the following example, the same track is removed from the same playlist on both clients. When the track that is deleted is included in the first query, we have two anti-dependencies from the `SELECT_1` to the `DELETE_2` events which is a critical cycle.

In some applications it is ensured that the user cannot login on multiple clients at the same time. If this is the case, the violation shown in Figure 4.6 is a false positive. Therefore, an option can be set in the analysis that ensures that client local variables are unique per client. In this example, this would mean that the username cannot be equal on both clients. As the username is part of the primary key of the `playlist_tracks` table, the events of one client commute with the events of the other client, so there are no anti-dependencies in this example when unique client local variables are enabled.

4.3.2.6 Display Code

In most of the analyzed examples, some results of queries are directly displayed to the user, so there is no application logic that builds on these results. Usually it is acceptable if these results are not strongly consistent. Such parts can be annotated in the program and an option can be set that excludes such display code from being checked. If all events of a transaction are only executed to get some data that is directly displayed to the user, the whole transaction can be marked as display code. If single queries or subsets of columns in a query are used only for displaying, a comment can be added to the CQL-query that specifies these columns.

In the violation in Figure 4.7 we have one transaction on each client in which a playlist for a user is deleted. On each user row, there is a set named `playlists` that contains the names of the playlists of the user. In the `deletePlaylist` transaction, the user is loaded from the database for application logic, but the set with the playlist names is only used for displaying. It is therefore not

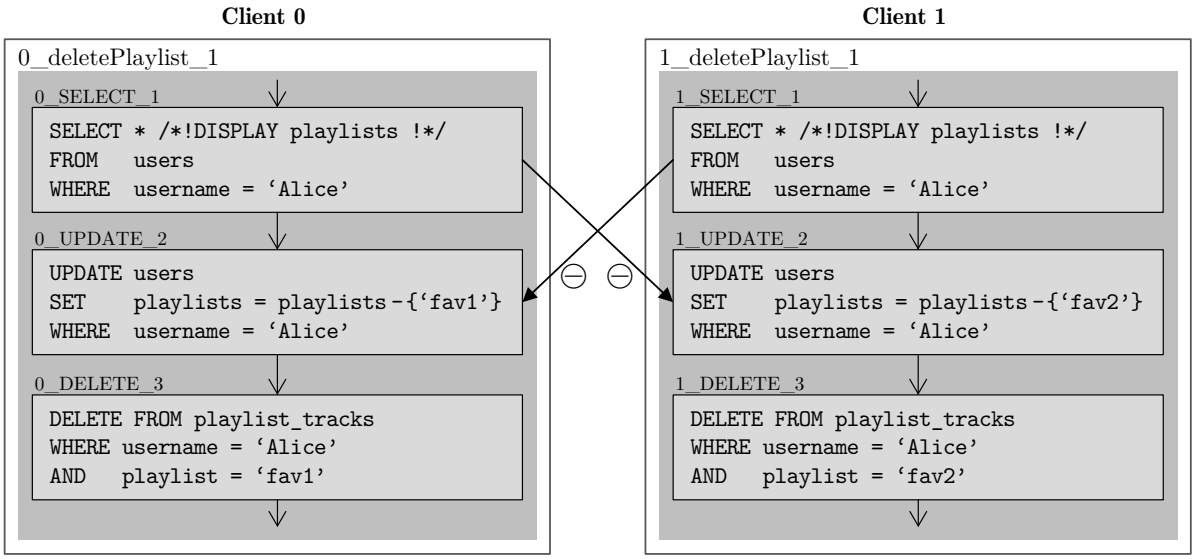


Figure 4.7: When display code is excluded from the analysis, this violation can be removed.

a real violation that if two playlists are deleted simultaneously on two clients, both clients may still display the playlist deleted on the other client. When display code is enabled, the `SELECT_1` event commutes with the `UPDATE_2` event, as the update only removes a value from a display column.

4.3.2.7 Strict Updates

In the example shown in Figure 4.8, we have two clients that execute a chat application. Client 0 adds a message to the "Cassandra" chat room and executes a query to read the user details afterwards. Client 1 leaves the "Java" chat room first and then reads the messages from the "Cassandra" chat room. The chat rooms where a user participates in are stored as a set on the users

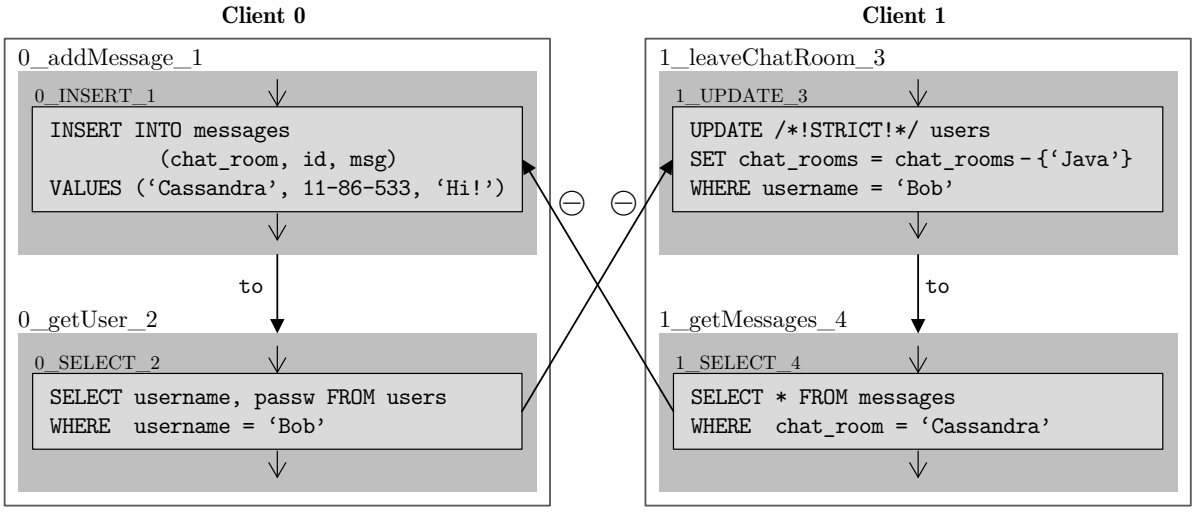


Figure 4.8: This violation is only possible when the `1_UPDATE_3` event may insert a new row. If this is not the case, it can be annotated as `strict`, which removes the violation.

row in the `users` table.

As client 1 may not see the message posted by client 0, there is an anti-dependency from `1_SELECT_4` to `0_INSERT_1`. Due to the fact that an `UPDATE` event may also add new rows in a table, it is possible that `1_UPDATE_3` adds a new user row that is not observed in `0_SELECT_2`, which is why there is a second anti-dependency.

The programmer may know that the user row exists when a `leaveChatRoom` transaction is executed (e.g. because the user had to login first). If the row already exists, the `1_UPDATE_3` event cannot add a new row, which would mean that the `1_UPDATE_3` commutes with `0_SELECT_2`. That an `UPDATE` does not insert any rows can be specified by adding a comment `/*!STRICT!*/` to the CQL-query. Therefore if this annotation is enabled, the analysis does not report the false positive.

4.3.2.8 Program Order Constraints

The example shown in Figure 4.9 is from a Twitter application. Client 0 first registers "Alice" as a new user and then checks if she has followers. On client 1, "Bob" wants to follow "Alice". The `follow` transaction first checks if both user exist and only if this is the case, a new row is added to the `followers` table. The violation that is reported happens if client 1 does not see "Alice" in the `users` table and "Alice" does not see that "Bob" is following her. So there are two anti-dependencies.

If the constraints on the program order are also considered, we can resolve this false positive. If there is an anti-dependency from `1_SELECT_4` to `0_INSERT_1`, client 1 does not see "Alice" yet, so `1_INSERT_5` is not executed. If client 1 sees "Alice", there is no anti-dependency in the first case.

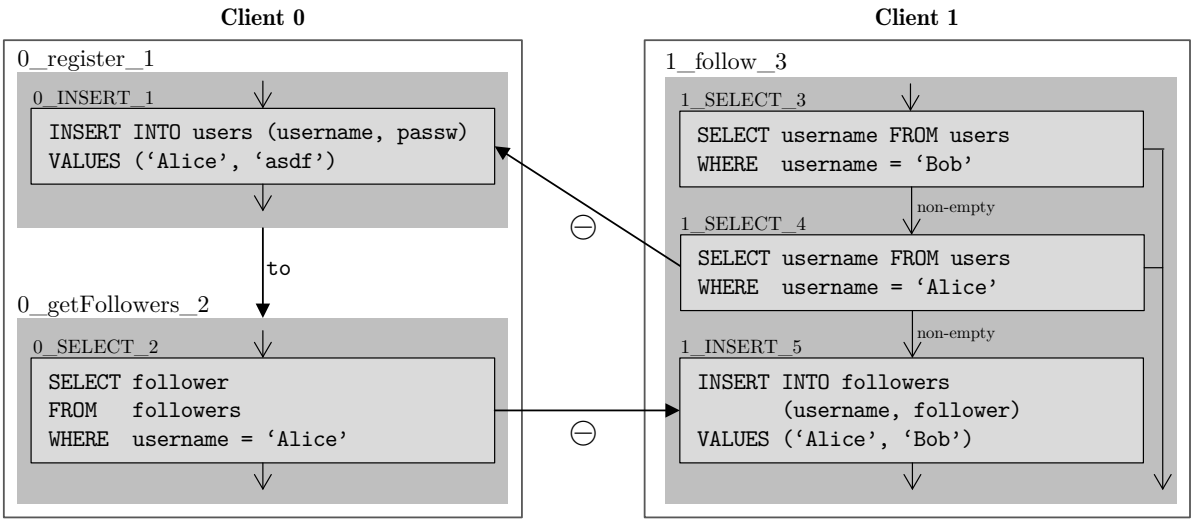


Figure 4.9: Encoding the constraints on the program order removes this false positive.

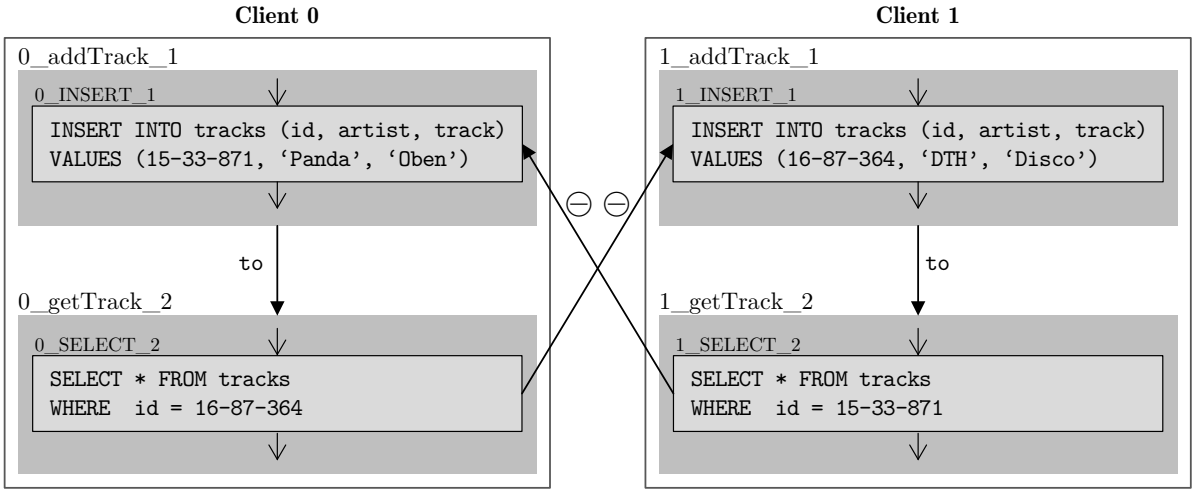


Figure 4.10: When specifying that `1_SELECT_2` asymmetrically commutes with `0_INSERT_1`, this violation can be removed.

4.3.2.9 Asymmetric Commutativity

We have two transactions in the example shown in Figure 4.10: The first transaction adds a new track to the database. A random UUID (cf. Section 4.3.2.3) is generated as the identifier of the track. The other transaction queries the database to obtain the details for a given track. The possible violation occurs as both clients add a new track and then request the details of the track the other client has added, but do not see the track yet.

In reality, the clients cannot behave in this way. As a UUID is unique and random, client 0 cannot know which UUID is used by client 1 when inserting a new track (except when client 0 has another channel than the database to communicate with client 1). If the programmer sets the option that no such side channel exist, the analysis specifies for this example that the event `SELECT_2` does commute asymmetrically with `INSERT_1` if the events happen on different clients. This is due to the fact that `SELECT_2` must read other rows than the one that may be added by a succeeding `INSERT_1`, as the identifier that `INSERT_1` uses cannot be known in `SELECT_2`.

Another false positive that can be removed by asymmetric commutativity is illustrated in Figure 4.11: On client 0, "Alice" is registered first. Afterwards, the tracks from the "favorites" playlist are queried. Client 1 wants to delete the "favorites" playlist. First, the user details are loaded and only if the result is not empty, the playlist is removed.

We can specify that the events `1_SELECT_3` and `0_INSERT_1` commute in this order if the query does return a non-empty result. As the query can return at most one row (i.e. all primary key columns are constrained), a non-empty result implies that the `0_INSERT_1` can only insert a new row with a different primary key.

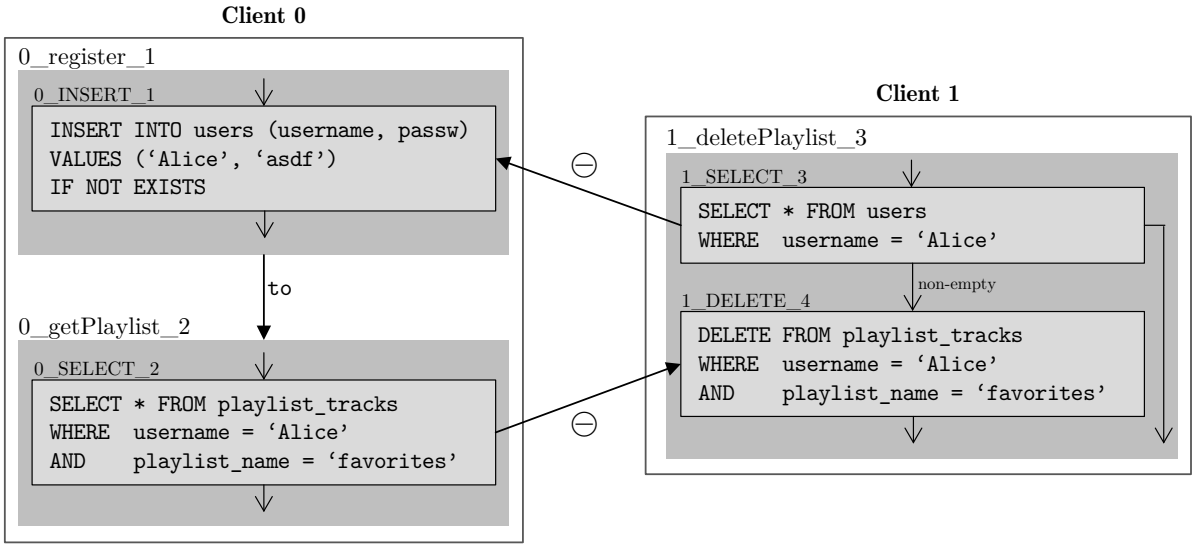


Figure 4.11: Another false positive that can be removed using asymmetric commutativity.

So the false positive can be resolved with asymmetric commutativity: If `1_SELECT` returns an empty result, the `1_DELETE_4` event is not executed, otherwise `1_SELECT` asymmetrically commutes with `0_INSERT_1`, which also breaks the cycle.

4.3.2.10 Synchronization

The example shown in Figure 4.12 makes use of lightweight transactions. Client 1 creates a new chat room if no other room with the same name already exists. Client 0 joins the chat room and

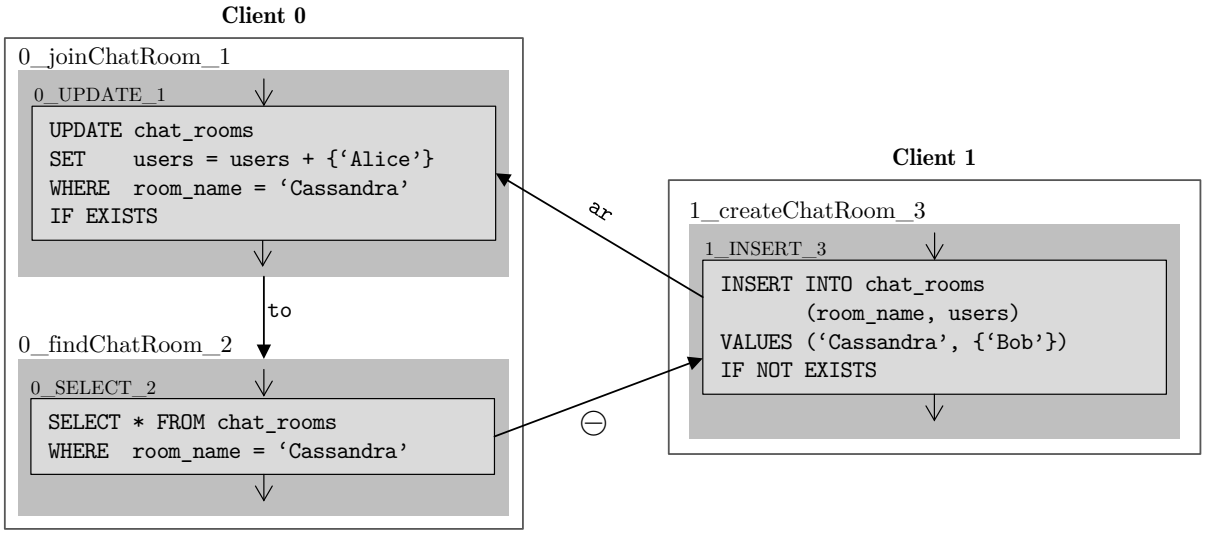


Figure 4.12: This false positive can be removed when encoding that two LWTs on the same row synchronize on each other.

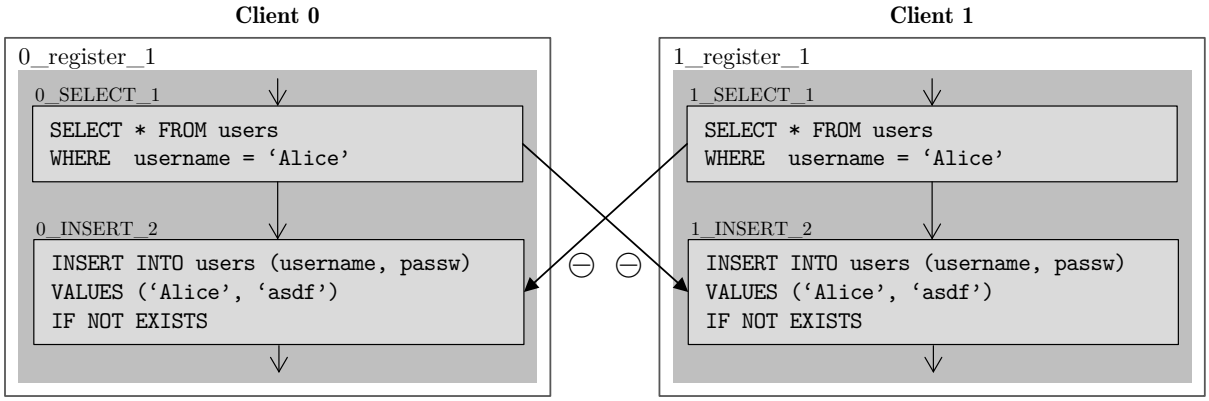


Figure 4.13: This real violation is not reported when synchronization is specified.

then queries all information for the room. Joining is also implemented using a LWT, as otherwise a room would be created when a user joins and the room does not exist (due to the upsert semantics of the `UPDATE`).

By specifying that `0_UPDATE_1` and `1_INSERT_3` synchronize if both use the same room name, we can break the critical cycle. Synchronization introduces a causality edge either from `1_createChatRoom_3` to `0_joinChatRoom_1`, in which case the anti-dependency does not exist, or from `0_joinChatRoom_1` to `1_createChatRoom_3`, in which case arbitration is not possible in the opposite direction.

The problem is that it is not always sound to introduce causality edges. The real violation illustrated in Figure 4.13 is not reported when the synchronization specification is enabled. The `INSERT` events either commute or synchronize. If they synchronize, it is still possible that both query events get an empty result, which is not serializable. Nonetheless, the violation is not reported as the transactions synchronize.

4.3.2.11 Legality

We use legality for tables where data is only inserted or modified, but never deleted. The `users` table in Figure 4.14 is such an example. Both clients execute the same transaction: First, the user joins a new chat room. Afterwards, both clients query the user information from the database. If the `SELECT_2` events do not return a result, an anti-dependency may exist to the `UPDATE_1` of the other client, which results in a possible violation.

Using legality, the analysis can specify that if the `UPDATE_1` either updates an existing row or inserts a new row, the `SELECT_2` cannot return an empty result if it is ordered after the `UPDATE_1` event in the causality order. This breaks the critical cycle as either the `UPDATE` does nothing or the `SELECT` returns a non-empty row, so both anti-dependencies are not feasible when legality is also considered.

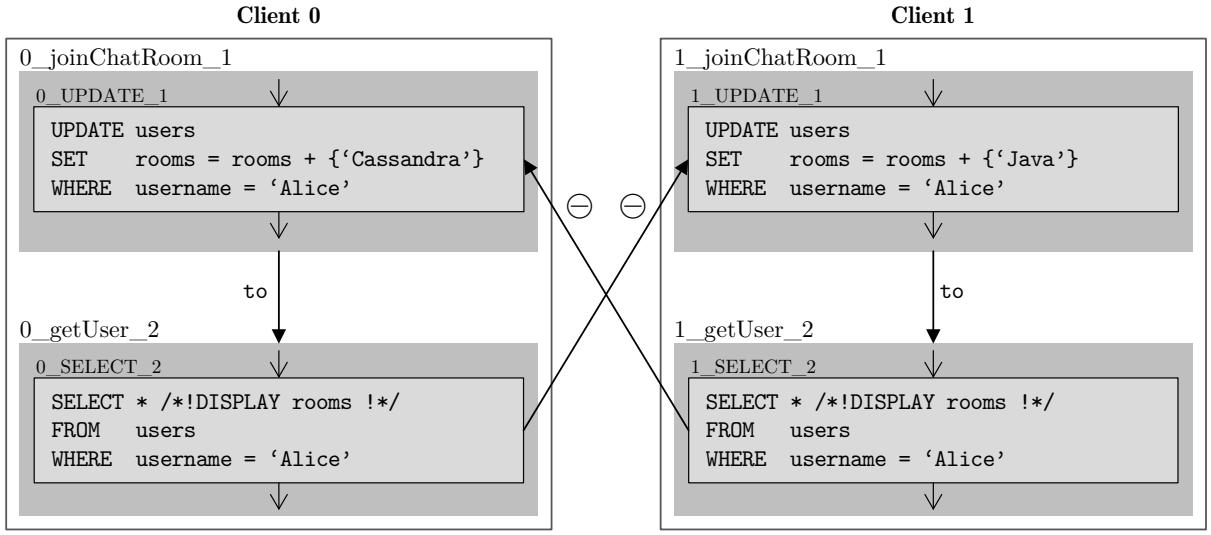


Figure 4.14: A false positive that is not reported when specifying legality.

The violation could also be resolved manually by the user, as he could use the strict update annotation (see Section 4.3.2.7) on the `UPDATE` CQL-query. This is no longer necessary for this case, as the analysis can resolve this false positive automatically using legality.

The example shown in Figure 4.15 was also used in the Section 4.3.2.10. If it would be the case that no deletes happen on the `chat_rooms` table, it can also be solved with the help of legality, so that synchronization is not even necessary. We can specify that if `0_UPDATE_1` updates a row,

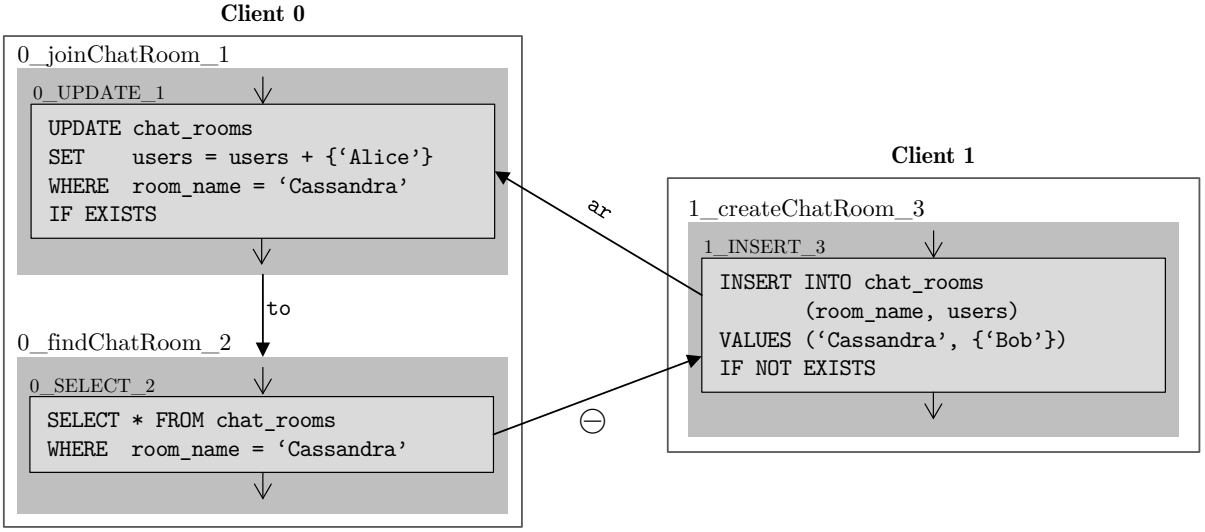


Figure 4.15: A violation that could be removed using legality if no deletes happen on the table `chat_rooms`.

then `0_SELECT_2` cannot return an empty result. So either the arbitration edge is not possible (if the `0_UPDATE_1` event does not update a row) or otherwise the anti-dependency is not possible (as `0_SELECT_0` does not return an empty result).

5 Evaluation

We evaluated the static analysis and both approaches for checking serializability on twelve programs that we found in public code repositories. All measurements were taken on a system with a 1.9 GHz 2-Core CPU (Intel i7-3517U) and 8 GB of RAM running Windows 10. The following section describes the examples we used for evaluation. Afterwards, the results of the evaluation for each part are summarized in a section.

5.1 Examples

In this section we describe the example programs found in public code repositories that we used to evaluate the static analysis. For all the examples we annotated transactions, display code and strict updates where appropriate. Additionally, we annotated username variables as client local values in the examples that have users. If further modifications were necessary, these are described below.

cassandra-lock[1]**:** This project provides a library that can be used for distributed locking. Locking is implemented using Cassandras lightweight transactions.

cassandra-twitter[2]**:** This project is a twitter clone that implements transactions for registering new users, following other users, adding and displaying tweets. Interaction with the user happens on the command line. To simplify the analysis, we have rewritten the code that checks if a user exists. In the original, a list of the rows returned from the database is created. If this list is empty, the user does not exist. We use the `isExhausted` function directly on the `ResultSet` to check whether a user exists.

cassatwitter[3]**:** Another implementation of a twitter clone. Similar to cassandra-twitter.

cassieq[4]**:** An implementation of a distributed queue. In some methods, lambda functions are used. As we cannot analyze the code of lambda functions (see Section 3.2.6), we have removed the lambdas and inserted the code directly in the method instead to enable a sound analysis. At some places in the code, an exception is thrown if some result of a query is required but the returned result is empty. In order to simplify the flow of the static analysis, we have moved the throwing of the exceptions inside the method that executes the query.

[1] `https://github.com/dekses/cassandra-lock`
[2] `https://github.com/edmundophie/cassandra-twitter`
[3] `https://bitbucket.org/ClearingPath/cassatwitter`
[4] `https://github.com/paradoxical-io/cassieq`

currency-exchange[5]**:** This project provides an overview of trade activities. New trades can be added to the system and trade data can be queried. Input to the `saveTrade` transaction is an object of class `Trade`. At the begin of the transaction, a new random UUID is set on the argument as the identifier of the trade. As the transaction argument is set to top by the analysis, the identifier is also abstracted with top in this case.
We manually cloned the trade object in the beginning of the transaction, so that the new UUID can also be abstracted as such by the analysis.

datastax-queueing[6]**:** This is an implementation of a single threaded queuing system.

killrchat[7]**:** This program implements a chat application. As the original implementation uses an unsupported driver for the communication with Cassandra, we have rewritten the application so that it makes use of the Datastax driver.

playlist[8]**:** This is an implementation of a playlist service. Users can add music tracks and combine them into playlists.

roomstore[9]**:** This is a implementation of a bot that logs all messages that are sent on the different channels of an IRC server. If the bot receives a message of a defined format (e.g. "∼ today"), it will reply with the requested information (e.g. a list of all messages posted today).

shopping-cart[10]**:** This is an implementation of an online shop. Cassandra is only used for querying product information.

simple-twitter: Inspired by cassandra-twitter, we implemented our own twitter clone.

twissandra[11]**:** This is another project that implements a twitter clone. We removed code that was never used (`afterBatchWorks` in `MyBatch`). Otherwise, an extension of the static analysis would have been necessary to handle twissandra precisely.

The analysis should be sound on the projects with the modifications that we have described above. Some of the examples are web applications that are multi-threaded, but these programs do not share global state. Reflection and asynchronous method calls are not used at all.

5.2 Building the Transaction Graphs

We have measured the performance of the different phases of the static analysis that builds the transaction graphs (see Section 3.2). Table 5.2 gives an overview of the runtime information. With the exception of cassieq, the static analysis needs at most 10 seconds to complete. In cassieq, there is one transaction with a lot of method calls, which results in a huge super-graph with more than

[5] `https://github.com/Haiyan2/Trade`
[6] `https://github.com/PatrickCallaghan/datastax-queueing-demo`
[7] `https://github.com/doanduyhai/killrchat`
[8] `https://github.com/DataStaxDocs/playlist`
[9] `https://github.com/mebigfatguy/roomstore`
[10] `https://github.com/nikhilswagle/Shopping_Cart_Angular_Cassandra`
[11] `https://github.com/cilesizemre/twissandra`

Example	# Transactions	# Violations	# Violations ignoring Display-Code
cassandra-lock	3	0	0
cassandra-twitter	5	5	3
cassatwitter	7	6	3
cassieq	18	16	13
currency-exchange	2	2	0
datastax-queueing	2	2	2
killrchat	11	11	3
playlist	11	11	6
roomstore	5	5	0
shopping-cart	6	0	0
simple-twitter	4	3	1
twissandra	7	6	2

Table 5.1: Number of transactions that are part of at least one critical cycle in the over-approximation of the dependency serialization graphs. Statements that are only used to display some information are excluded from one evaluation.

6'000 nodes and 20'000 edges. Also, there is a catch statement in the first method, which results in a lot of edges when sound exception handling is enabled. Therefore, this example takes longer to analyze.

5.3 Over-Approximation of the Dependency Serialization Graph

Table 5.1 contains the number of transactions t for which a critical cycle exists in the over-approximation of the dependency serialization graph that includes t. The runtime of the checker is less than one second for all the examples. We also evaluated the examples once with excluding all the statements whose results are only used for displaying.

This approach is rather imprecise, as two statements normally do not commute if both operate on the same table. The examples that have zero critical cycles do either only write data (cassandra-lock) or only read data (shopping-cart).

5.4 Checking Serializability for Two Clients

In this section, we evaluate the serializability checking for all combinations of up to two transactions on two clients using ECChecker. In the first section, we provide a classification of the reported violations and also show examples of real violations. In the following section, we compare the results from ECChecker with the results from a trivial analysis. We also analyzed the impact of the annotations and enhancements on the number of false positives. The results are listed in the last section.

We excluded parts of cassieq for the evaluation. The implementation of the queues is quite complicated, which results in transactions with lot of events (e.g. more than 60 for the `getMessage`

Example	# Classes / # Methods / # Transactions	Soot Setup	CFG Transformation	Field Values Collection	Transaction Graph	Parsing	Total
cassandra-lock	6 / 30 / 3	3.74	1.62	1.25	0.04 (0.04)	0.53	7.18 (7.18)
cassandra-twitter	1 / 19 / 5	4.08	1.32	0.63	0.09 (0.11)	0.45	6.57 (6.59)
cassatwitter	6 / 46 / 7	3.72	1.87	0.79	0.34 (0.76)	0.56	7.28 (7.70)
cassieq	280 / 1294 / 17	4.54	6.97	3.78	41.08 (283.27)	0.93	57.30 (299.49)
currency-exchange	21 / 96 / 2	3.98	1.89	1.30	0.05 (0.06)	0.35	7.57 (7.58)
datastax-queueing	6 / 56 / 2	3.56	1.47	0.68	0.12 (0.12)	0.35	6.18 (6.18)
killrchat	72 / 401 / 11	5.15	2.91	1.34	0.16 (0.17)	0.49	10.05 (10.06)
playlist	24 / 122 / 11	4.15	2.22	0.90	0.51 (0.59)	0.75	8.53 (8.61)
roomstore	9 / 47 / 5	3.81	1.84	1.01	0.08 (0.08)	0.50	7.24 (7.24)
shopping-cart	30 / 127 / 6	0.63	1.26	0.51	0.16 (0.12)	0.34	2.90 (2.86)
simple-twitter	2 / 13 / 4	4.30	0.80	0.81	0.07 (0.07)	0.96	6.94 (6.94)
twissandra	35 / 107 / 7	4.43	1.52	0.78	0.16 (0.16)	0.56	7.45 (7.45)

Table 5.2: Overview of the runtime (in seconds) of the different parts of the static analysis. The numbers in parentheses are the runtimes if sound exception handling was enabled.

transaction). This lead sometimes to timeouts in the SMT-solver on the one hand, but it was also not possible to classify the reported violations due to the complexity. Therefore we only evaluated the transactions that are related to user management, so that checking and classification was possible for all combinations.

5.4.1 Reported Violations and Runtime

We evaluated the output of ECChecker on the examples. Table 5.3 shows an overview of the results. We run the analysis on each example at most four times with different options. If display code (cf. Section 4.3.2.6) is included in the analysis, the options list contains "display". If the analysis assumed that client local variables are unique on different clients (cf. Section 4.3.2.5), "unique" is listed in the options. If a combination of options is missing in the table, this means that the example makes no use of the specific annotation.

ECChecker first checks all combinations that consist of two transactions. Afterwards, the combinations with three and then the ones with four transactions are checked. Combinations that include a smaller combination which was already reported as a violation are ignored. Symmetric combinations are also checked only once. For each combination, a trivial analysis checks first if a critical cycle can exist using the same approach as we used for the over-approximation of the dependency serialization graph (cf. Section 4.2). Only if this is the case, a logical formula is built and checked using the SMT-solver.

Due to this bottom-up checking, the less violations exist for a given example, the more formulas are checked using the SMT-solver. The number in the column "SMT Checks" indicates how many formulas were checked for the given example. The "SER" column contains the number of combinations consisting of 4 transactions that do not contain a violation (i.e. are serializable). The "Violations" column contains the reported violations, i.e. the minimal 2-, 3-, and 4-combinations. The runtime increases linearly with the number of SMT-checks. All runs completed in less than 2 minutes.

We classified all reported violations into the four categories false positive (FP), error (ERR), warning (W), and harmless (H). A false positive is a violation that cannot occur in practice. The other three categories are used to distinguish the violations that can lead to non-serializable executions. An error is a violation that is likely to be fixed by a programmer. A warning is used for violations where it is debatable if they are acceptable. A harmless violation is used if we think that a non-serializable execution is acceptable.

When display-code is not included, the analysis did report 6 false positives for killrchat. For cassandra-twitter, cassatwitter, cassieq, datastax-queueing, playlist, simple-twitter and twissandra, we found serious errors and some warnings. When display code is also considered, the analysis reports a lot of harmless violations. Also, we have some false positives for killrchat and one for cassandra-twitter. With the unique option and without display code, we could show that killrchat,

Example	Options	Time [s]	SMT Checks	SER	Violations	FP	ERR	W	H
cassandra-lock		0.2	0	0	0	0	0	0	0
cassandra-twitter		2.3	5	2	2	0	1	1	0
cassandra-twitter	unique	6.4	15	8	2	0	0	1	1
cassandra-twitter	display	8.4	25	10	8	1	1	1	5
cassandra-twitter	display unique	12.4	39	22	5	1	0	1	3
cassatwitter		3.1	2	0	2	0	1	1	0
cassatwitter	unique	7.4	5	3	0	0	0	0	0
cassatwitter	display	9	16	4	9	0	1	1	7
cassatwitter	display unique	16.6	27	19	1	0	0	0	1
cassieq*		12.3	103	65	6	0	5	0	1
cassieq*	unique	46.2	391	290	0	0	0	0	0
cassieq*	display	17.7	147	98	10	0	5	0	5
cassieq*	display unique	54.3	513	386	10	0	0	0	10
currency-exchange		0.1	0	0	0	0	0	0	0
currency-exchange	display	1.2	3	1	1	0	0	0	1
datastax-queueing		2	3	0	3	0	2	0	1
killrchat		5.5	23	15	6	6	0	0	0
killrchat	unique	5	23	21	0	0	0	0	0
killrchat	display	102.3	641	307	267	36	0	22	209
killrchat	display unique	99.4	731	507	148	21	0	11	116
playlist		26.3	126	79	5	0	0	4	1
playlist	unique	62	319	228	0	0	0	0	0
playlist	display	31.1	131	56	51	0	0	4	47
playlist	display unique	99	507	380	23	0	0	0	23
roomstore		0.2	0	0	0	0	0	0	0
roomstore	display	3.2	4	0	4	0	0	4	0
shopping-cart		0.1	0	0	0	0	0	0	0
simple-twitter		0.8	1	0	1	0	1	0	0
simple-twitter	unique	0.8	3	1	0	0	0	0	0
simple-twitter	display	1.7	4	1	2	0	1	0	1
simple-twitter	display unique	2.1	8	4	1	0	0	0	1
twissandra		3	3	0	2	0	0	2	0
twissandra	unique	3.2	3	1	1	0	0	1	0
twissandra	display	14.1	25	5	14	0	0	4	10
twissandra	display unique	14.6	29	17	5	0	0	2	3

Table 5.3: Overview of the results from checking serializability using ECChecker.

playlist, cassiq, cassatwitter and simple-twitter do not have a single violation. This means, that as long as a user is not logged in multiple times in the application, no serialization errors are expected.

The following illustrations show typical false positives and real errors that we found. Only the necessary parts of the real transactions are illustrated for improving readability. That is, all the events in a transaction that are not necessary for the violation are omitted.

5.4.1.1 False Positive in Cassandra-Twitter

Figure 5.1 illustrates the violation from cassandra-twitter that is a false positive. Client 0 is used by "Alice". She first adds a new tweet and than wants to follow the user "Bob". When a new tweet is added, the tweet is also added to the timeline of all followers (`0_SELECT_1` and `0_INSERT_2`). When a user wants to follow another user, there is an initial check if both users exist and only if this is the case, the follower is inserted. Client 1 is used by "Bob". In the first transaction he registers himself and then requests the timeline in the second transaction.

In the violation, we have an anti-dependency from the user check from left to right which means that the query `0_SELECT_4` returned an empty result. Another anti-dependency is from the query

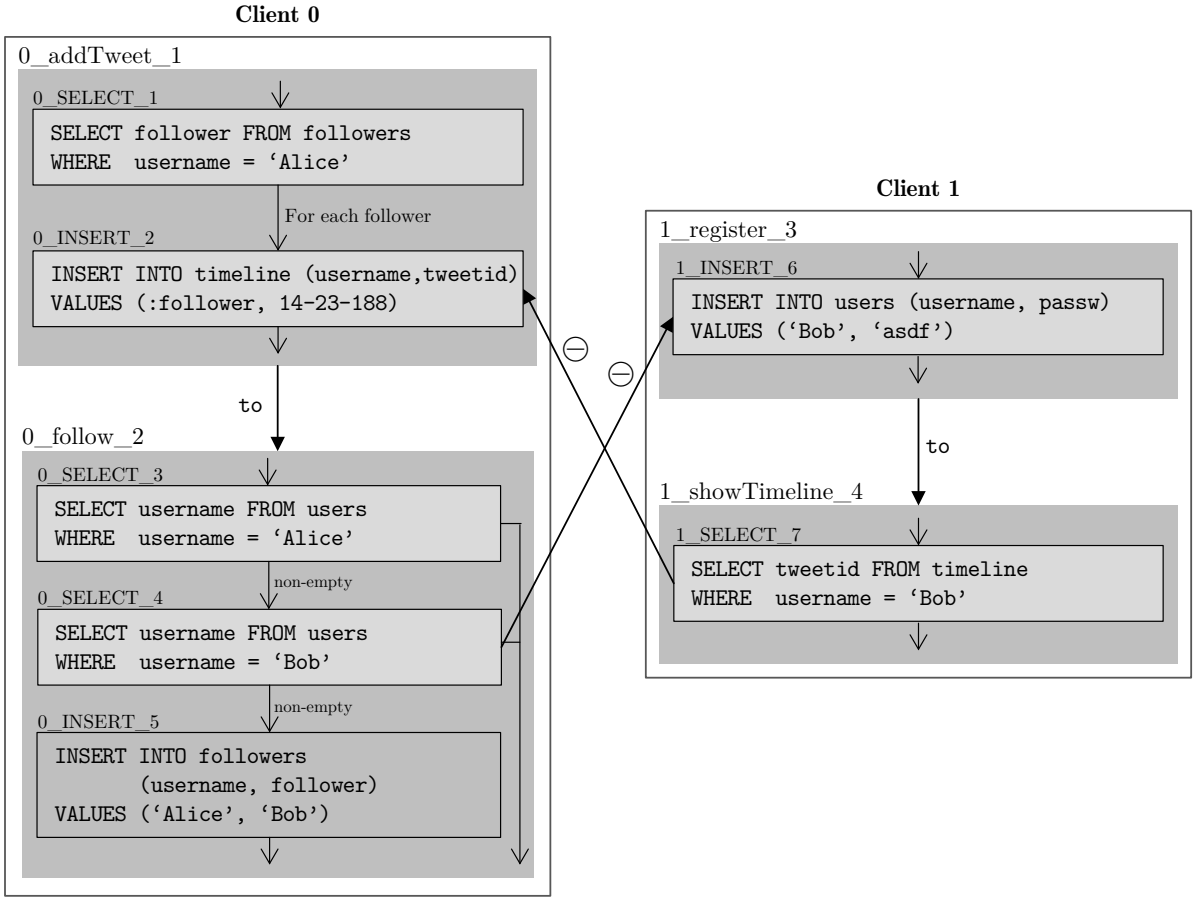


Figure 5.1: False positive that is reported for cassandra-twitter.

of the timeline to the insertion of the new tweet in the timeline, therefore the new tweet is added to the timeline of "Bob" in the `0_addTweet_1` transaction. This means that "Bob" has to be following "Alice" and client 0 has already observed this. Due to causality and the design of the `follow` transaction, the query `0_SELECT_4` cannot return an empty result on client 0, as a user is never deleted and the existence check for "Bob" happened before the insertion in the `followers` table.

In order to prevent the reporting of this false positive, the analysis would have to track somehow that the usernames that are returned from the query `0_SELECT_1` have to exist in the `users` table at this point.

5.4.1.2 False Positives in Killrchat

If display code is not included in the analysis, ECChecker reports 6 false positives that all base on the same reason. One of these false positives is shown in Figure 5.2. Client 0 first posts a new message in a chat room and then requests the details of the user. On client 1, a new user is created first and afterwards, the chat room where client 0 posted the message is deleted. We have arbitration from `1_DELETE_4` to `0_INSERT_1`. Also we have an anti-dependency from `0_SELECT_2` to `1_INSERT_3`, which means that the user query returned an empty result. A new message can only be posted if the user is logged in, which can only happen if the corresponding row exists in the `users` table. Therefore `0_SELECT_2` cannot return an empty row, so the anti-dependency is not possible.

The false positive could be avoided if it would be possible to specify properties for events that hold if the event happens after some transaction. This would enable e.g. the programmer to annotate that the query `0_SELECT_2` cannot return an empty result when executed after the `postMessage` transaction. We do have only six of these false positives as for some examples the analysis can deduce exactly the property that a row exists after a given transaction using legality. E.g. in the

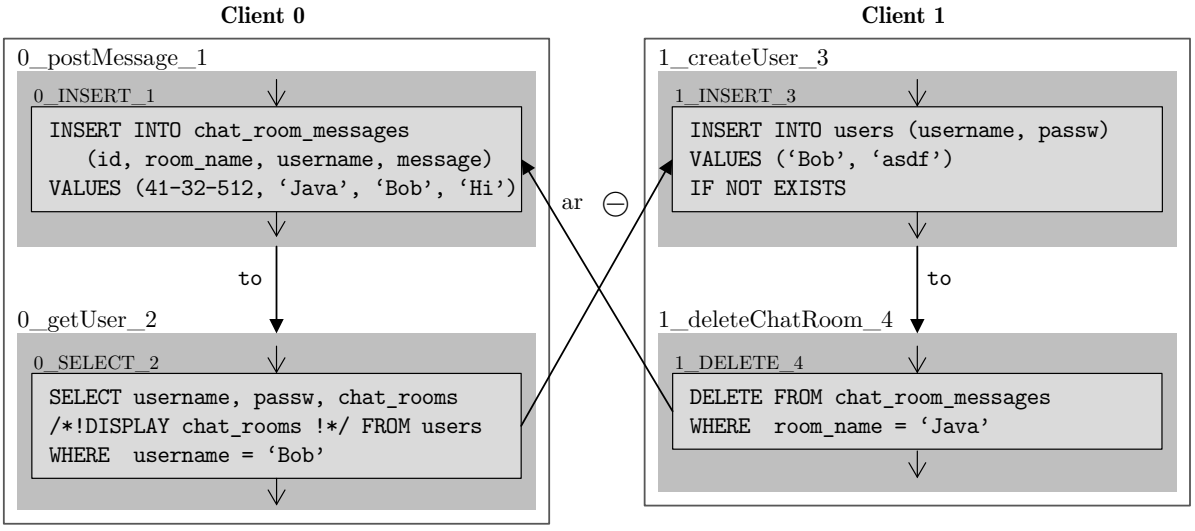


Figure 5.2: False positive that is reported for killrchat if display code is not included.

transaction that adds a user to a chat room, there is an upsert `UPDATE users SET chat_rooms = chat_rooms + {?}`. The analysis can specify in the legality specification that after this upsert, a `0_SELECT_2` must return a row. So for the combination where on client 0 a user is added to a chat room first and the user details are queried afterwards, the analysis does not report a violation, as due to legality, the user row exists in the second transaction on client 0.

With display code, we get additional false positives. In Figure 5.3, client 0 removes a chat room and afterwards queries the user information. Client 1 removes a single user from the chat room. On both clients, the same user is logged in. We have arbitration from client 1 to client 0 as the user posts a message when he leaves the chat, which is then deleted when the chat room is removed. We have an anti-dependency from the left to the right, as the set with the chat rooms a user participates in is also stored in the `users` table, which is updated in the `removeUserFromChatRoom` transaction.

A chat room can only be deleted from the creator, so "Bob" participates in the chat room he deletes. In the `deleteChatRoom` transaction, the chat room that is deleted is also removed from the user rows of all participants by executing `0_UPDATE_4` in a loop. As "Bob" is a participant, one

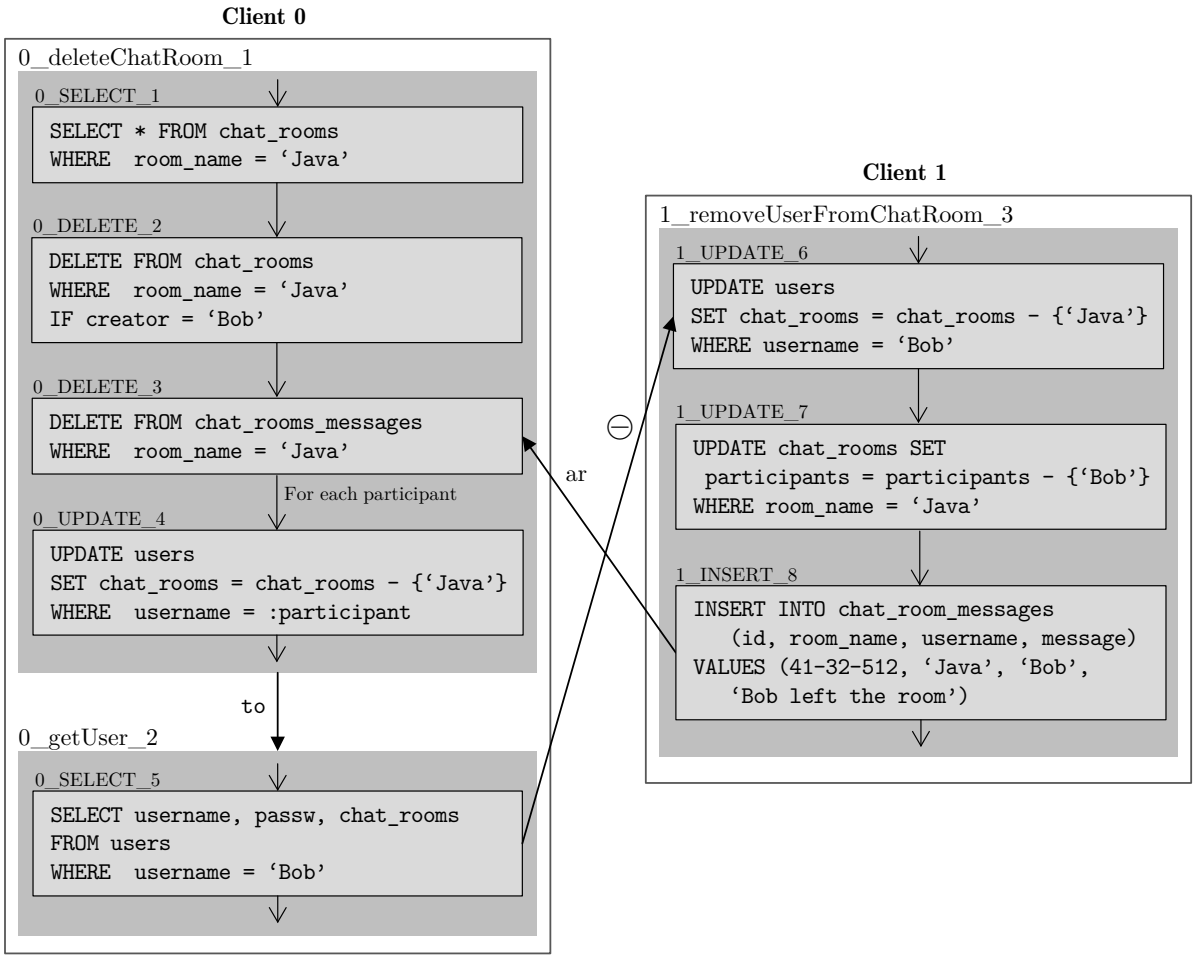


Figure 5.3: A false positive that is reported for killrchat if display code is included in the analysis.

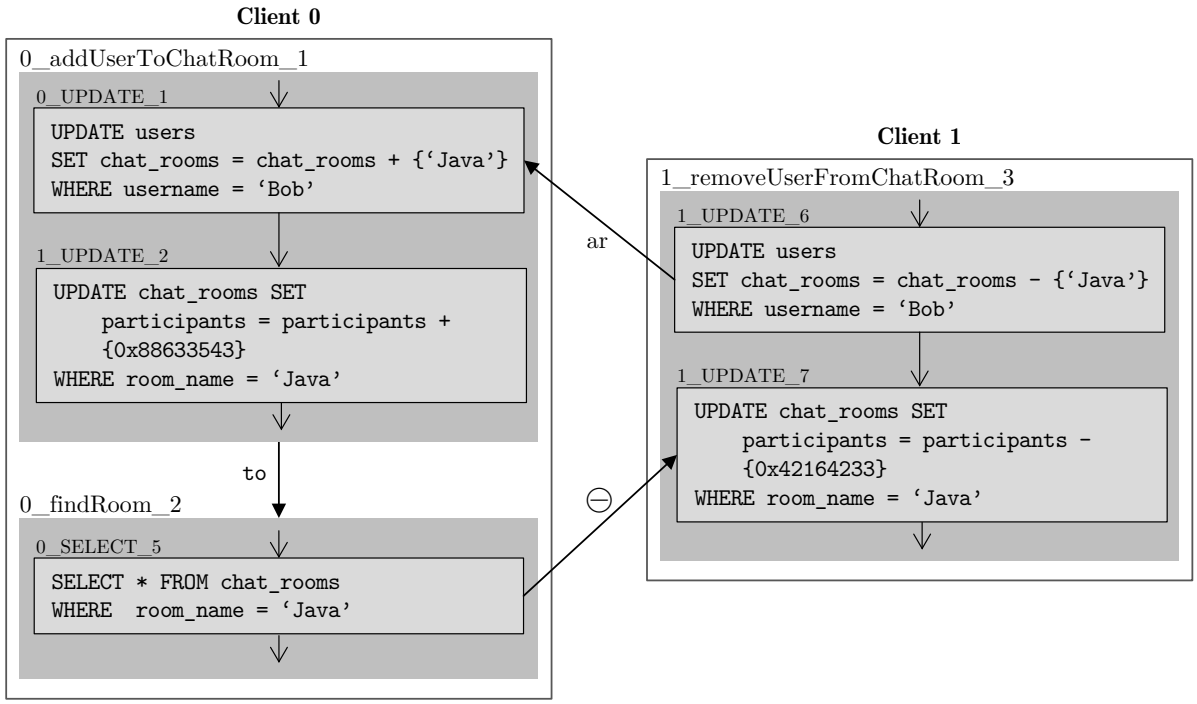


Figure 5.4: Another false positive that is reported for killrchat if display code is included.

concrete execution of the `0_UPDATE_4` event removes the chat room "Java" from the row of "Bob", which absorbs the `1_UPDATE_6` event. Therefore, the anti-dependency cannot occur.

The programmer could solve this false positive by explicitly updating the row of the current user in a single event instead of updating it also in the loop.

Figure 5.4 shows another false positive in which we have two clients where on one, the user joins a chat room while on the other the same user leaves the chat room. Each row in the `chat_rooms` table has a set containing all the participants of the chat room. The user data of which this participants set consists of is stored as a serialized object. In the violation, we have an anti-dependency from `0_SELECT_3` to `1_UPDATE_5`, which is not possible in reality as the update would be absorbed by `0_UPDATE_2`.

Currently, the analysis does only encode equality of immutable values. This false positive would be avoided if equality would be correctly encoded for serialized objects too.

These three types of imprecisions explain all the false positives that are reported for killrchat.

5.4.1.3 Errors and Warnings from the Twitter Clones

Cassandra-twitter, cassatwitter and simple-twitter all have the error shown in Figure 5.5 for the `register` transaction: If the user already exists is checked using a query first. If no result is returned, a new user is created. This means that if two users register simultaneously, both first

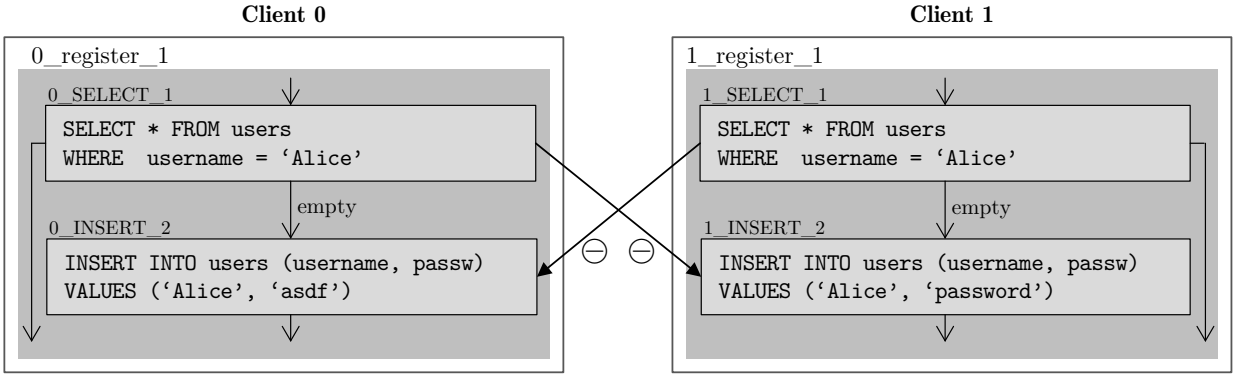


Figure 5.5: Error in the `register` transaction found in cassandra-twitter, cassatwitter and simple-twitter.

get an empty result as no other user with the same username already exists. Both think that they registered with their password, but only one is registered eventually. The error can be resolved in the programs by relying on lightweight transactions.

Also all twitter clones have the same type of warnings: Two different users follow each other. It is possible that then both add a tweet which is not displayed in the timeline of the other user. So it is possible that "Alice" follows "Bob" since 10:00, but she does not see the tweet that "Bob" posts at 10:01 on her timeline.

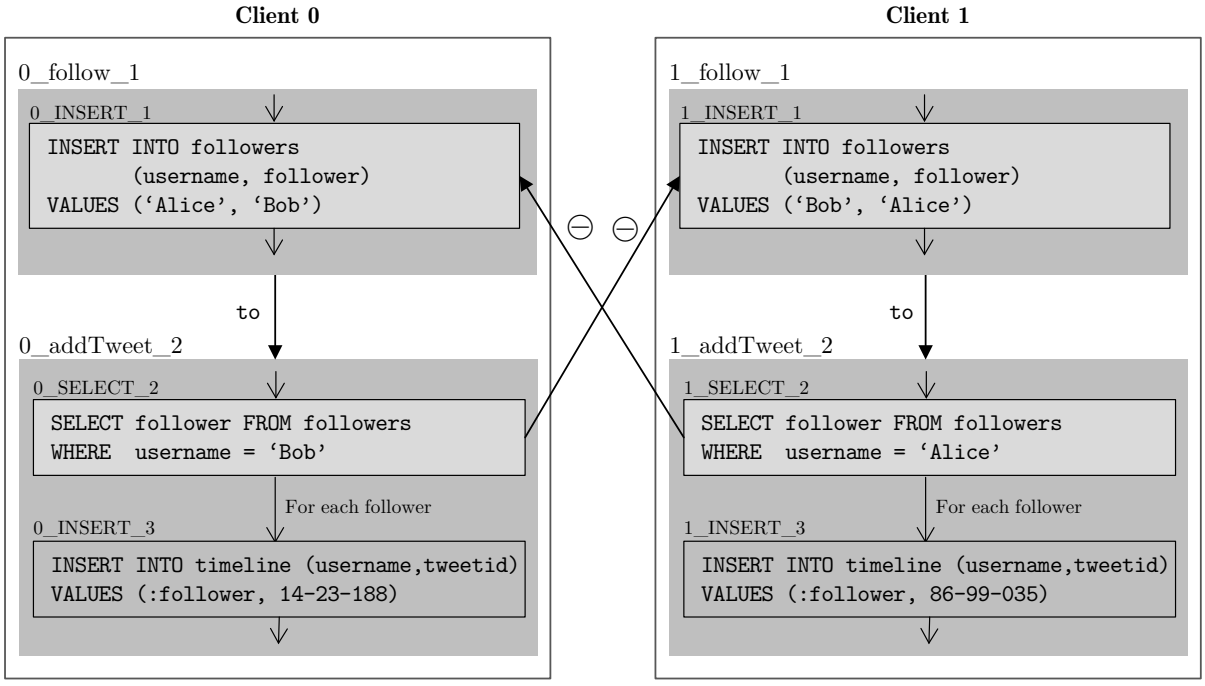


Figure 5.6: Serializability violation that is reported for all twitter clones.

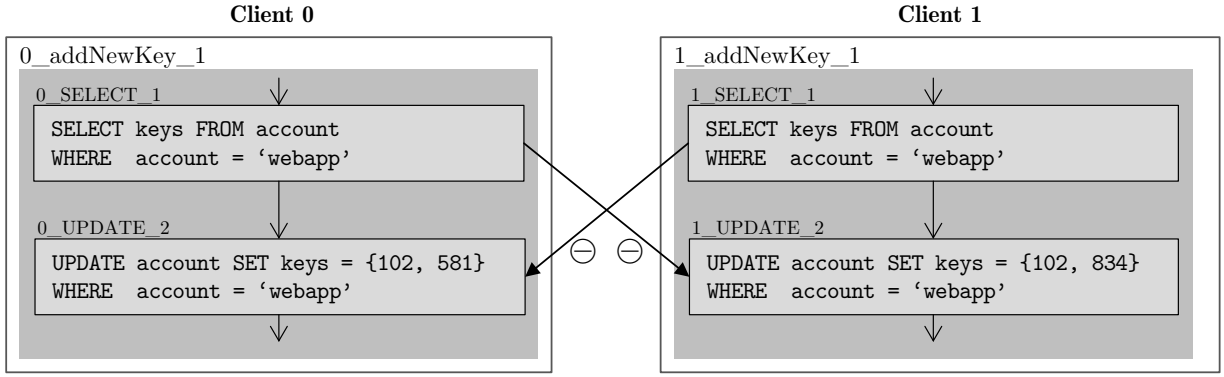


Figure 5.7: Serializability violation found in cassieq.

Figure 5.6 shows this type of violation. In the first transaction, both users follow the other user. When adding a tweet, they query the list of followers and add the tweet also to the timeline of all followers. In this example, on both clients, `SELECT_2` returns an empty result, which means that the tweet does not show up in the timeline of the other user.

Resolving this violation without a performance penalty is probably not possible. If a user does not follow a lot of other users, the timeline could also be built by fetching the timeline for each followed user separately.

5.4.1.4 Errors for Cassieq

The five errors that are reported for cassieq can be classified in two types. The first type of error is a lost update: Each account in cassieq has a set of keys that identify the account. Keys are also used for authentication. New keys can be added to an account using the `addNewKey` transaction. Cassieq reads the current set of keys, adds the new key to the set and updates the account with the new set. Two clients can therefore read the old set simultaneously and add a new key each. One of the updates is then lost, as the update that is arbitrated last overwrites the first. This violation is shown in Figure 5.7.

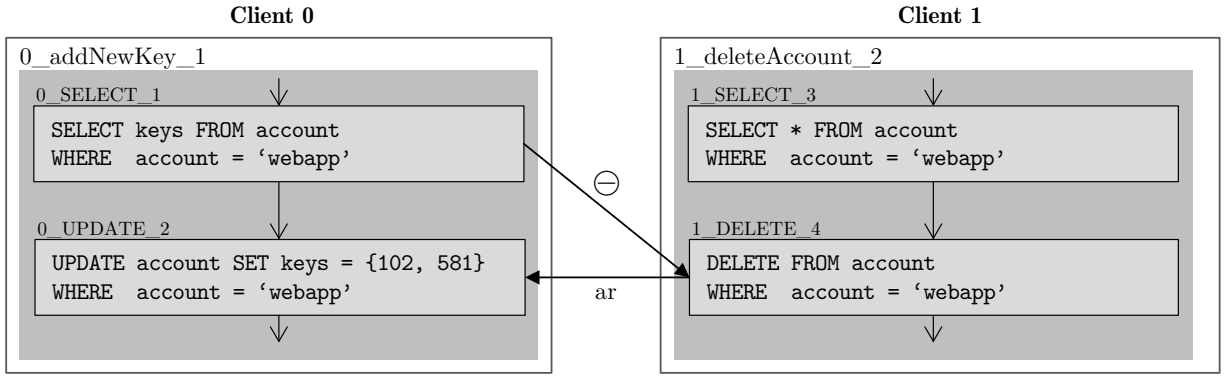


Figure 5.8: Another serializability violation found in cassieq.

The error can be resolved using the add to set operator of CQL. Instead of selecting and updating, one can update the set in place using the following CQL-query: `UPDATE account SET keys = keys + {:newKey} WHERE account = :account`.

In the other type of error, the account is updated while it is deleted on a second client. The deleted account can reappear eventually due to the upsert semantics of an `UPDATE` CQL-query.

Figure 5.8 illustrates this behavior: Client 0 does not observe the event `1_DELETE_4` which is arbitrated before `0_UPDATE_2`. Eventually, client 1 will observe the `0_UPDATE_2`, which means that the deleted account reappeared.

As the account is probably not modified a lot of times, lightweight transactions could be used to fix this error. The `UPDATE` should check if the record exists using the `IF EXISTS` condition in order to avoid the recreation of a deleted account.

5.4.1.5 Errors in Datastax-Queueing

Datastax-Queueing is a small program that implements a queueing system. It is designed for a single reader and a single writer. The system is implemented using a circular buffer and two indexes that indicate the next position to read and the next position to write. Both errors that are reported for this example are due to a double read or double write on the same index.

Figure 5.9 shows what may happen if two writers send a new job to the system: Both writers query the current index, write the job at that index and update the index in the table. If both writers fetch the same index initially, one of the jobs is overwritten by the other.

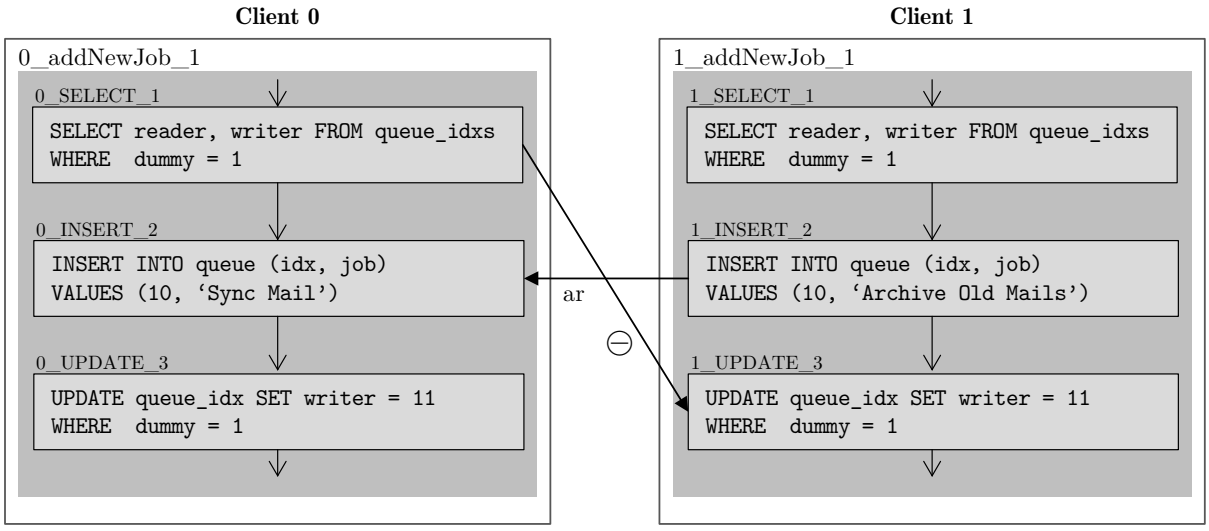


Figure 5.9: A serializability violation found in the datastax-queueing example.

The error can be resolved using lightweight transactions and sets with indexes of jobs that are ready to be processed. The next index on the `queue_ids` table is incremented using a LWT that checks if the current index equals the selected one. If the LWT succeeds, the job is written to the table and the index of the job is added into a ready set. The reader can poll this set, remove an index using an LWT and if the LWT is successful, process the job stored at the given index.

5.4.1.6 Warnings for KillrChat

For killrchat, there is only one type of warning: Messages are sometimes posted in a chat room where a user does not belong to. This may lead to strange behavior, e.g. if a new chat room is created and a message is already there or if a user posts in a chat room he does not participate in.

In the violation shown in Figure 5.10, client 0 creates a new chat room and checks afterwards the messages from the new chat room. Client 1 does remove a user from the same chat room that is created by client 0. As the `0_INSERT_1` is an LWT, a new chat room is only created if no other room with the same name exists. Client 1 is arbitrated before client 0 which means that client 1 posts his leaving message before the chat room exists. Note that `1_UPDATE_5` is not an upsert: This update behaves like a delete because the only update is the removal from a set. Therefore the LWT of client 0 still succeeds. Even though that the `0_fetchMessages_2` does not fetch the leaving message in this violation, it eventually will.

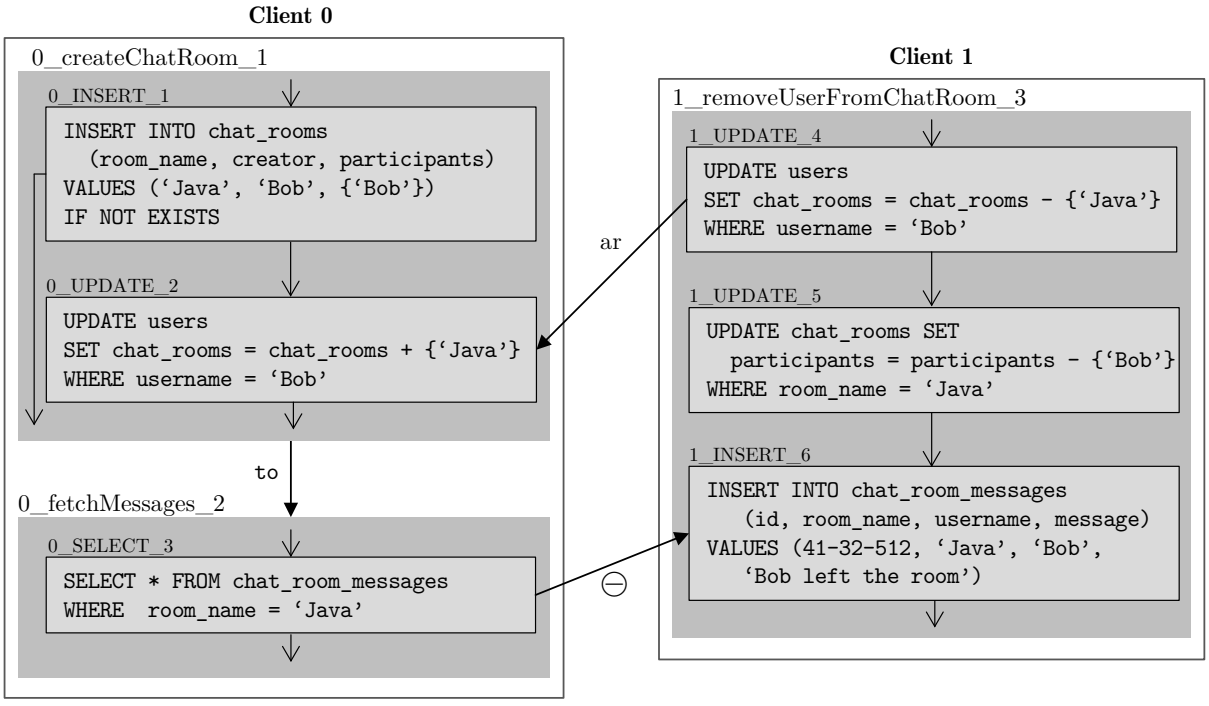


Figure 5.10: A serializability violation found in killrchat.

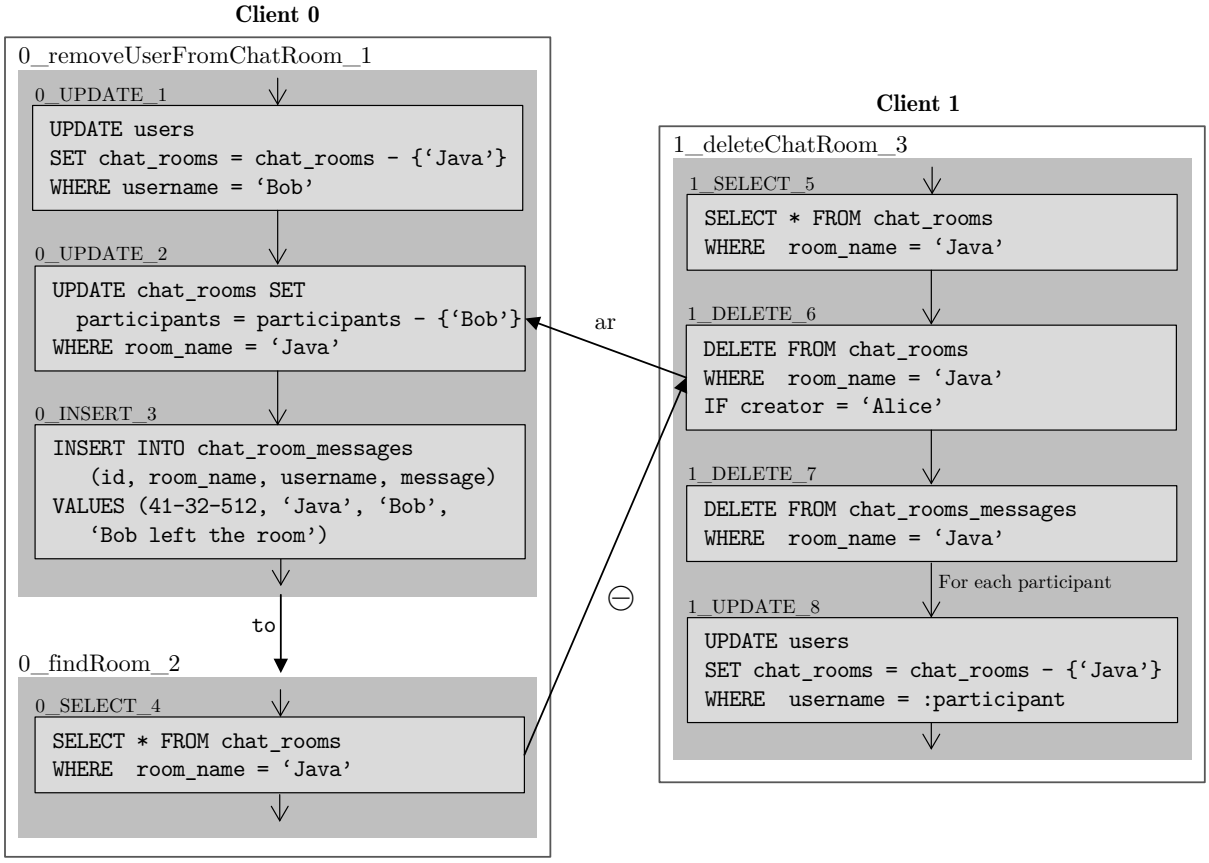


Figure 5.11: Another serializability violation found in killrchat.

Figure 5.11 shows how it can occur that client 1 from the previous example left a chat room he did not participate in: Client 1 deletes a chat room. The transaction `0_removeUserFromChatRoom_1`, in which client 0 leaves the same room, is arbitrated later. Therefore, client 0 posts his leaving message in a non-existing chat room. Also, in the second transaction on client 0, the chat room "Java" is still returned from the data-store (as there is an anti-dependency).

One solution to this problem is using a surrogate key for the chat rooms, e.g. a UUID. If each new chat room is created with a new UUID, messages that are posted after the deletion of a chat room will still remain in the database, but they will eventually not show up in the application.

5.4.1.7 Warnings for Playlist

The reported violations that we classified as warnings for playlist can be split in two categories. Tracks are combined in playlists where each playlist has a name and a user it belongs to. The primary key of the table `playlist_tracks`, which is used to store the playlists, consists of `username`, `playlist_name` and `sequence_no`. The `sequence_no` column represents the timestamp in milliseconds when a track was added to the playlist.

If two tracks are added to the same playlist at exactly the same time, we have a lost update. Figure

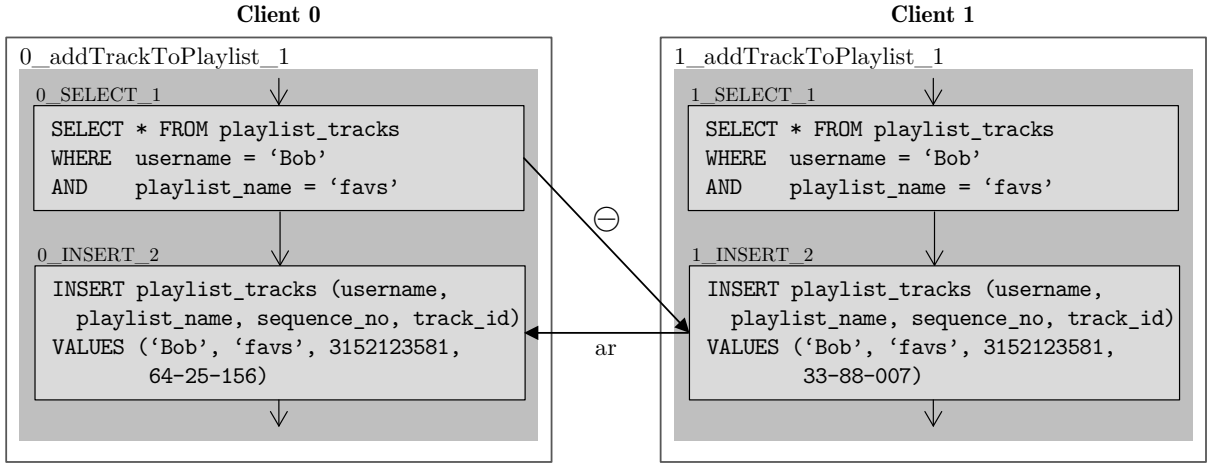


Figure 5.12: A serializability violation found in the playlist example.

5.12 shows the possible serializability violation that leads to a lost update. If the `sequence_no` is not equal on both clients, we still may have a serializability violation (with two anti-dependencies), but no lost update.

The other type of warning is illustrated in Figure 5.13. Client 1 deletes a playlist. Client 0 adds a new track to the same playlist. This is similar to the warnings reported for killrchat. Eventually the playlist is deleted but a track for the playlist is still in the database. If a playlist with the same name is recreated later, the new playlist already contains tracks.

The problem with the lost update can be solved with using a UUID for the `sequence_no` instead of a timestamp. UUIDs can be created based on the current time, which means that tracks would still be sorted in the partition in the order in which they were added to the playlist. But in contrast to a timestamp, a UUID is unique.

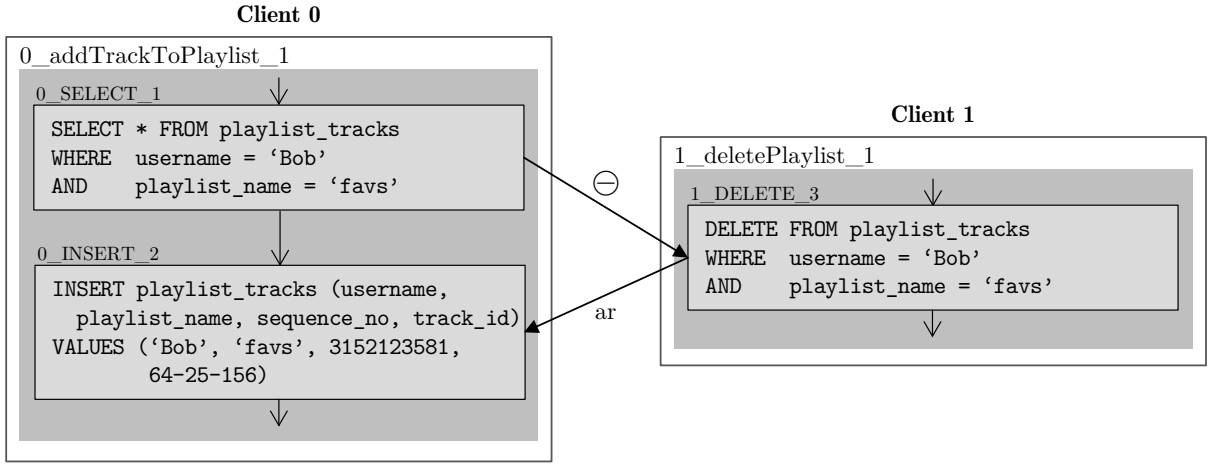


Figure 5.13: Another serializability violation found in the playlist example.

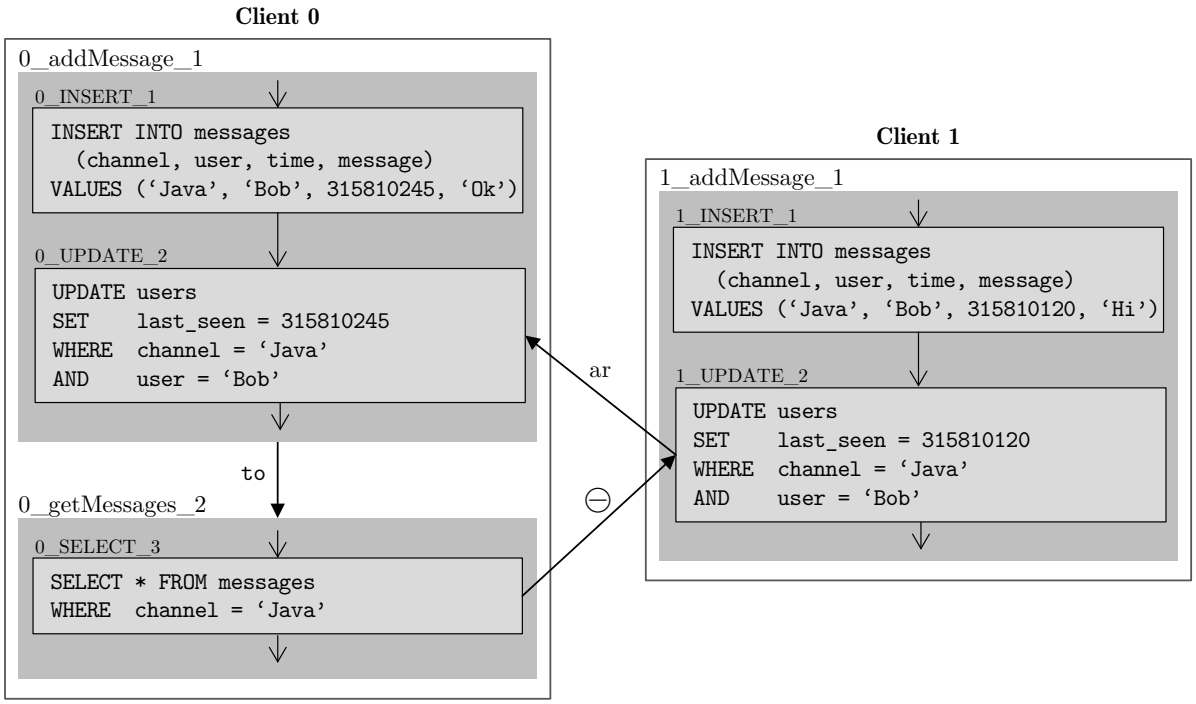


Figure 5.14: A serializability violation reported for the roomstore example.

5.4.1.8 Warnings for Roomstore

The violations that are reported for roomstore are only classified as warnings as they indicate a possible problem. Figure 5.14 shows the violation, which is quite simple: Two messages are added to the data-store and the timestamp when the user sent the last message is updated. This results in a potential arbitration edge and an anti-dependency when a query for a message is added on one client.

This violation is probably not a warning, it is rather harmless. But the problem with roomstore is also that the primary key consists of the username, the channel and the timestamp when a message is sent, which may not be unique. If multiple messages are sent at the same time, all but one are lost. That two messages are sent at exactly the same time from the same user can happen e.g. due to delays or leap-seconds. However, if two messages are sent using the same timestamp, we do not have a serializability violation due to absorption.

5.4.2 Comparison with Trivial Analysis

We compared the false positives that are reported from ECChecker with the false positives that are reported if we only check if a critical cycle exists once without any annotations or enhancements and once with all annotations and enhancements. For this comparison, only combinations that consist of two transactions per client are considered. Table 5.4 shows the results of the comparison. The column "violations" contains the number of serializability violations that we have classified for the given project. The column "CC" shows how many combinations exist with a critical cycle if the

Example	**Options**	**Violations**	**CC**	**FP CC**	**EnhCC**	**FP EnhCC**	$\mathbf{1 - \frac{Enh}{CC}}$	**ECC**	**FP ECC**	$\mathbf{1 - \frac{ECC}{CC}}$	$\mathbf{1 - \frac{ECC}{EnhCC}}$
cassandra-lock		0	0	0	0	0	-	0	0	-	-
cassandra-twitter		8	18	10	10	2	80%	8	0	100%	100%
cassandra-twitter	display	16	33	17	27	11	35%	17	1	94%	91%
cassatwitter		3	7	4	3	0	100%	3	0	100%	-
cassatwitter	display	16	25	9	20	4	56%	16	0	100%	100%
cassieq*		225	290	65	290	65	0%	225	0	100%	100%
cassieq*	display	298	396	98	396	98	0%	298	0	100%	100%
currency-exchange		0	0	0	0	0	-	0	0	-	-
currency-exchange	display	1	2	1	2	1	0%	1	0	100%	100%
datastax-queueing		10	10	0	10	0	-	10	0	-	-
killrchat		0	129	129	21	21	84%	6	6	95%	71%
killrchat	display	326	935	609	759	433	29%	452	126	79%	71%
playlist		149	228	79	228	79	0%	149	0	100%	100%
playlist	display	347	403	56	403	56	0%	347	0	100%	100%
roomstore		0	0	0	0	0	-	0	0	-	-
roomstore	display	14	14	0	14	0	-	14	0	-	-
shopping-cart	display	0	0	0	0	0	-	0	0	-	-
simple-twitter		1	2	1	1	0	100%	1	0	100%	-
simple-twitter	display	4	5	1	5	1	0%	4	0	100%	100%
twissandra		2	3	1	2	0	100%	2	0	100%	-
twissandra	display	20	25	5	25	5	0%	20	0	100%	100%

Table 5.4: False positives (FP) resulting from checking serializability using critical cycles (CC), enhanced critical cycles (EnhCC) and ECChecker (ECC). The reduction in the number of false positives a better analysis can achieve is shown in percent.

trivial analysis does not use any enhancements. Subtracting the real violations gives the number of false positives that are reported (FP CC). The trivial analysis does also profit from the enhancements, e.g. the schema information and asymmetric commutativity. In the column "EnhCC" is the number of critical cycles that are reported by the trivial analysis if all enhancements are enabled. The column "FP EnhCC" contains the number of false positives generated by the enhanced trivial analysis. The number of reported violations from ECChecker with all annotations and enhancements enabled is shown in the column "ECC". The reduction in the number of false positives a better analysis can achieve is shown in percent for each case.

For all of the examples, ECChecker reduces the number of false positives by more than 75% compared to a trivial analysis that only checks for critical cycles and by more than 70% compared to the enhanced trivial analysis.

5.4.3 Effects of Annotations and Enhancements

We analyzed the impact of the different annotations and enhancements on the reported violations. Like in Section 5.4.2 we only considered combinations for two clients with two transactions per client. We measured the number of false positives that are reported if we disable one enhancement or annotation and compared it to a trivial analysis that only checks if a critical cycle can occur in the combination without any enhancements. Only examples where at least one false positive is reported by the trivial analysis are considered for this measurement. The results can be found in Table 5.5.

If each option is considered on its own, the schema setup scripts have the biggest impact on the result. If these are not provided, the reduction of false positives is only 22%, compared to 98% when the default options are used. Nevertheless, e.g. for cassieq providing the schema has no effect. This is due to the fact that the part of cassieq that we analyze acts on a single table. As there is an update statement, the analysis can deduce from the `WHERE` part of the update which columns are part of the primary key. Hence, the schema does not provide additional information. Also commutativity (i.e. that a logical formula is used for commutativity instead of `true` and `false`), global and event constraints and the value analysis are crucial for a precise result. The encoding of these parts happens based on the results of the static analysis. This shows that it is important that the static analysis collects properties like equality, inequality and uniqueness for the arguments of the operations and uses them for serializability checking.

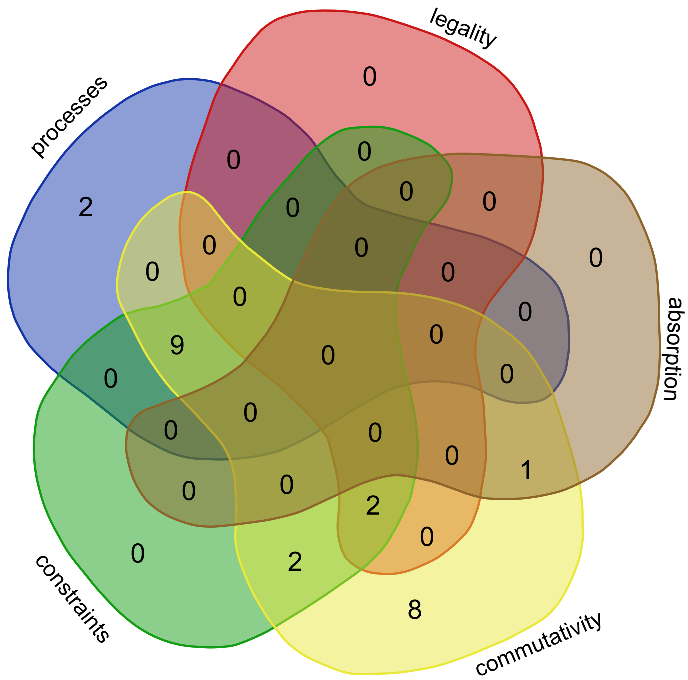
We can also see in the results that there is no useless option. Although there are sometimes only few examples that profit from a single annotation or enhancement, each of them has some impact for at least one project.

Example	Options	Schema Setup Scripts	Commutativity	Global & Event Constraints	Value Analysis	Program Order	Asymmetric Commutativity	Program Order Constraints	Client Local Variables	Absorption	Legality	Side-Channels	Synchronization	Strict Updates	Default
cassandra-twitter		0%	80%	80%	80%	80%	80%	80%	80%	100%	80%	100%	100%	100%	100%
cassandra-twitter	display	0%	35%	35%	35%	59%	59%	59%	65%	94%	65%	88%	94%	94%	94%
cassatwitter		0%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
cassatwitter	display	0%	56%	56%	56%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
cassieq*		100%	0%	3%	17%	8%	8%	8%	17%	12%	100%	100%	100%	100%	100%
cassieq*	display	100%	0%	3%	15%	8%	8%	8%	15%	11%	100%	100%	100%	100%	100%
currency-exchange	display	0%	0%	0%	0%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
killrchat		76%	84%	88%	88%	94%	94%	95%	91%	95%	94%	95%	92%	86%	95%
killrchat	display	25%	14%	37%	24%	57%	50%	79%	62%	72%	60%	79%	53%	73%	79%
playlist		6%	9%	0%	6%	9%	85%	27%	85%	100%	47%	92%	100%	100%	100%
playlist	display	21%	16%	4%	18%	50%	32%	75%	50%	96%	86%	57%	84%	100%	100%
simple-twitter		0%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
simple-twitter	display	0%	0%	0%	0%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
twissandra		0%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
twissandra	display	0%	0%	0%	0%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Total		22%	40%	40%	43%	71%	74%	75%	78%	85%	89%	94%	95%	97%	98%

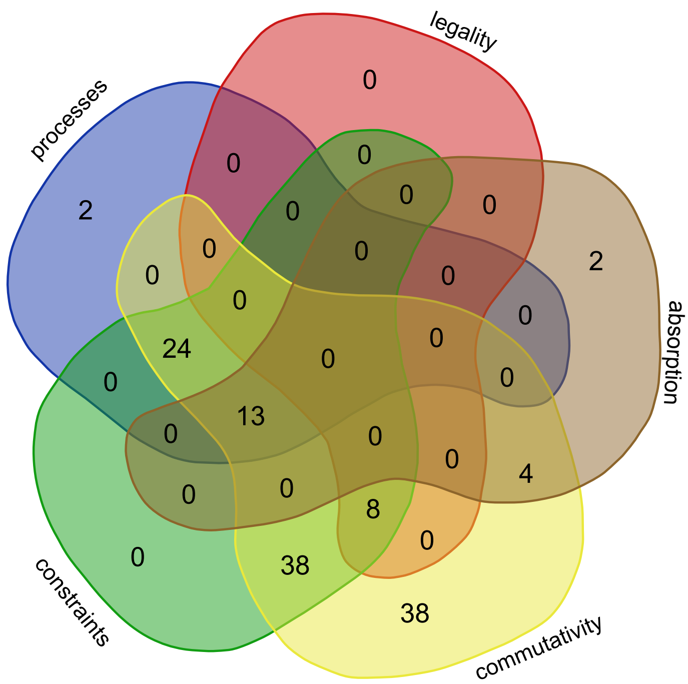
Table 5.5: Reduction of false positives with the default options and one missing annotation / enhancement. While the average reduction is 98% when the default options are used, reduction is only 22% on average if the analysis is executed without providing a schema setup script (all other options are still set to default).

Figure 5.15 shows the results of another evaluation of the effects of some enhancements. We created a set v of possible serializability violations by running the trivial analysis with all enhancements enabled in the following way: First for all combinations consisting of one transaction per client, the trivial analysis checked if a critical cycle is possible and reported all the combinations for which this is the case. Afterwards, all combinations with 2 transactions on one client and 1 transaction on the other client, which did not consist of a violation that was already reported, were checked for a critical cycle and reported if necessary. Finally all combinations consisting of 2 transactions on both clients were checked and reported if necessary. Then, we run ECChecker with the default options and one enhancement disabled and counted all violations that are part of v, but are not reported by an ECChecker analysis with all enhancements enabled (i.e. all reported violations that must be false positives). The numbers in Figure 5.15 show how many false-positives over all examples are reported if the respective enhancement is disabled. If a certain violation v_1 is reported if either the constraints are disabled or commutativity is disabled, v_1 is counted in the intersection of the areas for constraints and commutativity.

We can see that more than half of all false positives need more than one enhancement to be resolved. Commutativity, processes and constraints (global, event and program order constraints) have the biggest effect in this experiment. Legality and constraints only have an impact if also commutativity is enabled.



(a) without display code



(b) with display code

Figure 5.15: Number of false positives that are reported if one enhancement is missing. A violation that is reported if either commutativity or constraints are disabled, counts for the number in the intersection of constraints and commutativity.

6 Conclusions

In this thesis, we have implemented and evaluated a static analysis for checking serializability of programs written in Java and operating on a Cassandra database assuming causal-consistency and atomic visibility. The manual inspection of the reported violations revealed some serious serializability errors while only few false positives were reported.

The extension of the original criterion [2] with asymmetric commutativity, synchronization and legality proved to have a significant effect on reducing the number of false positives. The evaluation of the effects of the enhancements also showed that it is necessary for enough precision to know facts about equalities and inequalities of operation arguments. The design of the static analysis as an inter-procedural and context-sensitive data-flow analysis proved to be useful for collecting lots of these facts.

The program does not need a lot of annotations for the analysis to be precise. Therefore, the tool can be used easily and without a lot of effort during the development of a Java program that works on Cassandra. The output of the tool is minimal, i.e. no other transactions then the ones that are part of a violation are reported. The graphical representations of the violations also contain a model of the event arguments that lead to the violation. Harmless violations can be classified as such so that they are reported separately in later analyses. Therefore, the output of the tool should be useful and understandable for a programmer and help him find and fix serious serializability violations during the development.

6.1 Future Work

We see the following possibilities for future work:

Other Consistency Models: Serializability is currently checked assuming causal-consistency and atomic visibility. The design of the analysis allows to also encode other consistency models. Therefore the analysis could be extended easily with e.g. a model that is closer to the concrete semantics of Cassandra in order to reveal more bugs that are not possible under causal consistency, but may still occur in the real system.

Precision of the Static Analysis: Currently, the static analysis encodes equality only for predefined immutable objects like numbers, strings and UUIDs. The missing encoding of equality for other objects lead to false positives in the evaluation. Therefore, the precision of the anal-

ysis could be improved if the static analysis infers equality for general objects too. When an object is considered as equal depends on the serialization technique used, but as a start, an object could be considered as equal if all the fields are equal.

Bibliography

[1] G. Bernardi and A. Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR'16: International Conference on Concurrency Theory*, 2016.

[2] L. Brutschy, D. Dimitrov, P. Müller, and M. 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.

[3] S. Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, Oct. 2014.

[4] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[7] DataStax. Documentation of Apache Cassandra 3.0. `https://docs.datastax.com/en/cassandra/3.0/`. Accessed: 2017-03-16.

[8] DataStax. Documentation of CQL 3.3. `https://docs.datastax.com/en/cql/3.3/`. Accessed: 2017-03-16.

[9] DataStax. Documentation of prepared statements. `https://docs.datastax.com/en/developer/java-driver/3.0/manual/statements/prepared/`. Accessed: 2017-03-31.

[10] DataStax. Documentation of the java driver 3.0. `https://docs.datastax.com/en/developer/java-driver/3.0/`. Accessed: 2017-03-16.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[12] J. Ellis. Lightweight transactions in cassandra 2.0. `https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0`. Accessed: 2017-03-16.

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

[14] P. J. Leach, M. Mealling, and R. Salz. A universally unique identifier (uuid) urn namespace. RFC 4122, RFC Editor, July 2005. `http://www.rfc-editor.org/rfc/rfc4122.txt`.

[15] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[16] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

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):

Analyzing Serializability of Cassandra Applications

Authored by (in block letters):

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

Name(s):	**First name(s):**
Kurath	Arthur

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)**
Winterthur, 31.03.2017	

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