IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 February 2024, accepted 28 February 2024, date of publication 1 March 2024, date of current version 8 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3372618

APPLIED RESEARCH

Manipulating Data Lakes Intelligently With Java Annotations

LAP MAN HOI[1], (Member, IEEE), WEI KE[1], (Member, IEEE), AND SIO KEI IM[1,2], (Member, IEEE)

[1]Faculty of Applied Sciences, Macao Polytechnic University, Macau, SAR, China
[2]Engineering Research Centre of Applied Technology on Machine Translation and Artificial Intelligence, Ministry of Education, Macao Polytechnic University, Macau, China

Corresponding author: Lap Man Hoi (lmhoi@mpu.edu.mo)

This work was supported by Macao Polytechnic University under Project RP/ESCA-03/2020.

ABSTRACT Data lakes are typically large data repositories where enterprises store data in a variety of data formats. From the perspective of data storage, data can be categorized into structured, semi-structured, and unstructured data. On the one hand, due to the complexity of data forms and transformation procedures, many enterprises simply pour valuable data into data lakes without organizing and managing them effectively. This can create data silos (or data islands) or even data swamps, with the result that some data will be permanently invisible. Although data are integrated into a data lake, they are simply physically stored in the same environment and cannot be correlated with other data to leverage their precious value. On the other hand, processing data from a data lake into a desired format is always a difficult and tedious task that requires experienced programming skills, such as conversion from structured to semi-structured. In this article, a novel software framework called *Java Annotation for Manipulating Data Lakes* (JAMDL) that can manage heterogeneous data is proposed. This approach uses Java annotations to express the properties of data in metadata (data about data) so that the data can be converted into different formats and managed efficiently in a data lake. Furthermore, this article suggests using *artificial intelligence* (AI) translation models to generate *Data Manipulation Language* (DML) operations for data manipulation and uses AI recommendation models to improve the visibility of data when data precipitation occurs.

INDEX TERMS Data lake, data precipitation, data stewards, enterprise-level applications, impedance mismatch, java annotations, JAMDL, object-oriented, ORMapping, software framework.

I. INTRODUCTION

Data has always been regarded as one of the most valuable assets in business and academia. Talent will retire, technology will age and become obsolete, and only data can provide the insights and evidence to remain in an irreplaceable position. In retrospect, when the Internet began to boom in the business world, data warehouses were developed to store the rapidly growing volume of data coming from *Online Transaction Processing* (OLTP) systems. With the popularity and practicality of *Big Data* (BD), *Internet of Things* (IoT), and AI technologies in recent years, data warehouses are no longer able to meet the requirements of storing diverse forms of data. Therefore, the concept of a data lake was introduced to create a large data repository for storing structured data (spreadsheet, tabular, etc.), semi-structured data (JSON, XML, YAML, etc.), and unstructured data (audio, corpus, image, log, etc.).

Generally speaking, data in a data warehouse is more tightly coupled, while data in a data lake is more loosely coupled. Data warehouse projects use the *Extract-Transform-Load* (ETL) approach to data processing where everything is defined before writing, technically known as the ''schema-on-write''. On the other hand, the Data Lake project uses the *Extract-Load-Transform* (ELT) or ''schema-on-read'' approach [2]. Consequently, data in different formats

The associate editor coordinating the review of this manuscript and approving it for publication was Rahim Rahmani.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

can be saved and exported more flexibly. The data lake concept is designed to support a variety of data operations, such as persistent log files for BD and IoT projects, and domain-specific output datasets for training AI projects.

However, manipulating large volumes and complex forms of data has been a challenging problem for many years. Despite the many innovative solutions that continue to be offered, the demanding needs of data lakes are still not being met. Some of the outstanding issues are how to deal with differently structured data in a data lake with a single data model, which includes representing different data structures in an abstract model, transforming between them, retrieving data from different data sources, and storing them in a data lake. There are also questions about how to leverage modern AI technology to govern data lakes. Nevertheless, as more people become aware of these new requirements for data lakes, it will have a positive impact on the development of data lakes.

Therefore, this article focuses on managing diverse data structures in data lakes. A novel software framework *Java Annotation for Manipulating Data Lakes* (JAMDL) is proposed to help people quickly develop web applications for manipulating data in data lakes. As a result, people use Java annotations to define data models to manage different data structures in the data lake. In pursuance of creating JAMDL, some interesting questions and fundamental requirements are discussed below.

A. IMPEDANCE MISMATCH

In computer science, impedance mismatch is the study of how to match two things with different properties together. For instance, people often want to map database data with programming objects. It is one of the core problems for *Object-Relational Mapping* (ORMapping), i.e., the disparity between the object-oriented application development model and the object-relational database model [3]. Neward claimed that ORMapping is the Vietnam of Computer Science. It represents a quagmire that starts well, gets more complicated as time passes, and before long entraps its users in a commitment that has no clear demarcation point, no clear win conditions, and no clear exit strategy [4]. For example, persisting and retrieving tree-structured data (e.g., JSON) into the flat database table shown in Fig. 1 is not a straightforward process. It not only involves programming techniques for recursively reading schema-oblivious trees but also applies data normalization concepts to persist data in database tables.

Over the years, there have been a plethora of systems catering to the new needs of businesses resulting in a wide variety of data formats. Based on the exact nature of the records, datasets can be classified as structured, semi-structured, and unstructured datasets [1]. If an ORMapping software framework is to be developed to solve the problem of data transformation in a data lake, then how to represent and transform data of different structures is the core problem to be overcome.

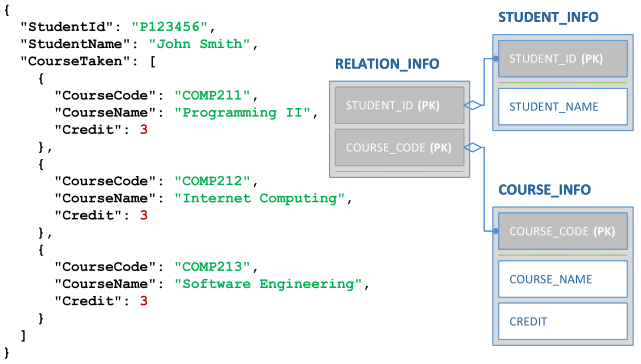

FIGURE 1. Transaction data can be represented in JSON (semi-structured data format) and multiple database tables (structured data format).

B. TEDIOUS CRUD TRANSACTIONS

In most cases, enterprise applications require at least one database engine to store transactional data. However, developing fundamental *Create, Read, Update, and Delete* (CRUD) transactions using SQL statements at the database level is a tedious task. Therefore, it is common practice to use ORMapping software frameworks for simplification. However, most ORMapping software frameworks only deal with structured data. There is relatively little research on ORMapping for semi-structured and unstructured data.

C. DATABASE NORMALIZATION PROBLEMS

Conforming to the database normalization design principles of reducing data redundancy and improving data integrity, the data in the transactions of the OLTP system needs to be decomposed and stored in multiple database tables [5]. ORMapping typically uses a single object to represent a *relational database* (RDB) table, which means that multiple objects are needed to represent a single transaction. Thus, if the transaction contains a substantial amount of information, the matching, persistence, and retrieval process can become very complex. As shown in Fig. 1, the student information is broken down into three tables and stored in the database.

A single transaction at the modern enterprise level usually contains a large amount of information. People are starting to debate whether it is certainly worth breaking up transactions into multiple tables for data persistence and then joining them for data retrieval. Some database engines such as *Not Only Structured Query Language* (NoSQL) or *Not Relational* (Non-SQL) recommend saving the entire transaction as an object and operating on it without modifying the structure [6]. As a result, it reduces the data granularity problems associated with database normalization and simplifies the data transformation process. Nonetheless, database normalization provides data integrity and powerful querying capabilities. Both are genuinely necessary and using ORMapping objects to represent informative datasets that combine different data structures is a major challenge for framework developers.

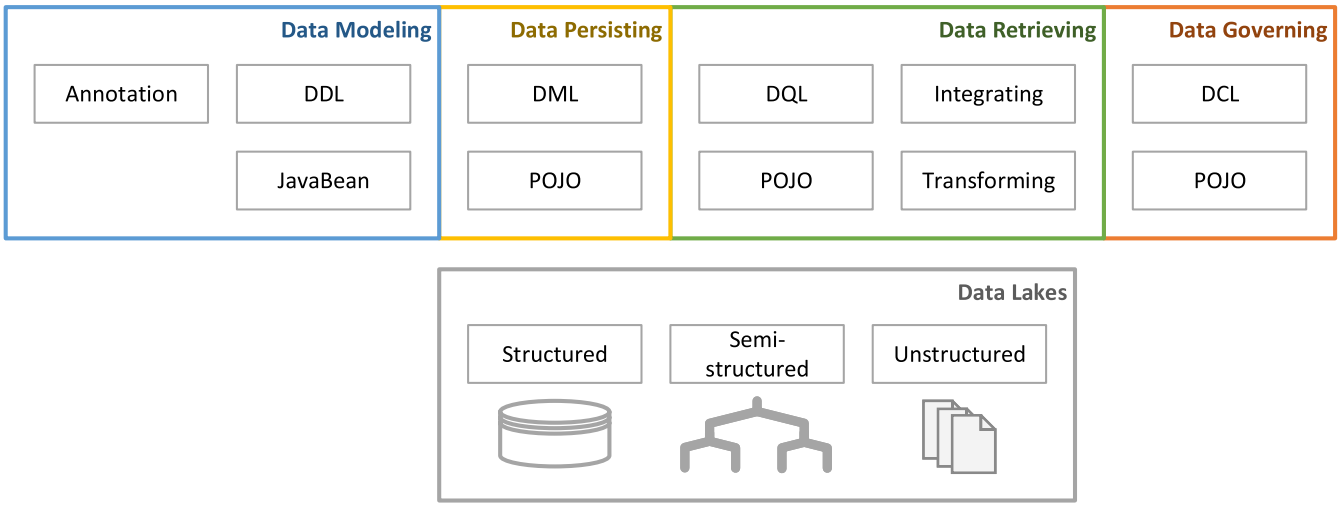

FIGURE 2. The modular software architecture for manipulating the data lake.

Furthermore, most modern ORMapping software frameworks only use inner join methods to join database tables and do not provide other table join methods (left join, right join, intersect join, full join, etc.) for data integration. Consequently, developers either have to change the table structure or write complex SQL statements themselves. Likewise, data integrity, especially the merging of various data structures, is one of the difficulties in manipulating data lakes.

D. BIG PICTURE

In this article, the JAMDL framework is presented comprehensively, and it is based on ORMapping for manipulating data in a data lake. The structure diagram is shown in Fig. 2. The software architecture is componentized into different modules (data modeling, data persisting, data retrieving, and data governing), and each module is discussed separately below.

The JAMDL framework is designed to address the problems listed in the previous sections and to provide a way for people to represent different data structures using an object model. As a result, one can use this object model to read, write, and convert data in a data lake to a desired format.

The rest of the article is organized as follows: Section I briefly describes the purpose of this study; Section II reviews state-of-the-art techniques and related research; Section III shows how to build a software framework for manipulating data lakes; Section IV evaluates and analyses the significance of all the results; and Section V concludes all the research study and discusses future work.

II. LITERATURE REVIEW

A. DATA LAKES

Enterprises collect digital footprints from a wide range of activities, with data coming in a heterogeneous form. The predefined table schema of the data warehouse architecture cannot meet the needs of storing unstructured data such as images, videos, and corpus files. Data lakes are one of the solutions for persisting data in various formats. However, data is often not well organized due to the complexity and diversity of data in data lakes. As a result, it is difficult for data to be fully utilized and analyzed to help decision-makers identify interesting issues and govern.

In other words, a data lake is simply a repository of all data (including raw data) for people to access at one point. The terms used to describe this large data repository are varied and include data puddles, data ponds, data pools, data oceans, and more. They differ mainly in their size, maturity, and purposes [7]. Nevertheless, it is more appropriate to use the term "data lake" in this article, as it is more relevant to enterprise-level applications.

Starting in 2010, different architectures have been suggested for building data lakes. Recently, many service providers have adopted data lakes in the cloud. Some well-known companies, such as *Amazon Web Services* (AWS), Azure Data Lake Store, *Google Cloud Platform* (GCP), Alibaba Cloud, and the Data Cloud from Snowflake, even offer powerful tools and user-friendly service interfaces for enterprises to build their own data lakes. In academia, scholars have never stopped to propose innovative solutions for constructing data lakes. According to the structure and function of a data lake, data lakes usually consist of four layers (Ingestion, Maintenance, Exploration, and Storage) [8], [9], [10].

Nonetheless, people are more concerned with the architecture than with manipulating the content of the data in the data lake. The demand for comprehensive solutions for manipulating data in data lakes continues to exist.

B. JAVA ANNOTATIONS

Java annotations, first released in 2004, are a form of metadata that provide information about a program rather

than being part of the program itself [11]. Java annotations provide three different retention policies (source, class, and runtime) for specifying how long to retain annotations. Therefore, once a Java program is annotated, the annotation information can be read at different stages of the program (compile time, deployment time, and runtime). Moreover, annotations can even be used to dynamically generate code that outputs a Java program, which is very much in line with the needs of framework developers.

Although annotations are not the core programming language by themselves, people often use them to extend the language's support for custom features such as compiler information, documentation, runtime logging, generating additional files, and so on [12]. In addition, some recent research studies have suggested the use of Java annotations for validation [13], [14], [15], mining [16], [17], [18], and maintenance [19].

People commonly use XML to define and configure different software frameworks. However, XML is considered heavyweight because it has too many tags and its tree structure is bulky when parsing content. Hence, Java annotations can be used to store configuration information of software systems instead of the verbose XML.

In the field of deep learning, data labeling is one of the important processes in machine-supervised training, and Java annotations can be the metadata describing various data structures in the data lake. As a result, the applications of Java annotations in different domains are yet to be explored by researchers.

C. TYPES OF SQL STATEMENTS

There are various types of *Structured Query Language* (SQL) statements used to process database records. The most commonly used SQL statements are *Data Control Language* (DCL), *Data Definition Language* (DDL), *Data Manipulation Language* (DML), and *Data Query Language* (DQL) [20]. They are sub-languages that perform all the basic operations in the database engine.

- **DCL** operations grant access to all elements in the database. Typical DCL statements are the GRANT and REVOKE statements.
- **DDL** operations define elements such as schema for storing data. Typical DDL statements are the CREATE and DROP statements.
- **DML** operations manipulate the contents of data records. Typical DML statements are INSERT, DELETE, and UPDATE statements.
- **DQL** operations retrieve data records and combine them into a subset of data. A typical DQL statement is a SELECT statement.

These sub-languages have traditionally been used only for manipulating structured data, i.e., database records. Hence, researchers should improve these sub-languages so that they can handle other types of data (semi-structured and unstructured data) and assist the JAMDL framework in managing the heterogeneous data in the data lakes.

D. ORMAPPING SOFTWARE FRAMEWORK

ORMapping is a mechanism for connecting classes in an *object-oriented* (OO) programming language to tables in a relational database [21]. ORMapping allows us to query and manipulate data stored in an RDB using OO approaches without the need to use DQL or DML [5]. As a result, programmers can retrieve and save data in a variety of database engines without having to write cumbersome SQL statements. In other words, developers can simply use their favorite programming language (Java, PHP, C#, etc.) without having to develop at the database level.

However, the popular ORMapping software frameworks on the market (Hibernate, MyBatis, TopLink, etc.) and the *Java Persistence API* (JPA) industry standard share many common issues that cannot fully satisfy the needs of developers [5]. For example, most ORMapping software frameworks are unable to address the impedance mismatch between structured, semi-structured, and unstructured data.

Although ORMapping has been around for a while, there are still some outstanding issues that cannot be fully settled [22]. A typical example of this is manipulating *JavaScript Object Notation* (JSON) data. JSON is one of the most commonly used data exchange formats in modern online systems, but existing ORMapping software frameworks only partially support the complex tree structure of JSON, which does not meet the expectations of modern developers. Most database engines that support JSON will keep the JSON objects intact, which loses flexibility and performance for searching and updating data.

To design a new ORMapping software framework to manipulate data in a data lake, conversion between different data formats is unequivocally a challenging problem. In general, there are three problems (mapping, retrieving, and persisting) that need to be overcome to design a new ORMapping software framework [23]. In the proposed solution, Java annotations are used as data objects to represent different data structures to manipulate the data in the data lake. Thus, the JAMDL framework based on ORMapping must overcome the following problems.

- Map data in different formats to data objects.
- Store data objects to different datasets.
- Retrieve data from multiple datasets and convert it back to a single data object.

E. DATA STEWARDS

Over the years, people have had different names for people who work with data, such as data engineers, data analysts, and data scientists. Roughly speaking, data engineers are responsible for underlying data processing, data analysts for business insight analysis, and data scientists for academic research. People usually categorize employees into specific roles based on their interests and job characteristics in the company. More recently, enterprises can even appoint data stewards for data governance, data quality control, data pipeline management, business definition regulation, glossary creation, and sensitive data operations.

Data governance is a very broad term that can cover many areas. It can be linked to plans, policies, and procedures for managing and implementing data accountability [24], [25], [26]. Data governance is now often used to represent the job responsibilities of a data steward. As the role of the data stewards becomes more important, enterprises require them to have a more holistic view of the data lake conduce to effectively manage its dynamic nature. It has also become necessary to use modern AI tools to help them in their governance efforts.

With the help of cutting-edge AI technologies, many hidden issues and problems can be detected in advance and data stewards can react quickly. Therefore, modern software frameworks should also offer the ability to include AI technologies to predict and recommend management strategies to data stewards.

F. AI TECHNIQUES

Since the rise of neural networks and AI-related techniques, they have immediately dominated the field of academic research. They specialize in automation and prediction and can be applied to many different domains.

AI techniques can be used in many different areas to help the JAMDL framework provide powerful features for secondary developers. Some common but not limited to these neural networks can help are *Convolutional Neural Network* (CNN) for building classification model [27], [28], *Recurrent Neural Network* (RNN), *Long Short-Term Memory* (LSTM), and *Gated Recurrent Unit* (GRU) for building models that require to understand the context of the sentences [29], [30], *end-to-end* (E2E) network for building translation models [32], and *Emphasized Channel Attention, Propagation and Aggregation in Time Delay Neural Network* (ECAPA-TDNN) for voice and speaker recognition [31].

In this article, AI techniques are mainly used to generate SQL statements and to help data stewards manage data lakes. In a previous study [33], an E2E network was implemented to generate DQL-type SQL statements to query a database engine using natural language. E2E is very elegant and has been popularized for deep learning [32]. The idea of using a single model to specialize in predicting outputs directly from inputs can handle extremely complex systems and is arguably the most advanced deep learning technique. By the same token, people can use AI to generate complex DML-type SQL statements for processing enterprise-level transactions.

Data stewards also need AI technologies to provide alerts and recommendations to create insightful analyses of *business intelligence* (BI) type reports and visualization diagrams. Furthermore, people can build AI models that rate data based on its accessibility, and validate the data in the data lakes after the JAMDL object models are built. Subsequently, many areas where AI can be used to enhance the JAMDL framework are yet to be explored by researchers.

III. METHODOLOGY

To propose a new type of ORMapping software to comprehensively address the salient issues of data lakes, it is necessary to address the fundamental issues of data manipulation (mapping, retrieval, and persistence). All the modules listed in Fig. 2 are discussed below. The JAMDL framework is designed to handle different datasets, which will be demonstrated below using the simple dataset mentioned in Fig. 1.

A. DATA MODELING

The mapping process is the most critical part of the proposed software framework. There are several conventional methods for mapping relational data to program objects. The common practice in today's software frameworks is to define an XML file for the object mapping process. However, popular software frameworks (Spring Boot, iBaits, etc.) require writing and managing many XML files for configuration. As the project evolved, the content and syntax became lengthy and complex. Instead, Java annotation is recommended to be used as an object to describe different types of data in a data lake.

1) OBJECT MODEL

The principle of OO is that everything can be object-based. Data in various formats can also be conceptually represented by corresponding objects. In a previous study [34], object models are illustrated to process unstructured data (parallel corpora). Therefore, the object model can help to represent and manage complex data. Abstraction refers to the basic characteristics of an object that distinguish it from all other types of objects, thus providing a clearly defined conceptual boundary relative to the perspective of the viewer [35]. This concept helps in discovering the characteristics of various data. Furthermore, it is also applicable in dealing with semi-structured data, which requires some higher level of abstract description.

The ease of use of Java annotations is unparalleled in the history of Java metadata. Java annotations are flexible enough to provide a retention policy that specifies how marked annotations are stored, whether they are stored only in code, compiled into classes, or available at runtime through reflection [11]. With Java annotations, the JAMDL framework is fully implemented based on ORMapping for manipulating data in the data lakes, the source code of which is available on Github [36]. Fig. 3 shows the essential classes used to form the business logic of this software framework.

2) MAPPING APPROACH

To the extent that Java objects represent various types of data, the most fundamental metadata encompass *field name*, *field type*, *entity type*, and *entity path*. The terms ''field'' (table columns, log records, JSON attributes, etc.) and ''entity'' (database tables, log files, JSON files, etc.) here have more

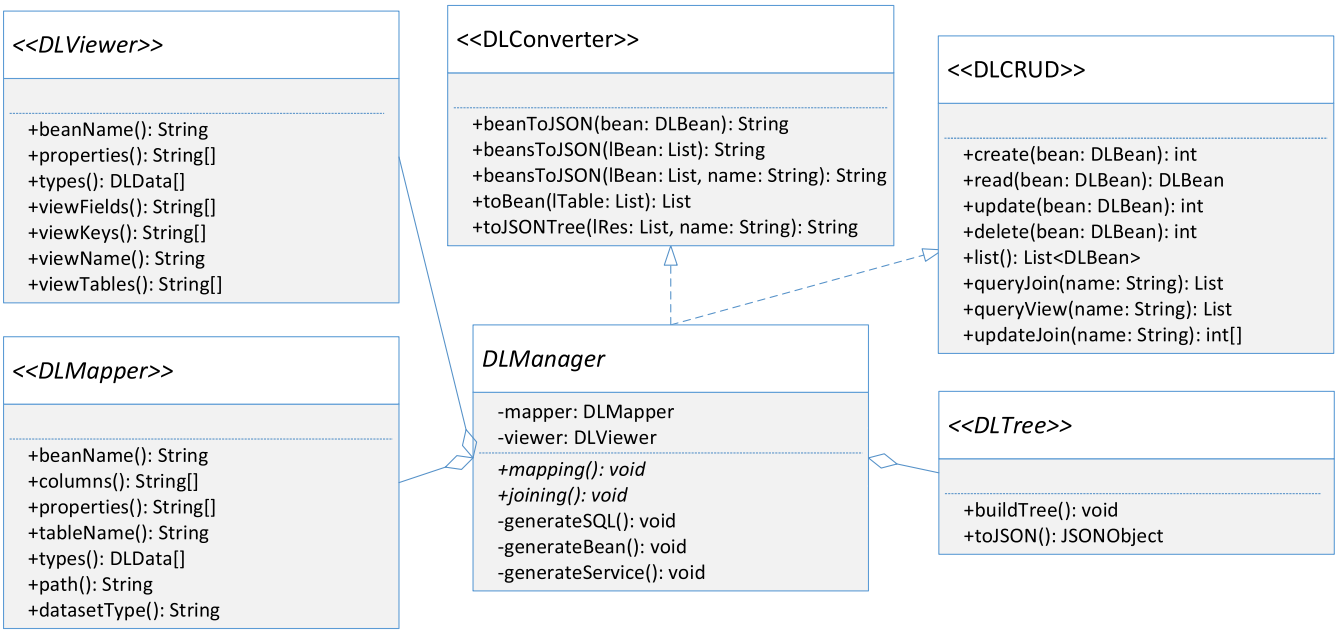

FIGURE 3. Class diagram: Business logic of the software framework.

abstract meanings and are used to support different data structures.

Once this metadata is defined in a Java class, the software framework generates JavaBeans for developers to manipulate the data and DML and/or DQL operations for the software framework to manipulate the data in the data lake. Note that DML and DQL here have been enhanced to support the processing of data other than database records using *Plain Old Java Objects* (POJOs).

- **field name:** This attribute is the name of the field that will be used to generate the JavaBeans and DML and/or DQL operations.
- **field type:** This attribute is the data field (double, integer, string, etc.) of the data type that will be used to generate JavaBeans and DML and/or DQL operations. The field type is the data type of structured data table columns, semi-structured data attributes, and unstructured data records.
- **entity type:** This attribute indicates the type of entity (log file, JSON, RDB, etc.). Unlike traditional ORMapping software frameworks that can only handle structured data (i.e., database records), it can associate semi-structured and unstructured data.
- **entity path:** This attribute indicates the location of the entity (file path, RDB name, etc.) in the data lake.

3) CORE PROGRAMS

The implementation of developers begins with the construction of the *DLMapper* program, an interface class for developers to annotate metadata. *DLManager* is a superclass for constructing CRUD transactions to manipulate data in a data lake after the *DLMapper* class has been defined. Fig. 3 shows the architecture of the *DLManager* and its relationship to the auxiliary Java classes. All the roles of these auxiliary classes are summarized below.

- *DLConverter*: An interface class that contains APIs for converting JavaBeans to JSON objects, tabular database records to JavaBeans, and tabular database records to JSON objects.
- *DLCRUD*: An interface class that provides developers with a CRUD transaction API. It also provides advanced APIs for manipulating multiple tables and records at the same time.
- *DLManager*: An abstract class that implements CRUD transactions by generating DML and DQL operations. It also generates JavaBeans for developers.
- *DLMapper*: An interface class that stores annotation information provided by the developer for the ORMapping process.
- *DLTree*: A class that provides developers with an API for dynamically building JSON objects.
- *DLViewer*: An interface class for storing information to combine transactions from multiple tables using specific table join methods.

4) MAPPING PROCESS

The *DLManager* class triggers the `generateSQL()`, `generateBean()`, and `generateService()` APIs to take care of the tedious tasks for us. The following pseudocode demonstrates the mapping process using Java annotations in a subclass of *DLManager*. The database table (student_info) has two columns (student_id, student_name) that can be mapped to a JavaBean (*StudentBean*) whose properties (studentId, studentName) are of type integer and

string respectively. Moreover, the `mapping()` API specifies the primary key of the table.

```
1  public class StudentManager extends DLManager {
2  @Override
3  @DLMapper(tableName = ‘‘student_info’’,
4  beanName = ‘‘dl.model.StudentBean’’,
5  columns = {"student_id", "student_name"},
6  properties = {"StudentId", "StudentName"},
7  types = {DLData.Integer, DLData.String},
8  path = ‘‘StudentDB’’, datasetType = 0)
9  public void mapping() { super.setPkNum(1); }
10
11 @Override
12 public void joining() {}
13 }
```

All the mapping process is done in this ordinary Java subclass. The annotations occur in front of the `mapping()` API. Since Java annotations are defined to be retained in the software framework at compile time, JavaBeans and the related DML and DQL operations will be generated after the subclass is compiled.

B. DATA RETRIEVING

The object retrieval mechanism involves a READ transaction of CRUD. The software framework uses the industry standard *Java Database Connectivity* (JDBC) for database connectivity, which is very mature and robust and supports most database engines.

There are four different types of JDBC connections (bridge, native API, middleware, and driver-only) to cater to the diversity of enterprise environments [37]. Furthermore, this software framework is designed to support the schema-on-read approach that allows developers to self-define their special schema when querying a data lake. Since JAMDL is developed entirely in Java, the database connection methods use type four (driver-only) which is considered the most efficient. Consequently, unlike most of the well-known ORMapping software frameworks on the market that use other complex database connection types, the concise architecture of JAMDL should run faster than they do.

In most ORMapping software frameworks, the READ transaction can retrieve only one record from the database per query, which is insufficient for handling semi-structured and unstructured data. Therefore, this software framework provides a `list()` API to retrieve multiple records. It also provides developers with custom JSON objects to dynamically output data in the desired format.

1) RETRIEVING PROCESS

Once the mapping process is complete, CRUD transactions become very simple. The READ transaction can be implemented by simply calling *StudentManager* with a JavaBean. The following pseudocode demonstrates a READ transaction where *StudentManager* retrieves a record based on a primary key.

```
1  public class ReadStudent {
2    public static void main(String[] args) {
3      // Data Source Connection
4      DSConn dsObject = new DSConn();
5      StudentBean bean = new StudentBean();
6      bean.setStudentId(123123);
7      bean = (StudentBean) StudentManager.noSQL()
8              .read(dsObject, bean);
9      System.out.println(bean.getStudentName());
10   }
11 }
```

The READ API provided by the software framework is very concise, as the abstract class *DLManager* already handles all the heavy lifting. First, it reads the annotation information from the `mapping()` API of the subclass. It then validates the number of attributes in the annotation and generates a JavaBean and DQL operation. DQL operations are performed by using two APIs (`callGetter()` and `callSetter()`) to manipulate data in the data lake. Algorithm 1 and Algorithm 2 show the implementation of these two APIs inside the software framework.

Algorithm 1 Retrieving Data From a JavaBean

Input: *bName*, *mName*, *bean*
Output: *value*

```
function callGetter()
    Class<?> c ← Class.forName(bName)
    Method m ← c.getDeclaredMethod(mName)
    Object value ← m.invoke(bean)
    return value
```

Algorithm 2 Storing Data to a JavaBean

Input: *bName*, *mName*, *bean*, *value*

```
function callSetter()
    Class<?> c ← Class.forName(bName)
    Class<?>[] arg ← new Class[1]
    Method m ← c.getDeclaredMethod(mName, arg)
    m.invoke(bean, value)
```

2) OBJECT INTEGRATION

According to the database normalization concept, a single transaction can be separated into multiple tables for persistence [38]. Querying back a transaction requires consolidating them together. The query statements become very complex if a transaction involves more than three tables, and joining database tables is doubtless a challenge in building novel software frameworks. As in the previous example in Fig. 1, student information is normalized into three different database tables. To retrieve complete information about a student, these three tables need to be joined.

In this software framework, *DLManager* provides a `joining()` API for developers to specify how to join tables together. Developers simply create a *JoinBean*, fill in the basic table join information (data fields, table name, and join

key), and the software framework fetches the data from the database table accordingly.

DLManager also provides developers with two result formats through the interface class *DLCRUD*: JavaBean and tabular data (list of strings).

```
+queryJoin(n: String): List<DLBean>
+queryView(n: String): List<List<String\gg
```

Instead of retrieving all records directly, *DLManager* can choose to limit the records fetched. Developers can use the `setFilters()` API of *JoinBean* to insert the where clause of a DQL operation.

```
+setFilters(f: String[]): void
```

In addition, *DLManager* is capable of specifying different join methods. Since not all database engines can support various join methods, the JAMDL framework provides the most commonly used method to support most database engines, and the default method in this software framework is an inner join. These join methods (inner join, left join, right join, full outer join, intersect, union, minus, etc.) are commonly found in most database engines for processing queries [1]. Moreover, these join methods are slightly modified to accommodate all database engines. Table 1 summarizes the syntax of the different join methods provided.

TABLE 1. Syntax for different join methods.

Symbol	Join Method
=	Inner join *
∧	Intercept
<	Left join
>	Right join
+	Full outer join
−	Minus
&	Union

* *default method*

The following pseudocode shows how to query student information by constructing a *JoinBean*. The table join method is specified in the `setKeys()` API on line 7.

```
1  @Override
2  public void joining() {
3  JoinBean bean = new JoinBean();
4  bean.setJoinName(‘‘JoinStudent’’);
5  bean.setColumns({"ID", ‘‘NAME’’, "COURSE"});
6  bean.setTables({"STUD", ‘‘RELA’’, "COURSE"});
7  bean.setKeys({{"STUD.ID=", "RELA.ID"}\ldots });
8  bean.setFilters({"ID = 123456"});
9  super.setJoinTables(bean);
10 }
```

The most critical part of making this data retrieval mechanism work flawlessly is the process of correctly generating SQL statements (DQL). While query statements can be dynamic and varied, SQL statements can be broken down into different parts (data fields, tables, keys, and filters) [33].

There are two main SQL patterns for table joins in this software framework, and the syntax diagram is shown in Fig. 4. As mentioned earlier, the DQL keywords (SELECT, FROM, ON, HAVING, and WHERE) separate the SQL into different parts. This software framework then reads the object information from the annotations and uses these predefined SQL templates to generate the requesting queries.

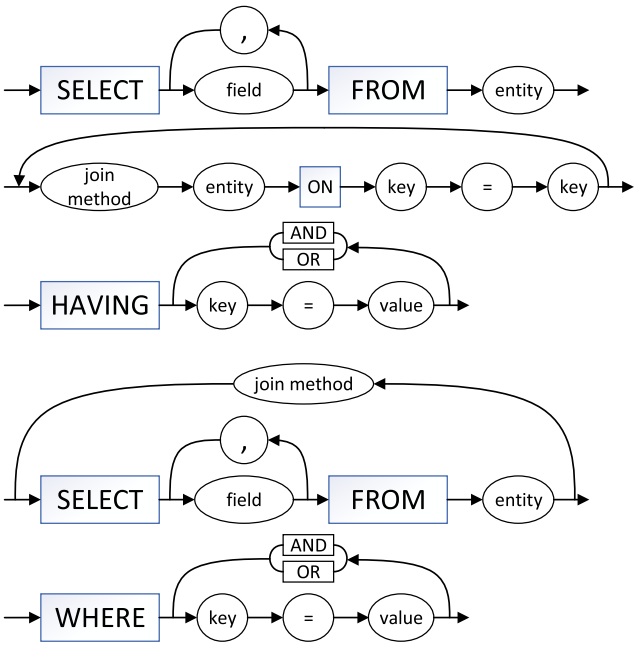

FIGURE 4. DQL type SQL patterns for different join methods.

By the same token, semi-structured and unstructured data can be queried by implementing the `joining()` API. Unstructured datasets are retrieved in the form of a data field. The following DQL operations show enhanced versions that support data other than database records. Therefore, POJOs handle queries by processing data at the file system level. It is important to note that the syntax of enhanced DQL is the same as the normal DQL SQL statement.

```
SELECT row FROM abc.log WHERE lineNum = 6;
SELECT attribute FROM def.json WHERE
       attribute = 'id' and level = 3;
```

The first enhanced DQL retrieves a record from the log file. This result set is a subset of unstructured data or just a part of the log file. Therefore, developers can now manipulate the log file through the JAMDL framework.

3) SEMI-STRUCTURED DATA TRANSFORMATION

Since JAMDL framework uses JavaBean as the default data format for developers to manipulate data in the data lake, it provides advanced APIs for converting bland, two-dimensional data into the JSON tree format. This software framework provides two APIs (`beanToJson()` and `toJsonTree()`) from the *DLConverter* interface

class (described in Fig. 3) to construct JSON objects. The `beanToJson()` API converts JavaBean or table data directly into flattened JSON objects. This is an overloaded method that can output single or multiple records from the database. In other words, the output JSON object has either a one-level (*JSONObject*) or two-level (*JSONArray*) tree structure.

In reality, JSON objects can be very complex, with multiple layers, especially for enterprise systems. To address this need, the JAMDL framework provides the `toJSONTree()` API for dynamically constructing tree-structured objects. Moreover, the ''Bracket Schema'' notation is introduced for the software framework to understand the self-defined JSON structure. Fig. 5 shows the syntax of bracket notation with the upper part representing *JSONObject*, and the lower part representing *JSONArray* objects.

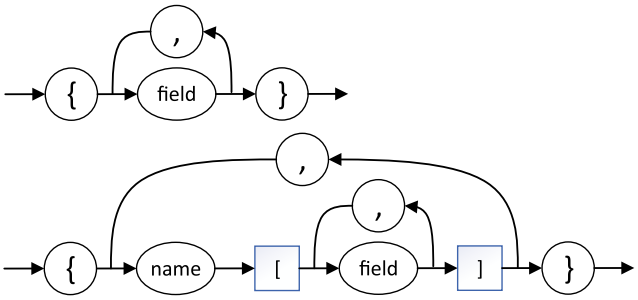

FIGURE 5. The "Bracket Schema" notation for building the JSON objects.

As a result, developers can dynamically define the JSON format and the software framework generates JSON on the fly, for example, by converting data records from three database tables into JSON objects, as shown in Fig. 1.

Developers can dynamically create different tree structures by simply embedding these bracket symbols in a Java program (pseudocode shown below). In addition, the JSON structure can be changed by modifying lines 7 through 10.

```
1  @Override
2  public static void main(String[] args) {
3     DSConn dsObject = new DSConn();
4     list lBean = new ArrayList();
5     lBean = StudentManager.noSQL().list(dsObject, bean);
6     Bean2Tree b2t = new Bean2Tree();
7     b2t.setFields(new String[] {
8        "StudentID", "StudentName",
9        "CourseTaken[CourseCode", "CourseName", "Credit]"
10    });
11    b2t.buildTree();
12    b2t.setlRecord(lRow);
13    JSONObject json = b2t.toJSON();
14 }
```

Developers can change the bracket notation in plain text to represent more complex JSON object structures such as nested brackets. Two additional JSON objects are demonstrated below, with the results shown in Fig. 6.

```
// JSON format 1
b2t.setFields(new String[] {
   "StudentID", "StudentName",
   "CourseTaken[CourseCode", "CourseName]",
   "TotalCredit"
});
// JSON format 2
b2t.setFields(new String[] {
   "CourseCode", "CourseName", "Credit",
   "Students[StudentID", "StudentName]"
});
```

```
{
  "StudentId": "P123456",
  "StudentName": "John Smith",
  "CourseTaken": [
    {
      "CourseCode": "COMP211",
      "CourseName": "Programming II"
    },
    {
      "CourseCode": "COMP212",
      "CourseName": "Internet Computing"
    },
    {
      "CourseCode": "COMP213",
      "CourseName": "Software Engineering"
    }
  ],
  "TotalCredit": 9
}
```

```
{
  "CourseCode": "COMP211",
  "CourseName": "Programming II",
  "Credit": 3,
  "Students": [
    {
      "StudentId": "P123456",
      "StudentName": "John Smith"
    },
    {
      "StudentId": "P222222",
      "StudentName": "Paul Chan"
    },
    {
      "StudentId": "P333333",
      "StudentName": "Mary Lee"
    }
  ]
}
```

FIGURE 6. Different JSON objects.

Inside the JAMDL framework, the `toJSONTree()` API is implemented in the *DLTree* class (as described in Fig. 3). As shown in Algorithm 3, it reads the data fields and determines the sequential number, tree level number, and column name of the database table for each data field. It then uses indirect recursive methods to add either *JSONObject* or *JSONArray* objects to build the JSON tree. Algorithm 4 shows the business logic for transforming JSON objects from database records.

Algorithm 3 Parsing the Bracket Schema

```
1:  function readNodes()
2:      levelId ← 1
3:      initialize nodeList
4:      while loop over each field do
5:          if field has left bracket then
6:              levelId ← levelId + 1
7:          if field has right bracket then
8:              levelId ← levelId − 1
9:          nodeList.add(field)
10:     return nodeList
```

C. DATA PERSISTING

This software framework provides two different approaches to handle DML operations beneficial to persist data in a data lake.

1) PERSISTING PROCESS

DML operations (INSERT, UPDATE, and DELETE) are used to change the contents of database tables. After the data mapping process is complete, the *DLManager* reads the object information from the annotations and

Algorithm 4 JSON Objects Transformation

```
function toTreeArr(levelId, array)
    TreeBean bean ← nodeList.get(levelId)
    JSONObject obj ← new JSONObject()
    obj.put(bean.getName(), bean.getRecord())
    if levelId + 1 = nodeList.size() then
        array.add(obj)
        return array
    else
        TreeBean next ← nodeList.get(levelId + 1)
        if next.getLevel() > next.getLevel() then
            obj.put(next.getName(),
            toTreeArr(levelId + 1, new JSONArray()))
            array.add(obj)
        else
            array.add(toTreeObj(levelId + 1, obj))
        return array
function toTreeObj(levelId, obj)
    TreeBean bean ← nodeList.get(id)
    obj.put(bean.getName(), bean.getValue())
    if levelId + 1 = nodeList.size() then
        return obj
    else
        TreeBean next ← nodeList.get(levelId + 1)
        if next.getLevel() > next.getLevel() then
            obj.put(next.getName(),
            toTreeArr(levelId + 1, new JSONArray()))
            return obj
        else
            return toTreeObj(levelId + 1, obj)
```

generates DML to process the corresponding transactions. All SQL statements rely on the primary key defined in the `mapping()` API (described in Section III-A4) as the default generation method.

The business logic is similar to the READ transaction. DML templates (shown in Fig. 7) can be created by breaking down SQL statements using keywords (INSERT INTO, VALUES, UPDATE, SET, DELETE FROM, and WHERE). The basic DML can be further divided into different parts: action (DELETE, INSERT, UPDATE), filters (WHERE), and arguments (entity names, field names, and field values) for automatic generation by the framework.

Developers can then use the `create()`, `update()`, and `delete()` APIs provided by the *DLCRUD* interface class to modify the content of entities.

Changing one record at a time cannot satisfy the requirements of an enterprise-level system. Therefore, this software framework provides three additional APIs (`bCreate()`, `bUpdate()`, and `bDelete()`) to modify multiple records in batch mode or "transaction" mode. Most database engines offer the advanced feature of transaction mode. A transaction is a set of SQL queries that are treated "atomically" into a unit of work [39].

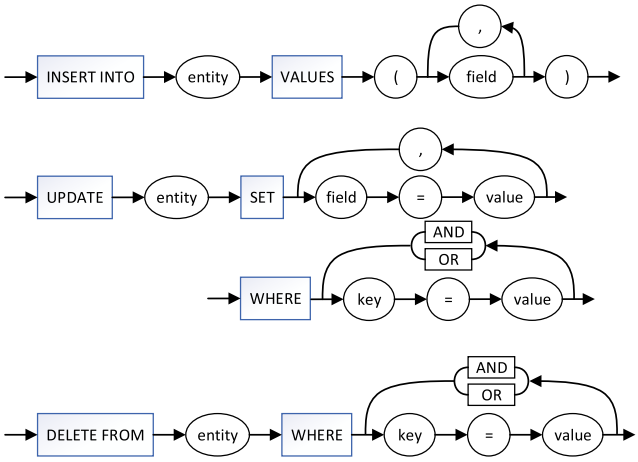

FIGURE 7. The DML operation template.

These three APIs first turn off the auto-commit feature of the database engine and then evaluate each SQL statement. If all SQL statements are successfully executed, the modified data will be permanently stored in the database. Finally, they turned on auto-commit again. This transaction mode process improves the overall performance significantly. The implementation is almost identical to a normal CRUD transaction, with the addition of auto-commit configuration steps.

Semi-structured and unstructured data usually exist as files in the data lake. The framework treats them as a normal data field with their data path in the JavaBean. Persisting these types of data requires direct modification of files rather than database tables. Thus, the DML operation consists not only of an SQL statement that modifies a database record but also of a POJO that modifies a JSON or log file.

Although DML operations that modify semi-structured and unstructured data are more like commands than SQL statements, they follow SQL syntax structures. For example, two DML operations to delete specific lines of a log file and update an attribute of a JSON object with a specific value are shown below. Then, the POJO translates these command-type DML operations and modifies them at the file system level.

```
DELETE FROM abc.log WHERE lineNum = 6;
UPDATE def.json SET name = value WHERE
level = 3;
```

2) DML FOR JOINED TABLES

Section III-C1 demonstrates how JAMDL framework manipulates simple DML operations. An enterprise-level transaction indeed can involve several data sources and various data formats. The `updateJoin()` API provided by the *DLCRUD* interface class is used to persist online transactions involving multiple database tables. The *DLManager* first reads object information from the `joining()` API. It then generates DML operations from the corresponding entities, fields, and keys. To handle more complex DML operations,

the software framework uses AI technologies to generate these DML operations.

In previous research on building *question answering* (QA) systems, LSTM and GRU neural networks are illustrated to generate natural language questions and DQL answers with high accuracy [33]. To exploit this, the JAMDL framework also uses LSTM and GRU models to generate DML for accessing multiple data sources, the performance of which is discussed in detail in the next section.

D. DATA GOVERNING

Although the responsibilities of data stewards vary from enterprise to enterprise, their role is becoming increasingly important. They also need to react quickly to new rules, policies, and instructions set by local governments in some countries. Since DCL operations can only control user access to structured data, DCL can be augmented with POJOs to control other structured data (semi-structured and unstructured data).

However, the role of data stewards is not only to safeguard data but also to provide high-quality data to the enterprise. Therefore, the following areas can utilize AI technology to help them better govern the data in the data lake.

1) DATA PRECIPITATION

In the field of chemistry, precipitation is the separation of solid substances from a liquid. This process occurs naturally over a while without external intervention.

As data stays in the data lake for longer and longer periods, polarization occurs due to the increase in the number of visits, leading to data precipitation. Fig. 8 shows that data near the surface is most frequently accessed, while data located at the bottom is rarely accessed, or is referred to as ''zombie data''. Nevertheless, data stewards are responsible for various definitions of data, including zombie data, which is data that has not been accessed within a specific time.

Subsequently, there are a lot of interesting things that can be done when data precipitation appears. On the one hand, people can rank the data based on how often it is accessed. On the other hand, people can classify data that may end up as zombie data. With the help of AI technology, people can also build recommendation models that provide not only high-quality data but also rarely accessed data that may be useful to users. As a result, data that would otherwise form data silos can increase their visibility.

2) FAULT DETECTION

Most data stewards are probably more concerned about system failures and errors. However, many factors and combinations can cause system failure. Therefore, big data concepts can be used to build a linear regression model and incorporate all relevant parameters that may lead to system failures. Some common parameters include new system deployments, system updates, age of hardware, new security threats, peak times, holidays, historical high-risk days, and dates of special events and activities.

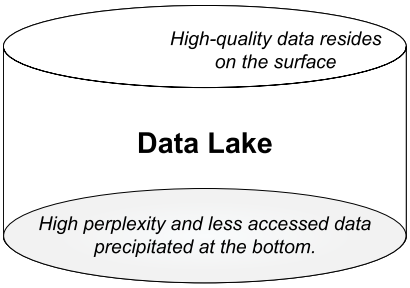

FIGURE 8. Data Precipitation.

3) RESOURCE PLANNING

Modern system architectures rely on cloud computing to create virtual machines for each server node. Hardware configurations (CPU, memory, hard drives, etc.) can be adjusted more flexibly as server usage changes. Consequently, a predictive model can be built to estimate the resources (hardware configuration, network bandwidth, etc.) consumed by each server node through past usage statistics and possible patterns.

IV. EXPERIMENTS AND DISCUSSIONS

A. GENERATING DML BY LSTM AND GRU NETWORKS

In most cases, building accurate AI models requires a large amount of high-quality training data. To prepare the training data, all possible DML operations from each entity (database tables, JSON, and log files) are created. This can be accomplished by writing a simple recursive program that iterates through every file and column in the database table. Then, form a transaction by randomly selecting some fields from different entities (files or tables). Simultaneously, the corresponding DMLs are generated from each entity. As a result, each transaction containing different fields from different entities is paired with multiple DML operations. The business logic of this program is shown in Algorithm 5.

Algorithm 5 Training Data Generation

Input: n ▷ Number of records
Output: *train.txt*, *test.txt* ▷ Output files

procedure *gen_training_data*(n)
  while loop over each file and table **do**
    $list(field, entity) \leftarrow$ {files or tables}
  $counter = 0$
  while $counter < n$ **do**
    randomly picks some records from the *list*
    put all the *field* together as a transaction
    iterate over all three DML types of operations
    generate DML for each *entity* containing *field*
  output the transaction and DMLs pairs to files
    write 90% of the data to *train.txt*
    write 10% of the data to *test.txt*

Consequently, the training and testing datasets are prepared. Then, translation models are built to allow the software

framework to generate DML statements automatically. Translation models require source and target texts as a pair or parallel corpus to train the AI model. Below is an example of the parallel corpus generated by Algorithm 5.

SOURCE TEXT (TRANSACTIONS):

```
UPDATE logA.lineNum=num,
tableA.fieldA=valueA, tableA.fieldB=valueB,
tableB.fieldA=valueA, tableB.fieldC=valueC
```

TARGET TEXT (DML OPERATIONS):

```
UPDATE logA.log SET lineNum=value;
UPDATE tableA SET fieldA=valueA,fieldB=valueB;
UPDATE tableB SET fieldA=valueA,fieldC=valueC;
```

B. EVALUATING LSTM AND GRU NETWORKS

The next step is to build some translation models for evaluation. With the emergence of the Google Transformer model in 2017 [28], it has rapidly dominated the deep learning field, particularly *natural language processing* (NLP) research projects. Hence, a *sequence-to-sequence* (S2S) transformer model is used to encode the transactions and decode the DML operations with the Keras APIs. The construction of LSTM and GRU neural networks follows some textbooks [40], [41].

The translation model was built by using a Keras GRU-based encoder and decoder and SoftMax as a starter function, with the embedding dimension set to 256. Since the vocabulary of all field names in the data lake is not rich, the maximum value of vocabulary is set to 18,000. Moreover, the "forget gate" in GRU is activated which can help predict words in a sequence accurately and efficiently [42].

During the experiments, six test cases were designed to test the performance of different networks and datasets (3, 6, and 9 hundred thousand). After running each test case for 60 epochs, the results are summarized in Table 2.

TABLE 2. Test case results.

Case	Network	Size*	Epoch	Loss	Accuracy
1	LSTM	3.0	60	0.0322	0.6644
2	LSTM	6.0	60	0.0217	0.7358
3	LSTM	9.0	60	0.0182	0.8203
4	GRU	3.0	60	0.0024	0.7647
5	GRU	6.0	60	0.0018	0.8634
6	GRU	9.0	60	0.0013	**0.9680**

* *one hundred thousand per unit.*

Fig.9 shows the accuracy values for both networks. The curves show that the overall performance of the GRU neural network outperforms LSTM. After more than 50 epochs of training, the score of the model was roughly stable, so training was stopped at 60 epochs. Furthermore, large datasets can lead to higher accuracy. As a consequence, this software framework uses GRU neural networks to generate DML operations.

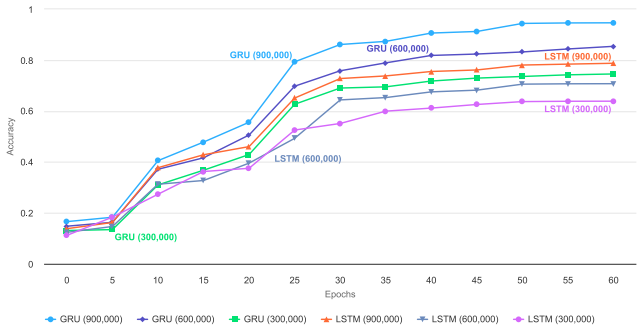

FIGURE 9. Evaluation of GRU and LSTM neural networks.

C. PERFORMANCE EVALUATION

Section III introduced the features and usage of the JAMDL framework. This section provides a performance evaluation of the JAMDL framework and compares it with well-known ORMapping software frameworks in the market (Hibernate and MyBatis).

Honestly, it is not easy to fairly compare different ORMapping software frameworks. Performance may depend on the environment (operating system, hardware configuration, database engine, data complexity, etc.). Since neither the Hibernate nor MyBatis software frameworks fully support semi-structured and unstructured data, performance testing can only be done on structured data.

Four test cases are designed for evaluation. Case 1 has a table with less than a hundred data records. Case 2 has a table with about one million data records. Case 3 has three separate tables, each with fewer than a hundred data records. Case 4 has three separate tables, each with about one million data records. Then, the read and write speeds of the three software frameworks are measured for each case. Four cases are tested on the same environment and the results are summarized in Table 3.

TABLE 3. Performance evaluation results.

Case	JAMDL		MyBatis		Hibernate	
	Read	Write	Read	Write	Read	Write
1	0.018	0.644	0.016	2.847	0.022	3.219
2	0.282	5.954	0.318	61.262	0.422	63.436
3	10.735	15.435	10.032	99.395	12.483	101.556
4	15.250	18.434	13.322	110.862	18.032	126.233

* *Read and write speeds are measured in seconds.*

The resulting data in Table 3 are then plotted into pie charts for comparison, as shown in Fig. 10. Each pie chart is either a read or a write operation to the database for each of the four test cases. Moreover, the resultant data (database operation time) in the pie charts was converted into a percentage for comparison and visualization, where the higher the percentage, the longer it took to complete the database operation.

The first impression of these pie charts in Fig. 10 is that the speed of the read operations on the left are similar, while

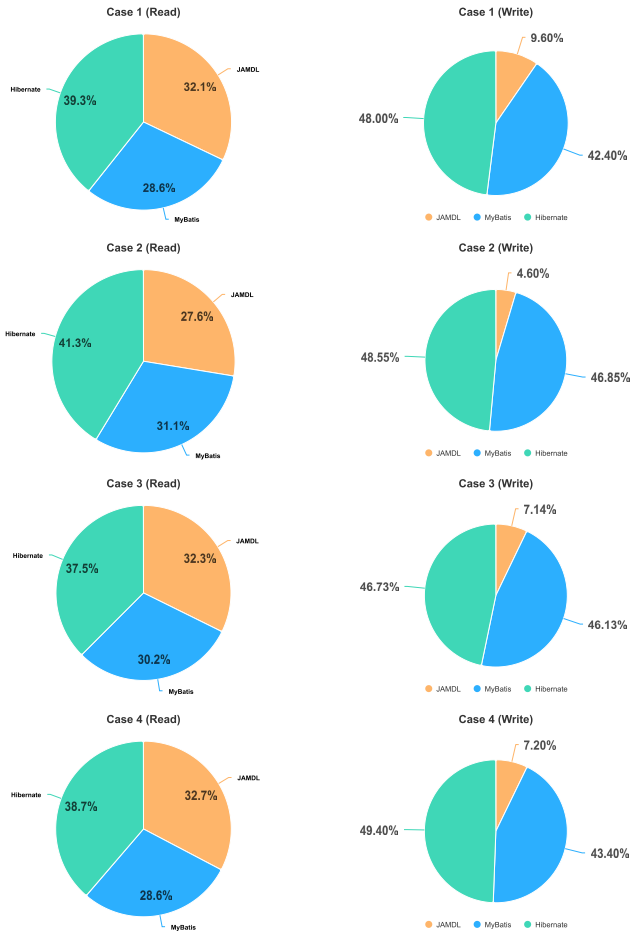

FIGURE 10. Comparison of database running speeds of different software frameworks.

JAMDL (in orange) takes less time in the write operations on the right. There are several reasons why these pie charts are distributed in this pattern, which are summarized below.

1) DIFFERENT PURPOSES

These three software frameworks were designed for different purposes, and are more concerned with functionality than just speed of operation. Hibernate is designed for larger applications can be used in most development environments and supports most database engines. Therefore, it is considered to be more heavyweight than the others because it has more libraries consuming the memory. On the other hand, MyBatis is designed for small and medium-sized applications and is therefore easier to use. The JAMDL framework focuses on processing different structured data in a data lake. Hence, it is less compromised for database and operating system environments.

2) BATCH OPERATIONS

Since the JAMDL framework implements batch processing, it takes less time to write database records. Batch processing allows developers to combine related SQL statements into a single batch and commit it to the database with a single call [37]. As a result, it reduces communication overhead and logging, thereby improving performance. For example, if a transaction needs to write to fifty data records in a table, the database engine treats the batch processing as a single operation, not fifty separate operations.

In another aspect, Hibernate has heavily adopted the database optimizers and the caching and buffering features of database engines allow previously executed queries to perform significantly better and faster compared to the other software frameworks. Therefore, the JAMDL framework should also employ some database optimizers to improve robustness.

3) TABLE JOIN

Since neither Hibernate nor MyBatis supports complex queries connecting to multiple database tables, people usually need to write their own SQL statements. Therefore, performance becomes similar due to executing SQL directly instead of going through the framework to generate the SQL statements. However, JAMDL better supports the generation of complex queries, saving developers time in developing SQL. Consequently, using the JAMDL framework for manipulating multiple tables is more efficient than the other two software frameworks.

4) IMPACT ON COMPLEXITY

The SQL statements generated in the JAMDL framework are high-level programming languages. Database engines can use the built-in functions to optimize these statements, and the optimization results are similar to those obtained by manually writing SQL. The purpose of the JAMDL framework is mainly to handle different structured data in the data lake through Java annotations and there should not be any impact on complexity.

D. DYNAMIC DATASETS

The manipulation of different structures in data lakes is fully demonstrated in Section III. One may realize that switching from one data structure to another as time goes by requires a huge amount of effort. Others may be aware of the feasibility of managing multiple data structures simultaneously.

The JAMDL framework abstracts data structures into an object model that can represent different data structures at the same time. In addition, the lightweight nature of Java annotations makes it easy to modify the object model to meet inconsistent demand. Therefore, the JAMDL framework can address the concerns when managing complex data.

V. CONCLUSION

In this article, a software framework JAMDL based on ORMapping is comprehensively presented. JAMDL aims to provide a solution for the manipulation of different data structures in a data lake. JAMDL solves the problem of managing diverse data and overcomes the difficulty of transforming data between different structures in the data

lake. Some important features of this software framework are summarized below, and some of the main features are compared in Table 4.

- Java annotations are used as objects that represent diverse data structures (structured, semi-structured, and unstructured data).
- Java objects are used to read and write different data structures in a data lake.
- In addition to inner joins, a variety of table joining methods are available.
- By adopting the *schema-on-read* approach, JSON objects are dynamically generated using the ''Bracket Schema'' notation.
- GRU neural networks are used to build E2E AI models to generate DML operations.
- A recommendation AI model is implemented to improve the visibility of data when data precipitation occurs.

TABLE 4. ORMapping software framework comparison.

	Hibernate	JPA	MyBatis	TopLink	JAMDL
Purpose	Complete	Standard	Easy to use	Commercial	Data Lake
Mapping	XML	XML	XML	GUI	Java Annotation
Table Join	inner	inner	inner	inner	inner, intercept, full, left, right, minus, union
Data Type	structured	structured	structured	structured	structured, unstructured, semi-structured
Query	HQL	JPQL	Dynamic SQL	GUI	Objects, SQL
Use of AI	n/a	n/a	n/a	n/a	DML, DQL, rarely access data suggestion

This article demonstrates how to prepare training and testing data for building LSTM and GRU neural networks. In addition, six test cases were used to evaluate the effectiveness of the resulting DML operations. The results show that GRU has the highest accuracy of 0.9589 using 900,000 training data. Moreover, the JAMDL framework can write faster than the other software frameworks.

The job responsibilities of an enterprise data steward have created many new and interesting areas of research. There is ample space to explore further and help them manage and deliver high-quality data.

Data lake is one of the important development projects in enterprise recently. The JAMDL framework is designed to rapidly develop web applications and data warehouse-type software applications for manipulating different structures of data in data lakes. Unlike the majority of data lake projects that focus merely on architecture, JAMDL contributes people to governing big data holistically and efficiently. It also inspires people to manage data lakes from different perspectives using AI techniques.

Data are facts and evidence that deserve to be preserved forever. Even though computer science has evolved over many eras, data always tells us new stories with new technologies. Unlike other fields of study, data is not confined to a specific domain but is cross-domain and ubiquitous. As the appearance of data continues to change (structured, semi-structured, unstructured, etc.), people will continue to face new challenges to either play with data or be played by data.

REFERENCES

[1] A. Badia, *SQL for Data Science: Data Cleaning, Wrangling and Analytics With Relational Databases*. Cham, Switzerland: Springer, Nov. 2020.
[2] J. Reis and M. H. Housley, *Fundamentals of Data Engineering: Plan and Build Robust Data Systems*. Sebastopol, CA, USA: O'Reilly Media, Jul. 2022.
[3] D. Colley and C. Stanier, ''Identifying new directions in database performance tuning,'' *Proc. Comput. Sci.*, vol. 121, pp. 260–265, Jan. 2017.
[4] T. Neward, ''The Vietnam of computer science,'' in *Political Science*, Jun. 2006.
[5] M. Keith, M. Schincariol, and M. Nardone, *Pro JPA 2 in Java EE 8: An in-Depth Guide to Java Persistence*, 3rd ed. Apress, Feb. 2018.
[6] L. H. Z. Santana and R. D. S. Mello, ''Persistence of RDF data into NoSQL: A survey and a reference architecture,'' *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 3, pp. 1370–1389, Mar. 2022.
[7] A. Gorelik, *The Enterprise Big Data Lake: Delivering the Promise of Big Data and Data Science*. Sebastopol, CA, USA: O'Reilly Media, Mar. 2019.
[8] R. Hai, C. Koutras, C. Quix, and M. Jarke, ''Data lakes: A survey of functions and systems,'' *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12571–12590, Dec. 2023.
[9] E. Zagan and M. Danubianu, ''Data lake architecture for storing and transforming web server access log files,'' *IEEE Access*, vol. 11, pp. 40916–40929, 2023.
[10] A. A. Munshi and Y. A. I. Mohamed, ''Data lake lambda architecture for smart grids big data analytics,'' *IEEE Access*, vol. 6, pp. 40463–40471, 2018.
[11] C. Horstmann, *Core Java: Advanced Features*, vol. 2. Oracle Press, Apr. 2022.
[12] H. Rocha and M. T. Valente, ''How annotations are used in Java: An empirical study,'' in *Proc. 23rd Int. Conf. Softw. Eng. Knowl. Eng.*, Jul. 2011, pp. 426–431.
[13] S. Roubtsov, A. Serebrenik, and M. van den Brand, ''Detecting modularity 'smells'' in dependencies injected with java annotations,'' in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2010, pp. 244–247.
[14] Y. Liu, Y. Yan, C. Sha, X. Peng, B. Chen, and C. Wang, ''DeepAnna: Deep learning based Java annotation recommendation and misuse detection,'' in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2022, pp. 685–696.
[15] F. Mancini, D. Hovland, and K. A. Mughal, ''The SHIP validator: An annotation-based content-validation framework for Java applications,'' in *Proc. 5th Int. Conf. Internet Web Appl. Services*, May 2010, pp. 122–128.
[16] C. He, Z. Li, and K. He, ''Identification and extraction of design pattern information in Java program,'' in *Proc. 9th ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw., Parallel/Distrib. Comput.*, 2008, pp. 828–834.
[17] L. Jicheng, Y. Hui, and W. Yabo, ''A novel implementation of observer pattern by aspect based on Java annotation,'' in *Proc. 3rd Int. Conf. Comput. Sci. Inf. Technol.*, vol. 1, Jul. 2010, pp. 284–288.
[18] B. Nuryyev, A. Kumar Jha, S. Nadi, Y.-K. Chang, E. Jiang, and V. Sundaresan, ''Mining annotation usage rules: A case study with MicroProfile,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2022, pp. 553–562.
[19] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, ''Characterizing the usage, evolution and impact of Java annotations in practice,'' *IEEE Trans. Softw. Eng.*, vol. 47, no. 5, pp. 969–986, May 2021.
[20] M. Chatham, *Structured Query Language by Example—Volume I: Data Query Language*, 1st ed. Nov. 2012.
[21] M. Lorenz, G. Hesse, and J.-P. Rudolph, ''Object-relational mapping revised—A guideline review and consolidation,'' in *Proc. 11th Int. Joint Conf. Softw. Technol.*, 2016, pp. 157–168.
[22] C. Richardson, *Microservices Patterns With Examples in Java*, 1st ed. Manning, Oct. 2018.
[23] G. Liyanaarachchi, L. Kasun, M. Nimesha, K. Lahiru, and A. Karunasena, ''MigDB–relational to NoSQL mapper,'' in *Proc. IEEE Int. Conf. Inf. Autom. for Sustainability (ICIAfS)*, Dec. 2016, pp. 1–6.
[24] A. Gandomi and M. Haider, ''Beyond the hype: Big data concepts, methods, and analytics,'' *Int. J. Inf. Manage.*, vol. 35, no. 2, pp. 137–144, Apr. 2015.
[25] P. Kaewkamol, ''Data governance framework as initiative for higher educational organisation,'' in *Proc. Joint Int. Conf. Digit. Arts, Media Technol. ECTI Northern Sect. Conf. Electr., Electron., Comput. Telecommun. Eng.*, Jan. 2022, pp. 175–178.

[26] Y. Demchenko and L. Stoy, "Research data management and data stewardship competences in university curriculum," in *Proc. IEEE Global Eng. Educ. Conf. (EDUCON)*, Apr. 2021, pp. 1717–1726.
[27] I. H. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," *Social Netw. Comput. Sci.*, vol. 2, no. 6, pp. 1–20, Aug. 2021.
[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30. Red Hook, NY, USA: Curran Associates, Dec. 2017, pp. 5999–6009.
[29] F. Mortezapour Shiri, T. Perumal, N. Mustapha, and R. Mohamed, "A comprehensive overview and comparative analysis on deep learning models: CNN, RNN, LSTM, GRU," 2023, *arXiv:2305.17473*.
[30] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proc. SSST EMNLP*, Sep. 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:11336213
[31] B. Desplanques, J. Thienpondt, and K. Demuynck, "ECAPA-TDNN: Emphasized channel attention, propagation and aggregation in TDNN based speaker verification," in *Proc. Interspeech*, Oct. 2020, pp. 3830–3834.
[32] T. Glasmachers, "Limits of end-to-end learning," in *Proc. 9th Asian Conf. Mach. Learn. (ACML)*, pp. 17–32, Apr. 2017.
[33] L. M. Hoi, W. Ke, and S. K. Im, "Data augmentation for building QA systems based on object models with star schema," in *Proc. IEEE 3rd Int. Conf. Power, Electron. Comput. Appl. (ICPECA)*, Jan. 2023, pp. 244–249.
[34] L. M. Hoi, W. Ke, and S. K. Im, "Corpus database management design for chinese-portuguese bidirectional parallel corpora," in *Proc. IEEE 3rd Int. Conf. Comput. Commun. Artif. Intell. (CCAI)*, May 2023, pp. 103–108.
[35] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design With Applications*, 3rd ed. Reading, MA, USA: Addison-Wesley Professional, Apr. 2008.
[36] L. M. Hoi. (Mar. 2023). *An Open-Source Software Framework for Manipulating Data Lakes*. [Online]. Available: https://github.com/LapmanHoi/Annotation
[37] Y. Bai, *JDBC API and JDBC Drivers*, 1st ed. Hoboken, NJ, USA: Wiley, May 2012.
[38] C. Beeri, P. A. Bernstein, and N. Goodman, "A sophisticate's introduction to database normalization theory," in *Proc. 4th Int. Conf. Very Large Data Bases*, Sep. 1978, pp. 113–124.
[39] S. Botros, *High Performance MySQL: Proven Strategies for Operating at Scale*, 4th ed. O'Reilly Media, Dec. 2021.
[40] A. Géron, *Hands-on Machine Learning With Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2nd ed. O'Reilly Media, Oct. 2019.
[41] F. Chollet, *Deep Learning With Python*, 2nd ed. Manning, Dec. 2021.
[42] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, "End-to-end memory networks," in *Proc. Conf. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 2440–2448.

LAP MAN HOI (Member, IEEE) received the bachelor's degree in computer science from York University, Canada, and the master's degree in internet computing from the Queen Mary University of London. He is currently pursuing the Ph.D. degree in computer applied technology with the Faculty of Applied Sciences, Macao Polytechnic University (MPU). He was a Researcher of gaming and entertainment. He is also a Researcher of machine translation with the Faculty of Applied Sciences, MPU. His research interests include internet computing, data warehouse, data science, gaming, deep learning, machine translation, and voice recognition.

WEI KE (Member, IEEE) received the Ph.D. degree from the School of Computer Science and Engineering, Beihang University. He is currently a Professor with the Faculty of Applied Sciences, Macao Polytechnic University. His research interests include programming languages, image processing, computer graphics, tool support for object-oriented and component-based engineering and systems, the design and implementation of open platforms for applications of computer graphics, and pattern recognition, including programming tools, environments, and frameworks.

SIO KEI IM (Member, IEEE) received the degree in computer science and the master's degree in enterprise information systems from the King's College London, University of London, U.K., in 1998 and 1999, respectively, and the Ph.D. degree in electronic engineering from the Queen Mary University of London (QMUL), U.K., in 2007. He gained the position of a Lecturer with the Computing Program, Macao Polytechnic Institute (MPI), in 2001. In 2005, he became the Operations Manager of the MPI-QMUL Information Systems Research Center jointly operated by MPI and QMUL, where he carried out signal processing work. He was promoted to a Professor with MPI, in 2015. He was a Visiting Scholar with the School of Engineering, University of California, Los Angeles (UCLA), and an Honorary Professor with The Open University of Hong Kong.