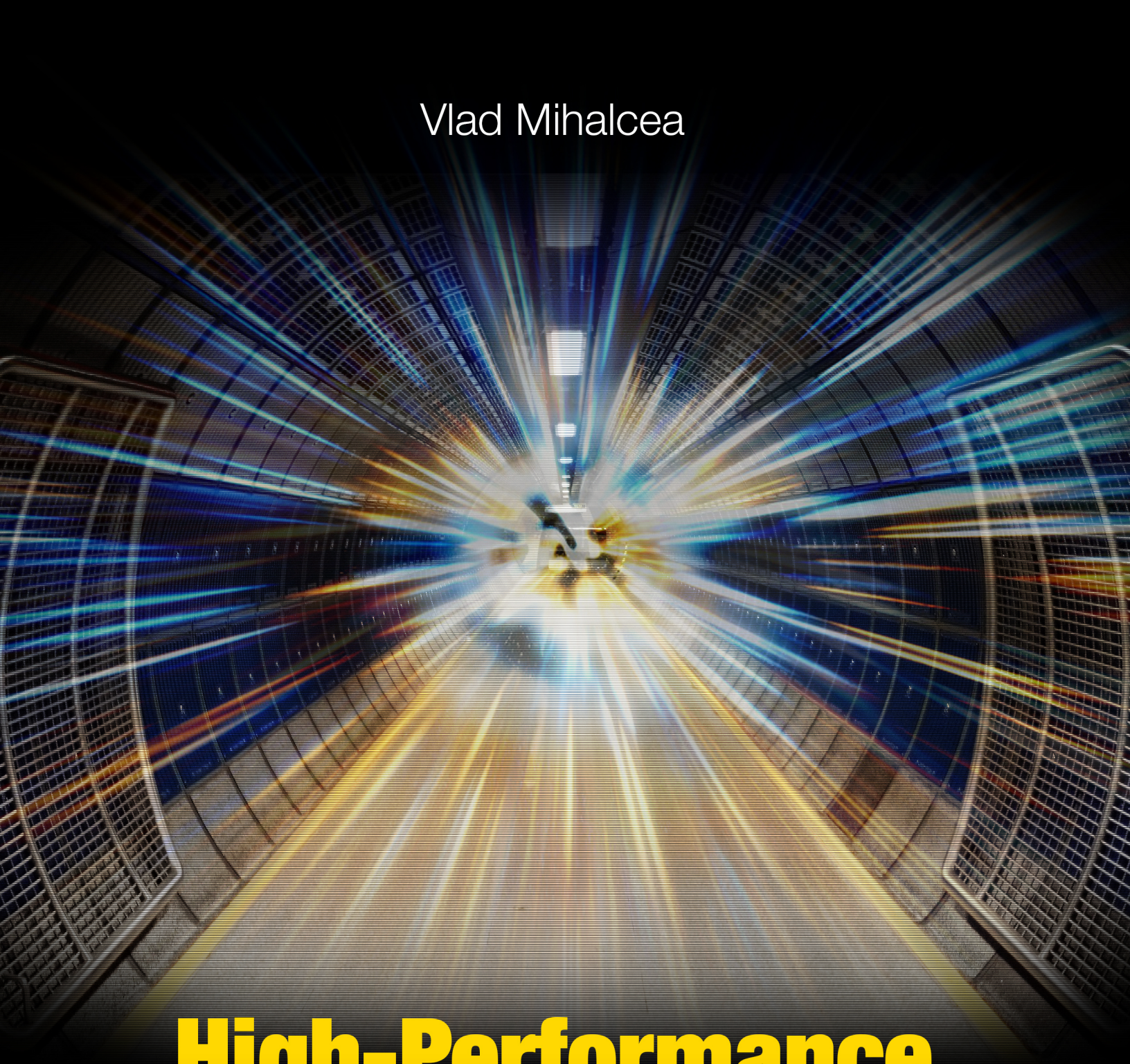
Vlad Mihalcea



High-Performance Java Persistence

Get the most out of your persistence layer

High-Performance Java Persistence

Get the most out of your persistence layer

Vlad Mihalcea

This book is for sale at http://leanpub.com/high-performance-java-persistence

This version was published on 2016-07-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Vlad Mihalcea by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just bought ”High-Performance Java Persistence” https://leanpub.com/high-performance-java-persistence/ by @vlad_mihalcea #Java #JPA #Hibernate

To my wife and kids

Contents

I Introduction

1. Preface

In an enterprise system, a properly designed database access layer can have a great impact on the overall application performance. According to Appdynamics[1]

> More than half of application performance bottlenecks originate in the database

Data is spread across various structures (table rows, index nodes), and database records can be read and written by multiple concurrent users. From a concurrency point of view, this is a very challenging task, and, to get the most out of a persistence layer, the data access logic must resonate with the underlying database system.

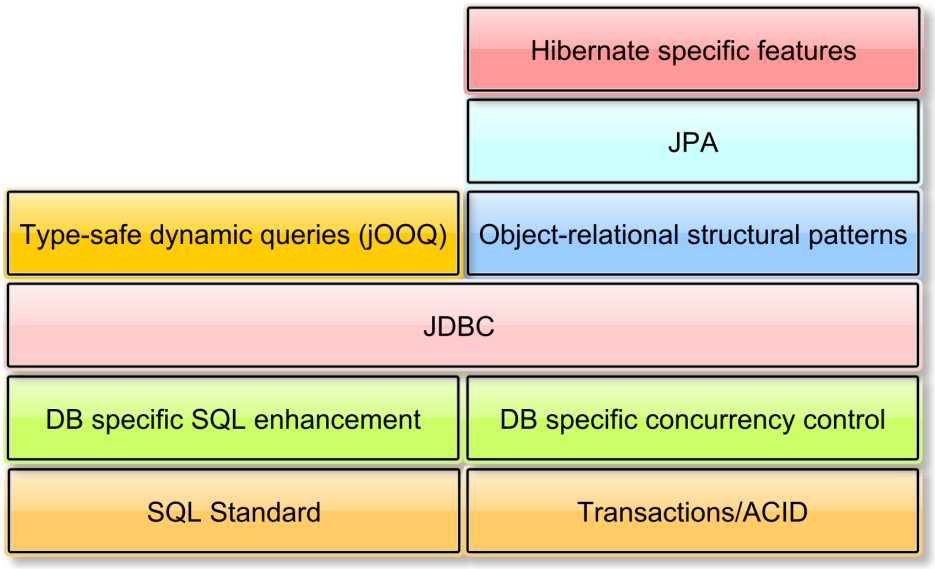


Figure 1.1: Data access skill stack

A typical RDBMS (Relational Database Management System) data access layer requires mastering various technologies, and the overall enterprise solution is only as strong as the team's weakest skills. Before advancing to higher abstraction layers such as ORM (Object-Relational Mapping) frameworks, it is better to conquer the lower layers first.

[1] http://www.appdynamics.com/solutions/database-monitoring/

1.1 The database server and the connectivity layer

The database manual is not only meant for database administrators. Interacting with a database, without knowing how it works, is like driving a racing car without taking any driving lesson. Getting familiar with the SQL standard and the database-specific features can make the difference between a high-performance application and one that barely crawls.

The fear of database portability can lead to avoiding highly effective features just because they are not interchangeable across various database systems. In reality, it is more common to end up with a sluggish database layer than having to port an already running system to a new database solution.

All data access frameworks rely on JDBC (Java Database Connectivity) API for communicating with a database server. JDBC offers many performance optimization techniques, aiming to reduce transaction response time and accommodate more traffic.

The first part of the book is therefore dedicated to JDBC, and it covers topics such as database connection management, statement batching, result set fetching, and database transaction essentials.

1.2 The application data access layer

There are data access patterns that have proven their effectiveness in many enterprise application scenarios. Martin Fowler's Patterns of Enterprise Application Architecture[2] is a must read for every enterprise application developer. Beside the object-relational mapping pattern, most ORM frameworks also employ techniques such as *Unit of Work*, *Identity Map*, *Lazy Loading*, *Embedded Value*, *Entity Inheritance* or *Optimistic and Pessimistic Locking*.

1.2.1 The ORM framework

ORM tools can boost application development speed, but the learning curve is undoubtedly steep. The only way to address the inherent complexity of bridging relational data with the application domain model is to fully understand the ORM framework in use.

Sometimes even the reference documentation might not be enough, and getting familiar with the source code is inevitable when facing performance related problems. JPA (Java Persistence API) excels in writing data because all DML (Data Manipulation Language) statements are automatically updated whenever the persistence model changes, therefore speeding up the iterative development process.

The second part of this book describes various Hibernate-specific optimization techniques like identifier generators, effective entity fetching, and state transitions, application-level transactions and entity caching.

[2] http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420

1.2.2 The native query builder framework

JPA and Hibernate were never meant to substitute SQL[3], and native queries are unavoidable in any non-trivial enterprise application. While JPA makes it possible to abstract DML statements and common entity retrieval queries, when it comes to reading and processing data, nothing can beat native SQL.

JPQL (Java Persistence Querying Language) abstracts the common SQL syntax that is supported by most relation databases. Because of this, JPQL cannot take advantage of Window Functions, Common Table Expressions, Derived tables or PIVOT.

As opposed to JPA, jOOQ (Java Object Oriented Query)[4] offers a type-safe API, which embraces any database-specific querying feature offered by the underlying database system. Just like Criteria API protects against SQL injection attacks when generating entity queries dynamically, jOOQ offers the same safety guarantee when building native SQL statements.

For this reason, the third part of the book is about advanced querying techniques with jOOQ.

About database performance benchmarks

Throughout this book, there are benchmarks aimed to demonstrate the relative gain of a certain performance optimization. The benchmarks results are always dependent on the underlying hardware, operating system and database server configuration, database size and concurrency patterns. For this reason, the absolute values are not as important as the relative optimization gain. In reality, the most relevant benchmark results are the ones against the actual production system.

To prevent the reader from comparing one database against another and drawing a wrong conclusion based on some use case specific benchmarks, the database names are obfuscated as *DB_A*, *DB_B*, *DB_C*, and *DB_D*.

All the source code, for every example that was used in this book is available on GitHub[a].

[3] https://plus.google.com/+GavinKing/posts/LGJU1NorAvY
[4] http://www.jooq.org/

2. Batch Updates

JDBC 2.0 introduced *batch updates* so that multiple DML statements can be grouped into a single database request. Sending multiple statements in a single request reduces the number of database roundtrips, therefore decreasing transaction response time. Even if the reference specification uses the term *updates*, any *insert*, *update* or *delete* statement can be batched, and JDBC supports batching for `java.sql.Statement`, `java.sql.PreparedStatement` and `java.sql.CallableStatement` too.

Not only each database driver is distinct, but even different versions of the same driver might require implementation-specific configurations.

2.1 Batching Statements

For executing static SQL statements, JDBC defines the `Statement` interface, which comes with a batching API as well. Other than for test sake, using a `Statement` for CRUD (Create, Read, Update, Delete), as in the example below, should be avoided for it's prone to SQL injection attacks.

```
statement.addBatch(
    "INSERT INTO post (title, version, id) " +
    "VALUES ('Post no. 1', 0, default)");

statement.addBatch(
    "INSERT INTO post_comment (post_id, review, version, id) " +
    "VALUES (1, 'Post comment 1.1', 0, default)");

int[] updateCounts = statement.executeBatch();
```

The numbers of database rows affected by each statement is included in the return value of the `executeBatch()` method.

Oracle

For `Statement` and `CallableStatement`, the Oracle JDBC Driver[a] does not actually support batching For anything but `PreparedStatement`, the driver ignores batching, and each statement is executed separately.

[a] http://docs.oracle.com/cd/E11882_01/java.112/e16548/oraperf.htm#JJDBC28752

The following graph depicts how different JDBC drivers behave when varying batch size, the test measuring the time it takes to insert 1000 *post* rows with 4 *comments* each:

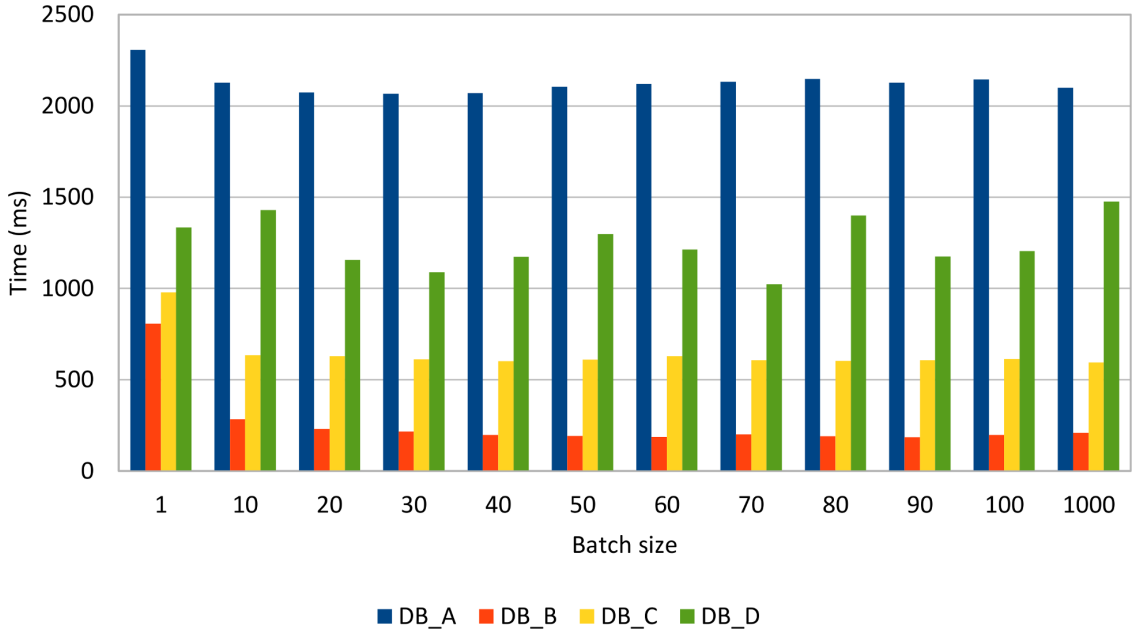


Figure 4.1: Statement batching

Reordering inserts, so that all *posts* are inserted before the *comment* rows, gives the following results:

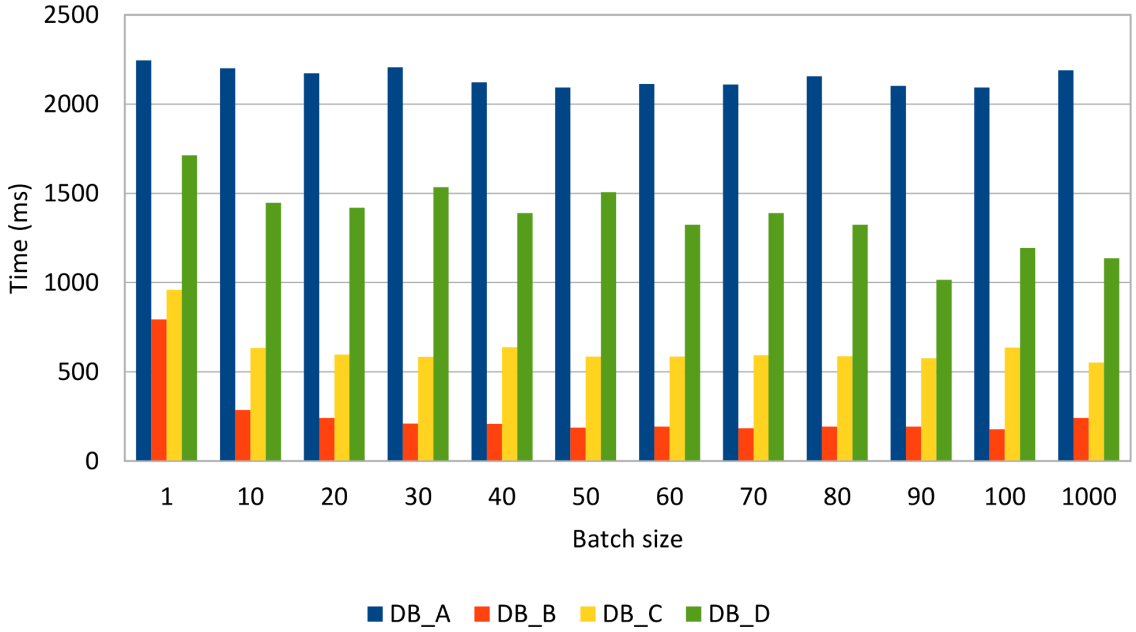


Figure 4.2: Reordered statement batching

Reordering statements does not seem to improve performance noticeably, although some drivers (e.g. MySQL) might take advantage of this optimization.

MySQL

Although it implements the JDBC specification, by default, the MySQL JDBC driver does not send the batched statements in a single request.

For this purpose, the JDBC driver defines the `rewriteBatchedStatements`[a] connection property, so that statements get rewritten into a single `String` buffer. In order to fetch the auto-generated row keys, the batch must contain insert statements only.

For `PreparedStatement`, this property rewrites the batched insert statements into a multi-value insert. Unfortunately, the driver is not able to use server-side prepared statements when enabling rewriting.

Without setting this property, the MySQL driver simply executes each DML statement separately, therefore defeating the purpose of batching.

[a] http://dev.mysql.com/doc/connector-j/en/connector-j-reference-configuration-properties.html

The following graph demonstrates how statement rewriting performs against the default behavior of the MySQL JDBC driver:

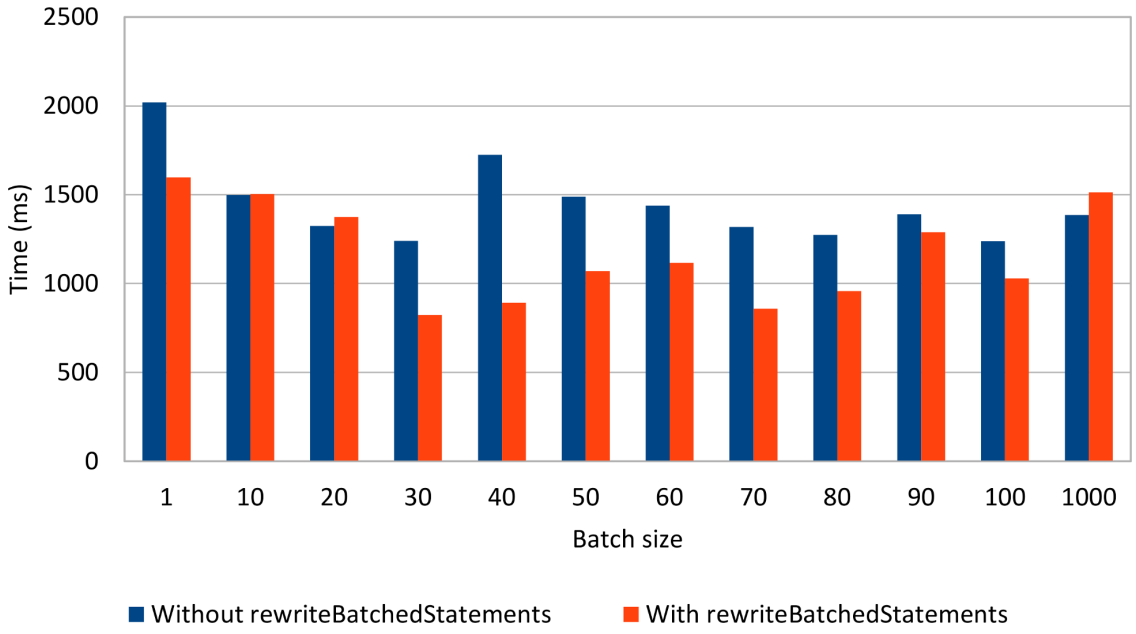


Figure 4.3: MySQL Statement Batching

Rewriting non-parameterized statements seems to make a difference, as long as the batch size is not too large. In practice, it is common to use a relatively small batch size, to reduce both the client-side memory footprint and to avoid congesting the server from suddenly processing a huge batch load.

2.2 Batching PreparedStatements

For parameterized statements (a very common enterprise application requirement), the JDBC `Statement` is a poor fit because the only option for varying the executing SQL statement is through `String` manipulation. Using a `String` template or concatenating `String` tokens is risky as it makes the data access logic vulnerable to SQL injection attacks.

To address this shortcoming, JDBC offers the `PreparedStatement` interface for binding parameters in a safe manner. The driver must validate the provided parameter at runtime, therefore discarding unexpected input values.

Because a `PreparedStatement` is associated with a single DML statement, the batch update can group multiple parameter values belonging to the same prepared statement.

```
PreparedStatement postStatement = connection.prepareStatement(
    "INSERT INTO post (title, version, id) " +
    "VALUES (?, ?, ?)");

postStatement.setString(1, String.format("Post no. %1$d", 1));
postStatement.setInt(2, 0);
postStatement.setLong(3, 1);
postStatement.addBatch();

postStatement.setString(1, String.format("Post no. %1$d", 2));
postStatement.setInt(2, 0);
postStatement.setLong(3, 2);
postStatement.addBatch();

int[] updateCounts = postStatement.executeBatch();
```

SQL injection

For an enterprise application, security is a very important technical requirement. The *SQL Injection* attack exploits data access layers that do not use bind parameters. When the SQL statement is the result of `String` concatenation, an attacker could inject a malicious SQL routine that is sent to the database along the current executing statement.

SQL injection is usually done by ending the current statement with the `;` character and continuing it with a rogue SQL command, like modifying the database structure (deleting a table or modifying authorization rights) or even extracting sensitive information.

All DML statements can benefit from batching as the following tests demonstrate. Just like for the JDBC `Statement` test case, the same amount of data (1000 *post* and 4000 *comments*) is inserted, updated, and deleted while varying the batch size.

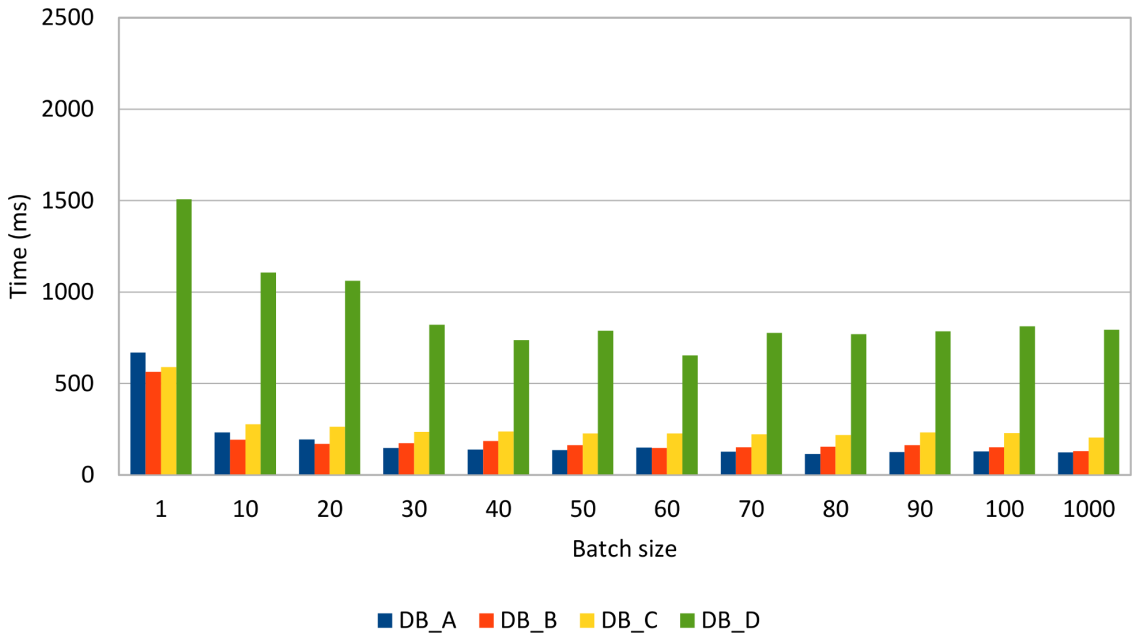


Figure 4.4: Insert PreparedStatement batch size

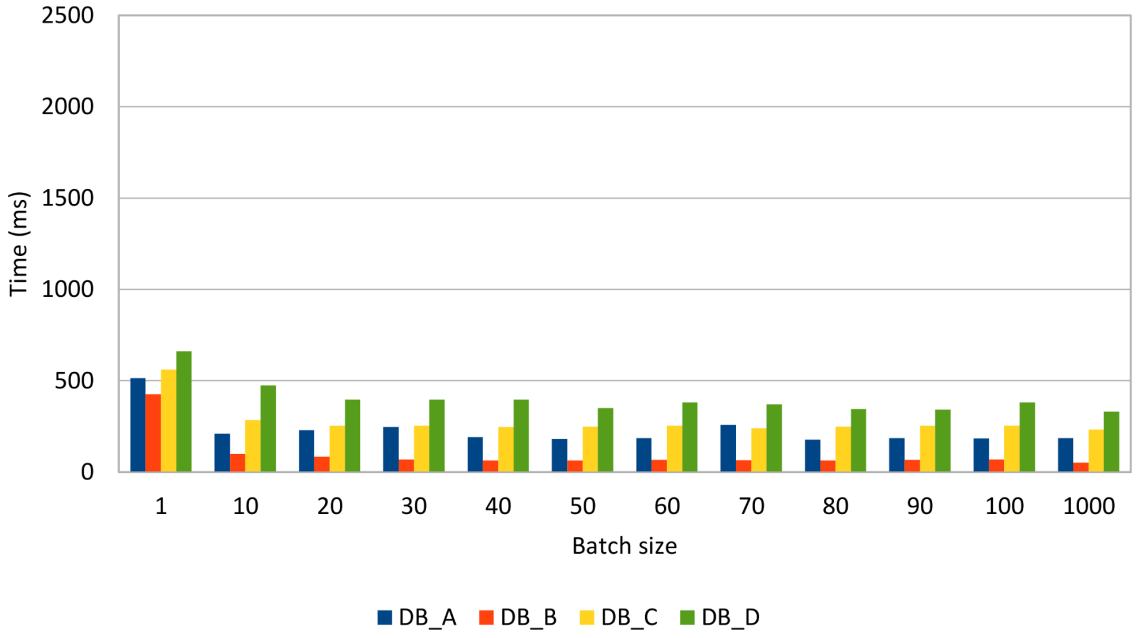


Figure 4.5: Update PreparedStatement batch size

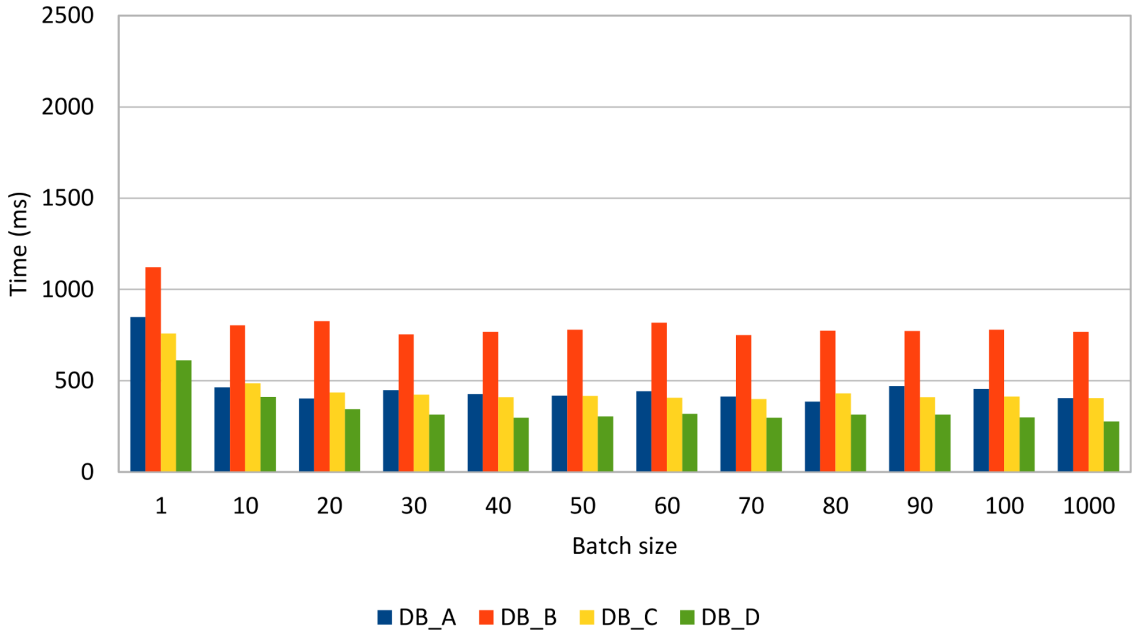


Figure 4.6: Delete PreparedStatement batch size

All database systems show a significant performance improvement when batching prepared statements. Some database systems are very fast when inserting or updating rows, while others perform very well when deleting data.

Compared to the previous `Statement` batch insert results, it is clear that, for the same data load, the `PreparedStatement` use case performs just better. In fact, `Statement(s)` should not be used for batching CRUD operations, being more suitable for bulk processing:

```
DELETE from post
WHERE spam = true AND created_on < current_timestamp - INTERVAL '30' day;
```

2.2.1 Choosing the right batch size

Finding the right batch size is not a trivial thing to do as there is no mathematical equation to solve the appropriate batch size for any enterprise application.

Like any other performance optimization technique, measuring the application performance gain in response to a certain batch size value remains the most reliable tuning option.

The astute reader has already figured out that even a low batch size can reduce the transaction response time, and the performance gain does not grow linearly with batch size. Although a larger batch value can save more database roundtrips, the overall performance gain does not necessarily increase linearly. In fact, a very large batch size can hurt application performance if the transaction takes too long to be executed.

As a rule of thumb, you should always measure the performance improvement for various batch sizes. In practice, a relatively low value (between 10 and 30) is usually a good choice.

2.2.2 Bulk processing

Apart from batching, SQL offers bulk operations to modify all rows that satisfy a given filtering criteria. *Bulk update* or *delete* statements can also benefit from indexing, just like select statements.

To update all records from the previous example, one would have to execute the following statements:

```
UPDATE post SET version = version + 1;
UPDATE post_comment SET version = version + 1;
```

Table 4.1: Bulk update time

DB_A time (ms)	DB_B time (ms)	DB_C time (ms)	DB_D time (ms)
26	13	58	9

The bulk alternative is one order of magnitude faster than batch updates. However, batch updates can benefit from application-level optimistic locking mechanisms, which are suitable for preventing *lost updates* when data is loaded in a read-only database transaction and written back in a successive transaction.

Like with updates, bulk deleting is also much faster than deleting in batches.

```
DELETE FROM post_comment WHERE version > 0;
DELETE FROM post WHERE version > 0;
```

Table 4.2: Bulk delete time

DB_A time (ms)	DB_B time (ms)	DB_C time (ms)	DB_D time (ms)
3	12	1	2

Long-running transaction caveats

Processing too much data in a single transaction can degrade application performance, especially in a highly concurrent environment. Whether if using 2PL (Two-Phase Locking) or MVCC (Multiversion Concurrency Control), writers always block other conflicting writers.

Long running transactions can affect both batch updates and bulk operations if the current transaction modifies a very large number of records. For this reason, it is more practical to break a large batch processing task into smaller manageable ones that can release locks in a timely fashion.

2.3 Retrieving auto-generated keys

It is common practice to delegate the row identifier generation to the database system. This way, the developer does not have to provide a monotonically incrementing primary key since the database takes care of this upon inserting a new record.

As convenient as this practice may be, it is important to know that auto-generated database identifiers might conflict with the batch insert process.

Like many other database features, setting the auto incremented identifier strategy is database-specific so the choice goes between an *identity* column or a database *sequence* generator.

Oracle

Prior to Oracle 12c, an auto incremented generator had to be implemented on top of a database sequence.

```
CREATE SEQUENCE post_seq;

CREATE TABLE post (
    id NUMBER(19,0) NOT NULL,
    title VARCHAR2(255 CHAR),
    version NUMBER(10,0) NOT NULL,
    PRIMARY KEY (id));

CREATE OR REPLACE TRIGGER post_identity
BEFORE INSERT ON post
FOR EACH ROW
BEGIN
    SELECT post_seq.NEXTVAL
    INTO   :NEW.id
    FROM   dual;
end;
```

Oracle 12c adds support for identity columns as well, so the previous example can be simplified as follows.

```
CREATE TABLE post (
    id NUMBER(19,0) NOT NULL GENERATED ALWAYS AS IDENTITY,
    title VARCHAR2(255 CHAR),
    version NUMBER(10,0) NOT NULL,
    PRIMARY KEY (id));
```

SQL Server

Traditionally, SQL Server offered identity column generators, but, since SQL Server 2012, it now supports database sequences as well.

```
CREATE TABLE post (
    id BIGINT IDENTITY NOT NULL,
    title VARCHAR(255),
    version INT NOT NULL,
    PRIMARY KEY (id));
```

PostgreSQL

PostgreSQL 9.5 does not support identity columns natively, although it offers the `SERIAL` column type which can emulate an identity column.

```
CREATE TABLE post (
    id SERIAL NOT NULL,
    title VARCHAR(255),
    version INT4 NOT NULL,
    PRIMARY KEY (id));
```

The `SERIAL` (4 bytes) and `BIGSERIAL` (8 bytes) types are just a syntactic sugar expression as, behind the scenes, PostgreSQL relies on a database sequence anyway.

The previous definition is therefore equivalent to:

```
CREATE SEQUENCE post_id_seq;

CREATE TABLE post (
    id INTEGER DEFAULT NEXTVAL('post_id_seq') NOT NULL,
    title VARCHAR(255),
    version INT4 NOT NULL,
    PRIMARY KEY (id));
);
```

MySQL

MyQL 5.7 only supports identity columns through the `AUTO_INCREMENT` attribute.

```
CREATE TABLE post (
    id BIGINT NOT NULL AUTO_INCREMENT,
    title VARCHAR(255),
    version INTEGER NOT NULL,
    PRIMARY KEY (id));
```

Many database developers like this approach since the client does not have to care about supplying a database identifier upon inserting a new row.

```
INSERT INTO post (title, version) VALUES (?, ?);
```

To retrieve the newly created row identifier, the JDBC `PreparedStatement` must be instructed to return the auto-generated keys.

```
PreparedStatement postStatement = connection.prepareStatement(
    "INSERT INTO post (title, version) VALUES (?, ?)",
    Statement.RETURN_GENERATED_KEYS
);
```

One alternative is to hint the driver about the column index holding the auto-generated key column.

```
PreparedStatement postStatement = connection.prepareStatement(
    "INSERT INTO post (title, version) VALUES (?, ?)",
    new int[] {1}
);
```

The column name can also be used to instruct the driver about the auto-generated key column.

```
PreparedStatement postStatement = connection.prepareStatement(
    "INSERT INTO post (title, version) VALUES (?, ?)",
    new String[] {"id"}
);
```

It is better to know all these three alternatives because they are not interchangeable on all database systems.

Oracle auto-generated key retrieval gotcha

When using `Statement.RETURN_GENERATED_KEYS`, Oracle returns a `ROWID` instead of the actually generated column value. A workaround is to supply the column index or the column name, and so the auto-generated value can be extracted after executing the statement.

According to the JDBC 4.2 specification, every driver must implement the `supportsGetGeneratedKeys()` method and specify whether it supports auto-generated key retrieval. Unfortunately, this only applies to single statement updates as the specification does not make it mandatory for drivers to support generated key retrieval for batch statements. That being said, not all database systems support fetching auto-generated keys from a batch of statements.

Table 4.3: Driver support for retrieving generated keys

Returns generated keys after calling	Oracle JDBC driver (11.2.0.4)	Oracle JDBC driver (12.1.0.1)	SQL Server JDBC driver (4.2)	PostgreSQL JDBC driver (9.4-1201-jdbc41)	MySQL JDBC driver (5.1.36)
`executeUpdate()`	Yes	Yes	Yes	Yes	Yes
`executeBatch()`	No	Yes	No	Yes	Yes

If the Oracle JDBC driver 11.2.0.4 cannot retrieve auto-generated batch keys, the 12.1.0.1 version works just fine. When trying to get the auto-generated batch keys, the SQL Server JDBC driver throws this exception: *The statement must be executed before any results can be obtained.*

2.3.1 Sequences to the rescue

As opposed to identity columns, database sequences offer the advantage of decoupling the identifier generation from the actual row insert. To make use of batch inserts, the identifier must be fetched prior to setting the insert statement parameter values.

```
private long getNextSequenceValue(Connection connection)
    throws SQLException {
    try(Statement statement = connection.createStatement()) {
        try(ResultSet resultSet = statement.executeQuery(
            callSequenceSyntax())) {
            resultSet.next();
            return resultSet.getLong(1);
        }
    }
}
```

For calling a sequence, every database offers a specific syntax:

Oracle

```
SELECT post_seq.NEXTVAL FROM dual;
```

SQL Server

```
SELECT NEXT VALUE FOR post_seq;
```

PostgreSQL

```
SELECT NEXTVAL('post_seq');
```

Because the primary key is generated up-front, there is no need to call the `getGeneratedKeys()` method, and so batch inserts are not driver dependent anymore.

```
try(PreparedStatement postStatement = connection.prepareStatement(
    "INSERT INTO post (id, title, version) VALUES (?, ?, ?)")) {
    for (int i = 0; i < postCount; i++) {
        if(i > 0 && i % batchSize == 0) {
            postStatement.executeBatch();
        }
        postStatement.setLong(1, getNextSequenceValue(connection));
        postStatement.setString(2, String.format("Post no. %1$d", i));
        postStatement.setInt(3, 0);
        postStatement.addBatch();
    }
    postStatement.executeBatch();
}
```

Many database engines use sequence number generation optimizations to lower the sequence call execution as much as possible. If the number of inserted records is relatively low, then the sequence call overhead (extra database roundtrips) will be insignificant. However, for batch processors inserting large amounts of data, the extra sequence calls can add up.

Optimizing sequence calls

The data access layer does not need to go to the database to fetch a unique identifier if the sequence incrementation step is greater than 1. For a step of *N*, the sequence numbers are 1, N + 1, 2N + 1, 3N + 1, etc. The data access logic can assign identifiers in-between the database sequence calls (e.g. 2, 3, 4, …, N -1, N), and so it can mitigate the extra network roundtrips penalty.

This strategy is going to be discussed in greater detail in the Hibernate types and identifiers chapter.

II JPA and Hibernate

3. Why JPA and Hibernate matter

Although JDBC does a very good job of exposing a common API that hides the database vendor-specific communication protocol, it suffers from the following shortcomings:

- The API is undoubtedly verbose, even for trivial tasks.
- Batching is not transparent from the data access layer perspective, requiring a specific API than its non-batched statement counterpart.
- Lack of built-in support for explicit locking and optimistic concurrency control.
- For local transactions, the data access is tangled with transaction management semantics.
- Fetching joined relations requires additional processing to transform the `ResultSet` into Domain Models or DTO (Data Transfer Object) graphs.

Although the primary goal of an ORM (Object-Relational Mapping) tool is to automatically translate object state transitions into SQL statements, this chapter aims to demonstrate that Hibernate can address all the aforementioned JDBC shortcomings.

Java Persistence history

The EJB 1.1 release offered a higher-level persistence abstraction through scalable enterprise components, known as Entity Beans. Although the design looked good on paper, in reality, the heavyweight RMI-based implementation proved to be disastrous from a performance perspective. Neither the EJB 2.0 support for *local interfaces* could revive the Entity Beans popularity, and, due to high-complexity and vendor-specific implementation details, most projects chose JDBC instead.

Hibernate was born out of all the frustration of using the Entity Bean developing model. As an open-source project, Hibernate managed to gain a lot of popularity, and so it soon became the *de facto* Java persistence framework.

In response to all the criticism associated with Entity Bean persistence, the Java Community Process advanced a lightweight POJO-based approach, and the JDO specification was born. Although JDO is data source agnostic, being capable of operating with both relation databases as well as NoSQL or even flat files, it never managed to hit mainstream popularity. For this reason, the Java Community Process decided that EJB3 would be based on a new specification, inspired by Hibernate and TopLink, and JPA (Java Persistence API) became the standard Java enterprise persistence technology.

The morale of this is that persistence is a very complex topic, and it demands a great deal of knowledge of both the database and the data access usage patterns.

3.1 The impedance mismatch

When a relational database is manipulated through an object-oriented program, the two different data representations start conflicting.

In a relational database, data is stored in tables, and the relational algebra defines how data associations are formed. On the other hand, an object-oriented programming (OOP) language allows objects to have both state and behavior, and bidirectional associations are permitted.

The burden of converging these two distinct approaches has generated much tension, and it has been haunting enterprise systems for a very long time.

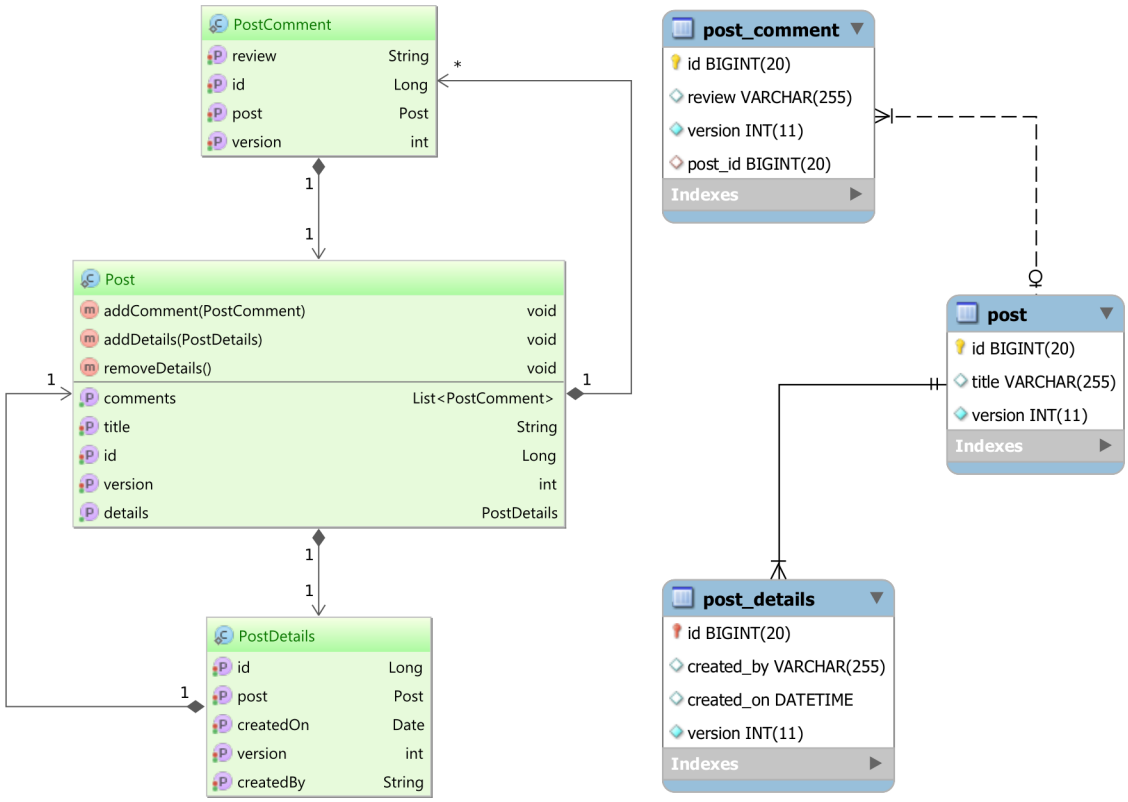


Figure 8.1: Object/Relational Mapping

The above diagram portrays the two different schemas that the data access layer needs to correlate. While the database schema is driven by the SQL standard specification, the Domain Model comes with an object-oriented schema representation as well.

The Domain Model encapsulates the business logic specifications and captures both data structures and the behavior that governs business requirements. OOP facilitates Domain Modeling, and many modern enterprise systems are implemented on top of an object-oriented programming language (e.g. Java, C#).

Because the underlying data resides in a relational database, the Domain Model must be adapted to the database schema and the SQL-driven communication protocol. The ORM design pattern helps to bridge these two different data representations and close the technological gap between them. Every database row is associated with a Domain Model object (*Entity* in JPA terminology), and so the ORM tool can translate the entity state transitions into DML statements.

From an application development point of view, this is very convenient since it is much easier to manipulate Domain Model relationships rather than visualizing the business logic through its underlying SQL statements.

3.2 JPA vs. Hibernate

JPA is only a specification. It describes the interfaces that the client operates with and the standard object-relational mapping metadata (Java annotations or XML descriptors). Beyond the API definition, JPA also explains (although not exhaustively) how these specifications are ought to be implemented by the JPA providers. JPA evolves with the Java EE platform itself (Java EE 6 featuring JPA 2.0 and Java EE 7 introducing JPA 2.1).

Hibernate was already a full-featured Java ORM implementation by the time the JPA specification was released for the first time. Although it implements the JPA specification, Hibernate retains its native API for both backward compatibility and to accommodate non-standard features.

Even if it is best to adhere to the JPA standard, in reality, many JPA providers offer additional features targeting a high-performance data access layer requirements. For this purpose, Hibernate comes with the following non-JPA compliant features:

- extended identifier generators (hi/lo, pooled, pooled-lo)
- transparent prepared statement batching
- customizable CRUD (`@SQLInsert`, `@SQLUpdate`, `@SQLDelete`) statements
- static or dynamic collection filters (e.g. `@FilterDef`, `@Filter`, `@Where`)
- entity filters (e.g. `@Where`)
- mapping properties to SQL fragments (e.g. `@Formula`)
- immutable entities (e.g. `@Immutable`)
- more flush modes (e.g. `FlushMode.MANUAL`, `FlushMode.ALWAYS`)
- querying the second-level cache by the natural key of a given entity
- entity-level cache concurrency strategies (e.g. `Cache(usage = CacheConcurrencyStrategy.READ_WRITE)`)
- versioned bulk updates through HQL
- exclude fields from optimistic locking check (e.g. `@OptimisticLock(excluded = true)`)
- versionless optimistic locking (e.g. `OptimisticLockType.ALL`, `OptimisticLockType.DIRTY`)
- support for skipping (without waiting) pessimistic lock requests
- support for Java 8 Date and Time
- support for multitenancy

If JPA is the interface, Hibernate is one implementation and implementation details always matter from a performance perspective.

The JPA implementation details leak and ignoring them might hinder application performance or even lead to data inconsistency issues. As an example, the following JPA attributes have a peculiar behavior, which can surprise someone who is familiar with the JPA specification only:

- The FlushModeType.AUTO[1] does not trigger a flush for native SQL queries like it does for JPQL or Criteria API.
- The FetchType.EAGER[2] might choose a SQL join or a secondary select whether the entity is fetched directly from the `EntityManager` or through a JPQL (Java Persistence Query Language) or a Criteria API query.

That is why this book is focused on how Hibernate manages to implement both the JPA specification and its non-standard native features (that are relevant from an efficiency perspective).

Portability concerns

Like other non-functional requirements, portability is a feature, and there is still a widespread fear of embracing database-specific or framework-specific features. In reality, it is more common to encounter enterprise applications facing data access performance issues than having to migrate from one technology to the other (be it a relation database or a JPA provider).

The lowest common denominator of many RDBMS is a superset of the SQL-92 standard (although not entirely supported either). SQL-99 supports Common Table Expressions, but MySQL 5.7 does not. Although SQL-2003 defines the `MERGE` operator, PostgreSQL 9.5 favored the `UPSERT` operation instead. By adhering to a SQL-92 syntax, one could achieve a higher degree of database portability, but the price of giving up database-specific features can take a toll on application performance. Portability can be addressed either by subtracting non-common features or through specialization. By offering different implementations, for each supported database system (like the jOOQ framework does), portability can still be achieved.

The same argument is valid for JPA providers too. By layering the application, it is already much easier to swap JPA providers, if there is even a compelling reason for switching one mature JPA implementation with another.

[1] https://docs.oracle.com/javaee/7/api/javax/persistence/FlushModeType.html
[2] https://docs.oracle.com/javaee/7/api/javax/persistence/FetchType.html#EAGER

3.3 Schema ownership

Because of data representation duality, there has been a rivalry between taking ownership of the underlying schema. Although theoretically, both the database and the Domain Model could drive the schema evolution, for practical reasons, the schema belongs to the database.

An enterprise system might be too large to fit into a single application, so it is not uncommon to split in into multiple subsystems, each one serving a specific goal. As an example, there can be front-end web applications, integration web services, email schedulers, full-text search engines and back-end batch processors that need to load data into the system. All these subsystems need to use the underlying database, whether it is for displaying content to the users, or dumping data into the system.

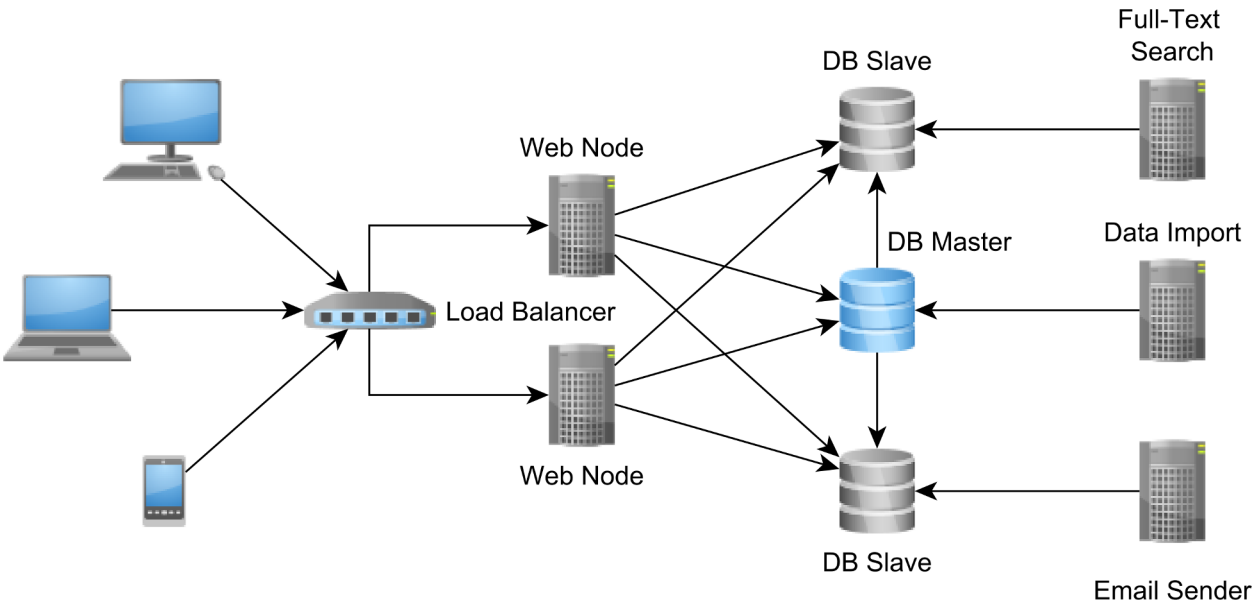


Figure 8.2: Database-centric integration

Although it might not fit any enterprise system, having the database as a central integration point can still be a choice for many reasonable size enterprise systems.

The relational database concurrency models offer strong consistency guarantees, therefore having a significant advantage to application development. If the integration point does not provide transactional semantics, it will be much more difficult to implement a distributed concurrency control mechanism.

Most database systems already offer support for various replication topologies, which can provide more capacity for accommodating an increase in the incoming request traffic. Even if the demand for more data continues to grow, the hardware is always getting better and better (and cheaper too), and database vendors keep on improving their engines to cope with more data.

For these reasons, having the database as an integration point is still a relevant enterprise system design consideration.

The distributed commit log

For very large enterprise systems, where data is split among different providers (relational database systems, caches, Hadoop, Spark), it is no longer possible to rely on the relational database to integrate all disparate subsystems.

In this case, Apache Kafka[a] offers a fault-tolerant and scalable append-only log structure, which every participating subsystem can read and write concurrently.

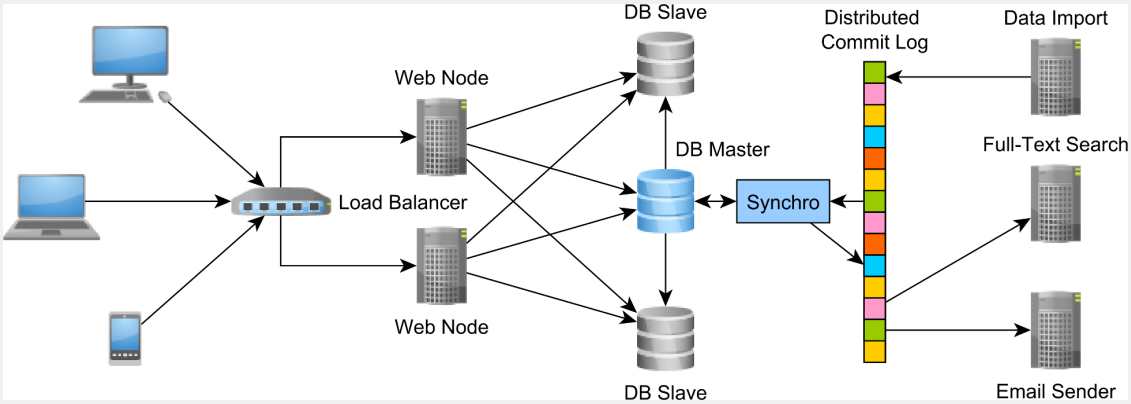


Figure 8.3: Distributed commit log integration

The commit log becomes the integration point, each distributed node individually traversing it and maintaining client-specific pointers in the sequential log structure. This design resembles a database replication mechanism, and so it offers durability (the log is persisted on disk), write performance (append-only logs do not require random access) and read performance (concurrent reads do not require blocking) as well.

[a] http://kafka.apache.org/

No matter what architecture style is chosen, there is still need to correlate the transient Domain Model with the underlying persistent data.

The data schema evolves along the enterprise system itself, and so the two different schema representations must remain congruent at all times.

Even if the data access framework can auto-generate the database schema, the schema must be migrated incrementally, and all changes need to be traceable in the VCS (Version Control System) as well. Along with table structure, indexes and triggers, the database schema is, therefore, accompanying the Domain Model source code itself. A tool like Flywaydb[3] can automate the database schema migration, and the system can be deployed continuously, whether it is a test or a production environment.

[3] http://flywaydb.org/

The schema ownership goes to the database, and the data access layer must assist the Domain Model to communicate with the underlying data.

3.4 Write-based optimizations

JPA shifts the developer mindset from SQL statements to entity state transitions. An entity can be in one of the following states:

Table 8.1: JPA entity states

State	Description
New (Transient)	A newly created entity which is not mapped to any database row is considered to be in the New or Transient state. Once it becomes managed, the Persistence Context issues an insert statement at flush time.
Managed (Persistent)	A Persistent entity is associated with a database row, and it is being managed by the currently running Persistence Context. State changes are detected by the *dirty checking* mechanism and propagated to the database as update statements at flush time.
Detached	Once the currently running Persistence Context is closed, all the previously managed entities become detached. Successive changes are no longer tracked, and no automatic database synchronization is going to happen.
Removed	A removed entity is only scheduled for deletion, and the actual database delete statement is executed during Persistence Context flushing.

The Persistence Context captures entity state changes, and, during flushing, it translates them into SQL statements. The JPA `EntityManager`[4] and the Hibernate `Session`[5] (which includes additional methods for moving an entity from one state to the other) interfaces are gateways towards the underlying Persistence Context, and they define all the entity state transition operations.

[4] http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#persist-java.lang.Object-

[5] https://docs.jboss.org/hibernate/orm/current/javadocs/org/hibernate/Session.html

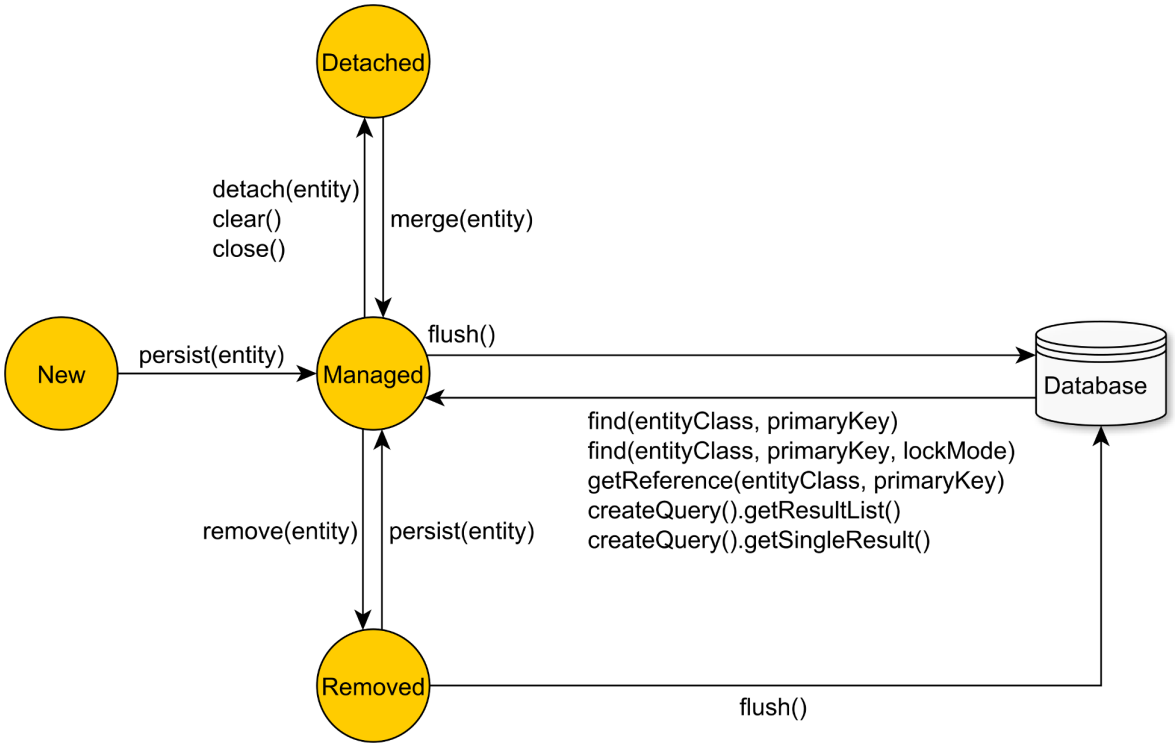


Figure 8.4: JPA entity state transitions

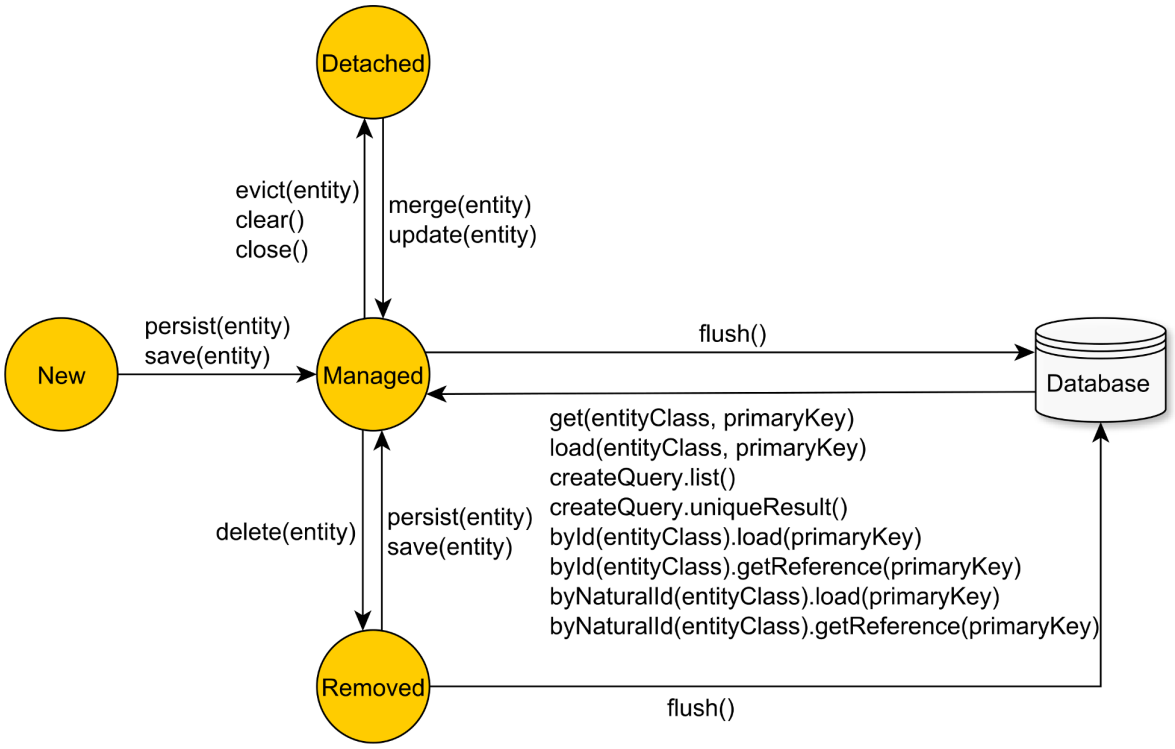


Figure 8.5: Hibernate entity state transitions

SQL injection prevention

By managing the SQL statement generation, the JPA tool can assist in minimizing the risk of SQL injection attacks. The less the chance of manipulating SQL `String` statements, the safer the application can get. The risk is not completely eliminated because the application developer can still recur to concatenating SQL or JPQL fragments, so rigor is advised.

Hibernate uses `PreparedStatement(s)` exclusively, so not only it protects against SQL injection, but the data access layer can better take advantage of server-side and client-side statement caching as well.

Auto-generated DML statements

The enterprise system database schema evolves with time, and the data access layer must mirror all these modifications as well.

Because the JPA provider auto-generates insert and update statements, the data access layer can easily accommodate database table structure modifications. By updating the entity model schema, Hibernate can automatically adjust the modifying statements accordingly.

This applies to changing database column types as well. If the database schema needs to migrate a postal code from an `INT` database type to a `VARCHAR(6)`, the data access layer will need only to change the associated Domain Model property type from an `Integer` to a `String`, and all statements are going to be automatically updated. Hibernate defines a highly customizable JDBC-to-database type mapping system, and the application developer can override a default type association, or even add support for new database types (that are not currently supported by Hibernate).

The entity fetching process is automatically managed by the JPA implementation, which auto-generates the select statements of the associated database tables. This way, JPA can free the application developer from maintaining entity selection queries as well.

Hibernate allows customizing all the CRUD statements, in which case the application developer is responsible for maintaining the associated DML statements.

Although it takes care of the entity selection process, most enterprise systems need to take advantage of the underlying database querying capabilities. For this reason, whenever the database schema changes, all the native SQL queries need to be updated manually (according to their associated business logic requirements).

Write-behind cache

The Persistence Context acts as a transactional write-behind cache, deferring entity state flushing up until the last possible moment.

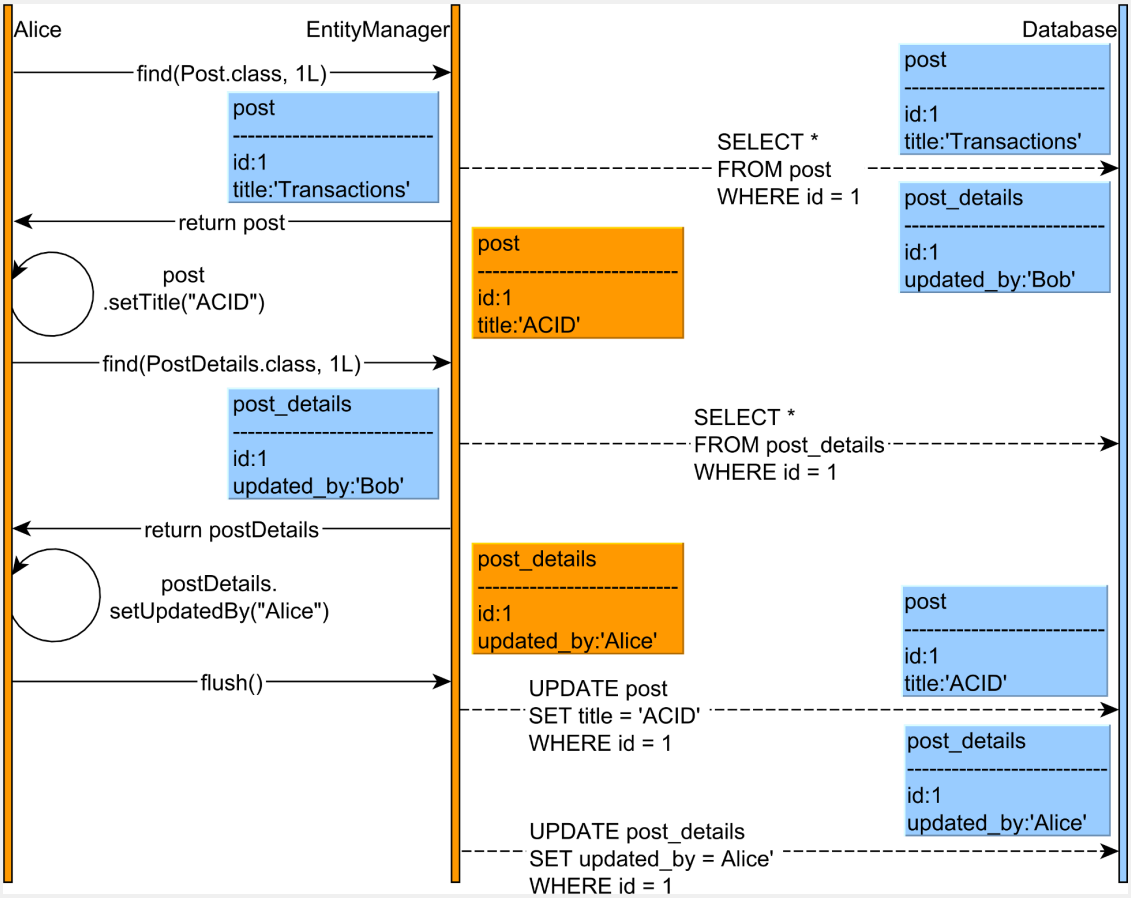


Figure 8.6: Persistence context

Because every modifying DML statement requires locking (to prevent dirty writes), the write behind cache can reduce the database lock acquisition interval, therefore increasing concurrency.

However, caches introduce consistency challenges, and the Persistence Context requires a flush prior to executing any JPQL or native SQL query (as otherwise it might break the read-your-own-write consistency guarantee).

As detailed in the following chapters, Hibernate does not automatically flush pending changes when a native query is about to be executed, and the application developer must explicitly instruct what database tables are needed to be synchronized.

Transparent statement batching

Since all changes are being flushed at once, Hibernate may benefit from batching JDBC statements. Batch updates can be enabled transparently, even after the data access logic has been implemented. Most often, performance tuning is postponed until the system is already running in production, and switching to batching statements should not require a considerable development effort.

With just one configuration, Hibernate can execute all prepared statements in batches.

Application-level concurrency control

As previously explained, no database isolation level can protect against losing updates when executing a multi-request long conversation. JPA supports both optimistic and pessimistic locking.

The JPA optimistic locking mechanism allows preventing lost updates because it imposes a *happens-before* event ordering. However, in multi-request conversations, optimistic locking requires maintaining old entity snapshots, and JPA makes it possible through *Extended Persistence Contexts* or detached entities.

A Java EE application server can preserve a given Persistence Context across several web requests, therefore providing application-level repeatable reads. However, this strategy is not free since the application developer must make sure the Persistence Context is not bloated with too many entities, which, apart from consuming memory, it can also affect the performance of the Hibernate default dirty checking mechanism.

Even when not using Java EE, the same goal can be achieved using detached entities, which provide fine-grained control over the amount of data needed to be preserved from one web request to the other. JPA allows merging detached entities, which rebecome managed and automatically synchronized with the underlying database system.

JPA also supports a pessimistic locking query abstraction, which comes in handy when using lower-level transaction isolation modes.

Hibernate has a native pessimistic locking API, which brings support for timing out lock acquisition requests or skipping already acquired locks.

3.5 Read-based optimizations

Following the SQL standard, the JDBC `ResultSet` is a tabular representation of the underlying fetched data. The Domain Model being constructed as an entity graph, the data access layer must transform the flat `ResultSet` into a hierarchical structure.

Although the goal of the ORM tool is to reduce the gap between the object-oriented Domain Model and its relational counterpart, it is very important to remember that the source of data is not an in-memory repository, and the fetching behavior influences the overall data access efficiency.

The database cannot be abstracted out of this context, and pretending that entities can be manipulated just like any other plain objects is very detrimental to application performance. When it comes to reading data, the impedance mismatch becomes even more apparent, and, for performance reasons, it is mandatory to keep in mind the SQL statements associated with every fetching operation.

In the following example, the *posts* records are fetched along with all their associated *comments*. Using JDBC, this task can be accomplished using the following code snippet:

```
doInJDBC(connection -> {
    try (PreparedStatement statement = connection.prepareStatement(
        "SELECT * " +
        "FROM post AS p " +
        "JOIN post_comment AS pc ON p.id = pc.post_id " +
        "WHERE " +
        "   p.id BETWEEN ? AND ? + 1"
    )) {
        statement.setInt(1, id);
        statement.setInt(2, id);
        try (ResultSet resultSet = statement.executeQuery()) {
            List<Post> posts = toPosts(resultSet);
            assertEquals(expectedCount, posts.size());
        }
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
});
```

When joining *many-to-one* or *one-to-one* associations, each `ResultSet` record corresponds to a pair of entities, so both the parent and the child can be resolved in each iteration. For *one-to-many* or *many-to-many* relationships, because of how the SQL join works, the `ResultSet` contains a duplicated parent record for each associated child.

Constructing the hierarchical entity structure requires manual `ResultSet` transformation, and, to resolve duplicates, the parent entity references are stored in a `Map` structure.

```
List<Post> toPosts(ResultSet resultSet) throws SQLException {
    Map<Long, Post> postMap = new LinkedHashMap<>();
    while (resultSet.next()) {
        Long postId = resultSet.getLong(1);
        Post post = postMap.get(postId);
        if(post == null) {
            post = new Post(postId);
            postMap.put(postId, post);
            post.setTitle(resultSet.getString(2));
            post.setVersion(resultSet.getInt(3));
        }
        PostComment comment = new PostComment();
        comment.setId(resultSet.getLong(4));
        comment.setReview(resultSet.getString(5));
        comment.setVersion(resultSet.getInt(6));
        post.addComment(comment);
    }
    return new ArrayList<>(postMap.values());
}
```

The JDBC 4.2 `PreparedStatement` supports only positional parameters, and the first ordinal starts from 1. JPA allows named parameters as well, which are especially useful when a parameter needs to be referenced multiple times, so the previous example can be rewritten as follows:

```
doInJPA(entityManager -> {
    List<Post> posts = entityManager.createQuery(
        "select distinct p " +
        "from Post p " +
        "join fetch p.comments " +
        "where " +
        "   p.id BETWEEN :id AND :id + 1", Post.class)
    .setParameter("id", id)
    .getResultList();
    assertEquals(expectedCount, posts.size());
});
```

In both examples, the object-relation transformation takes place either implicitly or explicitly. In the JDBC use case, the associations must be manually resolved, while JPA does it automatically (based on the entity schema).

The fetching responsibility

Besides mapping database columns to entity properties, the entity associations can also be represented in terms of object relationships. More, the fetching behavior can be hard-wired to the entity schema itself, which is most often a terrible thing to do.

Fetching multiple one-to-many or many-to-many associations is even more problematic because they might require a Cartesian Product, and performance becomes tied to the children count. Controlling the hard-wired schema fetching policy is cumbersome as it prevents overriding an eager retrieval with a lazy loading mechanism.

Each business use case has different data access requirements, and one policy cannot anticipate all possible use cases, so the fetching strategy should always be set up on a query basis.

Prefer projections for read-only views

Although it is very convenient to fetch entities along with all their associated relationships, it is better to take into consideration the performance impact as well. As previously explained, fetching too much data is not suitable because it increases the transaction response time.

In reality, not all use cases require loading entities, and not all read operations need to be served by the same fetching mechanism. Sometimes a custom projection (selecting only a few columns from an entity) is much more suitable, and the data access logic can even take advantage of database-specific SQL constructs that might not be supported by the JPA query abstraction.

As a rule of thumb, fetching entities is suitable when the logical transaction requires modifying them, even if that only happens in a successive web request. With this is mind, it is much easier to reason on which fetching mechanism to employ for a given business logic use case.

The second-level cache

If the Persistence Context acts as a transactional write-behind cache, its lifetime will be bound to that of a logical transaction. For this reason, the Persistence Context is also known as the first-level cache, and so it cannot be shared by multiple concurrent transactions.

On the other hand, the second-level cache is associated with an `EntityManagerFactory`, and all Persistence Contexts have access to it. The second-level cache can store entities as well as entity associations (one-to-many and many-to-many relationships) and even entity query results. Because JPA does not make it mandatory, each provider takes a different approach to caching (as opposed to EclipseLink, by default, Hibernate disables the second-level cache).

Most often, caching is a trade-off between consistency and performance. Because the cache becomes another source of truth, inconsistencies might occur, and they can be prevented only when all database modifications happen through a single `EntityManagerFactory` or a synchronized distributed caching solution. In reality, this is not practical since the application might be clustered on multiple nodes (each one with its own `EntityManagerFactory`) and the database might be accessed by multiple applications.

Although the second-level cache can mitigate the entity fetching performance issues, it requires a distributed caching implementation, which might not elude the networking penalties anyway.

3.6 Wrap-up

Bridging two highly-specific technologies is always a difficult problem to solve. When the enterprise system is built on top of an object-oriented language, the object-relational impedance mismatch becomes inevitable. The ORM pattern aims to close this gap although it cannot completely abstract it out.

In the end, all the communication flows through JDBC and every execution happens in the database engine itself. A high-performance enterprise application must resonate with the underlying database system, and the ORM tool must not disrupt this relationship.

Just like the problem it tries to solve, Hibernate is a very complex framework with many subtleties that require a thorough knowledge of both database systems, JDBC, and the framework itself. This chapter is only a summary, meant to present JPA and Hibernate into a different perspective that prepares the reader for high-performance object-relational mapping. There is no need to worry if some topics are not entirely clear because the upcoming chapters analyze all these concepts in greater detail.