International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

ADVANCED HIBERNATE TECHNIQUES FOR OPTIMIZING DATABASE INTERACTIONS

Anishkumar Sargunakumar
Independent Researcher
Plainsboro, New Jersey

Abstract

Hibernate, a popular Object-Relational Mapping (ORM) framework for Java, offers developers powerful tools for managing database interactions. While its basic features are sufficient for most applications, advanced techniques enable optimized performance, scalability, and maintainability. This paper explores advanced Hibernate techniques, detailing their benefits and providing practical implementation examples to demonstrate their application.

Keywords: Hibernate, Object-Relational Mapping(ORM), Java, Object Mapping, Serialization and Deserialization

I. INTRODUCTION

Efficient database interaction is critical for modern applications to maintain high performance and scalability. Hibernate simplifies database operations by abstracting complexities, but default configurations can lead to performance bottlenecks. Advanced techniques like second-level caching, query optimization, custom user types, and multi-tenancy support empower developers to fine-tune database interactions. This paper delves into advanced Hibernate features and their implementations to guide developers in optimizing database performance [1][2][3].

II. LITERATURE REVIEW

A. Second-Level Cache

Overview

The second-level cache is a session-independent cache shared across sessions. By storing frequently accessed entities, collections, or query results, it minimizes redundant database interactions. This is especially useful in applications where the same data is queried repeatedly across different sessions. Implementing a second-level cache can drastically reduce the load on the database and improve response times, making it easier for developers to build high-performance applications without extensive query optimization [1][9][10].

International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

Implementation Example

- **Add Cache Dependency:**

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>6.x.x</version>
</dependency>
```

Fig. 1. Hibernate dependency in pom.xml

- **Enable Second-Level Cache:**

```
hibernate.cache.use_second_level_cache=true
hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory
hibernate.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider
```

Fig. 2. Hibernate configurations from property file

Second-level caching is an essential Hibernate feature that reduces the number of database queries by storing frequently accessed data in a shared cache across sessions. This is particularly useful for improving application performance when the same data is accessed repeatedly by different users or sessions. By caching entities, collections, and query results, second-level caching minimizes the load on the database and accelerates response times.

The Figure 2 illustrates the properties that are necessary to activate and manage this feature
hibernate.cache.use_second_level_cache=true activates the second-level cache mechanism in Hibernate.
hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory: specifies the caching provider and region factory to use. Here, JCache (JSR 107) is used as the caching standard.
hibernate.javax.cache.provider=org.ehcache.jsr107. EhcacheCachingProvider defines Ehcache as the JCache provider, enabling it to handle the caching operations efficiently.

- **Annotate Entity:**

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Product {
@Id
private Long id;
private String name;
private Double price;
}
```

Fig. 3. Cache annotation in class file

The **@Cache** annotation in Hibernate is used to specify that an entity should be cached in the second-level cache. This annotation helps Hibernate understand how to manage the caching

International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

behaviour for the annotated entity.

usage = CacheConcurrencyStrategy.READ_WRITE: This parameter defines the concurrency strategy for the cache. READ_WRITE ensures that the cached data is consistent with the database by using a read-write locking mechanism. This strategy is ideal for applications where entities are frequently read and occasionally updated, as it provides a balance between performance and consistency.

B. Query Optimization

Overview

From Hibernate Documentation, Query optimization techniques in Hibernate help reduce the time and resources required to fetch data. By leveraging features like query caching, batch fetching, and fetch strategies, developers can minimize database calls and improve application performance [1][8]. These techniques also simplify coding by automating complex optimizations, allowing developers to focus on business logic rather than query tuning.

Implementation Example

- **Enable Query Caching:**

```
hibernate.cache.use_query_cache=true
```

Fig. 4. Query Optimization property

This property enables query caching in Hibernate, allowing the results of frequently executed queries to be stored in the second-level cache. By caching query results, Hibernate avoids repeatedly executing the same query on the database, reducing database load and improving response times.

- **Cache Query Results:**

```
Query<Product> query = session.createQuery("FROM Product WHERE price > :price", Product.class);
query.setParameter("price", 100.0);
query.setCacheable(true);
List<Product> products = query.list();
```

Fig. 5. Query Cache in code

The first line creates a Query object using HQL (Hibernate Query Language) to fetch all Product entities with a price greater than a specified value. **session.createQuery** initializes the query, specifying the HQL statement and the expected result type **(Product.class).** The second line sets the value for the price parameter in the query. Here, it replaces price with 100.0, dynamically binding the parameter. The third line enables caching for the query results. When **setCacheable**(true) is used, hibernate stores the query results in the query cache (part of the second-level cache). Subsequent executions of the same query with identical parameters will fetch results from the cache instead of querying the database, improving performance. The fourth line in the figure 5 executes the query and retrieves the result as a List of Product objects. If the query result is already cached, hibernate fetches the data directly from the cache. Otherwise, it queries the database, stores the result in the cache, and returns the data.

International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

- **Batch Fetching:**

```
hibernate.default_batch_fetch_size=10
```

Figure 6. Hibernate default batch size

This property sets a default batch size for batch fetching. It instructs Hibernate to load collections or entities in batches of 10, reducing the number of database queries. Without batch fetching, Hibernate might generate multiple SELECT statements, one for each entity or collection. For example, if 100 orders need to be fetched, Hibernate might execute 100 queries. With batch fetching, Hibernate groups these fetches into fewer queries, significantly improving performance. If **default_batch_fetch_size=10** is set, Hibernate will group related entities into batches of 10 and execute fewer SELECT statements.

```
@OneToMany(fetch = FetchType.LAZY)
@BatchSize(size = 10)
private List<Order> orders;
```

Figure 7. Batch fetching java code

@BatchSize annotation is applied at the entity or collection level to override the default batch fetch size. **@OneToMany(fetch = FetchType.LAZY**): Specifies that the orders collection is lazily loaded, meaning the associated Order entities are fetched only when accessed. **@BatchSize(size = 10):** Indicates that Hibernate should fetch up to 10 Order entities in a single query when loading this collection.

C. Custom User Types

Overview

Custom user types in Hibernate allow developers to map non-standard Java objects, such as JSON structures or custom data types, to database columns. This feature simplifies the handling of complex data structures by automating serialization and deserialization, reducing boilerplate code and enhancing maintainability [6][7].

Implementation Example

- **Create a Custom UserType:**

```
public class JsonUserType implements UserType {
    @Override
    public Object nullSafeGet(ResultSet rs, String[] names, SharedSessionContractImplementor session, Object owner) throws SQLException {
        String json = rs.getString(names[0]);
        return json != null ? new ObjectMapper().readValue(json, Map.class) : null;
    }

    @Override
    public void nullSafeSet(PreparedStatement st, Object value, int index, SharedSessionContractImplementor session) throws SQLException {
        if (value == null) {
            st.setNull(index, Types.VARCHAR);
        } else {
            st.setString(index, new ObjectMapper().writeValueAsString(value));
        }
    }
    // Other required methods...
}
```

Fig. 8. Creating custom user type

International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

The JsonUserType class implements Hibernate's UserType interface. This interface provides methods to control how custom data types are read from and written to the database. The class is designed to map JSON data stored in a database column to a Java Map<String, Object>. The method nullSafeGet fetches Data from the Database. The ResultSet object (rs) represents a row in the database result and the rs.getString(names[0]) retrieves the value of the specified column (by name) as a JSON string. It then deserializes JSON to a Java Object. If the JSON string is not null, the code uses Jackson's ObjectMapper to deserialize the JSON into a Map<String, Object>. If the value is null, the method returns null. The purpose of the method is to ensure that JSON data stored in the database is converted into a usable Java object (a Map) when Hibernate retrieves it.
The method nullSafeSet and handles null values and serializes java object to JSON. If the value (Java object) is null, it sets the corresponding database column to NULL using st.setNull. If the value is not null, it uses Jackson's ObjectMapper to serialize the Map<String, Object> into a JSON string. This JSON string is then set as the value of the column in the database using st.setString. This method ensures that the Java object (a Map) is converted into a JSON string and stored in the database.

- **Apply Custom UserType:**

```
@Entity
public class Config {
    @Id
    private Long id;
    @Type(type = "com.example.JsonUserType")
    private Map<String, Object> settings;
}
```

Fig. 9. Apply custom user type

The @Entity annotation marks the Config class as a Hibernate-managed entity, meaning it maps to a database table. The @Id annotation indicates that the id field is the primary key of the table. The @Type annotation specifies that the settings field should use a custom UserType implementation (JsonUserType) to map the data between the database and Java. JsonUserType handles converting the Map<String, Object> to a JSON string when persisting to the database and converting the JSON string back to a Map when fetching from the database. The application of custom usertype works in Serialization and Deserialization.

1. **Serialization:** When saving or updating the Config entity, Hibernate calls the nullSafeSet method in the JsonUserType class. This method serializes the Map<String, Object> (stored in the settings field) into a JSON string and stores it in the corresponding database column.
2. **Deserialization:** When retrieving the Config entity, Hibernate calls the nullSafeGet method in the JsonUserType class. This method deserializes the JSON string stored in the database into a Map<String, Object> and assigns it to the settings field.

D. Multi-Tenancy Support

Overview

From Hibernate Community Forums, Multi-tenancy is a critical feature for applications serving multiple clients (tenants) with shared infrastructure. Hibernates multi-tenancy support enables

International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

developers to isolate data for different tenants efficiently [1][3][5]. This eliminates the need for separate codebases or extensive database configuration, making it easier to manage multi-tenant applications while maintaining security and scalability.

Implementation Example

- **Configure Multi-Tenancy:**

```
hibernate.multiTenancy=SCHEMA
hibernate.tenant_identifier_resolver=com.example.TenantIdentifierResolver
hibernate.multi_tenant_connection_provider=com.example.MultiTenantConnectionProvider
```

Fig. 10. Hibernate Multi-Tenancy properties

hibernate.multiTenancy=SCHEMA specifies the multi-tenancy strategy to use. In this case, SCHEMA indicates that each tenant's data is stored in a separate database schema. Other possible strategies include DATABASE (separate databases for each tenant) and DISCRIMINATOR (a column in shared tables differentiates tenants). The choice of strategy directly affects how data is segregated for different tenants. The SCHEMA approach is often preferred for shared database setups, as it isolates data while sharing a single database connection pool.

Hibernate .tenant_ identifier_ resolver= com .example. Tenant Identifier Resolver specifies the implementation of the **CurrentTenantIdentifierResolver** interface. This interface is responsible for determining the current tenant identifier (e.g., schema name) for each session or transaction. The **TenantIdentifierResolver** is critical in multi-tenant applications because it dynamically identifies which tenant's schema should be used based on the current request or session context. This property allows developers to manage tenant-specific schema switching seamlessly. By providing a custom implementation, developers can use request headers, JWT claims, or any other contextual data to resolve the tenant identifier.

```
public class TenantIdentifierResolver implements CurrentTenantIdentifierResolver {
    @Override
    public String resolveCurrentTenantIdentifier() {
        return TenantContext.getCurrentTenant(); // Fetch tenant from thread-local storage
    }

    @Override
    public boolean validateExistingCurrentSessions() {
        return true;
    }
}
```

Fig. 11. Implementation Tenant Identifier Resolver

hibernate.multi_tenant_connection_provider=com.example.MultiTenantConnectionProvider specifies the implementation of the MultiTenantConnectionProvider interface. This interface provides database connections for a specific tenant based on the resolved tenant identifier. This property is essential for managing database connections in a multi-tenant environment. It enables Hibernate to route queries to the correct schema or database based on the tenant identifier. By implementing this interface, developers can define how connections are created or pooled for each tenant. This ensures efficient and secure data access for all tenants.

International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

```java
public class MultiTenantConnectionProvider implements org.hibernate.engine.jdbc.connections.spi.MultiTenantConnectionProvider {
    @Override
    public Connection getConnection(String tenantIdentifier) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/" + tenantIdentifier; // Append tenant schema
        return DriverManager.getConnection(url, "user", "password");
    }

    @Override
    public Connection getAnyConnection() throws SQLException {
        return DriverManager.getConnection("jdbc:mysql://localhost:3306/default_schema", "user", "password");
    }

    @Override
    public void releaseConnection(String tenantIdentifier, Connection connection) throws SQLException {
        connection.close();
    }

    @Override
    public boolean supportsAggressiveRelease() {
        return true;
    }
}
```

Fig. 12. Implementation of Connection Provider

E. Hibernate Envers

Overview

Hibernate Envers provides a built-in auditing framework for tracking changes to entity data. This feature is essential for applications requiring historical data, compliance with auditing regulations, or version tracking. By automatically recording revisions, Envers reduces the need for manual logging, simplifying the development process and ensuring data integrity.

Implementation Example

- **Add Dependency:**

```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-envers</artifactId>
    <version>6.x.x</version>
</dependency>
```

Fig. 13. Hibernate-envers dependency in maven

- **Annotate Entity:**

```java
@Entity
@Audited
public class Employee {
    @Id
    private Long id;
    private String name;
}
```

Fig. 14. Annotate Entity

@Entity marks the Employee class as a persistent entity that maps to a table in the database. Hibernate uses this annotation to manage the class's lifecycle and its interactions with the

International Journal of Core Engineering & Management
Volume-6, Issue-08, 2020 ISSN No: 2348-9510

database. @Audited comes from Hibernate Envers and is used to enable auditing for the entity. When this annotation is added to an entity class, Hibernate Envers tracks changes made to instances of the entity over time. Each change to the entity creates a revision, which is stored in a separate audit table. This feature is essential for maintaining a history of changes for compliance, debugging, or version tracking. For example, you can query what the Employee entity looked like at a specific point in time or retrieve a list of all revisions.

- **Retrieve Revisions:**

```
AuditReader reader = AuditReaderFactory.get(session);
List<Number> revisions = reader.getRevisions(Employee.class, employeeId);
Employee oldVersion = reader.find(Employee.class, employeeId, revisions.get(0));
```

Fig. 15. Retrieve Revision code

Using Hibernate Envers' AuditReader, you can query historical data. This code retrieves all revisions of an Employee and fetches its state at a specific revision.

III. LIMITATIONS/CHALLENGES

While advanced Hibernate techniques offer significant performance benefits, they also come with challenges:

- **Complex Configuration:** Advanced features often require intricate setup, which can be error-prone [6].
- **Caching Issues**: Improper use of caching can lead to stale data or inconsistencies [9][10].
- **Debugging Difficulty**: Identifying issues in optimized queries or custom user types can be challenging due to abstraction layers [8].
- **Multi-Tenancy Overhead**: Implementing multi-tenancy increases complexity, especially when scaling [3][5].

IV. FUTURE SCOPE

Future research and development in Hibernate optimization can focus on:

- **Enhanced Tooling:** Development of intuitive tools for configuring and monitoring advanced Hibernate features [8].
- **AI-Driven Optimization**: Leveraging machine learning to predict and optimize query patterns dynamically [6].
- **Integration with Cloud Platforms**: Improved support for distributed databases and cloud-native environments [1][7].
- **Expanded Multi-Tenancy Models**: Incorporating more flexible strategies for tenant isolation and management [5].

V. CONCLUSIONS

- Advanced Hibernate techniques, such as second-level caching and query optimization,

significantly enhance application performance by reducing database interactions and improving response times [1][8].

- Custom user types in Hibernate simplify the management of complex data structures, enabling seamless integration with non-standard Java objects [6][7].
- Multi-tenancy support in Hibernate allows for efficient data isolation across tenants, ensuring scalability and security in shared infrastructure environments [3][5].
- Despite their benefits, these advanced features introduce complexity, requiring careful configuration and maintenance to avoid issues like stale data and debugging difficulties [6][9].
- The adoption of AI-driven optimization and enhanced tooling can further simplify the implementation of advanced Hibernate techniques, making them accessible to a broader audience [6][8].
- Future advancements in multi-tenancy strategies and cloud-native integration hold promise for addressing challenges in distributed and large-scale applications [5][7].
- Practical implementation examples provided in this paper serve as a guide for developers to adopt and effectively utilize advanced Hibernate features [1][8].

REFERENCES

1. Hibernate Documentation: https://hibernate.org/documentation
2. Hibernate Community Forums: https://discourse.hibernate.org
3. Java Persistence API (JPA) Specification
4. “Use of Hibernate in modern technology: Project Management” Sharayu Lokhande, Rushali Patil, Anup Kadam. International Journal of Computer Communication and Information System (IJCCIS)– Vol2. No1. ISSN: 0976–1349 July – Dec 2010.
5. “Research of Persistence Solution Based on ORM and Hibernate Technology”. International Journal of Advanced Research in Computer Science and Software Engineering Volume 7, Issue 4, April 2017 ISSN: 2277 128X.
6. Vlad Mihalcea. "High-Performance Java Persistence." https://vladmihalcea.com
7. Christian Bauer, Gavin King, and Gary Gregory. "Java Persistence with Hibernate, Second Edition." Manning Publications, 2015.
8. "Hibernate Performance Tuning Tips." Baeldung. https://www.baeldung.com/hibernate-performance-tips
9. Günter Tischler. "Understanding Hibernate Caching Mechanisms." https://tischler.net
10. Thorben Janssen. "A Beginner's Guide to Hibernate Second-Level Cache." https://thoughts-on-java.org/hibernate-second-level-cache