ORM-X: A Light-Weight Object-Relational Mapping Framework for MySQL

Sichen Tao
The Communication University of China, China
Corresponding Author: Sichen Tao (3209748898@qq.com)

Abstract: At present, ORM frameworks in the Java ecosystem cannot simultaneously satisfy the dual demands of high-concurrency business and rapid development, exhibiting a clear polarization. Semi-automatic ORM frameworks allow developers full control over SQL statements and make performance tuning straightforward, but they require repetitive coding for basic CRUD operations. The logic for assembling dynamic SQL is complex, resulting in high development redundancy. Fully automatic ORM frameworks simplify data operations and significantly accelerate development speed, but their automatically generated underlying SQL lacks transparency, easily leading to performance issues such as N+1 queries and redundant queries. Debugging and optimization in high-concurrency scenarios become extremely difficult. In addition, most mainstream existing frameworks are primarily designed for traditional monolithic architectures and provide inadequate support for modern microservice features such as cloud-native capabilities, multi-tenancy, and dynamic data sources, failing to meet the development needs of contemporary distributed projects [1-3]. Accordingly, this paper proposes a new ORM framework called ORM-X, which balances development efficiency and runtime performance while adapting to cloud-native architectures. Featuring a concise architecture, outstanding performance, and low system overhead, ORMX significantly improves the development efficiency of Java back-end systems and offers high practical application value.

Keywords: Object-Relational Mapping; ORM Framework; SQL; Java Backend

1. System Architecture

The ORM-X technical architecture is divided from bottom to top into four layers: Data Storage Layer, SQL Engine Layer, Data Table Access Layer, and Data Access Service Layer (see Figure 1)[4].

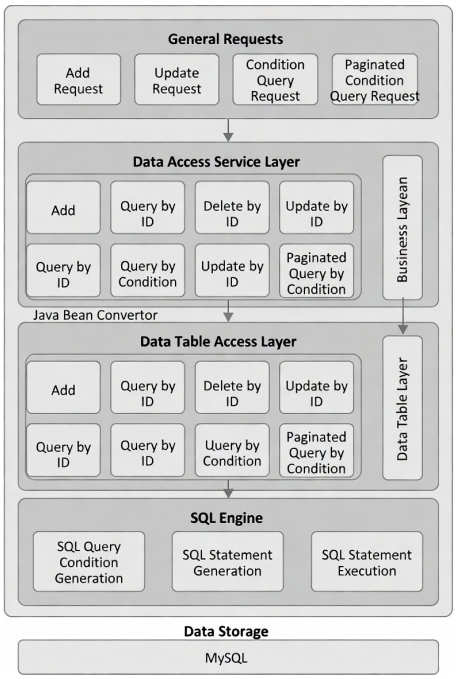

Figure 1: ORM-X Technical Architecture.

Responsibilities of each logical layer:

- **Data Storage Layer**: Data is stored in MySQL.
- **SQL Engine Layer**: Automatically generates SQL statements based on Java Beans, including query condition generation, complete SQL statement generation, and SQL execution.
- **Data Table Access Layer**: Provides generic implementations for data insertion, deletion, update, query by ID, query by condition, and paginated query by condition at the table level.
- **Data Access Service Layer**: Provides service-oriented implementations of the above operations and a set of encapsulated common request handlers. For example, the data insertion operation includes: (1) pre-insertion data validation (common checks include data correctness and existence validation; different business data can define custom validation), and (2) calling the Data Table Access Layer to perform the actual insertion.
- **Java Bean Converter**: Responsible for bidirectional conversion between business model Java Beans and data table model Java Beans. For the same data, business Java Beans are visible to callers, while data table Java Beans directly map to database tables and are invisible to callers. The separation exists because some business fields require different formats in the database versus external APIs (e.g., enum fields may be stored as names in the database but exposed as enum values externally; the converter handles this transformation).

2. Core Technical Implementation

ORM-X must solve a fundamental problem: how the SQL Engine Layer automatically generates SQL statements [5].

2.1 Column Access in SQL

Common single-table operation SQL statements (where f stands for field) are summarized as follows:

- Insert:
  Paginated Query by Condition:
- Delete:
  delete from <table> where id=<id>
- Update:
  update <table> set <f1>=<f1_val>,<f2>=<f2_val>,... where id=<id>
- Query by ID:
  select <f1>,<f2>,... from <table> where id=<id>
- Query by Condition:
  select <f1>,<f2>,... from <table>
  where <f1>=<f1_val> and <f2>=<f2_val> and ...
- Paginated Query by Condition:
  select <f1>,<f2>,... from <table>
  where <f1>=<f1_val> and <f2>=<f2_val> and ...
  limit <begin>, <length>

Field names and values are derived from the business Java Bean. To better determine whether a field has been assigned a value, all Java Bean fields are required to use wrapper types (e.g., Integer, Long, Boolean, String, Float, Date) rather than primitives (int, long, boolean, etc.).

2.2 Supported Query Conditions

ORM-X analyzes whether different query conditions can be expressed via Java Beans (using TestObj as an example):

```
public class TestObj {
    private String name;
private Integer age;
}
```

- Equality conditions: By convention, if a field of a Java Bean object is assigned a value, all fields are considered equal. Query conditions generated from multiple fields are connected using the AND operator.
- Inequality conditions: By convention, if a field of a Java Bean object is assigned a value, all fields are considered as inequality conditions. Query conditions generated from multiple fields are connected using the AND operator.
- Less than the query condition: By convention, when a field of a Java Bean object is assigned a value, that field is treated as a less-than condition. Multiple fields are joined together in the query condition using AND. This only applies to fields of type Comparable.
- LIKE conditions: By convention, if a field of a Java Bean object is assigned a value, that field will be treated as a condition in a LIKE query. Query conditions generated from multiple fields are connected using AND.
- BETWEEN conditions: Unlike other query conditions, `between` describes the range of values a field can take. Therefore, it includes the three elements `<field, begin, end>`, which translates to the following query condition: <field> between <begin> and <end>.
- AND/OR composite conditions: AND/OR query conditions are combined query conditions, and

the corresponding BNF is shown below:

```
combine_condition ::= condition <operator> condition
conditon ::= combine_condition | equal_condition | not_equal_conditon |
             less_condition | less_or_equal_condition |
             greater_condition | greater_or_equal_condition |
             like_condition | between_condition
```

For single-table or simple primary-key join scenarios, using Java Bean objects to automatically generate SQL is feasible, replacing the verbose MyBatis development model. The following sections detail ORM-X's class design and implementation.

3. Core Component Design

3.1 Transaction Template

All insert, update, and delete operations are wrapped in transactions. ORM-X's core idea is to generate SQL automatically from Java Beans; write operations must execute within transactions (see Figure 2) [6].

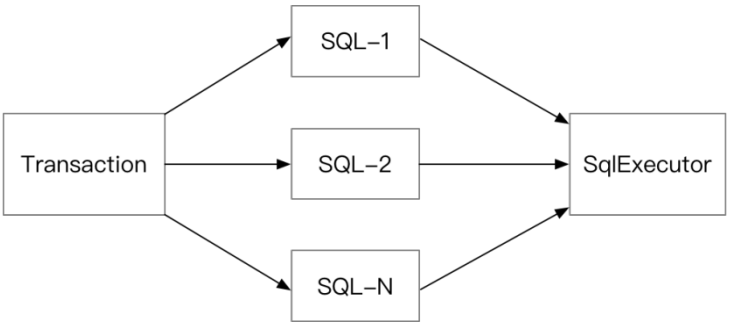

Figure 2: Relationship between Transactions and SQL Execution

Multiple SQL statements must share the same Connection object to ensure proper rollback. SqlExecutor therefore needs access to both the SQL and the Connection.

Using Spring's TransactionTemplate and Alibaba's Druid connection pool, the dependency path is shown in Figure 3.

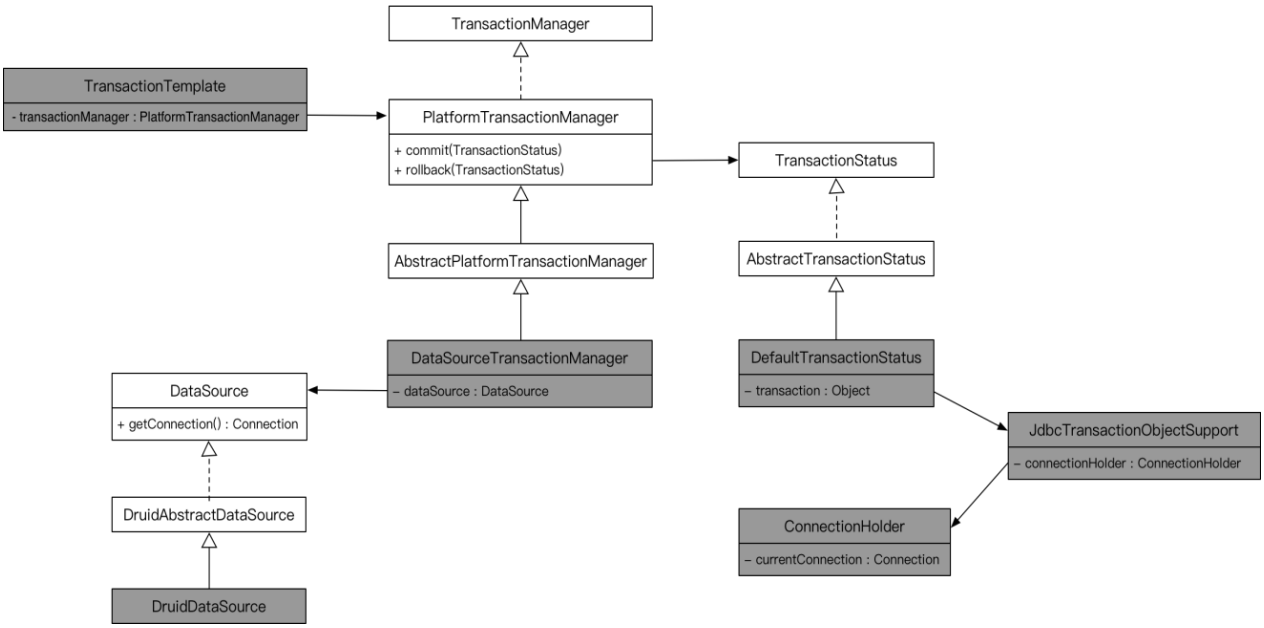

Figure 3: Dependency Path from TransactionTemplate to Connection.

In application development, TransactionTemplate is generally initialized by Spring injection [7]. The corresponding gray class in Figure 3 is shown in the following pseudocode.

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource. DataSourceTransactionManager" >
     <property name="   dataSource" ref="dataSource" />
</bean>

<bean id="transactionTemplate"
class="org.springframework.transaction.support.TransactionTemplate">
     <property name="transactionManager" ref="transactionManager" />
     <property name="isolationLevelName" value="ISOLATION_DEFAULT" />
     <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED" />
</bean>
```

The execute method of TransactionTemplate executes a transaction, as shown in the following code:

```
transactionTemplate.execute(transactionStatus -> {
// SqlExecutor.run(sql-1)
// SqlExecurotr.run(sql-2)
// ...
// SqlExecutor.run(sql-n)
return T;
});
```

Referring to Figure 2 and the pseudocode above, the execution of SQL statements is handled by SqlExecutor. SqlExecutor needs to see not only the SQL statement but also the Connection object held

by the transaction, which can be obtained through TransactionStatus. For SqlExecutor, regardless of which SQL statement is executed, it must be able to obtain the TransactionStatus. This can be maintained using threads, i.e., maintaining a mapping of <thread, TransactionStatus>. This paper enhances the transaction processing.

3.2 SqlExecutor

Write operations (SQL queries) are wrapped in transactions and executed using a custom SqlExecutor. However, read operations (SQL queries) do not require transactions. So how is a Connection obtained when a read operation is executed? [7] The SqlExecutor implementation maintains a TransactionManagerEx instance. The details of obtaining the `Connection` for read and write operations are as follows:

1) For read operations (SQL queries), they do not need to be wrapped in transactions for execution. The code implementation for obtaining the Connection in the SqlExecutor implementation is shown below.

```
TransactionManager transMgr = transactionManagerEx.getTransactionTemplate()
        .getTransactionManager();
DataSource dataSource = ((DataSourceTransactionManager) transMgr).getDataSource();
Connection conn = dataSource.getConnection();
```

Connections must be closed manually, otherwise Connection resources will be leaked.

2) For write SQL operations, execution needs to be wrapped in a transaction. In the SqlExecutor implementation, the code implementation corresponding to the Connection retrieval path is shown below.

```
TransactionStatus transactionStatus = transactionManagerEx.getCurrentTransactionStatus();
 Preconditions.checkNotNull(transactionStatus, "insert() operation not wrapped by transaction");

JdbcTransactionObjectSupport txobj = (JdbcTransactionObjectSupport)
      ((DefaultTransactionStatus) transactionStatus).getTransaction();
Connection conn = txobj.getConnectionHolder().getConnection();
```

The acquired Connection cannot be closed because it is opened internally by Spring TX and will be closed after the transaction ends. If it is manually closed, the rollback operation on the data tables will fail during transaction rollback, resulting in a functional exception.

Because different types of operations require different return values from the SQL statements, SqlExecutor provides different interfaces for executing SQL statements of different operation types.

4. Application Integration

ORM-X is distributed as a JAR package that can be referenced by other applications [8-9].

```
<dependency>
    <groupId>com.github.taosicheng</groupId>
    <artifactId>orm-x</artifactId>
    <version>1.0.0</version>
</dependency>
```

5. Performance Comparison

5.1 Test Environment

Hardware: 8-core CPU, 32GB RAM

Software: JDK 17, MySQL 8.0, JMH benchmarking tool

Data volume: 100,000 records per table

Table 1: ORM Framework Performance Comparison

Test Scenario	Unit	ORMX	MyBatis-Plus	Spring Data JPA (Hibernate)	Hibernate Native
Single Insert	ms	0.91	0.93	1.10	1.15
Single Query by ID	ms	0.36	0.37	0.51	0.54
1,000-row Batch Insert	ms	14.2	14.8	23.5	24.7
1,000-row Batch Query	ms	19.5	20.1	27.2	28.1
Three-table Join Query	ms	24.3	24.8	33.6	35.2
1,000-thread Concurrent QPS	ops/sec	11150	11200	8100	7680
Peak Memory Usage	MB	132	138	175	183

Test Conclusions:

ORM-X delivers the best overall performance.

It combines annotation-driven simplicity with SQL controllability, removes redundant entity state management, and outperforms MyBatis-Plus slightly (especially in high-concurrency and batch scenarios). It shows significant improvement over traditional Hibernate-based solutions.

While Hibernate can improve query performance with lazy loading and second-level caching, cache contention in concurrent scenarios further reduces throughput [10].

6. Conclusion

This paper presents a lightweight, high-performance new Java ORM framework: ORM-X. By combining the strengths of fully automatic and semi-automatic ORM designs, it uses annotations for entity-table mapping, provides a type-safe chained query DSL, eliminates redundant entity state management, and natively supports dynamic data sources and batch operations. ORM-X retains the development simplicity of full-automatic ORMs while delivering performance close to MyBatis. It meets the needs of both rapid development for small/medium projects and high-concurrency requirements of large-scale internet applications.

References

[1] Ambler, S.W., Krumbein, F. and Sun, Y. 2003. Object-Relational Mapping (ORM): A Practical Approach. IBM

Press. DOI:10.1002/9780470134921.

[2] Begin, C., Reddy, S.P. and Li, J. 2009. MyBatis: SQL Mapping Framework for Java. Manning Publications.

[3] Bauer, C., King, G. and Tudose, C. 2021. Java Persistence API (JPA) 3.0: Specification and Practice. In Proceedings of the 2021 17th International Conference on Computer Science and Control Systems (CSCS). IEEE, 412–417. DOI:10.1109/CSCS52396.2021.00076.

[4] Jones, N., Gough, K. and Zhang, H. 2018. Abstract syntax trees in compiler design and ORM parsing. ACM SIGPLAN Notices. 53, 9 (2018), 27–34. DOI:10.1145/3274809.3274815.

[5] Gray, J., Reuter, A. and Wang, L. 1993. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc.

[6] Pan, Z., Smith, R. and Garcia, M. 2012. Druid: A high-performance and monitoring-enabled JDBC connection pool. In Proceedings of Java EE Summit. Alibaba Open Source. https://github.com/alibaba/druid.

[7] Walls, C., Vogels, W. and Liu, X. 2020. Spring in Action: Dependency Injection & Application Context, 6th ed. Manning Publications.

[8] Bauer, C., King, G. and Gregory, G. 2015. Java Persistence with Hibernate, 2nd ed. Manning Publications.

[9] Bauer, C., Odubăşteanu, C. and Li, Y. 2021. Entity lifecycle management in ORM systems. In Proceedings of the 2021 17th International Conference on Computer Science and Control Systems (CSCS). IEEE, 501–506. DOI:10.1109/CSCS52396.2021.00076.

[10] Yadgar, G., Factor, M., Li, K. and Schuster, A. 2011. Management of multilevel, multiclient cache hierarchies. ACM Transactions on Computer Systems. 29, 2 (2011), 1–31. DOI:10.1145/1963559.1963561.