FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Richard Eliáš

Analyzing Data Lineage in Database Frameworks

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	RNDr. Pavel Parízek, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Inteligence

Prague 2019

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the "Metodický pokyn o etické přípravě vysokoškolských závěrečných prací".

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the "Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs ("software"), I hereby grant the so-called MIT License.

The MIT License represents a license to use the software free of charge. I grant this license to every person interested in using the software. Each person is entitled to obtain a copy of the software (including the related documentation) without any limitation, and may, without limitation, use, copy, modify, merge, publish, distribute, sublicense and/or sell copies of the software, and allow any person to whom the software is further provided to exercise the aforementioned rights. Ways of using the software or the extent of this use are not limited in any way.

The person interested in using the software is obliged to attach the text of the license terms as follows:

In . . . . date . . . .

I would like to thank my brothers Marek and Erik, my mother and my father and all those who have always supported and moved me forward during my studies.

I dedicate this thesis to my beautiful wife Anička, without the help of which I would not be able to master my studies.

Title: Analyzing Data Lineage in Database Frameworks

Author: Richard Eliáš

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Large information systems are typically implemented using frameworks and libraries. An important property of such systems is data lineage - the flow of data loaded from one system (e.g. database), through the program code, and back to another system. We implemented the Java Resolver tool for data lineage analysis of Java programs based on the Symbolic analysis library for computing data lineage of simple Java applications. The library supports only JDBC and I/O APIs to identify the sources and sinks of data flow. We proposed some architecture changes to the library to make easily extensible by plugins that can add support for new data processing frameworks. We implemented such plugins for few frameworks with different approach for accessing the data, including Spring JDBC, MyBatis and Kafka. Our tests show that this approach works and can be usable in practice.

Keywords: data lineage, data flow visualization, static program analysis, Java frameworks

Contents

1 Introduction **3**
1.1 MANTA Flow . . . 3
1.2 Goals . . . 3
1.3 Structure of the Work . . . 4

2 Popular Data Processing Frameworks for Java **5**
2.1 JDBC API . . . 6
2.2 Spring JDBC Framework . . . 8
2.3 MyBatis Framework . . . 8
2.3.1 Mapper definition . . . 9
2.3.2 Configuration . . . 10
2.3.3 Loading Data from Database . . . 11
2.4 Kafka . . . 13
2.4.1 Producer API . . . 14
2.4.2 Consumer API . . . 14
2.4.3 Stream API . . . 15
2.4.4 Connector API . . . 16
2.5 Requirements for Data Lineage Library . . . 16

3 Static Analysis **18**
3.1 Program Analysis . . . 18
3.2 WALA Framework . . . 18
3.2.1 Pointer analysis . . . 19
3.2.2 Class hierarchy . . . 19
3.2.3 Call Graph . . . 19
3.3 Symbolic Analysis Library . . . 19
3.3.1 Symbolic Analysis Algorithm . . . 20
3.3.2 Flow Propagation . . . 20
3.3.3 Identifying the Sources and Sinks of a Data . . . 21
3.3.4 Data Lineage Visualization . . . 22

4 Data Lineage Analysis for Frameworks **25**
4.1 Symbolic Analysis Library Plugins . . . 25
4.2 Interface to Symbolic Analysis . . . 26
4.2.1 FrameworkAnalysisPlugin . . . 26
4.2.2 MethodCallDescription . . . 26
4.2.3 MethodEffectsDescription . . . 26
4.2.4 DataEndpointFlowInfo . . . 28
4.3 JDBC API . . . 28
4.3.1 DataSource . . . 28
4.4 Spring JDBC Framework . . . 28
4.4.1 Callbacks . . . 29
4.5 MyBatis . . . 29
4.5.1 Mapper interfaces . . . 29
4.5.2 Database connection configuration . . . 31

4.6 Kafka . . . 31
4.6.1 Callbacks . . . 31
4.6.2 Properties . . . 32

5 User Documentation for the Java Resolver Tool 33
5.1 Software Requirements . . . 33
5.2 Installation . . . 33
5.3 Running the Java Resolver Tool . . . 34
5.4 Result Graphs . . . 34
5.5 Implementing Support for new Framework . . . 35

6 Evaluation 36
6.1 Data Flow Graph Visualizations for Frameworks . . . 36
6.2 Limitations of the Java Resolver Tool . . . 37
6.3 Fulfillment of the Requirements . . . 37

7 Conclusion 42

References 43

List of Figures 45

List of Listings 46

1. Introduction

For modern data processing applications, an important aspect is a data lineage. Data lineage, or provenance, describes where data came from, how it was derived and where the results are stored.

Lineage can be useful in many domains. Molecular biology databases, which mostly store copied data, can use lineage to verify the copied data by tracking the original sources as was demonstrated by Buneman et al. [1]. Probabilistic databases can exploit lineage for confidence computation as was stated by Ré and Suciu [11]. The important usage of the data lineage is also in machine learning, where for the purpose of a verification and validation can be important to know data sources of used data sets. Data lineage is also important in Business Intelligence (BI) when achieving full regulatory compliance and improving data governance.

In all these areas, Java applications together with data manipulation frameworks are often used to access data in databases, files or in network storages. The tool that is able to identify the sources and sinks of a data and create data lineage of an application can be very useful.

Up to now, we are aware of the only tool for creating data lineage of Java applications - the Symbolic analysis library presented in Parízek [9]. The Symbolic analysis library performs a specific kind of static program analysis. It can create data lineage of an analysed application when only Java I/O and JDBC APIs are used in addition to the core Java libraries (collections, etc.).

1.1 MANTA Flow

An example of a data lineage analysis tool is the MANTA Flow [7]. The MANTA Flow helps enterprises to get end-to-end data lineage including custom SQL code. That allows customers to fulfill compliance regulations or improve data governance.

It extracts and analyses metadata from report definitions, custom SQL code, and extract-transform-load (ETL) workflows, to create data flow graphs which span multiple systems and a range of technologies. Lineages are computed based on analysis of actual code. All entities detected by MANTA Flow, such as database tables, columns or procedures, are visualized to help users utilize this information.

Systems, for which MANTA Flow is used to analyse their data lineage, often use Java applications and data processing frameworks to work with data. MANTA Flow integrates the Symbolic analysis library to create data lineage of Java applications to cover whole system, not only its database parts. However, the support for computing data lineage of complex data processing frameworks is missing.

1.2 Goals

In our work, we *propose architecture changes* to Symbolic analysis library to be easily extensible by plugins that can add support for new data processing frameworks to identify its data sources and sinks.

We also *implement Symbolic analysis library plugins* for few selected frameworks (MyBatis, Spring JDBC and Apache Kafka). Each framework has different approach for accessing the data. Thereby we want to demonstrate that such plugins can be used to add support for data lineage analysis of other Java frameworks. The feature to easily extend the Symbolic analysis library to add support for a new frameworks is very important, as new frameworks are being developed today.

1.3 Structure of the Work

The rest of the thesis has the following structure.

In chapter 2 we give overview of popular Java frameworks for data processing.

In chapter 3 we introduce libraries that are used for static analysis of Java programs to create data lineage.

In chapter 4 we present design of the Java Resolver tool for data lineage of frameworks. We also describe some selected implementation details of the tool.

Chapter 5 contains user documentation for the Java Resolver tool.

Chapter 6 present our results for selected frameworks, limitations of our solution and plans for the future work.

2. Popular Data Processing Frameworks for Java

In this chapter, we show basic features of chosen frameworks that are used for manipulating data. We focus on the basic features relevant for the data lineage and ignore most of their advanced features.

We show data lineage basic examples of how data can be loaded or stored by Java application and how sources and sinks are identified, as this is the main topic of our work.

All frameworks that are using database, would work with the example of storing and loading objects of the class `DatabaseValue` in the Listing 2.1 from database structured as in Table 2.2.

```java
public class DatabaseValue {
  private Integer id;
  private String value;

  public Integer getId() {
    return id;
  }

  public void setId(Integer id) {
    this.id = id;
  }

  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;
  }
}
```

Listing 2.1: Example of object loaded from database

ID	VALUE
1	A
2	B
3	C

Table 2.2: Example of Database content

2.1 JDBC API

For accessing database in Java applications, there exists the standard Java Database Connectivity (JDBC) API. The API is described in more detail in JDBC Introduction [5] and here we show just the features that are relevant for this work.

Database vendors usually provide JDBC API implementation. This API is generic, so there should be no difference for connecting to different database types.

The main component of the API is the `java.sql` [3] package. We present main interfaces controlling database calls:

- `Connection`
- `Statement`, `PreparedStatement`, `CallableStatement`
- `ResultSet`

`Connection` object should hold database connection and through this connection database queries can be executed using any of `Statement` calls. When data are returned to application from statement, it is done through `ResultSet`.

Getting connection to database is through `DriverManager`, or since JDBC 2.0 it can be done using `DataSource` in `javax.sql` [4] package. Using `DataSource` is now the preferred way of connecting to database, but many applications still continue to use old-fashioned `DriverManager`.

Listing 2.3 shows how `DataSource` can be created for Oracle database. Database is listening on url `jdbc:oracle:thin:@//192.168.0.16:1521/orcl` and `User` user and `Password` password is used when connecting to it.

The `createDataSource()` method would be also used in next database examples.

```java
public static DataSource createDataSource() throws SQLException {
  OracleDataSource dataSource = new OracleDataSource();
  dataSource.setURL("jdbc:oracle:thin:@//192.168.0.16:1521/orcl");
  dataSource.setUser("User");
  dataSource.setPassword("Password");
  return dataSource;
}
```

Listing 2.3: Example of creating `DataSource` for connecting to Oracle database

Listing 2.4 illustrates the two basic use cases - loading and storing the data.

Firstly, we will describe the loading part - the `getForId` method. On line 3, connection to database is created. Then on lines 5–6 database query is created to select just rows matching `id` argument. The query is then executed on line 7 and its results are stored in `ResultSet` object. Then the `DatabaseValue` is created and its properties are set to values of `ID` and `VALUE` columns on lines 9–11 and the result is returned on line 12.

Secondly, we will describe storing part - the `insert` method. Apart from different SQL statement and setting two arguments to `PreparedStatement` instead of just one, the only difference from previous query is the usage of `executeUpdate` method on line 32 for executing insert statement, as no output is expected.

```java
public DatabaseValue getForId(int id) throws SQLException {
  DataSource dataSource = createDataSource();
  try (Connection connection = dataSource.getConnection()) {
    String query = "SELECT ID, VALUE FROM T WHERE ID = ?";
    try (PreparedStatement preparedStatement =
        connection.prepareStatement(query)) {
      preparedStatement.setInt(1, id);
      try (ResultSet resultSet = preparedStatement.executeQuery()) {
        resultSet.next();
        DatabaseValue databaseValue = new DatabaseValue();
        databaseValue.setId(resultSet.getInt("ID"));
        databaseValue.setValue(resultSet.getString("VALUE"));
        return databaseValue;
      } catch (SQLException e) {
        // handle exception
      }
    } catch (SQLException e) {
      // handle exception
    }
  } catch (SQLException e) {
    // handle exception
  }
  return null;
}

public void insert(DatabaseValue value) throws SQLException {
  DataSource dataSource = createDataSource();
  try (Connection connection = dataSource.getConnection()) {
    String query = "INSERT INTO T (ID, VALUE) VALUES (?, ?)";
    try (PreparedStatement preparedStatement =
        connection.prepareStatement(query)) {
      preparedStatement.setInt(1, value.getId());
      preparedStatement.setString(2, value.getValue());
      preparedStatement.executeUpdate();
    } catch (SQLException e) {
      // handle exception
    }
  } catch (SQLException e) {
    // handle exception
  }
}
```

Listing 2.4: Example of query and insert operations using JDBC API [5]

2.2 Spring JDBC Framework

Spring JDBC Framework [13] is an extension above JDBC API that tries to help developers to code only parts with application logic and it removes much of the boilerplate code. It is illustrated by next list from Spring JDBC Framework [13], from which only italicized items need to be coded by user, and all low level details are handled by framework:

- Define connection parameters
- Open the connection
- *Specify the statement*
- Prepare and execute the statement
- Set up the loop to iterate through the results (if any)
- *Do the work for each iteration*
- Process any exception
- Handle transactions
- Close the connection

Before Java 7, coding in standard JDBC tend to be errorneus because of forgetting to close database resources (user needs to close resources in finally block as try-with-resources did not exist yet) and the boilerplate code does not help in readability of code.

Because of this, Spring JDBC Framework starts to use other objects to wrap application logic that can be used by framework in later execution.

The Listing 2.5 illustrates that principe. The `RowMapper` defined on line 6 defines transformation logic (how result row should be transformed to `DatabaseValue` object) and the `PreparedStatementSetter` on line 23, which is used to set up parameters for database call.

We can notice the similarity between code in that wrapper objects and the Listing 2.4 when plain JDBC API was used. We can also notice that all the boilerplate code vanished, and when we use named classes instead of inline anonymous ones, the database call would fit just one line of code.

2.3 MyBatis Framework

MyBatis Framework [8] is one of Object-Relational Mapping (ORM) frameworks. It internally uses JDBC API to communicate with the database, but in almost all cases, there is no need to work with the low level JDBC.

Unlike other ORM frameworks, it does not map Java objects to database tables, but Java methods to SQL statements. Framework uses concept of `Mapper` interfaces (but differently as Spring JDBC Framework). Developer define `Mapper` interface with some methods that executes SQL queries, and all communication with database is always happening through these methods.

```java
public DatabaseValue getForId(int id) throws SQLException {
  DataSource dataSource = createDataSource();
  JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
  return jdbcTemplate.queryForObject(
      "SELECT ID, VALUE FROM T WHERE ID = ?",
      new RowMapper<DatabaseValue>() {
        @Override
        public DatabaseValue mapRow(ResultSet resultSet, int rowNum)
            throws SQLException {
          DatabaseValue databaseValue = new DatabaseValue();
          databaseValue.setId(resultSet.getInt("ID"));
          databaseValue.setValue(resultSet.getString("VALUE"));
          return databaseValue;
        }
      },
      id);
}

public void insert(DatabaseValue value) throws SQLException {
  DataSource dataSource = createDataSource();
  JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
  jdbcTemplate.update(
      "INSERT INTO T (ID, VALUE) VALUES (?, ?)",
      new PreparedStatementSetter() {
        @Override
        public void setValues(PreparedStatement preparedStatement)
            throws SQLException {
          preparedStatement.setInt(1, value.getId());
          preparedStatement.setString(2, value.getValue());
        }
      });
}
```

Listing 2.5: Example of query and insert operations using Spring JDBC Framework [13]

For each method SQL query is defined and when some output is expected, mapping of rows to Java objects should be provided. The name “Mapper” is derived from the feature of mapping database query result to Java objects. Definition of queries and mappings could be done by external XML files or by annotations in these interfaces.

In section 2.3.1 we show examples of `Mapper` interfaces, in section 2.3.2 we show how database configuration can be created and in section 2.3.3 we finally show how to use these parts to load data from database.

2.3.1 Mapper definition

In this section, we show examples of `Mapper` interface definition for MyBatis Framework done by XML and also by annotations.

When using `Mapper` interface, framework internally create implementation of it and by calling its methods, the same logic is done as in previous JDBC example, when connection to database is open and `PreparedStatement` is created with defined query and provided arguments, and after execution is done, resulting object values are mapped from `ResultSet`.

Listing 2.6 shows the use of annotations to store defitions of queries. In case of `getForId` method, also mapping is defined.

Firstly, we describe loading of `DatabaseValues` objects from database. Line 6 contains query definition using the `@Select` annotation and lines 2–5 contains mapping of result columns to `DatabaseValue` attributes using the `@Results` and `@Result` annotations.

The storing of `DatabaseValue` is done using `insert` method, for which the insert statement is located on line 9 using `@Insert` annotation.

The framework substitutes tags `#{id}` and `#{value}` in statements by values from method arguments (in case of `DatabaseValue` argument, objects properties are used).

```
 1 public interface Mapper {
 2   @Results(value = {
 3    @Result(column = "ID", property = "id"),
 4    @Result(column = "VALUE", property = "value")
 5   })
 6   @Select("SELECT ID, VALUE FROM T WHERE ID = #{id}")
 7   DatabaseValue getForId(int id);
 8 
 9   @Insert("INSERT INTO T (ID, VALUE) VALUES (#{id}, #{value})")
10   void insert(DatabaseValue value);
11 }
```

Listing 2.6: Definition of Mapper interface using annotations

MyBatis provide rich feature set for XML mappers. Basic mapper contains an SQL statements definition in XML tags `<select>`, `<insert>`, `<delete>` and `<update>` and `<resultMap>` for mapping of columns to object properties.

From more advanced features, MyBatis supports reusable fragments. It means that one can define an SQL fragment and reuse it in more queries.

Another advanced feature is dynamic SQLs, where queries are created dynamically as some conditions hold.

All of its features are described in more detail in MyBatis Framework [8].

Listing 2.7 shows the basic mapper definition using XML. First, methods `getForId` and `insert` are defined in `Mapper` interface and the rest, SQL statements and result mapping, are stored in XML mapper file.

Query for `getForId` is located on line 12 in `<select>` tag. Tag contains a reference to the correct `resultMap` mapping on lines 6–9, which contains mapping of result columns to `DatabaseValue` object attributes. The `insert` SQL statement is defined on line 15 using `<insert>` tag.

```java
public interface Mapper {
  DatabaseValue getForId(int id);

  void insert(DatabaseValue value);
}
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="Mapper">
  <resultMap id="resultMap" type="DatabaseValue">
    <result property="id" column="ID"/>
    <result property="value" column="VALUE"/>
  </resultMap>

  <select id="getForId" resultMap="resultMap" parameterType="int">
    SELECT ID, VALUE FROM T WHERE ID = #{id}
  </select>
  <insert id="insert">
    INSERT INTO T (ID, VALUE) VALUES (#{id}, #{value})
  </insert>
</mapper>
```

Listing 2.7: Definition of XML Mapper interface

2.3.2 Configuration

The main MyBatis component responsible for creating database connections is `SqlSessionFactory`. It needs to be configured to successfully connect to database. Configuration can be done in Java and also using external XML configuration file.

In Listing 2.8 we show, how configuration can be done in Java. On line 3 we use `DataSource` that we previously define in Listing 2.3. `JdbcTransactionFactory` was used to handle database transactions. On line 7 `Mapper` interface is registered to be known by MyBatis and finally `SqlSessionFactory` is created.

In Listing 2.9 we show, how configuration can be done using external XML file. `SqlSessionFactory` is created on line 3. Configuration file configures `DataSource` on lines 9–14 and `Mapper` interface is registered on line 18.

2.3.3 Loading Data from Database

Since we know how to configure database connections and how to define mappers, we are ready to show how data can be loaded (and stored) from (to) database. Listing 2.10 show all pieces together.

The `getForId` method creates on line 2 previously configured `SqlSessionFactory` from which `SqlSession` is opened to access database. On line 4 implementation of `Mapper` interface is retrieved and on line 5 database query is executed and its result is returned.

```java
public SqlSessionFactory createSqlSessionFactory()
    throws SQLException {
  Environment environment = new Environment.Builder("environmentId")
      .dataSource(createDataSource())
      .transactionFactory(new JdbcTransactionFactory())
      .build();
  Configuration configuration = new Configuration(environment);
  configuration.addMapper(Mapper.class);
  return new SqlSessionFactoryBuilder()
      .build(sessionFactory);
}
```

Listing 2.8: Configuration of `SqlSessionFactory` in Java

```java
public SqlSessionFactory createSqlSessionFactory()
    throws IOException {
  return new SqlSessionFactoryBuilder()
      .build(new FileInputStream("configuration.xml"));
}
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="environmentId">
    <environment id="environmentId">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver"
            value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url"
            value="jdbc:oracle:thin:@//192.168.0.16:1521/orcl"/>
        <property name="username" value="User"/>
        <property name="password" value="Password"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper class="Mapper"/>
  </mappers>
</configuration>
```

Listing 2.9: Configuration of `SqlSessionFactory` using external XML configuration file

The `insert` method just executes different mapper method (`insert`) and all remaining code is the same.

```java
public DatabaseValue getForId(int id) throws Exception {
  SqlSessionFactory sqlSessionFactory = createSqlSessionFactory();
  try (SqlSession sqlSession = sqlSessionFactory.openSession()) {
    Mapper mapper = sqlSession.getMapper(Mapper.class);
    return mapper.getForId(id);
  }
}

public void insert(DatabaseValue value) throws Exception {
  SqlSessionFactory sqlSessionFactory = createSqlSessionFactory();
  try (SqlSession sqlSession = sqlSessionFactory.openSession()) {
    Mapper mapper = sqlSession.getMapper(Mapper.class);
    mapper.insert(value);
  }
}
```

Listing 2.10: Example of query and insert operations using MyBatis Framework [8]

2.4 Kafka

Apache Kafka Framework [6] is a distributed streaming platform. It means that application can publish or subscribe records and process them, as they occur.

Kafka can run as a cluster on one or more servers. Each cluster stores stream of records in categories that are called *topics*. Topic can be viewed as a database table. Producers write to a topic's table and consumers read new data from that table. When application wants to publish some records (producer) in the topic, it sends them to server from which data are forwarded to all its subscribers (consumers).

Topics can be also partitioned, so distributed computations can be made on them - each partition can be handled by different server (or producer or consumer). Partitions are ordered, immutable sequences of records that are continually appended to.

Structure of data in records can be arbitrary. Application just need to handle correct transformation of used Java object to (or from) byte array using `Serializer` (or `Deserializer`) objects. However, this feature is not very important considering data lineage problem[1].

The Kafka cluster durably persists all published records (whether or not they have been consumed) using a configurable retention period. For that period, any consumer can access to any published record. This feature is illustrated by Figure 2.11.

[1]We cannot distinguish values that belong to the same attribute, as in the case of databases, where rows are divided into columns and we could create lineage on the attribute/column level.

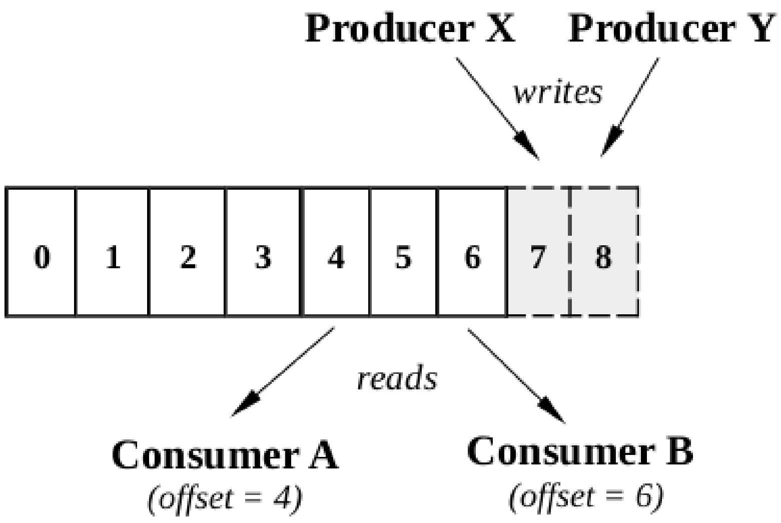

Figure 2.11: Structure of records in a topic. Many producers can write to such topic and consumers can work with the records in arbitrary order.

Figure 2.12 and next list taken from Kafka Framework documentation [6] illustrates usage of four Kafka core client APIs:

- The **Producer API** (see Section 2.4.1) allows an application to publish a stream of records to Kafka topics.
- The **Consumer API** (see Section 2.4.2) allows an application to subscribe to topics and process the stream of records produced to them.
- The **Streams API** (see Section 2.4.3) allows an application to act as a stream processor, consuming an input stream from topics and producing an output stream to output topics, effectively transforming the input streams to output streams.
- The **Connector API** (see Section 2.4.4) allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

2.4.1 Producer API

Using Kafka Producer API, application can create new records and send them to topic of Kafka server. It is illustrated by the example in Listing 2.13. We first configure `KafkaProducer` and then we just send records to topic `Topic` to Kafka server by calling the `send` method. Here we do not need to deal with the `Serializers`, as they are provided for some basic Java classes (like `String`) by framework itself.

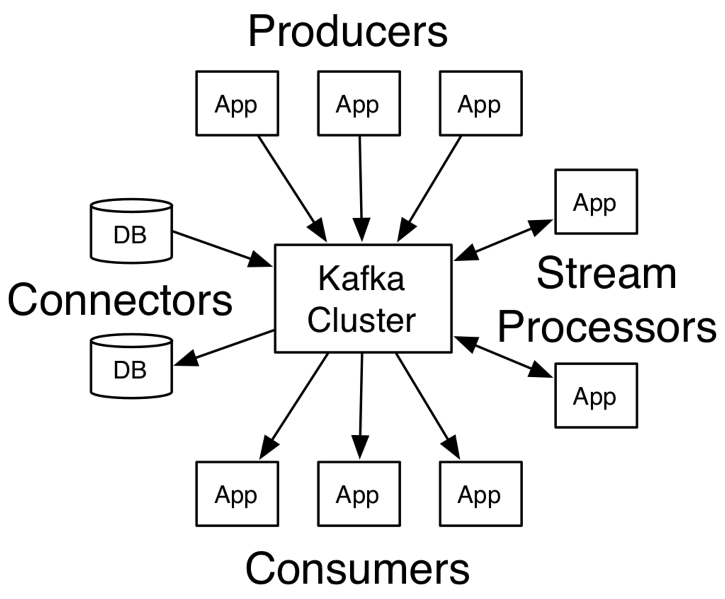

Figure 2.12: Applications using differend kinds of Kafka APIs

```java
1 public void sendData(String data) {
2   Properties configuration = new Properties();
3   configuration.setProperty(
        CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG,
        "localhost:9092");
4   try (Producer<String, String> producer = new
      KafkaProducer<>(configuration)) {
5     producer.send(new ProducerRecord<>("Topic", data));
6   }
7 }
```

Listing 2.13: Example of publishing data using Kafka Producer

2.4.2 Consumer API

Using Kafka Consumer API, application can handle new records that are arriving from server. Listing 2.13 shows receiving such data. We also configure `KafkaConsumer` as in previous section. On line 5 the program calls `subscribe` to register consumer to receive records in topic `Topic`. On line 6 server is queried for new data. We use 1 second as maximal time limit, for which Kafka waits for new records to arrive and then such records are returned and we can handle them.

2.4.3 Stream API

A Kafka Stream represents an unbounded, continuously updating data set. It is an ordered, replayable, and fault-tolerant sequence of immutable data records. Application defines its computational logic through processor topologies, where processor topology is a graph of stream processors (nodes) that are connected by

```java
public void readData() {
  Properties configuration = new Properties();
  configuration.setProperty(
       CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG,
       "localhost:9092");
  try (Consumer<String, String> consumer = new
      KafkaConsumer<>(configuration)) {
    consumer.subscribe(Collections.singletonList("Topic"));
    ConsumerRecords<String, String> data =
        consumer.poll(Duration.ofSeconds(1));
    // handle incoming data
  }
}
```

Listing 2.14: Example of reading data using Kafka Consumer

streams (edges).

Stream processor is a node in the processor topology that represents a processing step to transform data in streams by receiving one input record at a time from its upstream processors in topology, applying its operation on it and then produce one or more output records to its downstream processors.

In topology, there are two special processors:

- **Source Processor** is a stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its down-stream processors.

- **Sink Processor** is a stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.

The way how topology can be defined is using the Kafka Streams Domain Specific Language (DSL). It provides the most common data transformation operations, such as `map`, `filter`, `join` and `aggregations`. There exists also the low level Processor API that allows developers define and connect custom processors and also interact with state stores.

2.4.4 Connector API

A Kafka Connector is a tool for scalably and reliably streaming data between Kafka and other systems. It makes it simple to quickly define connectors that move large collections of data into and out of Kafka. Kafka Connector can ingest entire databases or collect metrics from all application servers into Kafka topics, making the data available for stream processing with low latency.

2.5 Requirements for Data Lineage Library

In previous sections we demonstrated usage of different frameworks that are usually used in Java applications to manipulate data. We ignored their advanced features and showed some examples of reading and writing data.

We described four types of data processing APIs / frameworks:

- **JDBC API**, which is the Java API for working with SQL databases.
- **Spring JDBC Framework**, which simplifies the JDBC API.
- **MyBatis Framework**, which is the ORM framework.
- **Kafka Framework**, which is the distributed streaming platform.

Based on the overview of data processing frameworks, in the rest of this section, we point out requirements on the library that have to be fulfilled to successfully compute data lineage of applications, where such frameworks are used for accessing the data.

1. **Analysis of whole Java application.**
   The analysis should process whole application with all its dependencies.
2. **Identifying the data sources and sinks.**
   The analysis should identify the places in application, where the data are read or written.
3. **Correctness, accuracy and efficiency of computing data lineage.**
   The analysis should compute correct and accurate data lineage graphs in a reasonable time.
4. **Work with external files.**
   The analysis should be able to handle not only Java classes, but also external files as many frameworks store their configuration outside of the code.
5. **Use of concrete values.**
   The analysis should know the concrete values of Java primitives and `Strings`. This is necessary for identifying sources and sinks of data, such as files, database tables, etc.
6. **Handle callbacks.**
   The analysis should work well with callbacks, as they are often used by frameworks to asynchronously notify application about some results of performed operation, or even to simplify the usage of other API.
7. **Easy extendability.**
   The analysis need to be easily extended as new frameworks are developed and the core of symbolic analysis library needs to be prepared to add support for computing their data lineage.

3. Static Analysis

In this chapter, we will introduce concept of static analysis, its possible applications and its limitations, focusing especially on aspects relevant for computing data lineage.

3.1 Program Analysis

Program analysis is the process of automatically analyzing the behavior of computer programs. There are two principal approaches to such analysis:

- **Dynamic program analysis** is performed during program runtime. To perform dynamic analysis, both executable program and its inputs are required. To be effective, the target program must be executed with sufficient test inputs, as its results are limited only to observed executions of the analyzed program.

- **Static program analysis** is the program analysis that is actually performed without executing the input program. The analysis is usually performed on the intermediate representation (IR) of the program source code (used e.g. in compilers), or bytecode. Results of static program analysis cover all execution paths of analysed program.

  Static program analysis technique is very popular. It is thanks to its speed, reliability and sometimes it is the only possible way for complex systems in reasonable time. It is often used to detect program errors, such as security vulnerabilities or performance issues.

Data flow analysis is a technique for gathering information about all possible values of variables during program execution.

3.2 WALA Framework

The T. J. Watson Libraries for Analysis, the WALA Framework [15], is a framework for static analysis capabilities for Java bytecode and related languages.

The main goals of WALA Framework are:

- Robustness
- Efficiency
- Extensibility

The key features WALA Framework provides are:

- Pointer analysis
- Class hierarchy
- Call graph

- Interprocedural dataflow analysis
- Context-sensitive slicing

We describe some of the features in the following subsections.

3.2.1 Pointer analysis

Smaragdakis and Balatsouras [12] define **pointer analysis**, or **points-to analysis** respectively, as a static program analysis that determines information on the values of pointer variables or expressions. It is near-synonym of **alias analysis** that use Sridharan et al. [14]. Pointer analysis typically answer question "what objects can a variable point to?", whereas alias analysis focus on closely related question "can a pair of variables point to the same object?".

Flow and Context Sensitivity

Flow sensitivity refers to ability of an analysis to take control flow into account when analyzing a program. In case, analysis considers statement ordering, it is called **flow-sensitive**, otherwise it is **flow-insensitive** analysis.

Contex sensitivity can be taken into account in an interprocedural analysis and then we call it **contex-sensitive**. Otherwise, when calling context is ommited, such an analysis is called **context-insensitive**.

3.2.2 Class hierarchy

A **class hierarchy** is a structure for set of classes of the analyzed program. It includes also information about the used programming language and the relationships between such classes.

Such relations in programs written in Java language are the `implements` and `extends` relations.

Each class also includes collection of its methods. Methods can be declared in the class, or inherited from parent.

3.2.3 Call Graph

A **call graph** represents calling relationships between subroutines in an analysed computer program. The nodes represents procedures and each directed edge represents procedure calls.

When used in context-sensitive manner, it means that for each procedure the graph contains a separate node for each call stack that procedure can be activated with.

3.3 Symbolic Analysis Library

Symbolic analysis library is a library for computing flow of information in Java program - the data lineage. The Symbolic analysis library is based on symbolic Java bytecode interpreter that was introduced in Parízek [9] and used in the ongoing whose results will be published in a short time in Parízek [10].

The library uses static analysis techniques to construct call graph and for each method it computes its summary for different invocation contexts (symbolic parameter values) based on symbolic bytecode interpretation.

The bytecode analysis is done using WALA Framework [15] that was described in Section 3.2.

Bytecode interpreter performs linear traversal of the method bytecode instructions and computes various information for symbolic variables and constant expressions.

The Symbolic analysis library computes method summaries that for each variable (local varibles, fields, arrays or complex expression) contain data sources and the concrete values (for `String`s and numbers).

3.3.1 Symbolic Analysis Algorithm

The symbolic analysis algorithm for computing static method summaries use:

1. A **fixpoint worklist algorithm** over the list of all methods reachable in the call graph.
2. A **linear symbolic interpretation** of the bytecode of methods.

The library is iteratively updating its method summaries until the fixpoint is reached, i.e. when summaries are not changing.

Several method invocation contexts (method parameters with their associated data flow information) are distinguished. When the newly computed summary for a given method is different from the previous one, all callers and calles of the method are added to the worklist in order to achieve soundness. The algorithm terminates when the summary of each method captures all its results and side effects.

3.3.2 Flow Propagation

In the previous section we described general algorithm for symbolic analysis. However, we did not specify, how the flow information is used in practice.

Now we describe selected important aspects and features of the Symbolic analysis library that are relevant for data lineage information.

Library Methods

As optimizations, library does not process all reachable methods. It analyses just methods of application code and procedures with special meaning that are described in next sections.

For ignored methods, the **identity** is returned. The identity function with respect to flow data propagation is defined as merge of flow data of receiver (the object on which method is called) and all arguments of that method. The result of that merge is then associated as flow data of returned value and receiver.

Strings

Knowing the concrete values of the `String` used in application is valuable when it comes to data lineage of application. `String`s identify file names that are accessed in application, or it can be SQL query to database, etc.

However, concrete values are known only when using as literal in the application. Sometimes, the symbolic analysis cannot determine the precise concrete string value. It is often when it is loaded from file or database, or after some operations like `trim`, `substring`, or even concatenation of `String`s (which works only for `String` literals). Then the actual value is in general unknown, but the flow information must be preserved.

Numeric types

Handling concrete values for numeric Java types works as in case of `String` literals. It is useful to know them, but often they cannot be determined. Flow information must be preserved after the operations with the numbers in any case.

Arrays and Collections

Arrays and collections are integral parts of Java. In this part we describe how they are handled in propagation of flow.

The basic approach is not to distinguish individual items and use just one abstract element summary. In that case, as over-approximation, all possible elements of a given array or collection are considered. Complete flow information is also used when creating `java.util.Iterator` from a collection.

Also, when using `java.util.Map` and `java.util.Properties` instances, analysis does distinguish the keys and corresponding values.

In case of collections of `String`s or numeric values, library provides the information about concrete values stored in collections whenever possible.

3.3.3 Identifying the Sources and Sinks of a Data

Symbolic analysis algorithm that we described in previous section is used to compute data flow in application. When such sources or sinks are identified, the algorithm would propagate such data flow.

Now, the problem is to identify the sources and sinks of a data in the application.

Java I/O

Java input and output (I/O) operations can be used to access data in application. It can be done through standard application `System.in`, `System.out` and `System.err`, where they are identified by the name "System.in", etc. or using external files, where its file name identifies the source (or sink respectively).

The library handles both cases and correctly identifies all the read and write operations of such inputs and outputs.

The library focuses only to inputs and outpus made by classes in `java.io` package but also the `java.nio` package would be supported soon.

JDBC API

The library also contains implementation for identification of database reads and writes using plain JDBC API. The related usage of the JDBC API was already described in Section 2.1.

The library can identify the connection url for the database that application is connecting to, SQL queries and the columns that are read from results in `ResultSet`.

The library focuses only on classes in the package `java.sql`, therefore `javax.sql` API is not supported.

Extendability

The symbolic analysis library can be extended to support other types of data sources and sinks. We will describe this feature in the Section 4.2.

3.3.4 Data Lineage Visualization

The data flow graph is created based on the results of symbolic analysis, based on the computed method summaries.

The graph nodes consist of a set of identified sources and sinks for the data, and its oriented edges are between pair of nodes between which the data flows.

The graph visualization is then created using Graphviz [2] tool in resulted PDF file. More details about the structure of the graph can be found in Section 5.4.

There are several types of graph nodes, from which the most important are:

- `Connection` - defines JDBC connection to database.
- `SQLCommand` - defines standard JDBC SQL query.
- `ParamIndex` - defines index of parameter for query or command.
- `ResultColumn` - defines column index or name from an SQL query result.
- `StreamAction` - defines action on stream I/O (files I/O, standard I/O).
- `EntryArgument` and `EntryArgumentData` - defines data flow from entry point method arguments.
- `EntryReturnVal` and `EntryReturnValData` - defines data flow to entry point method result.

Each node contains few attributes that describe the location in analysed code, where an operation was performed (`java_class_desc` and `java_method_desc` identifies the Java class and method) and more information about the data lineage for such operation. For example, `Connection` node contains the connection url in the `db_connection_desc` attribute and the user used to connect to the database is in the `db_connection_user_desc` attribute. Another example can be the `SQLCommand` node with the SQL statement in the attribute `db_statement_desc`, or `stream_location_desc` that identifies the name of used file in `StreamAction` node, or standard I/O (e.g. `System.in`, etc.).

The entry point nodes (`EntryArgument`, `EntryReturnVal`, etc.) show that data can flow into the entry method as its argument, or flow out from such method as its result.

Listing 3.1 show more complex example, where both database and file operations occurs. For simplicity we omit boilerplate code (such as closing connections and file stream, etc.).

On line 2, the database connection is opened, on lines 3–5 the query with `id` argument is created and executed. From `ResultSet` the `VALUE` column is loaded and on line 8 its value is written to the file `outputFile.txt`.

```
1 public static void writeValueForIdToFile(int id) throws Exception {
2   Connection connection = DriverManager.getConnection(
        "jdbc:oracle:thin:@192.168.0.16:1521:orcl",
        "User",
        "Password");
3   PreparedStatement queryStatement = connection.prepareStatement(
        "SELECT ID, VALUE FROM TABLE_NAME WHERE ID = ?");
4   queryStatement.setInt(1, id);
5   ResultSet resultSet = queryStatement.executeQuery();
6   String value = resultSet.getString("VALUE");
7   FileWriter output = new FileWriter("outputFile" + ".txt");
8   output.write(value);
9 }
```

Listing 3.1: Example of code with both database reading and file writing operations

Figure 3.2 contains visualization of the data flow graph of the `writeValueForIdToFile` method from Listing 3.1. We just omit some unessential information from nodes.

The visualization contains following nodes:

1. `EntryArgument0` and `EntryArgumentData0` nodes for identification of the `id` argument.
2. `Connection` node for identification of database connection - url and user.
3. `ParamIndex` and `SQLCommand` nodes for identification of a database query.
4. `ResultColumn` node for loading value from the query.
5. `StreamAction` nodes for identification of the output file.

Besides the main flow in the graph from the database query to the output file, we can notice that the correct file name (`outputFile.txt`) is computed by the library. Notice also flow from the `id` argument into SQL statement as its parameter.

In the visualization we can also notice two `StreamAction` nodes but with different tags - *Open* and *Write*. Tags represents the operations that were made to that stream. The flow goes from *Write* to *Open*, as the `write` call is made on the opened stream.

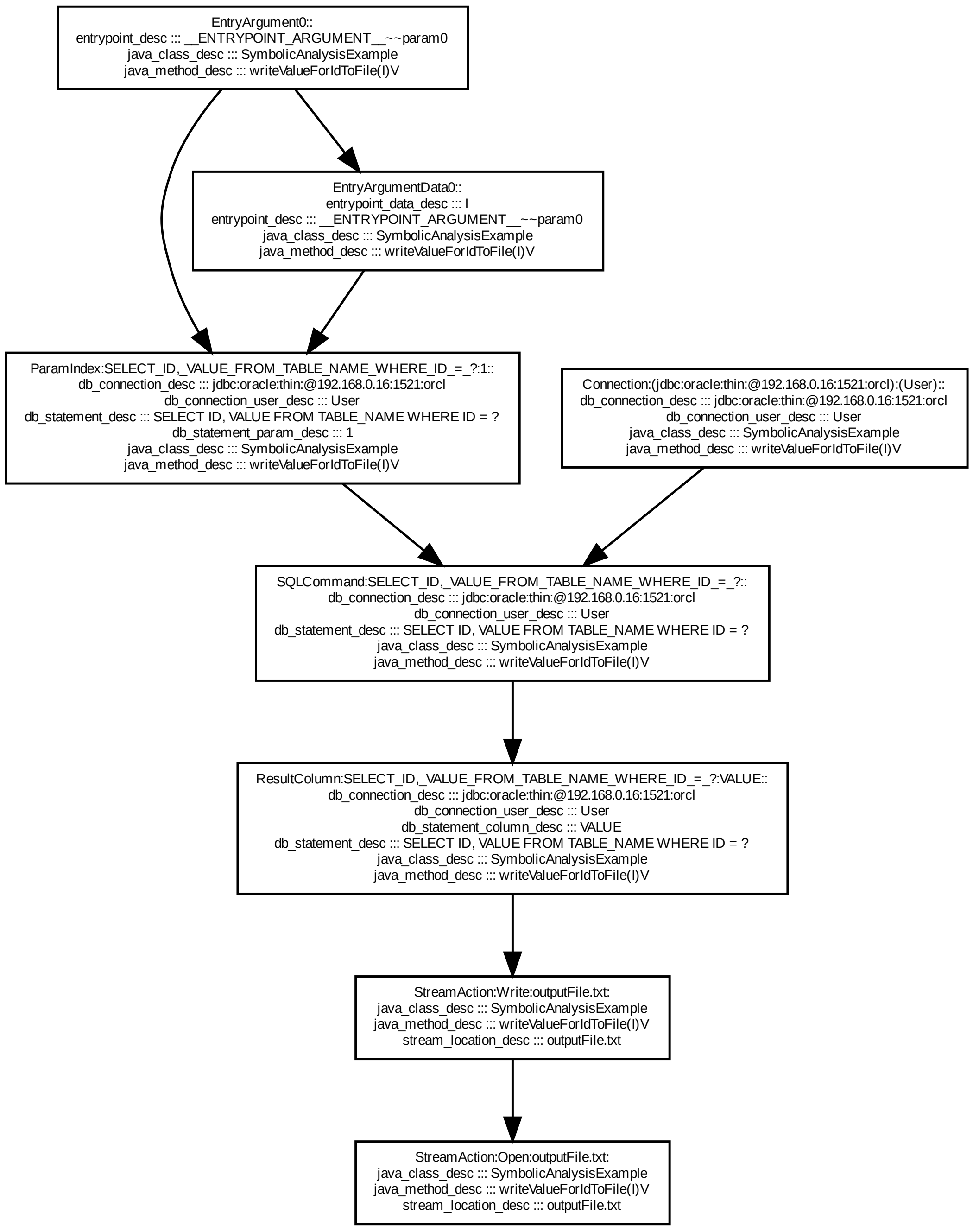
EntryArgument0::
entrypoint_desc ::: __ENTRYPOINT_ARGUMENT__~~param0
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
EntryArgumentData0::
entrypoint_data_desc ::: I
entrypoint_desc ::: __ENTRYPOINT_ARGUMENT__~~param0
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
ParamIndex:SELECT_ID,_VALUE_FROM_TABLE_NAME_WHERE_ID_=_?:1::
db_connection_desc ::: jdbc:oracle:thin:@192.168.0.16:1521:orcl
db_connection_user_desc ::: User
db_statement_desc ::: SELECT ID, VALUE FROM TABLE_NAME WHERE ID = ?
db_statement_param_desc ::: 1
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
Connection:(jdbc:oracle:thin:@192.168.0.16:1521:orcl):(User)::
db_connection_desc ::: jdbc:oracle:thin:@192.168.0.16:1521:orcl
db_connection_user_desc ::: User
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
SQLCommand:SELECT_ID,_VALUE_FROM_TABLE_NAME_WHERE_ID_=_?::
db_connection_desc ::: jdbc:oracle:thin:@192.168.0.16:1521:orcl
db_connection_user_desc ::: User
db_statement_desc ::: SELECT ID, VALUE FROM TABLE_NAME WHERE ID = ?
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
ResultColumn:SELECT_ID,_VALUE_FROM_TABLE_NAME_WHERE_ID_=_?:VALUE::
db_connection_desc ::: jdbc:oracle:thin:@192.168.0.16:1521:orcl
db_connection_user_desc ::: User
db_statement_column_desc ::: VALUE
db_statement_desc ::: SELECT ID, VALUE FROM TABLE_NAME WHERE ID = ?
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
StreamAction:Write:outputFile.txt:
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
stream_location_desc ::: outputFile.txt
StreamAction:Open:outputFile.txt:
java_class_desc ::: SymbolicAnalysisExample
java_method_desc ::: writeValueForIdToFile(I)V
stream_location_desc ::: outputFile.txt

Figure 3.2: Visualization of data lineage graph

4. Data Lineage Analysis for Frameworks

In Chapter 2 we described Java frameworks that are often used in applications to manipulate with data.

In Section 2.5 we described the key requirements on the library to analyse data lineage of such applications. The analysis library should:

- Identify the data sources and sinks.
- Create correct and accurate data lineage in reasonable time.
- Work with the concrete values for Java primitives and `String`s.
- Work with external files.
- Support the callback objects.
- Be easily extendible to support new frameworks.

In Section 3.3 we described the static program analysis approach used by the existing library for creating data lineage of plain Java programs. Using that library, data lineage can be created for Java I/O and JDBC API.

In this chapter, we will present our solution of data lineage analysis for database frameworks - the Java Resolver tool. The Java Resolver tool uses the Symbolic analysis library, which takes care of the data flow propagation between sources and sinks. The Java Resolver tool implements features for identifying the sources and sinks of data inside frameworks as plugins to the Symbolic analysis library.

We will discuss the design of the Java Resolver tool and some technical details of implemented Symbolic analysis library plugins.

4.1 Symbolic Analysis Library Plugins

In this section we describe the main ideas about plugin approach of our Java Resolver tool.

When the Symbolic analysis library performs the program analysis, the plugin handles the selected method calls on frameworks and provides information about the data sources and data sinks to the library.

An input of a plugin is computed data flow information about the receiver (`this` object) and the method call arguments. The Section 3.3.1 describes the Symbolic analysis algorithm as iterative updating of method summaries until a fixpoint is reached.

The same approach applies for the plugins. At each iteration, the flow information about the plugin inputs are computed by the library and based on their values, plugin provides information about the data sources and sinks of a framework method call.

4.2 Interface to Symbolic Analysis

We proposed the general interface between Symbolic analysis library and its plugins. Communication between the library and plugins is described in Figure 4.1.

In next sections we will describe all the interfaces used between the library and the plugins.

4.2.1 FrameworkAnalysisPlugin

The key interface for plugin to Symbolic analysis library is the `FrameworkAnalysisPlugin`. The interface defines few methods that are used for communication between the library and plugins. Here we describe their usage:

- `initialize` and `analyzeProgram` initialize plugins and precompute some information from the provided call graph and class hierarchy of an analysed program.

- `getAdditionalMethodsRequiredForAnalysis` specify framework methods that should be also analysed by the Symbolic analysis library, as the library does not analyse methods out of defined application package because of optimizations. Sometimes it is useful to analyse other framework methods (e.g. when there are many overloading methods that forward their calls to one method that can be handled by a plugin).

- `canHandleMethod` and `processMethodCall` query plugin whether method can be handled by the plugin and to compute flow information for such method.

- `canProvideArgument` and `getArgumentFlowInformation` provide flow information for an argument of a callback method.

- `destroy` closes all plugin resources. The method is used at the end of analysis, when the fixpoint is reached.

4.2.2 MethodCallDescription

The class `MethodCallDescription` serves as data flow information input for a plugin query (the `processMethodCall` method). It holds the attributes for all method inputs (for receiver (`this` object) and all the method arguments) and also flow information about the previously analysed callbacks.

4.2.3 MethodEffectsDescription

The class `MethodEffectsDescription` serves as output of the `processMethodCall` method. It holds the output data flow information and also identifies callbacks to be computed by the Symbolic analysis library.

Flow information are stored in the `DataEndpointFlowInfo` object that is described in the next section.

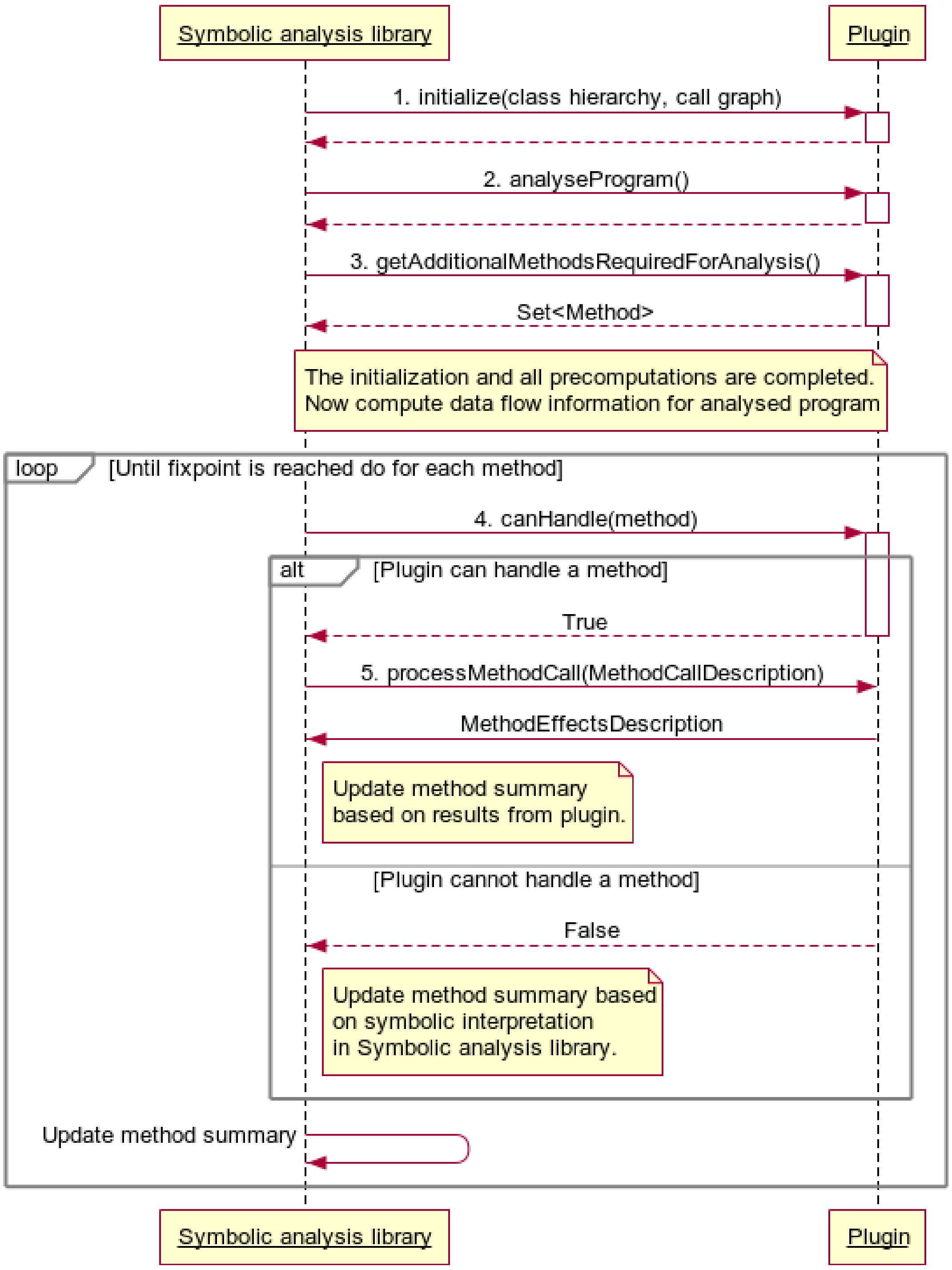
Symbolic analysis library
Plugin
1. initialize(class hierarchy, call graph)
2. analyseProgram()
3. getAdditionalMethodsRequiredForAnalysis()
Set<Method>
The initialization and all precomputations are completed.
Now compute data flow information for analysed program
loop
[Until fixpoint is reached do for each method]
4. canHandle(method)
alt
[Plugin can handle a method]
True
5. processMethodCall(MethodCallDescription)
MethodEffectsDescription
Update method summary
based on results from plugin.
[Plugin cannot handle a method]
False
Update method summary based
on symbolic interpretation
in Symbolic analysis library.
Update method summary
Symbolic analysis library
Plugin

Figure 4.1: Communication between Symbolic analysis and the plugins

4.2.4 DataEndpointFlowInfo

The class `DataEndpointFlowInfo` holds data flow information results for an analysed method, such as:

- Identification of the data source or sink.
- Mapping of fields in the resulting objects to data source or sink information (e.g. database column name). The results of a framework call can be returned by a method and also stored into a method attribute.
- Attributes for:
  - Data source or sink
  - Returned object
  - Receiver (`this`)
  - Method call arguments

The data source and sink identification can be the SQL statement in case of database frameworks, or the server identification, or other important information for data lineage.

Plugin can define its own attributes that can be used to identify any valuable information. All of the attributes are then propagated using the Symbolic analysis library.

4.3 JDBC API

Implementation of Data Lineage for standard JDBC API in `java.sql` package is done using the Symbolic analysis library, as we show in the Section 3.3.

In this section we will describe our solution for `DataSource` interfaces in `javax.sql` package. It is nowadays the preferred way of creating connections to database and the Symbolic Analysis library do not handle such information.

4.3.1 DataSource

Database vendors usually provide `DataSource` implementation, which still needs some configuration about database - connection url, credentials and probably much more.

To identify target database, we need information about connection url and credentials. These properties are usually set to `DataSource` objects and from these calls, we can get their values.

The plugins identify the methods where such information is set and return the concrete values of used `String`s for propagation by Symbolic analysis library.

4.4 Spring JDBC Framework

Spring JDBC Framework comes with approach of callbacks. It handles all boilerplate code and calls user defined callback objects to do their job and after finishing it, framework handles closing all resources that are not needed anymore.

When we looked to implementation of `JdbcTemplate` class, we found out that database calls are actually made only by 4 out of about 50 of its methods. All the other methods only use them when executing queries.

4.4.1 Callbacks

As we said, callbacks are used just to wrap application logic and separate it from the code for opening/closing connections, etc, as was shown by previous example in Listing 2.5.

When handling a particular method of the framework (e.g. the `update` method of `JdbcTemplate` class in Listing 2.5), we needed to find a way how to say to the Symbolic analysis library that a plugin wants to analyse that callback method. For that purpose, we use field `callbackMethodsToAnalyze` of the class `MethodEffectsDescription` from the interface. All callbacks have the method that is called in execution and we just leave analysis library to analyse it and return its data flow. After that, we append the resulting flow of the callback to flow of our handled method.

We will demonstrate it on simple example of the `insert` method from Listing 2.5. In the method, there is the call of the `jdbcTemplate.update()` method with two arguments - the SQL statement and the instance of the `PreparedStatementSetter`.

After data lineage analysis of the `insert` method is done, we would like to know that the `id` and `value` attributes of the `DatabaseValue` instance were arguments of the executed SQL statement. However, Symbolic analysis library does not know that, inside of `jdbcTemplate.update()`, the `setValues` method of the `PreparedStatementSetter` is called.

Our solution is that when the plugin is handling `jdbcTemplate.execute()` for the first time, it tells library that it wants to analyse that `setValues` call. The next time the plugin already has all the information about data lineage in the `setValues` and therefore it can propagate it to the result of the `jdbcTemplate.update()` call.

4.5 MyBatis

In chapter 2.3 we showed classic use case of loading data from database using the MyBatis framework. Now, we present solution of computing data lineage for this framework.

4.5.1 Mapper interfaces

First, it is necessary to find all the mapper interfaces. To do this, we iterate all classes in class hierarchy and search for interfaces with methods annotated with MyBatis annotations, or interfaces for which XML mapper definition exists.

XML mapper files

XML mapper file definition has the same name as the corresponding interface (but .xml extension is used instead of .java), and lays in same directory.

We made a parser of such XML files that can get information about SQL statements and mapping of columns to object fields.

Parser supports reusable SQL fragments. Each fragment has its own ID, so it is easy to find correct one and include it into query.

Parser also supports dynamic SQLs. As we do not know which conditions are valid at runtime, we made a simplification that we always use just first branch that was seen in the code. We also cannot determine the size of an input collection, so the parser provides no expansion.

Annotated mapper classes

When mappers use annotations, it is quite simple to get all data needed from WALA. We know that an SQL statement is always stored in annotations `@Select`, `@Insert`, `@Delete` or `@Update` and queries are stored in plain `String` array. Queries can also contain dynamic SQL as in XML definitions.

For mapping result object the `@Results` annotation is used with `@Result` for every column to object property mapping definition. Mapping can be done using `@ConstructorArgs`, when columns are mapped to arguments of a constructor. In that case, the plugin unfortunately does not know the name of target property, as arbitrary code can be executed in constructor[1].

Framework can provide result of an SQL query in two ways:

1. As a return statement of a mapper method.
2. As a method argument property.

In all previous mapper examples, we use the first approach, where result is returned from method in return statement.

However, when calling database procedure, we can define its output arguments and framework fill result values into that method argument. It can be `Map<String, Object>` object (where mapper references to key in that map), or any arbitrary object (where mapper references field, in which result will be stored).

Listing 4.2 illustrates the use of such `Map` as input and also output method argument. The database call of `DB_PROCEDURE` has two arguments.

First, the input integer argument, takes value of the `id` key from the map. Second, the output cursor argument, is created by MyBatis and perform database procedure call. The cursor is then transformed to list of target objects with result map `resultMap` and stored to the same map under the key `result`.

There also exist more advanced features of MyBatis[2] but we do not handle them. All of them can be added as new functionality in future development, if needed.

[1]Arbitrary code can be executed also in setters, but it is usually used just as setting new reference of a property.

[2]Such as using provider classes to create SQL queries (`@SelectProvider`, ...), generating IDs from sequence (`@SelectKey`) and using them in queries, generating maps from objects (`@MapKey`), associations for attribute classes (`@One` or `@Many`), mixing XML and annotation mappers (e.g. define `<resultMap>` in XML and reference it using `@ResultMap` annotation)

```xml
<select id="selectForMap" parameterType="java.util.Map"
    statementType="CALLABLE">
  {
    CALL DB_PROCEDURE
    (
      <!-- id is stored in map with key="id" -->
      #{id,    mode=IN, jdbcType=INTEGER, javaType=Integer},
      <!-- output will be stored in map with key="result" -->
      #{result, mode=OUT, jdbcType=CURSOR, javaType=ResultSet,
          resultMap=resultMap}
    )
  }
</select>
<resultMap id="resultMap">
  ...
</resultMap>
```

Listing 4.2: Using `Map` as argument for input and output

4.5.2 Database connection configuration

Configuration of MyBatis can be done in two ways, using XML configuration file or in Java using `Configuration` class. Second way needs no more attention, as we use `DataSource` classes that are already handled in section 4.3.1.

To handle XML configuration, our plugin needs to parse configuration file and find `<dataSource>` section and its `driver`, `url` and `username` properties. All this information we need for identifying target database.

4.6 Kafka

In section 2.4 we demonstrated the main use cases for data manipulation (producing or consuming) using Kafka framework. In this section, we show how data lineage for these examples can be computed using our plugin. In our work, we focused only on Consumer and Producer APIs, as the other two (Stream and Connector APIs) are much more advanced to use.

The plugin identifies the used Kafka server which is used in application to communicate with. It also identifies the methods that are used to set Kafka topics and the methods that are used to send and receive a data.

Together, the used server, topic and the place where data are produced and consumed, are the main data lineage information about the sources and sinks of a data.

4.6.1 Callbacks

Kafka Framework uses callbacks to asynchronously notify application. It can be callbacks to notify about performed operation result (like in case of sending data), or to notify about some state changes.

As in case of callbacks in Spring JDBC Framework, the plugin tells the Symbolic analysis library to analyse the data flow in such callback method and then plugin propagate computed data flow information to the result of handled method.

4.6.2 Properties

Kafka Framework uses Java `Properties` to its configuration.

`Properties` can be set in Java code (like it was done in Listing 2.14 and Listing 2.4.1). They can be also loaded from external file like in next code:

```
new Properties().load(new FileReader("file.txt"));
```

When the `Properties` values are set only using Java, the concrete values are resolved by Symbolic analysis library.

However, the library does not support the external files for loading properties. It provides only the stream identification (name of used file, "System.in", etc.).

If there exists a file with such identification, plugin loads its properties and tries to resolve what Kafka server was used in application.

5. User Documentation for the Java Resolver Tool

We created a set of plugins for Symbolic analysis library for extracting data lineage information from data processing frameworks used in Java applications. We also provide some test scenarios for frameworks we are handling.

We created the Java Resolver tool, which is composition of the Symbolic analysis library together with the plugins.

Here we describe installation and usage of the Java Resolver tool. We use standard Maven directory structure for source code. In `src/main` directory, application source and resource files are located and in `src/test` test sources and resources are located.

5.1 Software Requirements

To successfully install the Java Resolver tool, following is required:

- Java JDK 1.8 installed
- Apache Maven installed and configured
  - Recommended version is 3.3.9
- Graphviz installed with tool `dot` on PATH
  - Recommended version is 2.40.1

We recommend to use such versions of the software, as these versions were used for testing and some compatibility issues can be encountered using different versions.

5.2 Installation

To compile the Java Resolver tool and create runnable `JAR` file, use the command:

```
# mvn clean compile assembly:single
```

The command creates the executable `target/resolver.jar`. We recommend also to run all tests after instalation using:

```
# mvn test
```

When tests ends, the flow graph visualizations are saved in the `target/img` directory.

As it could take a long time to run all tests (more than an hour), we provide TestNG suites for each framework. This can be useful to verify that requested plugin is working before trying to use it.

The test suite files are located in subdirectories of `test/resources/tests` and can be run using command:

```
# mvn test -DtestSuite=<suiteFile>
```

5.3 Running the Java Resolver Tool

After instalation is completed, the Java Resolver tool is ready to be used for computing data lineage of target application using command line options:

```
# java -jar target/resolver.jar [OPTIONS]

OPTIONS:
  [--entry <className> <methodName>]
  [--application-jar <fileName>]
  [--library-jar <fileName>]
  [--application-package <package>]
  [--output-directory <directoryName>]
  [--help]
```

- `--entry` specify an entry point for the analysis. It can be some method, from which data lineage is computed. Fully qualified name of class should be provided and only methods without arguments and standard Java main method are supported.
- `--aplication-jar` specifies the application JAR file to be analysed by our Java Resolver tool.
- `--library-jar` specifies the library JAR file to be known by our Java Resolver tool. All library dependencies should be added for analysis.
- `--aplication-package` sets the root package of analysed application. This is needed, because of optimizations, when our Java Resolver tool does not analyse classes that are outside of that package.
- `--output-directory` specifies where the result files are stored.
- `--help` display usage and exit.

5.4 Result Graphs

The data lineage of an input program is represented by a flow graph, where nodes are data sources and sinks and oriented edges are between pair of nodes between which the data flows.

The graph can be visualized using Symbolic analysis library visualization tool described in Section 3.3.4.

The Java Resolver tool generates three types of files. First, the `.dot` file contains graph definition from which the `dot` tool creates `.pdf` and `.svg` files with visualized result graph.

The result visualization was described in Section 3.3.4 for JDBC and I/O APIs. Now we continue the list of node types from that section that are relevant to Symbolic analysis library plugins:

- `FrameworkDataSource` - defines source of data in framework (like SQL query statement).
- `FrameworkDataField` - defines resulting field from framework query.
- `FrameworkAction` - defines the data sink (like SQL insert statement).

Plugins define also own attributes for identification of some valuable information for data lineage. Such attribute can be `KAFKA_TOPIC` that is used for identification of used topic in Kafka Framework, or `FILE_NAME` that identifies a file name that was used as some input in program, like the configuration file in MyBatis. There are also much more attributes.

5.5 Implementing Support for new Framework

We implemented support for three data processing frameworks, but in general, any framework can be handled. It can be done by implementing a new plugin for Symbolic analysis library.

In Section 4.2 we described interface between Symbolic analysis library and its plugins. Plugins should identify the data sources and sinks and the library take care of the data flow between them.

Before the implementation of new plugin we strongly recomend to make familiar with already implemented plugins and other classes used in that plugins:

- `JdbcTemplateAnalysisPlugin` and `JdbcTemplateHandler` can be inspiration for the approach of callbacks.
- `MyBatisAnalysisPlugin` can be inspiration, when plugin should work with external files or annotations.
  - `MyBatisAnnotationMapperSqlReader` - working with annotations
  - `MyBatisXmlMapperSqlReader` - working with external XML files

6. Evaluation

In this chapter, we discuss the results of our Java Resolver tool. The tool provides implementation of plugins for three frameworks that are used for manipulating data - Spring JDBC, MyBatis and Kafka. It also provides implementation for the identification of many types of databases through the `DataSource` interface.

For the Java Resolver tool we created a comprehensive set of tests. They tests major part of the features of the tool and show that our tool is in a good condition.

The Java Resolver tool can also visualize data flow of JDBC and I/O API that was implemented in the Symbolic analysis library and is not part of our work.

In the next sections, we evaluate results of our Java Resolver tool for each of the selected frameworks. At the end, we discuss limitations of the current solution and fulfillment of the requirements from Section 2.5.

6.1 Data Flow Graph Visualizations for Frameworks

In this section we present some data flow visualizations done by our Java Resolver tool for each of the supported frameworks. The visualizations show only the important data lineage information from the computed data flow graphs.

Spring JDBC Framework

Figure 6.1 show created visualization for Spring JDBC Framework. The data flow graph was computed based on the analysis of the method `JdbcTemplateBasicTarget#runHandleExecuteWithConnectionCallback()`.

We can see the query statement done on the database table `TABLE_NAME` from which the `JdbcTemplateModel` is created and its property `value` is then used as the first parameter of the insert statement into the database table `OUTPUT_TABLE`.

MyBatis Framework

Figure 6.2 show created visualization for MyBatis Framework. The data flow graph was computed based on the analysis of the method `MyBatisMapperTarget#runSelectAllAndDelete()`.

The visualization show data flow between database select and delete statements. We can notice that database connection was configured using configuration file (see `FILE_NAME` attribute).

The mappers and their methods (`SelectAllAnnotatedMapper#selectAll()` and `DeleteAnnotatedMapper#delete()`) that were used to execute the database calls are also identified by the Java Resolver tool.

Kafka Framework

Figure 6.3 show created visualization for Kafka Framework. The data flow graph was computed based on the analysis of the method

`KafkaMixedTarget#runKafkaProducerAndConsumer()`.

The visualization show identified Kafka server for both data source and sink. We can notice two parallel data flows. One identifies only the source part (the call of `poll` method) and second identifies the sink part (the call of `send` method). However, the data flow from the source to the sink is not correctly identified by the Symbolic analysis library.

We can notice the over-approximation done by the Java Resolver tool. In the `send` method call only topic `ProducerTopic` is used. However, he tool cannot distinguish the correct topic that was used and therefore it identifies both of them (`ConsumerTopic` and `ProducerTopic`).

6.2 Limitations of the Java Resolver Tool

There are few limitations of our Java Resolver tool. We point them out in the following list:

1. **Slow data lineage computation.**
   The data lineage computation tests in our project last from a few tens of seconds to a few minutes. It depends on the number of used libraries, their complexity and also from the complexity of analysed application.

2. **There can be many algorithm iterations until fixpoint is reached.**
   The Symbolic analysis library computes method summaries iteratively until fixpoint is reached - when there are no changes in the method summaries. The number of iterations also depends on the number of methods in the analysed program.

3. **Some inaccuracies can occur based on the used over-approximation algorithm.**
   The Symbolic analysis library computes over-approximate data lineage information (e.g. when accessing an element of an array or collection, all elements are considered as output).

4. **Disadvantages of static analysis.**
   Many disadvantages come with the usage of static analysis (e.g. when values are computed dynamically based on the external inputs).

6.3 Fulfillment of the Requirements

In the Section 2.5 we pointed out the requirements on the Java Resolver tool to be able to successfully compute the data lineage of applications that use frameworks for accessing the data.

In the next list we try to explain how the requirements are fulfilled (or not) by the Java Resolver tool:

1. **Analysis of whole Java application.**
   Our tool analyses both application and its dependencies to distinguish all the data flows. The used WALA framework needs all dependencies to create a correct call graph.

2. **Identifying the data sources and sinks.**
   The plugins for Symbolic analysis library identify the data sources and sinks of an analysed application when the corresponding frameworks are used.

3. **Correctness, accuracy and efficiency of computing data lineage.**
   We explained some limitations of our Java Resolver tool in Section 6.2. The data lineage computation can take quite long time, depending on the complexity of an analysed application and the number of used libraries. However, it is expected that analysis of a huge program with many library dependencies will run longer than the analysis of other small Java application. As the tests in the Java Resolver tool show, the data lineage is computed in few minutes, therefore the tool is usable in practice.

4. **Work with external files.**
   The external configuration files are often handled by our plugins for the Symbolic analysis library.

5. **Use of concrete values.**
   The Symbolic analysis library always tries to resolve actual values of Java primitives and `String`s. However, even if in many cases the concrete values cannot be determined, the data flow between sources and sinks are preserved. As the frameworks often use external files for storing its data, that files can be often read by the Java Resolver tool to identify the data sources and sinks.

6. **Handle callbacks.**
   The Symbolic analysis library supports also computing the data flow in callbacks from the frameworks to applications.

7. **Easy extendability.**
   We proposed the pluggable architecture for the Symbolic analysis library. We demonstrated the possibilities that result from the changed architecture on several library plugins. The plugins identify the data sources and sinks of used data processing frameworks.

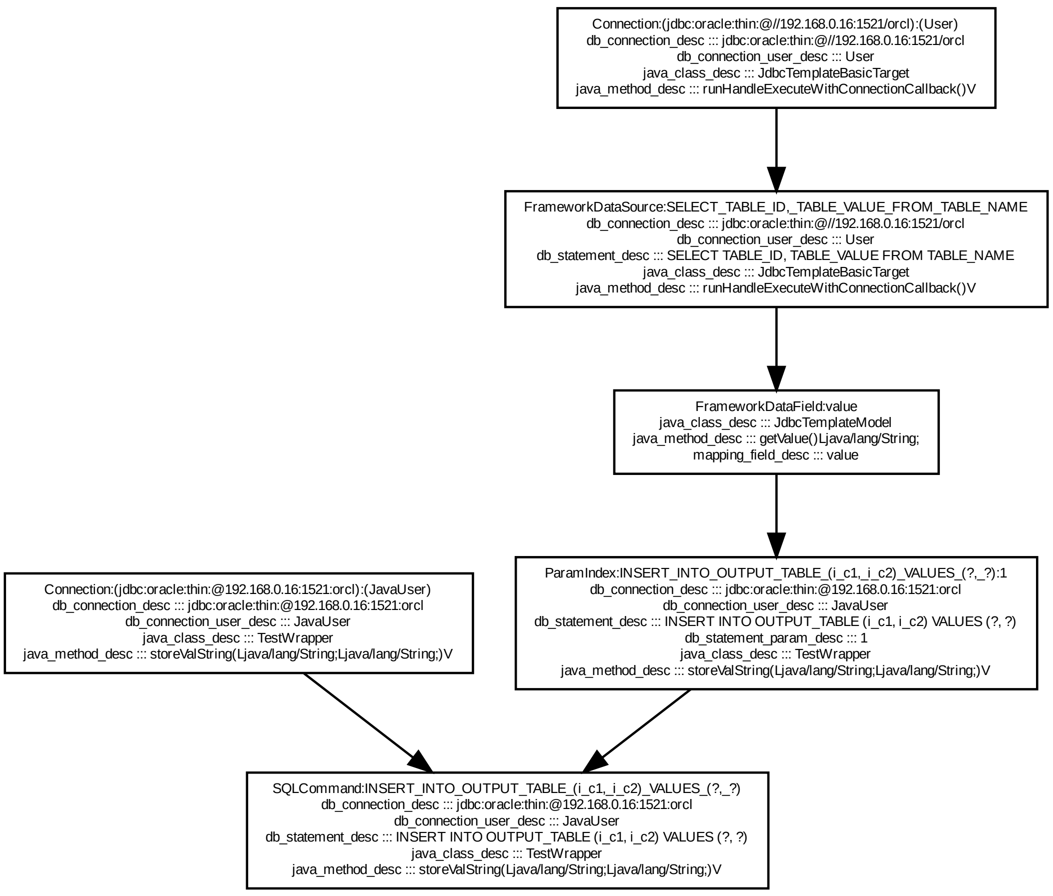
Connection:(jdbc:oracle:thin:@//192.168.0.16:1521/orcl):(User)
db_connection_desc ::: jdbc:oracle:thin:@//192.168.0.16:1521/orcl
db_connection_user_desc ::: User
java_class_desc ::: JdbcTemplateBasicTarget
java_method_desc ::: runHandleExecuteWithConnectionCallback()V
FrameworkDataSource:SELECT_TABLE_ID,_TABLE_VALUE_FROM_TABLE_NAME
db_connection_desc ::: jdbc:oracle:thin:@//192.168.0.16:1521/orcl
db_connection_user_desc ::: User
db_statement_desc ::: SELECT TABLE_ID, TABLE_VALUE FROM TABLE_NAME
java_class_desc ::: JdbcTemplateBasicTarget
java_method_desc ::: runHandleExecuteWithConnectionCallback()V
FrameworkDataField:value
java_class_desc ::: JdbcTemplateModel
java_method_desc ::: getValue()Ljava/lang/String;
mapping_field_desc ::: value
Connection:(jdbc:oracle:thin:@192.168.0.16:1521:orcl):(JavaUser)
db_connection_desc ::: jdbc:oracle:thin:@192.168.0.16:1521:orcl
db_connection_user_desc ::: JavaUser
java_class_desc ::: TestWrapper
java_method_desc ::: storeValString(Ljava/lang/String;Ljava/lang/String;)V
ParamIndex:INSERT_INTO_OUTPUT_TABLE_(i_c1,_i_c2)_VALUES_(?,_?):1
db_connection_desc ::: jdbc:oracle:thin:@192.168.0.16:1521:orcl
db_connection_user_desc ::: JavaUser
db_statement_desc ::: INSERT INTO OUTPUT_TABLE (i_c1, i_c2) VALUES (?, ?)
db_statement_param_desc ::: 1
java_class_desc ::: TestWrapper
java_method_desc ::: storeValString(Ljava/lang/String;Ljava/lang/String;)V
SQLCommand:INSERT_INTO_OUTPUT_TABLE_(i_c1,_i_c2)_VALUES_(?,_?)
db_connection_desc ::: jdbc:oracle:thin:@192.168.0.16:1521:orcl
db_connection_user_desc ::: JavaUser
db_statement_desc ::: INSERT INTO OUTPUT_TABLE (i_c1, i_c2) VALUES (?, ?)
java_class_desc ::: TestWrapper
java_method_desc ::: storeValString(Ljava/lang/String;Ljava/lang/String;)V

Figure 6.1: Visualization of data lineage graph for Spring JDBC Framework

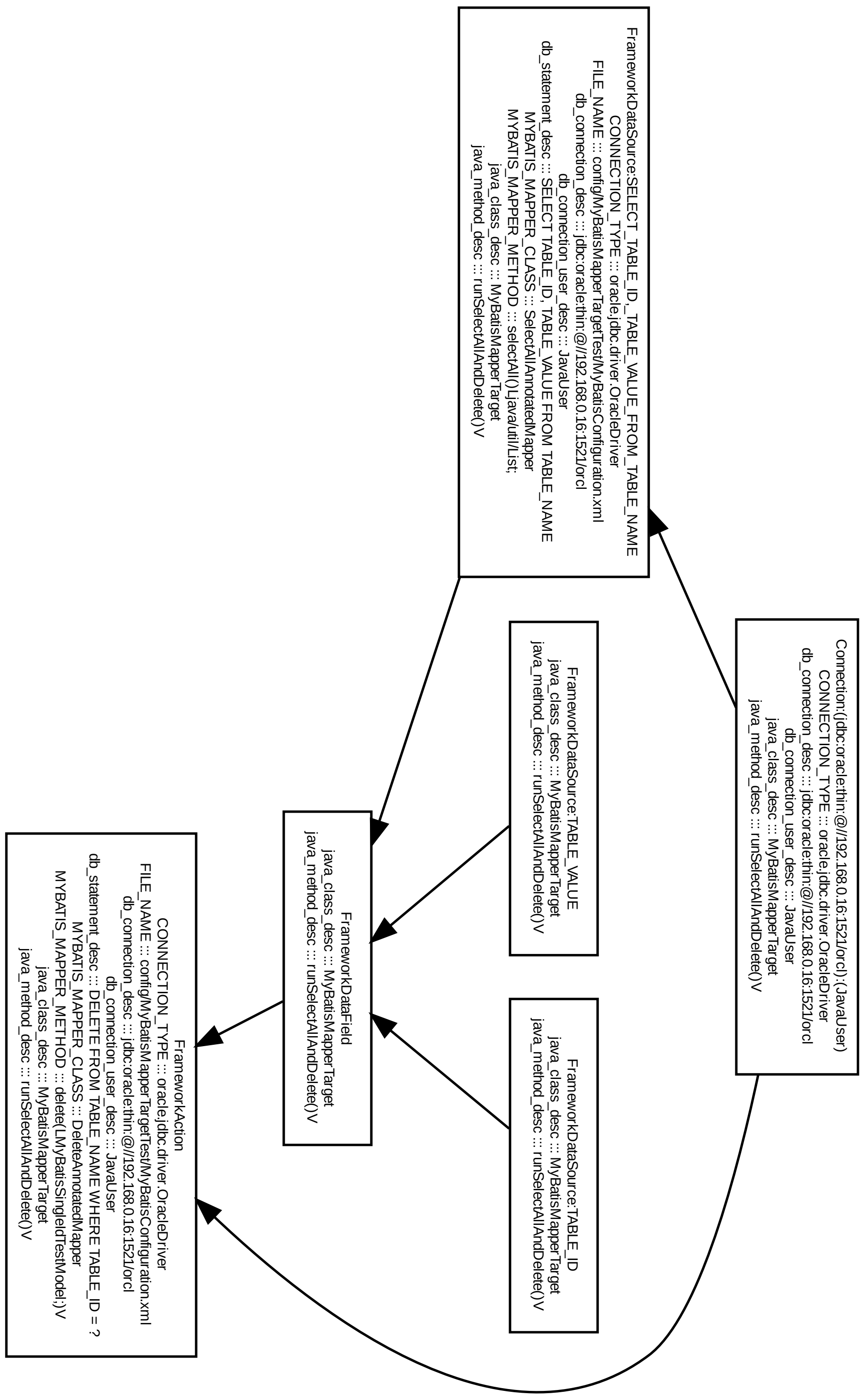
FrameworkDataSource:SELECT_TABLE_ID_TABLE_VALUE_FROM_TABLE_NAME
CONNECTION_TYPE ::: oracle.jdbc.driver.OracleDriver
FILE_NAME ::: config/MyBatisMapperTargetTest/MyBatisConfiguration.xml
db_connection_desc ::: jdbc:oracle:thin:@//192.168.0.16:1521/orcl
db_connection_user_desc ::: JavaUser
db_statement_desc ::: SELECT TABLE_ID, TABLE_VALUE FROM TABLE_NAME
MYBATIS_MAPPER_CLASS ::: SelectAllAnnotatedMapper
MYBATIS_MAPPER_METHOD ::: selectAll()Ljava/util/List;
java_class_desc ::: MyBatisMapperTarget
java_method_desc ::: runSelectAllAndDelete()V
FrameworkDataSource:TABLE_VALUE
java_class_desc ::: MyBatisMapperTarget
java_method_desc ::: runSelectAllAndDelete()V
Connection:(jdbc:oracle:thin:@//192.168.0.16:1521/orcl):(JavaUser)
CONNECTION_TYPE ::: oracle.jdbc.driver.OracleDriver
db_connection_desc ::: jdbc:oracle:thin:@//192.168.0.16:1521/orcl
db_connection_user_desc ::: JavaUser
java_class_desc ::: MyBatisMapperTarget
java_method_desc ::: runSelectAllAndDelete()V
FrameworkDataField
java_class_desc ::: MyBatisMapperTarget
java_method_desc ::: runSelectAllAndDelete()V
FrameworkDataSource:TABLE_ID
java_class_desc ::: MyBatisMapperTarget
java_method_desc ::: runSelectAllAndDelete()V
FrameworkAction
CONNECTION_TYPE ::: oracle.jdbc.driver.OracleDriver
FILE_NAME ::: config/MyBatisMapperTargetTest/MyBatisConfiguration.xml
db_connection_desc ::: jdbc:oracle:thin:@//192.168.0.16:1521/orcl
db_connection_user_desc ::: JavaUser
db_statement_desc ::: DELETE FROM TABLE_NAME WHERE TABLE_ID = ?
MYBATIS_MAPPER_CLASS ::: DeleteAnnotatedMapper
MYBATIS_MAPPER_METHOD ::: delete(LMyBatisSingleIdTestModel;)V
java_class_desc ::: MyBatisMapperTarget
java_method_desc ::: runSelectAllAndDelete()V

Figure 6.2: Visualization of data lineage graph for MyBatis Framework

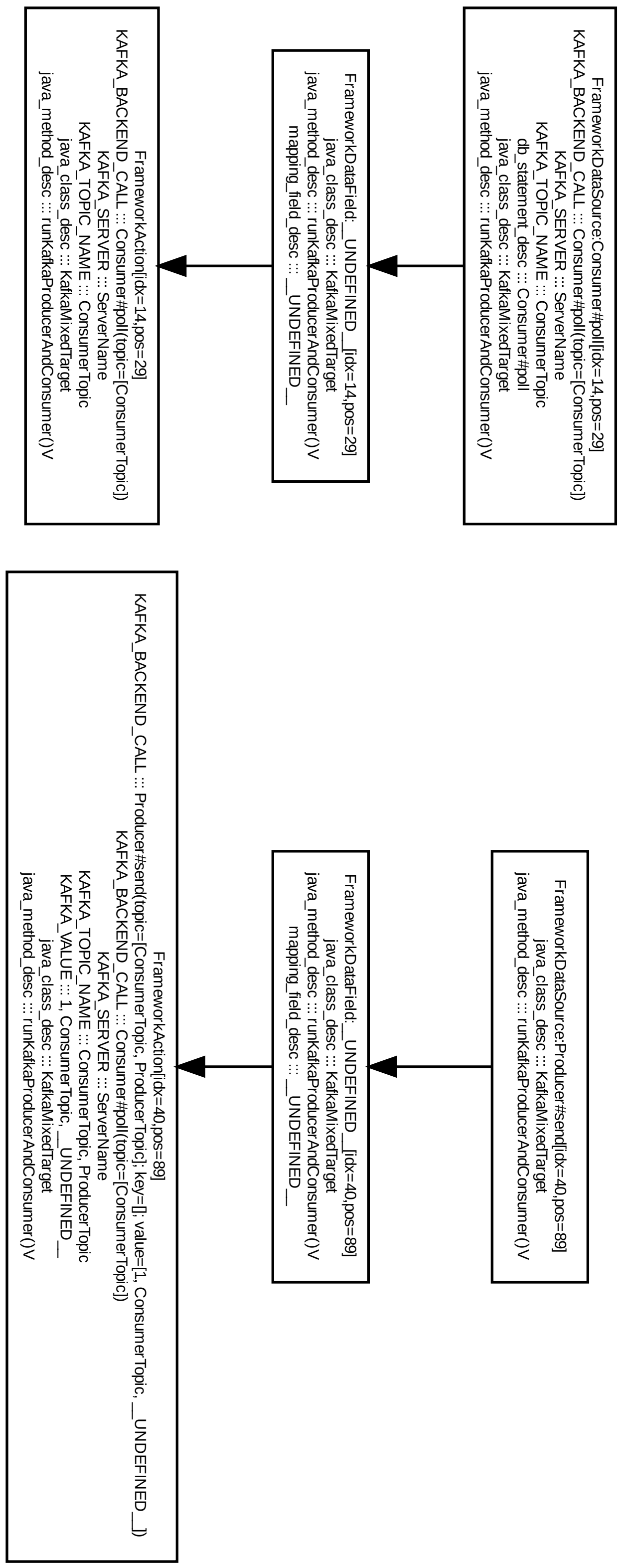
FrameworkDataSource:Consumer#poll[idx=14,pos=29]
KAFKA_BACKEND_CALL ::: Consumer#poll(topic=[ConsumerTopic])
KAFKA_SERVER ::: ServerName
KAFKA_TOPIC_NAME ::: ConsumerTopic
db_statement_desc ::: Consumer#poll
java_class_desc ::: KafkaMixedTarget
java_method_desc ::: runKafkaProducerAndConsumer()V
FrameworkDataField:__UNDEFINED__[idx=14,pos=29]
java_class_desc ::: KafkaMixedTarget
java_method_desc ::: runKafkaProducerAndConsumer()V
mapping_field_desc ::: __UNDEFINED__
FrameworkAction[idx=14,pos=29]
KAFKA_BACKEND_CALL ::: Consumer#poll(topic=[ConsumerTopic])
KAFKA_SERVER ::: ServerName
KAFKA_TOPIC_NAME ::: ConsumerTopic
java_class_desc ::: KafkaMixedTarget
java_method_desc ::: runKafkaProducerAndConsumer()V
FrameworkDataSource:Producer#send[idx=40,pos=89]
java_class_desc ::: KafkaMixedTarget
java_method_desc ::: runKafkaProducerAndConsumer()V
FrameworkDataField:__UNDEFINED__[idx=40,pos=89]
java_class_desc ::: KafkaMixedTarget
java_method_desc ::: runKafkaProducerAndConsumer()V
mapping_field_desc ::: __UNDEFINED__
FrameworkAction[idx=40,pos=89]
KAFKA_BACKEND_CALL ::: Producer#send(topic=[ConsumerTopic, ProducerTopic]; key=[]; value=[1, ConsumerTopic, __UNDEFINED__])
KAFKA_BACKEND_CALL ::: Consumer#poll(topic=[ConsumerTopic])
KAFKA_SERVER ::: ServerName
KAFKA_TOPIC_NAME ::: ConsumerTopic, ProducerTopic
KAFKA_VALUE ::: 1, ConsumerTopic, __UNDEFINED__
java_class_desc ::: KafkaMixedTarget
java_method_desc ::: runKafkaProducerAndConsumer()V

Figure 6.3: Visualization of data lineage graph for Kafka Framework

7. Conclusion

In the thesis we presented the Java Resolver tool that is able to compute data lineage information for Java applications that use complex data processing frameworks to access data.

A part of our work was to propose changes to the Symbolic analysis library such that it is possible to easily add support for new frameworks as the library plugins. Our main contribution is the implementation of such plugins for selected frameworks - Spring JDBC, MyBatis and Kafka. This demonstrates that such plugins can be used to provide support for data lineage analysis of other Java data processing frameworks with different approaches for accessing the data.

Each plugin should identify the sources and sinks of data inside the corresponding framework and then the Symbolic analysis library takes care of computing data flow in an analysed application.

We showed few limitations of our Java Resolver tool. Many of them are related to the static program analysis approach and the used Symbolic analysis library.

However, the tests of our Java Resolver tool show that the data flow is computed in reasonable time and thereof can be used in practice. The tests also create data lineage visualizations. The visualizations contain the key information about data sources and sinks and the flow between them.

In our work we focused on the basic features of selected data processing frameworks. We avoid the advanced features, such as Kafka Streams or Kafka Connector. We also did not try to provide the implementation for the the whole Spring JDBC Framework, as it provides quite large possibilities for accessing the data in an SQL database.

In the future development of the Java Resolver tool, we are planning to add support for all the mentioned advanced features of the frameworks. We will also develop plugins for new data processing frameworks such as Hibernate and Apache Spark.

References

[1] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. pages 539–550, 2006. doi: 10.1145/1142473.1142534. URL `http://doi.acm.org/10.1145/1142473.1142534`.

[2] Graphviz. Graphviz pages. URL `https://www.graphviz.org/`. Accessed 13.7.2018.

[3] java.sql. Documentation for package java.sql. URL `https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html`. Accessed 13.7.2018.

[4] javax.sql. Documentation for package javax.sql. URL `https://docs.oracle.com/javase/8/docs/api/javax/sql/package-summary.html`. Accessed 13.7.2018.

[5] JDBC Introduction. JDBC Introduction pages. URL `https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html`. Accessed 13.7.2018.

[6] Kafka Framework. Apache Kafka Framework manual pages. URL `http://kafka.apache.org/documentation/`. Accessed 13.7.2018.

[7] MANTA Flow. MANTA Flow pages. URL `https://getmanta.com/`. Accessed 13.7.2018.

[8] MyBatis Framework. MyBatis Framework manual pages. URL `http://www.mybatis.org/mybatis-3/`. Accessed 13.7.2018.

[9] Pavel Parízek. Hybrid Analysis for Partial Order Reduction of Programs with Arrays. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 291–310, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-662-49121-8. doi: 10.1007/978-3-662-49122-5_14. URL `http://dx.doi.org/10.1007/978-3-662-49122-5_14`.

[10] Pavel Parízek. BUBEN: Automated Library Abstractions Enabling Scalable Bug Detection for Large Programs with I/O and Complex Environment. 2019, to appear.

[11] Christopher Ré and Dan Suciu. Approximate lineage for probabilistic databases. *Proc. VLDB Endow.*, 1(1):797–808, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453943. URL `http://dx.doi.org/10.14778/1453856.1453943`.

[12] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015. ISSN 2325-1107. doi: 10.1561/2500000014. URL `http://dx.doi.org/10.1561/2500000014`.

[13] Spring JDBC Framework. Spring JDBC Framework manual pages. URL `https://docs.spring.io/spring/docs/2.0.x/reference/jdbc.html`. Accessed 13.7.2018.

[14] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. *Alias Analysis for Object-Oriented Programs*, pages 196–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36946-9. doi: 10.1007/978-3-642-36946-9_8. URL `https://doi.org/10.1007/978-3-642-36946-9_8`.

[15] WALA Framework. WALA Framework pages. URL `http://wala.sourceforge.net/wiki/index.php/Main_Page`. Accessed 13.7.2018.

List of Figures

2.11 Structure of records in a topic. Many producers can write to such topic and consumers can work with the records in arbitrary order. 14
2.12 Applications using differend kinds of Kafka APIs 15

3.2 Visualization of data lineage graph 24

4.1 Communication between Symbolic analysis and the plugins 27

6.1 Visualization of data lineage graph for Spring JDBC Framework . 39
6.2 Visualization of data lineage graph for MyBatis Framework 40
6.3 Visualization of data lineage graph for Kafka Framework 41

List of Listings

2.1 Example of object loaded from database 5
2.3 Example of creating `DataSource` for connecting to Oracle database 6
2.4 Example of query and insert operations using JDBC API [5] . . . 7
2.5 Example of query and insert operations using Spring JDBC Framework [13] 9
2.6 Definition of Mapper interface using annotations 10
2.7 Definition of XML Mapper interface 11
2.8 Configuration of `SqlSessionFactory` in Java 12
2.9 Configuration of `SqlSessionFactory` using external XML configuration file 12
2.10 Example of query and insert operations using MyBatis Framework [8] 13
2.13 Example of publishing data using Kafka Producer 15
2.14 Example of reading data using Kafka Consumer 16
3.1 Example of code with both database reading and file writing operations 23
4.2 Using `Map` as argument for input and output 31