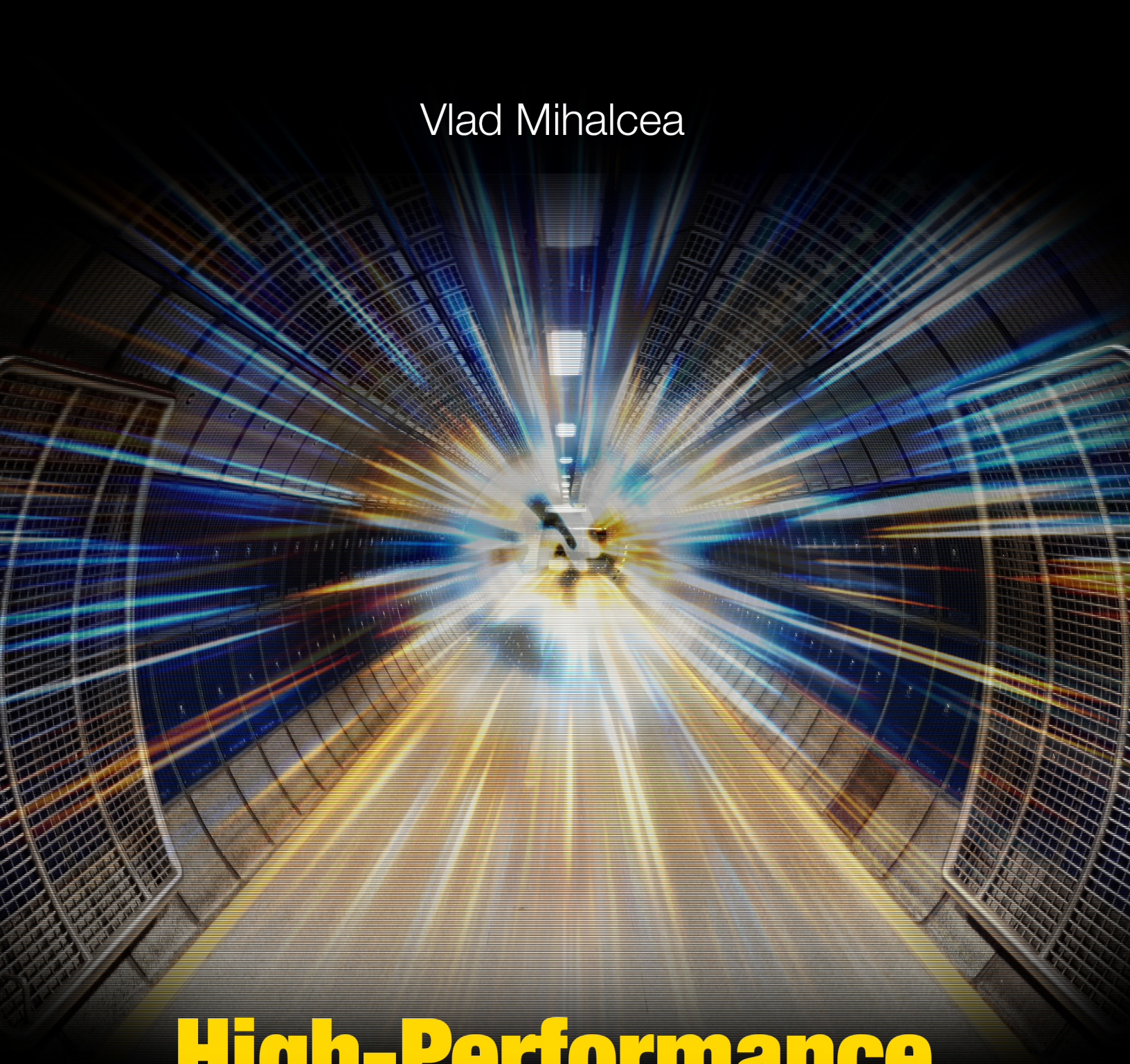
Vlad Mihalcea



High-Performance Java Persistence

Get the most out of your persistence layer

Contents

Publisher:

Vlad Mihalcea

Jupiter 9/27

900492 Cluj-Napoca

Romania

mihalcea.vlad@gmail.com

Cover design:

Dan Mihalcea danmihalcea@gmail.com

Cover photo:

Carlos ZGZ[1] - CC0 1.0[2]

[1] https://www.flickr.com/photos/carloszgz/19980799311/

[2] https://creativecommons.org/publicdomain/zero/1.0/

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.

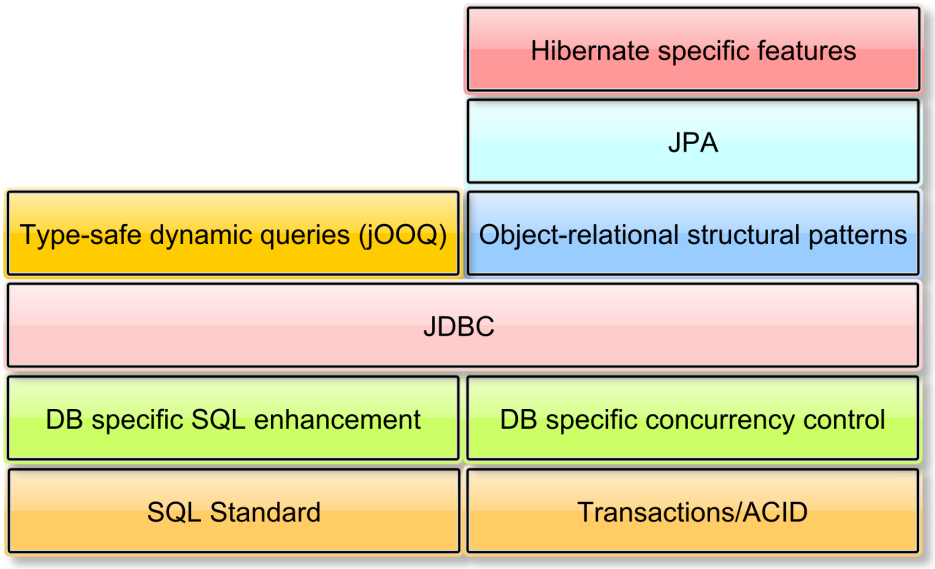


Figure 1.1: Data access skill stack

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's 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's 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 to 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, 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 by subtracting database specific querying features, so it lacks support for Window Functions, Common Table Expressions, Derived tables or PIVOT. As opposed to JPA, jOOQ (Java Object Oriented Query)[3] embraces database specific query features, and it provides a type-safe query builder which can protect the application against SQL injection attacks even for dynamic native queries.

For this reason, the third part of the book is about advance 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 anyway.

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] http://www.jooq.org/

2. Performance and Scaling

An enterprise application needs to store and retrieve as much data and as fast as possible. In application performance management, the two most important metrics are response time and throughput.

The lower the response time, the more responsive an application becomes. Response time is therefore the measure of performance. Scaling is about maintaining low latencies while increasing system load, so throughput is the measure of scalability.

2.1 Response time and throughput

Because this book is focused on high-performance data access, the boundaries of the system under test are located at the transaction manager level. The transaction response time is measured as the time it takes to complete a transaction, and so it encompasses the following time segments:

- the database connection acquisition time
- the time it takes to send all database statements over the wire
- the execution time for all incoming statements
- the time it takes for sending the result sets back to the database client
- the time the transaction is idle due to application-level computations prior to releasing the database connection.

$$T = t_{acq} + t_{req} + t_{exec} + t_{res} + t_{idle}$$

Throughput is defined as the rate of completing incoming load. In a database context, throughput can be calculated as the number of transactions executed within a given time interval.

$$X = \frac{transaction\,count}{time}$$

From this definition, we can conclude that by lowering the time it takes to execute a transaction, the system can accommodate more requests.

Testing against a single database connection, the measured throughput becomes the baseline for further concurrency-based improvements.

$$X\left(N\right) = X\left(1\right) \times C\left(N\right)$$

Ideally, if the system was scaling linearly, adding more database connections would yield a proportional throughput increase. Due to *contention* on database resources and the cost of maintaining *coherency* across multiple concurrent database sessions, the relative throughput gain follows a curve instead of a straight line.

USL (Universal Scalability Law)[1] can approximate the maximum relative throughput (system capacity) in relation to the number of load generators (database connections).

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

- C - the relative throughput gain for the given concurrency level
- α - the contention coefficient (the serializable portion of the data processing routine)
- β - the coherency coefficient (the cost for maintaining consistency across all concurrent database sessions).

When the coherency coefficient is zero, USL overlaps with Amdahl's Law[2]. Contention has the effect of leveling up scalability. On the other hand, coherency is responsible for the inflection point in the scalability curve, and its effect becomes more significant as the number of concurrent sessions increases.

The following graph depicts the relative throughput gain when the USL coefficients (α, β) are set to the following values (0.1, 0.0001). The x-axis represents the number of concurrent sessions (N) and the y-axis shows the relative capacity gain (C).

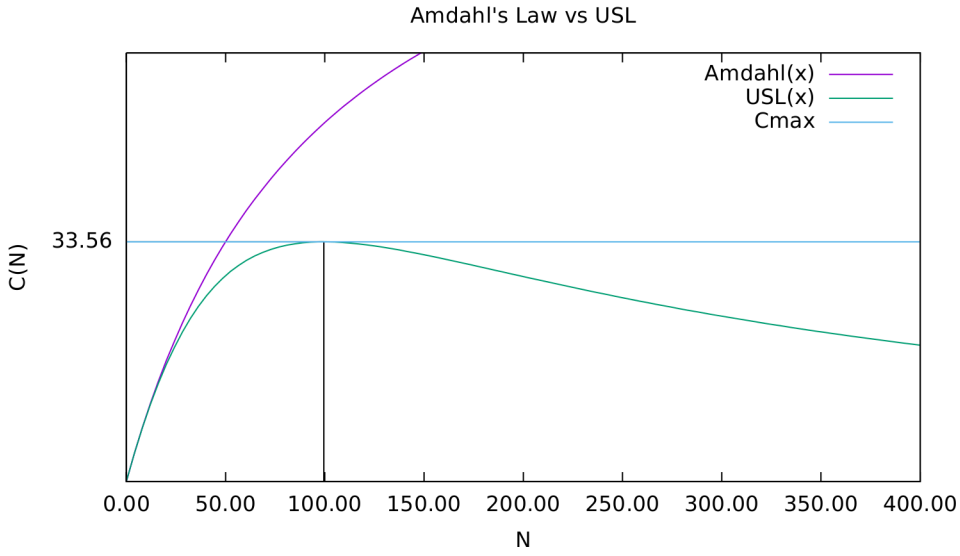


Figure 2.1: Universal Scalability Law

[1] http://www.perfdynamics.com/Manifesto/USLscalability.html

[2] http://en.wikipedia.org/wiki/Amdahl%27s_law

The number of load generators (database connections), for which the system hits its maximum capacity, depends on the USL coefficients solely.

$$Nmax = \sqrt{\frac{(1-\alpha)}{\beta}}$$

The resulting capacity gain is relative to the minimum throughput, so the absolute system capacity is obtained as follows:

$$Xmax = X\left(1\right) \times C\left(Nmax\right)$$

2.2 Database connections boundaries

Every connection requires a TCP socket from the client (application) to the server (database).

The total number of connections offered by a database server depends on the underlying hardware resources, and finding how many connections a server can handle is possible through measurements and proven scalability models.

> SQL Server 2016[a] and MySQL 5.7[b] use thread-based connection handling.
>
> PostgreSQL 9.5[c] uses one operating system process for each individual connection.
>
> On Windows systems, Oracle uses threads, while on Linux, it uses process-based connections. Oracle 12c[d] comes with a thread-based connection model for Linux systems too.
>
> [a] https://msdn.microsoft.com/en-us/library/ms190219.aspx
> [b] https://dev.mysql.com/doc/refman/5.7/en/connection-threads.html
> [c] http://www.postgresql.org/docs/9.5/static/connect-estab.html
> [d] http://docs.oracle.com/database/121/CNCPT/process.htm

A look into database system internals reveals the tight dependency on CPU, Memory and Disk resources. Because I/O operations are costly, the database uses a buffer pool to map into memory the underlying data and index pages. Changes are first applied in memory, and flushed to disk in batches to achieve better write performance.

Even if all indexes are entirely cached in memory, disk access might still occur (to fetch the associated data pages into the memory buffer pool). To provide data consistency, locks (shared and exclusive) are used to protect data blocks (rows and indexes) from being concurrently updated.

Using covering indexes, that fit into memory, can eliminate disk access because the querying data can be fetched without accessing the disk.

This means that high-throughput database applications will experience contention on CPU, Memory, Disk and Locks. When all the database server resources are in use, adding more work load will only increase contention, therefore lowering throughput.

Resources might get saturated due to improper system configuration, so the first step to improving a system throughput is to tune it according to the current data access patterns.

Lowering response time not only makes the application more responsive, but it can also increase throughput.

But response time alone is not sufficient in a highly concurrent environment. To maintain a fixed upper-bound response time, the system capacity must increase, relative to the incoming request throughput. Adding more resources can improve scalability up to a certain point, beyond which the capacity gain starts dropping.

At the Velocity conference[a], both Google Search and Microsoft Bing teams have concluded that higher response times can escalate and even impact the business metrics.

Capacity planning is a feedback-driven mechanism, and it requires constant application monitoring, and so, any optimization must be reinforced by application performance metrics.

[a] http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html

2.3 Scaling up and scaling out

Scaling is the effect of increasing capacity by adding more resources. Scaling vertically (scaling up) means adding resources to a single machine. Increasing the number of available machines is called horizontal scaling (scaling out).

Traditionally, adding more hardware resources to a database server has been the preferred way of increasing database capacity. Relational databases have emerged in the late seventies, and, for two and a half decades, the database vendors took advantage of the hardware advancements following the trends in Moore's Law.

Distributed systems are much more complex to manage than centralized ones, and that's why horizontal scaling is more challenging than scaling vertically. On the other hand, for the same price of a dedicated high-performance server, one could buy multiple commodity machines whose sum of available resources (CPU, Memory, Disk Storage) is greater than of the single dedicated server. When deciding which scaling method is better suited for a given enterprise system, one must take into account both the price (hardware and licenses) and the inherent developing and operational costs.

Being built on top of many open source projects (e.g. PHP, MySQL), Facebook[3] uses a horizontal scaling architecture to accommodate its massive amounts of traffic.

StackOverflow[4] is the best example of a vertical scaling architecture. In one blog post[5], Jeff Atwood explained that the price of Windows and SQL Server licenses was one of the reasons for not choosing a horizontal scaling approach.

No matter how powerful it might be, one dedicated server is still a single point of failure, and throughput drops to zero if the system is no longer available. Database replication is therefore not an option in most enterprise systems.

2.3.1 Master-Slave replication

For enterprise systems where the read/write ratio is high, a Master-Slave replication scheme is suitable for increasing availability.

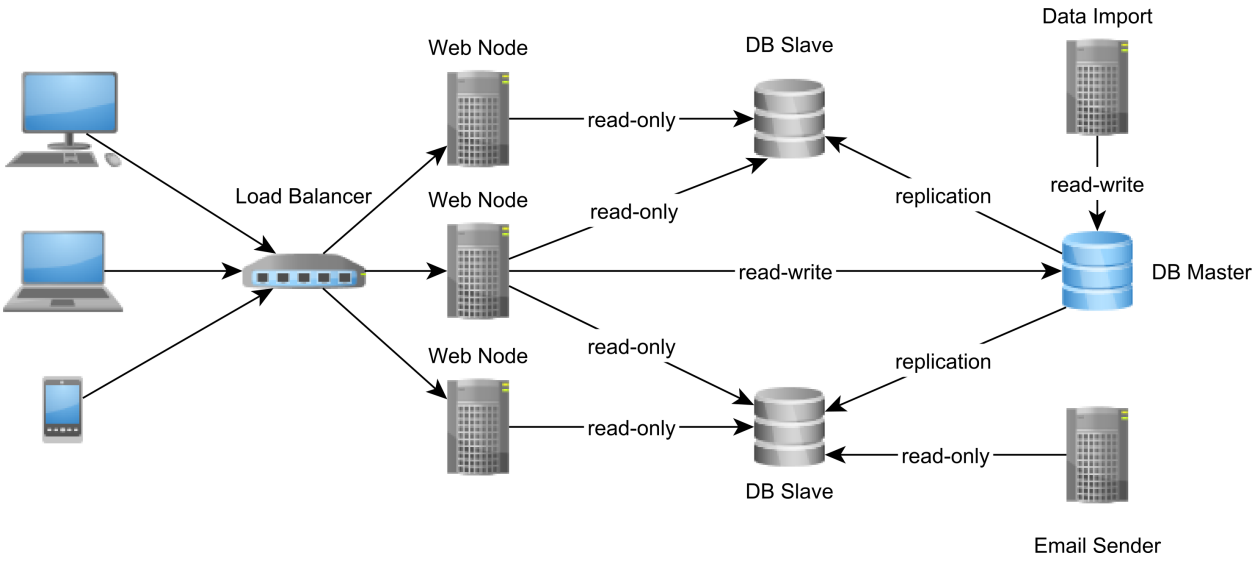


Figure 2.2: Master-Slave replication

The Master is the system of record and the only node accepting writes. All changes recorded by the Master node are replayed onto Slaves as well. A binary replication uses the Master node WAL (Write Ahead Log), while a statement-based replication replays on the Slave machines the exact statements executed on Master.

Asynchronous replication is very common, especially when there are more Slave nodes to update. The Slave nodes are eventual consistent as they might lag behind the Master. In case the Master node crashes, a cluster-wide voting process must elect the new Master (usually the node with the most recent update record) from the list of all available Slaves.

[3] https://www.facebook.com/note.php?note_id=409881258919

[4] http://stackexchange.com/performance

[5] http://blog.codinghorror.com/scaling-up-vs-scaling-out-hidden-costs/

The asynchronous replication topology is also referred as *warm standby* because the election process doesn't happen instantaneously.

Most database systems allow one synchronous Slave node, at the price of increasing transaction response time (the Master has to block waiting for the synchronous Slave node to acknowledge the update). In case of Master node failure, the automatic failover mechanism can promote the synchronous Slave node to become the new Master.

Having one synchronous Slave allows the system to ensure data consistency in case of Master node failures since the synchronous Slave is an exact copy of the Master. The synchronous Master-Slave replication is also called a *hot standby* topology because the synchronous Slave is readily available for replacing the Master node.

When only asynchronous Slave are available, the new elected Slave node might lag behind the failed Master, in which case consistency and durability are traded for lower latencies and higher throughput.

Aside from eliminating the single point of failure, database replication can also increase transaction throughput. In a Master-Slave topology, the Slave nodes can accept read-only transactions, therefore routing read traffic away from the Master node.

The Slave nodes increase the available read-only connections and reduce Master node resource contention, which, in turn, can also lower read-write transaction response time. If the Master node can no longer keep up with the ever increasing read-write traffic, a Multi-Master replication might be a better alternative.

2.3.2 Multi-Master replication

In a Multi-Master replication scheme, all nodes are equal and can accept both read-only and read-write transactions. Splitting the load among multiple nodes can only increase transaction throughput and reduce response time as well.

But because distributed systems are all about trade-offs, ensuring data consistency is challenging in a Multi-Master replication scheme because there is no longer a single source of truth. The same data can be modified concurrently on separate nodes, so there is a possibility of conflicting updates. The replication scheme can either avoid conflicts or it can detect them and apply an automatic conflict resolution algorithm.

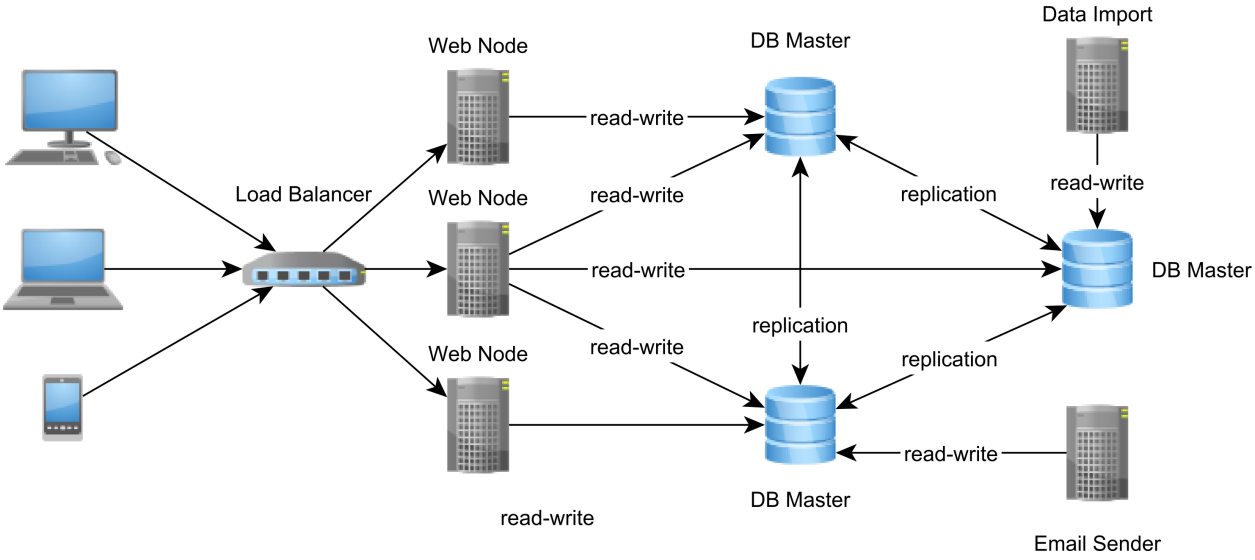


Figure 2.3: Multi-Master replication

To avoid conflicts, the two-phase commit protocol can be used to enlist all participating nodes in one distributed transaction. This design allows all nodes to be in-sync at all time, at the cost of increasing transaction response time (by slowing down write operations).

If nodes are separated by a WAN (Wide Area Network), synchronization latencies can increase dramatically. If one node is no longer reachable, the synchronization could fail, and the transaction would roll back on all Masters.

Although avoiding conflicts is better from a data consistency perspective, synchronous replication might incur high transaction response times. Asynchronous replication can provide better throughput, at the price of having to resolve update conflicts. The asynchronous Multi-Master replication requires a conflict detection and an automatic conflict resolution algorithm. When a conflict is detected, the automatic resolution tries to merge the two conflicting branches, and, in case it fails, manual intervention is required.

2.3.3 Sharding

When data size grows beyond the overall capacity of a replicated multi-node environment, splitting data becomes unavoidable. Sharding means distributing data across multiple nodes so each instance contains only a subset of the overall data.

Traditionally, relational databases have offered *horizontal partitioning* to distribute data across multiple tables within the same database server. As opposed to *horizontal partitioning*, sharding requires a distributed system topology so that data is spread over multiple machines.

Each shard must be self-contained because a user transaction can only use data from within a single shard. Joining across shards is usually prohibited because the cost of distributed locking and the networking overhead would lead to long transaction response times.

By reducing data size per node, indexes will also require less space, and they can better fit into main memory. With less data to query, the transaction response time can also get shorter too.

The typical sharding topology includes at least two separate data centers.

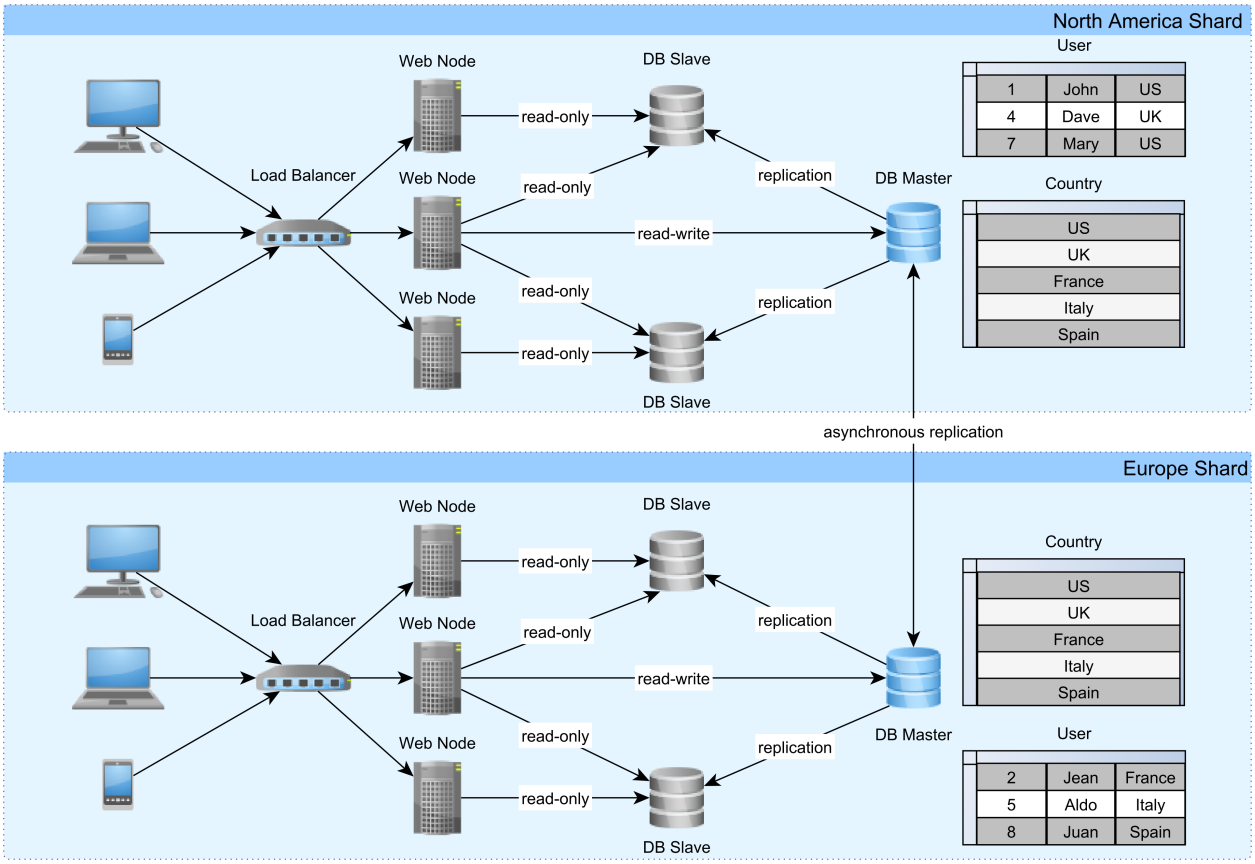


Figure 2.4: Sharding

Each data center can serve a dedicated geographical region, so load is balanced across geographical areas. Not all tables need to be partitioned across shards, smaller size ones being duplicated on each partition. To keep the shards in sync, an asynchronous replication mechanism can be employed.

In the previous diagram, the country table is mirrored from one data center to the other, and partitioning happens on the user table only. To eliminate the need for inter-shard data processing, each user along with all user-related data are contained in one data center only.

In the quest for increasing system capacity, sharding is usually a last resort strategy, employed after exhausting all other available options, such as:

- optimizing the data layer to deliver lower transaction response times
- scaling each replicated node to a cost-effective configuration
- adding more replicated nodes until synchronization latencies start dropping below an acceptable threshold.

MySQL cluster auto-sharding

MySQL Cluster[a] offers automatic sharding, so data is evenly distributed (using a primary key hashing function) over multiple commodity hardware machines. Every node accepts both read and write transactions and, just like Multi-Master replication, conflicts are automatically discovered and resolved.

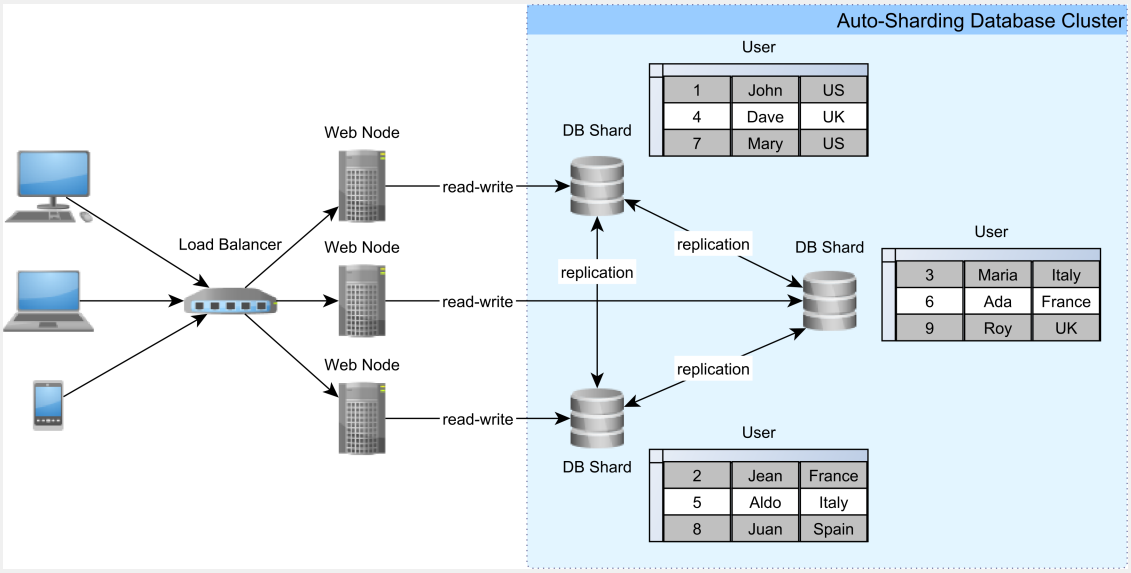


Figure 2.5: Auto-sharding

The auto-sharding topology is similar to the Multi-Master replication architecture as it can increase throughput by distributing incoming load to multiple machines. While in a Multi-Master replicated environment every node stores the whole database, the auto-sharding cluster distributes data so that each shard is only a subset of the whole database.

Because the cluster takes care of distributing data, the application doesn't have to provide a data shard routing layer, and SQL joins are possible even across different shards. MySQL Cluster 7.3 uses the *NDB* storage engine, and so it lacks some features provided by InnoDB[b] like multiple transaction isolation levels or MVCC (Multi Version Concurrency Control).

[a] https://www.mysql.com/products/cluster/scalability.html
[b] http://dev.mysql.com/doc/mysql-cluster-excerpt/5.6/en/mysql-cluster-ndb-innodb-engines.html

II JDBC and Database Essentials

3. JDBC Connection Management

The JDBC (Java Database Connectivity) API provides a common interface for communicating to a database server. All the networking logic and the database specific communication protocol are hidden away behind the vendor-independent JDBC API. For this reason, all the JDBC interfaces must be implemented according to the database vendor specific requirements. The `java.sql.Driver` is the main entry point for interacting with the JDBC API, defining the implementation version details and providing access to a database connection.

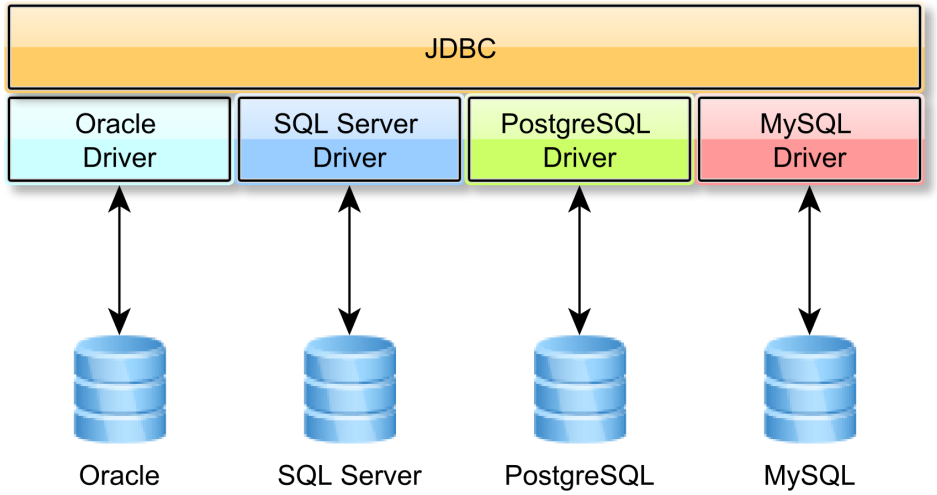


Figure 3.1: JDBC plugin architecture

JDBC defines four driver types:

- Type 1: It's only a bridge to an actual ODBC driver implementation
- Type 2: It uses a database specific native client implementation (e.g. *Oracle Call Interface*)
- Type 3: It delegates calls to an application server offering database connectivity support
- Type 4: The JDBC driver implements the database communication protocol solely in Java.

Being easier to setup and debug, the Type 4 driver is usually the preferred alternative.

To communicate to a database server, a Java program must first obtain a `java.sql.Connection`. Although the `java.sql.Driver` is the actual database connection provider, it's more convenient to use the `java.sql.DriverManager` since it can also resolve the JDBC driver associated with the current database connection URL.

> Previously, the application required to load the driver prior to establishing a connection but, since JDBC 4.0, the *Service Provider Interfaces* mechanism can automatically discover all the available drivers in the current application class-path.

3.1 DriverManager

The `DriverManager` defines the following methods:

```
public static Connection getConnection(
        String url, Properties info) throws SQLException;

public static Connection getConnection(
        String url, String user, String password) throws SQLException;

public static Connection getConnection(
        String url) throws SQLException;
```

Every time the `getConnection()` method is called, the `DriverManager` will request a new *physical* connection from the underlying `Driver`.

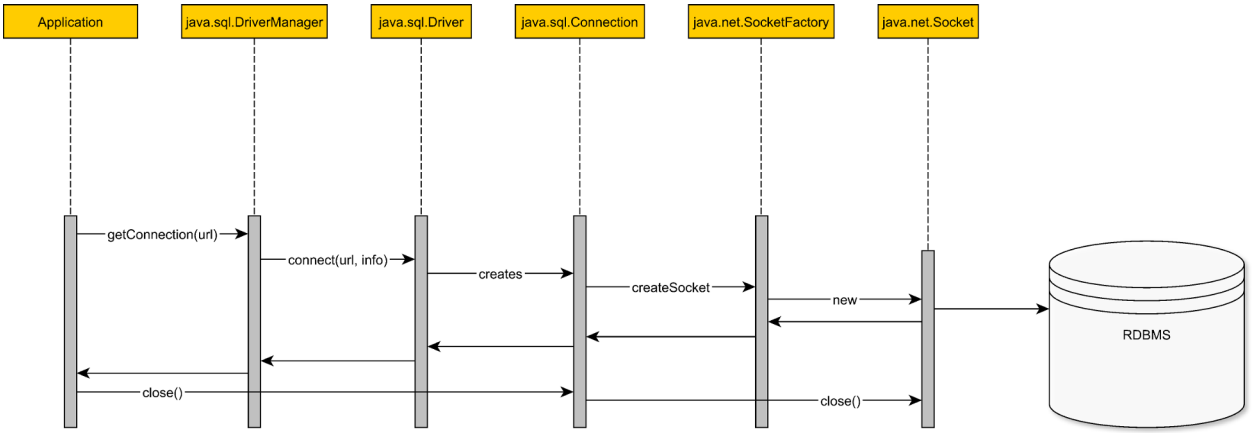


Figure 3.2: DriverManager connection

The first version of JDBC was launched in 1997, and it only supported the `DriverManager` utility for fetching database connections. Back then, Java was offering support for desktop applications which were often employing a two-tier architecture:

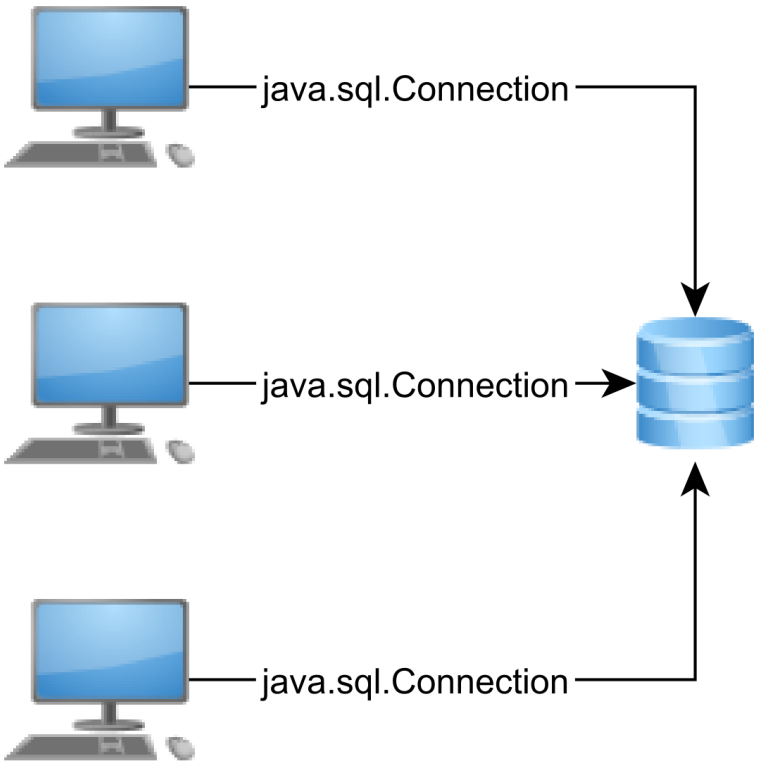


Figure 3.3: Two-tier architecture

In a two-tier architecture, the application is run by single user, and each instance uses a dedicated database connection. The more users, the more database connections are required, and based on the underlying resources (hardware, operating system or licensing restrictions), each database server can offer a limited number of connections.

Oracle mainframe legacy

Oracle has gained its popularity in the era of mainframe computers, when each client got a dedicated database connection.

Oracle assigns a distinct *schema* for each individual *user*, as opposed to other database systems where a schema is shared by multiple user accounts.

In PL/SQL, the *Packaged public variables* scope is bound to a *session*, instead of to the current running transaction. The application developer must be extra cautious to unbind these variables properly since connections are often reused and old values might leak into newer transactions.

3.2 DataSource

In 1999 J2EE was launched along with JDBC 2.0 and an initial draft of JTA (Java Transaction API)[1], marking the beginning of Enterprise Java. Enterprise applications use a three-tier architecture, where the middle tier acts as a bridge between user requests and various data sources (e.g. relational databases, messaging queues).

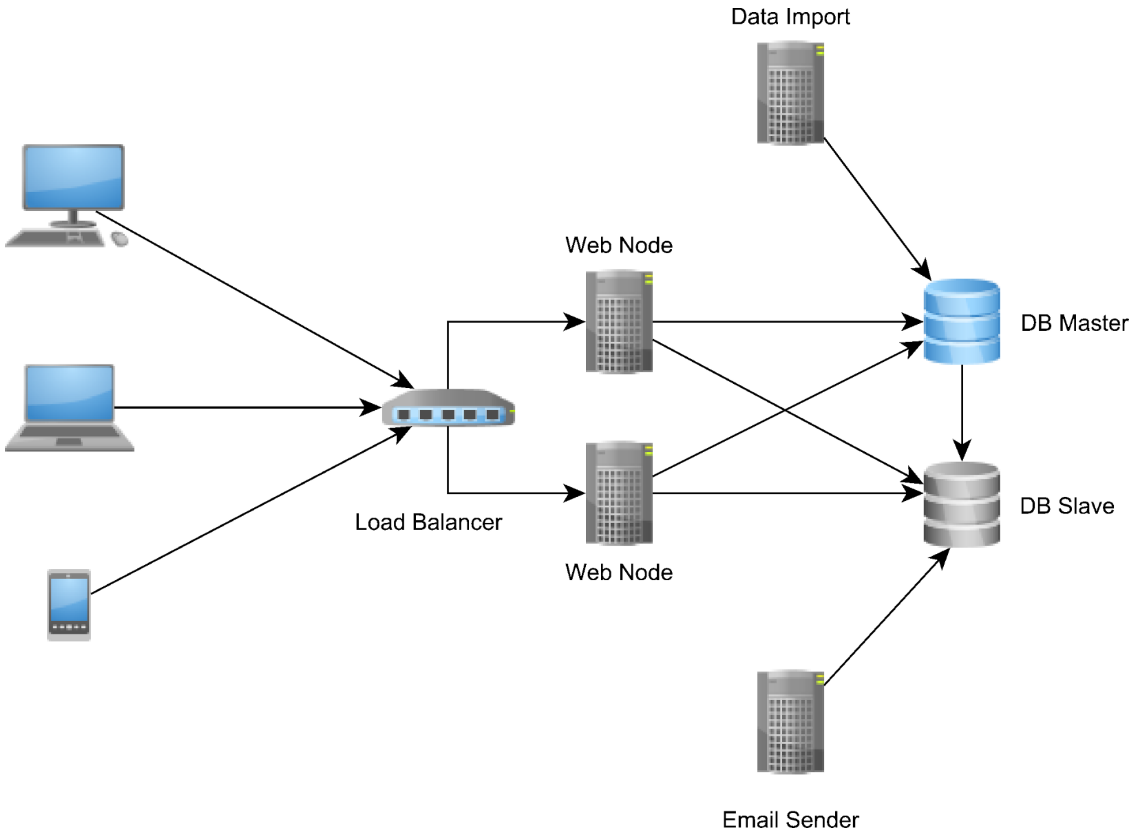


Figure 3.4: Three-tier architecture

Having an intermediate layer between the client and the database server has numerous advantages.

In a typical enterprise application, the user request throughput is greater than the available database connection capacity. As long as the connection acquisition time is tolerable (from the end-user perspective), the user request can wait for a database connection to become available. The middle layer acts as a database connection buffer that can mitigate user request traffic spikes by increasing request response time, without depleting database connections or discarding incoming traffic.

Because the intermediate layer manages database connections, the application server can also monitor connection usage and provide statistics to the operations team.

For this reason, instead of serving physical database connections, the application server provides

[1] https://jcp.org/en/jsr/detail?id=907

only logical connections (proxies or handles), so it can intercept and register how the client API interacts with the connection object.

A three-tier architecture can accommodate multiple data sources or messaging queue implementations. To span a single transaction over multiple sources of data, a distributed transaction manager becomes mandatory. In a JTA environment, the transaction manager must be aware of all logical connections the client has acquired as it has to commit or roll them back according to the global transaction outcome. By providing logical connections, the application server can decorate the database connection handles with JTA transaction semantics.

If the `DriverManager` is a physical connection factory, the `javax.sql.DataSource` interface is a logical connection provider:

```
public Connection getConnection() throws SQLException;

public Connection getConnection(
        String username, String password) throws SQLException;
```

The simplest `javax.sql.DataSource` implementation could delegate connection acquisition requests to the underlying `DriverManager`, and the connection request workflow would look like this:

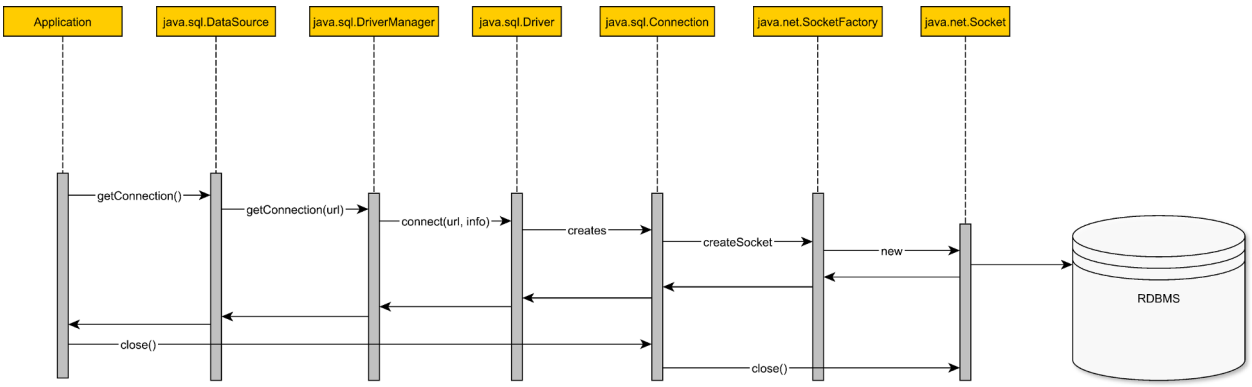


Figure 3.5: DataSource without connection pooling

1. The application data layer asks the `DataSource` for a database connection
2. The `DataSource` will use the underlying driver to open a physical connection
3. A physical connection is created, and a TCP socket is opened
4. The `DataSource` under test doesn't wrap the physical connection, and it simply lends it to the application layer
5. The application executes statements using the acquired database connection
6. When the connection is no longer needed, the application closes the physical connection along with the underlying TCP socket.

Opening and closing database connections is a very expensive operation, so reusing them has the following advantages:

- it avoids both the database and the driver overhead for establishing a TCP connection
- it prevents destroying the temporary memory buffers associated with each database connection
- it reduces client-side JVM object garbage.

To visualize the cumulated overhead of establishing and closing database connections, the following test compares the total time it takes to open and close *1000* database connections of four different RDBMS against HikariCP[2] (one of the fastest stand-alone connection pooling solutions in the Java ecosystem).

Table 3.1: Database connection establishing overhead vs connection pooling

Metric	Time (ms) DB_A	Time (ms) DB_B	Time (ms) DB_C	Time (ms) DB_D	Time (ms) HikariCP
min	11.174	5.441	24.468	0.860	0.001230
max	129.400	26.110	74.634	74.313	1.014051
mean	13.829	6.477	28.910	1.590	0.003458
p99	20.432	9.944	54.952	3.022	0.010263

When using a connection pooling solution, the connection acquisition time is between two and four orders of magnitude smaller. By reducing the connection acquisition interval, the overall transaction response time gets shorter too. All in all, in an enterprise application reusing connections is a much better choice than always establishing them on a transaction basis.

Oracle XE connection handling limitation

While the Enterprise Edition doesn't entail any limitations, the Oracle 11g Express Edition throws the following exception when running very short transactions without using a connection pooling solution:

> ORA-12516, TNS:listener could not find available handler with matching protocol stack

A connection pooling solution can prevent these intermittent connection establishing failures and reduce the connection acquisition time as well.

[2] http://brettwooldridge.github.io/HikariCP/

3.2.1 Why is pooling so much faster?

To understand why the connection pooling solution performs so much better, it's important to figure out the connection pooling mechanism:

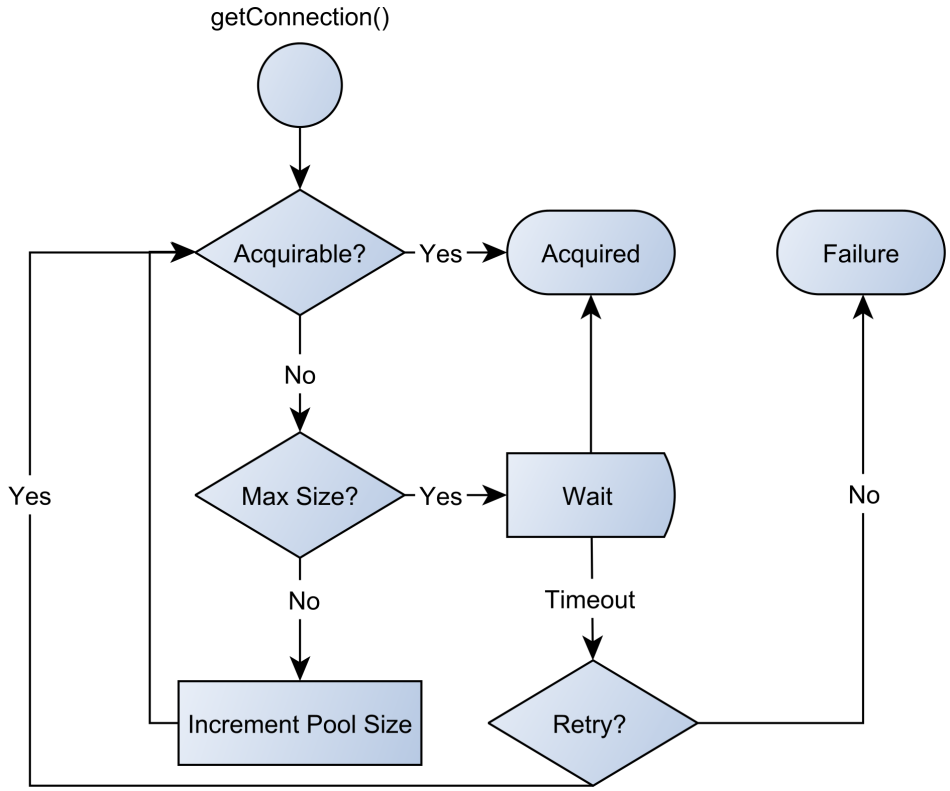


Figure 3.6: Connection acquisition request flow

1. When a connection is being requested, the pool looks for unallocated connections
2. If the pool finds a free one, it handles it to the client
3. If there is no free connection, the pool tries to grow to its maximum allowed size
4. If the pool already reached its maximum size, it will retry several times before giving up with a connection acquisition failure exception
5. When the client closes the logical connection, the connection is released and returns to the pool without closing the underlying physical connection.

Most connection pooling solutions expose a `DataSource` implementation that either wraps an actual database specific `DataSource` or the underlying `DriverManager` utility.

The logical connection lifecycle looks like this:

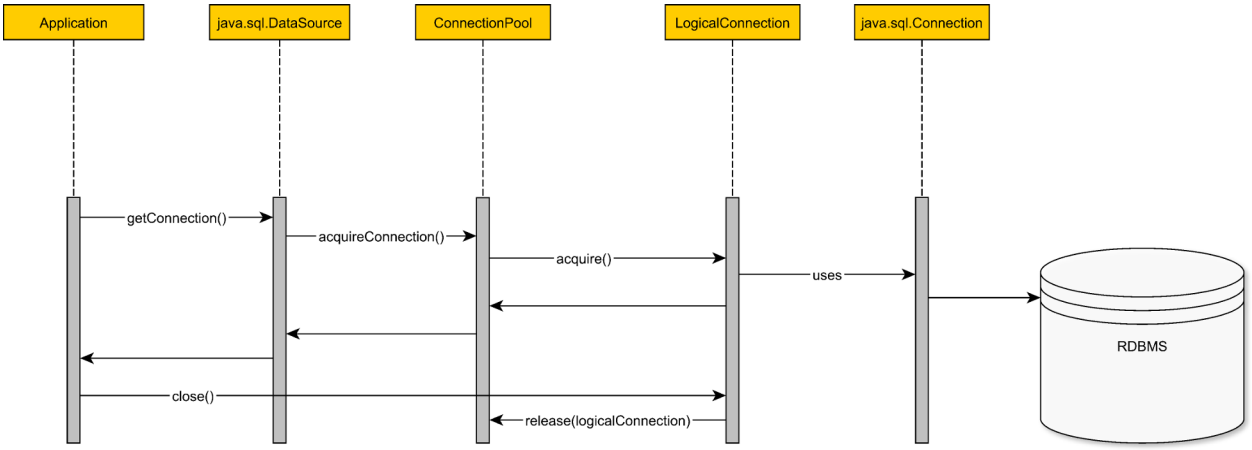


Figure 3.7: DataSource connection

The connection pool doesn't return the physical connection to the client, but instead it offers a proxy or a handle. When a connection is in use, the pool changes its state to *allocated* to prevent two concurrent threads from using the same database connection. The proxy intercepts the connection close method call, and it notifies the pool to change the connection state to *unallocated*.

Apart from reducing connection acquisition time, the pooling mechanism can also limit the number of connections an application can use at once.

The connection pool acts as a bounded buffer for the incoming connection requests. If there is a traffic spike, the connection pool will level it, instead of saturating all the available database resources.

All these benefits come at a price since configuring the right pool size is not a trivial thing to do. Provisioning the connection pool requires understanding the application-specific database access patterns and also connection usage monitoring.

Whenever the number of incoming requests surpasses the available request handlers, there are basically two options to avoid system overloading:

- discarding the overflowing traffic (affecting availability)
- queuing requests and wait for busy resources to become available (increasing response time).

Discarding the surplus traffic is usually a last resort measure, so most connection pooling solutions first attempt to enqueue overflowing incoming requests. By putting an upper bound on the connection request wait time, the queue is prevented from growing indefinitely and saturating application server resources.

For a given incoming request rate, the relation between the queue size and the average enqueuing time is given by one of the most fundamental laws of queuing theory.

3.3 Queuing theory capacity planning

Little's Law[3] is a general-purpose equation applicable to any queueing system being in a stable state (the arrival rate is not greater than the departure rate).

According to Little's Law, the average time for a request to be serviced depends only on the long-term request arrival rate and the average number of requests in the system.

$$L = \lambda \times W$$

- L - average number of requests in the system (including both the requests being serviced and the ones waiting in the queue)
- λ - long-term average arrival rate
- W - average time a request spends in a system.

Assuming that an application-level transaction uses the same database connection throughout its whole lifecycle, and the average transaction response time is *100* milliseconds,

$$W = 100\,ms = 0.1\,s$$

if the average connection acquisition rate is *50* requests per second,

$$\lambda = 50\,\frac{connection\,requests}{s}$$

then the average number of connection requests in the system is:

$$L = \lambda \times W = 50 \times 0.1 = 5\,connection\,requests$$

A pool size of *5* can accommodate the average incoming traffic without having to enqueue any connection request. If the pool size is *3*, then, on average, *2* requests are enqueued and waiting for connections to become available.

Little's Law operates with long-term averages and that might not be suitable when taking into consideration intermittent traffic bursts. In a real-life scenario, the connection pool must adapt to short-term traffic spikes, and so it's important to consider the actual connection pool throughput.

In queueing theory, throughput is represented by the departure rate (μ), and, for a connection pool, it represents the number of connections offered in a given unit of time:

$$\mu = \frac{Ls}{Ws} = \frac{pool\,size}{connection\,lease\,time}$$

[3] http://en.wikipedia.org/wiki/Little%27s_law

The following exercise demonstrates how queuing theory can help provisioning a connection pool to support various incoming traffic spikes.

Reusing the previous example configuration, the connection pool under test is within the following boundaries:

- there are at most *5* in-service requests (*Ls*), meaning that the pool can offer at most *5* connections
- the average service time (*Ws*) or the connection lease time is *100* milliseconds.

As expected, the connection pool can deliver up to *50* connections per second.

$$\mu = \frac{Ls}{Ws} = 50 \frac{connection\,requests}{s}$$

When the arrival rate equals departure rate, the system becomes saturated, all connections being in use.

$$\lambda = \mu = \frac{Ls}{Ws}$$

If the arrival rate outgrows the connection pool throughput, the overflowing requests must wait for connections to become available.

A one second traffic burst of *150* requests is handled as follows:

- the first *50* requests can be served in the first second
- the following *100* requests are first enqueued and processed in the following two seconds.

$$\mu = \frac{Ls}{Ws} = \frac{5}{0.1} = \frac{Lq}{Wq} = \frac{10}{0.2}$$

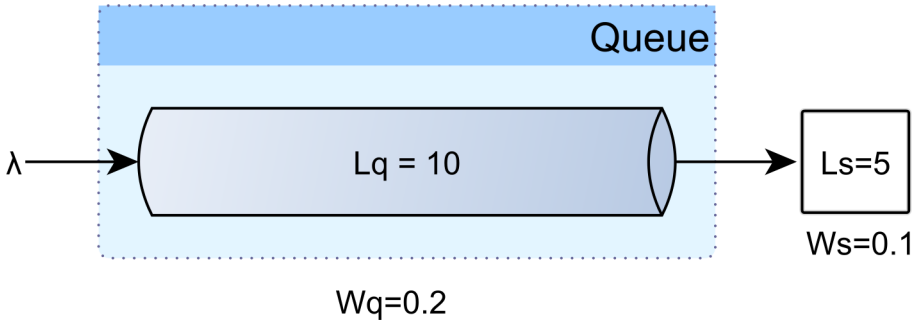


Figure 3.8: Little's Law queue

For a constant throughput, the number of enqueued connection requests (*Lq*) is proportional to the connection acquisition time (*Wq*).

The total number of requests in any given spike is calculated as follows:

$$Lspike = \lambda spike \times W spike$$

The total time required to process the spike is given by the following formula:

$$W = \frac{Lspike}{\mu} = \frac{\lambda spike \times W spike}{\lambda}$$

The number of enqueued connection requests and the time it takes to process them is expressed by the following equations:

$$Lq = Lspike - Ls$$

$$Wq = W - 1$$

Assuming there is a traffic spike of *250* requests per second, lasting for *3* seconds.

$$\lambda spike = 250\,\frac{requests}{s}$$

$$W spike = 3\,s$$

The *750* requests spike takes *15* seconds to be fully processed.

$$Lspike = 250\,\frac{requests}{s} \times 3\,s = 750\,requests$$

$$W = \frac{750\,requests}{50\,\frac{requests}{s}} = 15\,s$$

The queue size grows to *700* entries, and it requires *14* seconds for all connection requests to be serviced.

$$Lq = Lspike - Ls = 700\,requests$$

$$Wq = W - 1 = 14\,s$$

3.4 Practical database connection provisioning

Even if queuing theory provides an insight into the connection pool behavior, the dynamics of enterprise systems are much more difficult to express with general-purpose equations, and metrics become fundamental for resource provisioning. By continuously monitoring the connection usage patterns, it's much easier to react and adjust the pool size when the initial configuration doesn't hold anymore.

Unfortunately, many connection pooling solutions only offer limited support for monitoring and failover strategies, and that was the main reason for building FlexyPool[4]. Supporting the most common connection pooling frameworks, this open source project offers the following connection usage metrics:

Table 3.2: FlexyPool metrics

Name	Description
concurrent connection requests	How many connections are being requested at once
concurrent connections	How many connections are being used at once
maximum pool size	If the target `DataSource` uses adaptive pool sizing, this metric shows how the pool size varies with time
connection acquisition time	The time it takes to acquire a connection from the target `DataSource`
overall connection acquisition time	The total connection acquisition interval (including retries)
retry attempts	The connection acquisition retry attempts
overflow pool size	How much the pool size can grow over the maximum size until timing out the connection acquisition request
connection lease time	The duration between the moment a connection is acquired and the time it gets released

While metrics are important for visualizing connection usage trends, in case of an unforeseen traffic spike, the connection acquisition time could reach the `DataSource` timeout threshold.

The failover mechanism applies various strategies to prevent timed-out connection requests from being discarded. While a batch processor can retry a failing request (although it increases transaction response time), in a web application, the user is much more sensitive to unavailability or long-running transactions.

[4] https://github.com/vladmihalcea/flexy-pool

FlexyPool comes with the following default failover strategies:

Table 3.3: FlexyPool failover strategies

Name	Description
Increment pool size on timeout	The connection pool has a *minimum size* and, on demand, it can grow up to its *maximum size*. This strategy will increment the target connection pool maximum size on connection acquisition timeout. The *overflow* is a buffer of extra connections allowing the pool to grow beyond its initial *maximum size*, until it reaches the *overflow size* threshold
Retrying attempts	This strategy is useful for those connection pools lacking a connection acquiring retry mechanism, and it simply reattempts to fetch a connection for a given number of tries

3.4.1 A real-life connection pool monitoring example

The following example demonstrates how FlexyPool failover strategies can determine the right connection pool size. The application under test is a batch processor using Bitronix transaction manager[5] as the database connection pooling provider.

The batch processor is given a certain data load, and the pool size automatically grows upon detecting a connection acquisition timeout occurrence. The average and the maximum pool size are determined experimentally, without the need of any prior mathematical calculations.

Prior to running the load testing experiment, it's better to know the current application connection pool settings. According to the Bitronix connection pool documentation[6] the default `acquisitionTimeout` (the maximum time a connection request waits before throwing a timeout exception) is *30* seconds.

A connection acquisition timeout threshold of one second is sufficient for the current experiment, allowing the application to react more quickly to a traffic spike and apply a compensating failover strategy.

The initial `maxPoolSize` is set to one connection, and, upon receiving a connection acquisition timeout, it will grow until the `maxOverflow` threshold is reached.

The `retryAttempts` value is intentionally set to a reasonably large value because, for a batch processor, dropping a connection request is a much bigger problem than some occasional transaction response time spikes.

[5] https://github.com/bitronix/btm

[6] https://github.com/bitronix/btm/wiki/JDBC-pools-configuration

The experiment starts with the following initial connection pool settings:

Table 3.4: Initial connection pool settings

Name	Value	Description
minPoolSize	0	The pool starts with an initial size of 0
maxPoolSize	1	The pool starts with a maximum size of 1
acquisitionTimeout	1	A connection request will wait for 1s before giving up with a timeout exception
maxOverflow	4	The pool can grow up to 5 connections (initial `maxPoolSize` + `maxOverflow`)
retryAttempts	30	If the final maxPoolSize is reached, and there is no connection available, a request will retry 30 times before giving up.

3.4.1.1 Concurrent connection request count metric

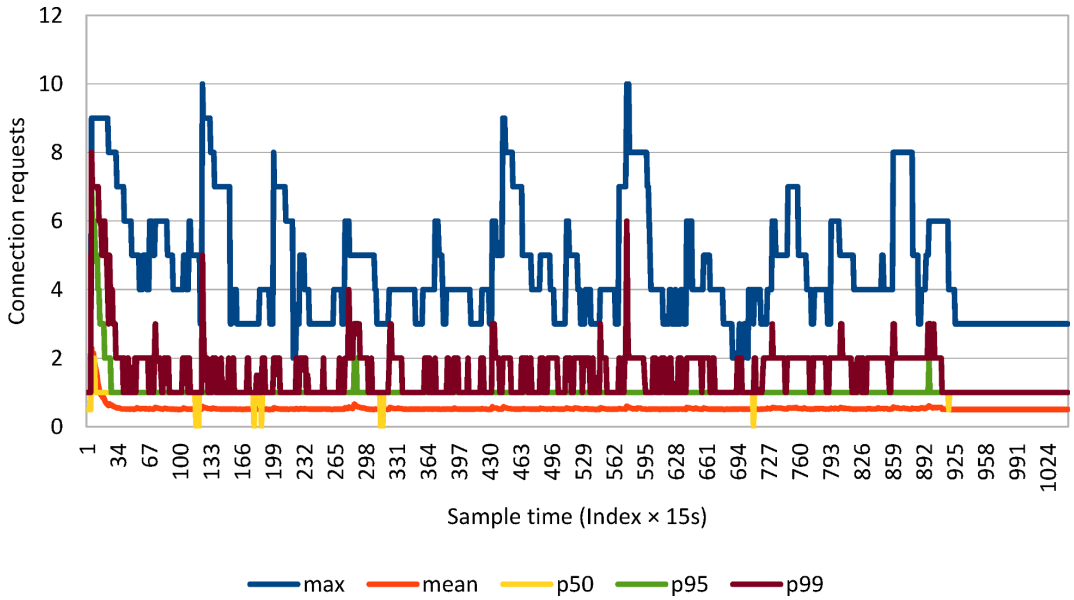


Figure 3.9: Concurrent connection requests

The more incoming concurrent connection requests, the higher the response time (for obtaining a pooled connection) gets. This graph shows the incoming request distribution, making it ideal for spotting traffic spikes.

The average value levels up all outliers, so it cannot reflect the application response to a given traffic spike.

When the recorded values fluctuate dramatically, the average and the maximum value alone offer only a limited view over the actual range of data, and that's why percentiles are preferred in application performance monitoring.

By offering the maximum value, relevant to only a percentage of the whole population, percentiles make outliers visible while capturing the immediate effect of a given traffic change.

3.4.1.2 Concurrent connection count metric

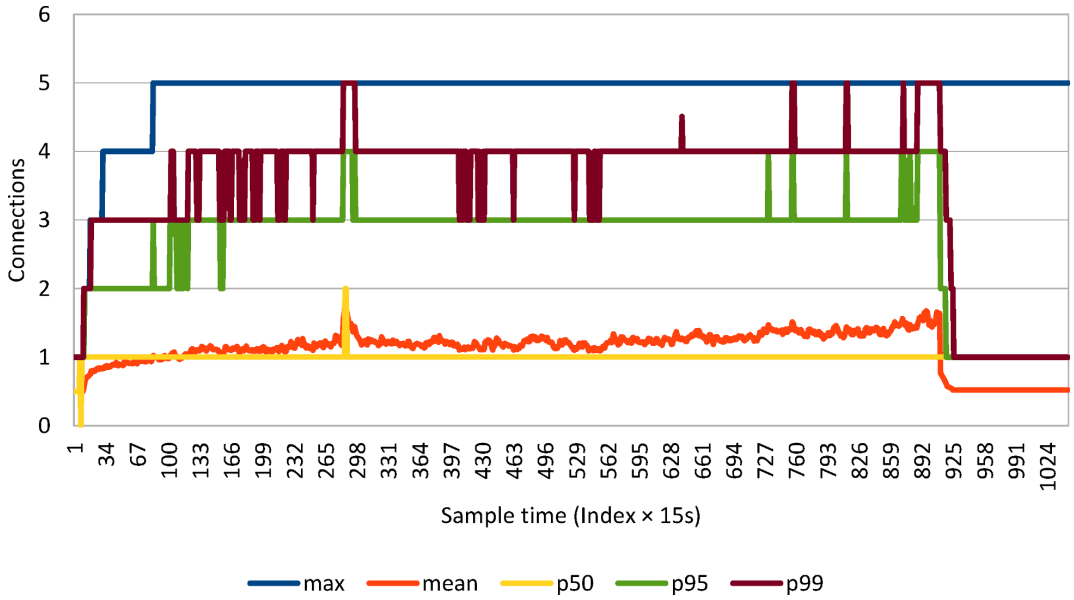


Figure 3.10: Concurrent connections

The average concurrent connection metric follows a gradual slope up to *1.5* connections. Unfortunately, this value is of little use for configuring the right pool size. On the other hand, the 99th percentile is much more informative, showing that *3* to *5* connections are sufficient. The maximum connections graph also confirms that the pool size should be limited to *5* connections (in case the connection acquisition time is acceptable).

If the connection pool supports it, it's very important to set the idle connection timeout threshold. This way, the pool can release unused connections, so the database can provide them to other clients as well.

3.4.1.3 Maximum pool size metric

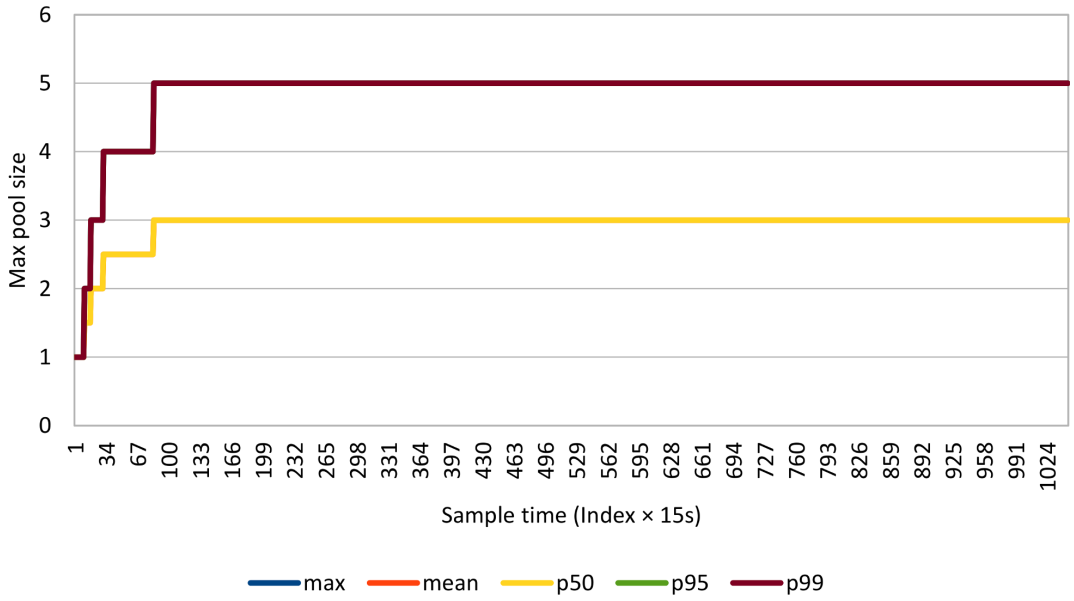


Figure 3.11: Maximum pool size

According to the 99th percentile, the pool gets saturated soon after the job process starts.

3.4.1.4 Connection acquisition time metric

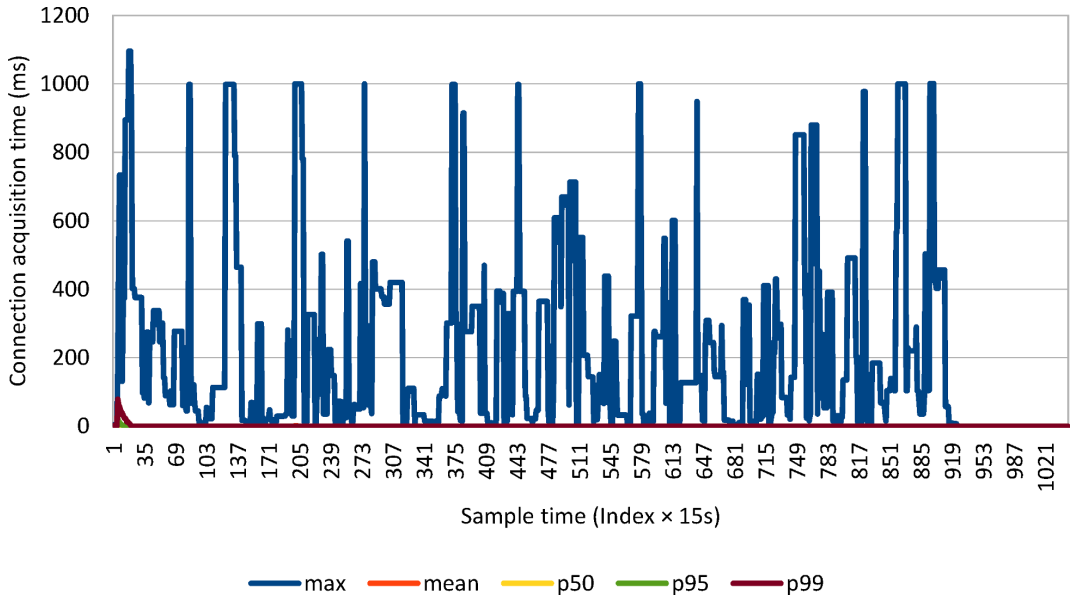


Figure 3.12: Connection acquisition time

The traffic spikes are captured by the maximum graph only. The timeout threshold is hit multiple times as the pool either grows its size or it retries the connection acquisition request.

3.4.1.5 Retry attempts metric

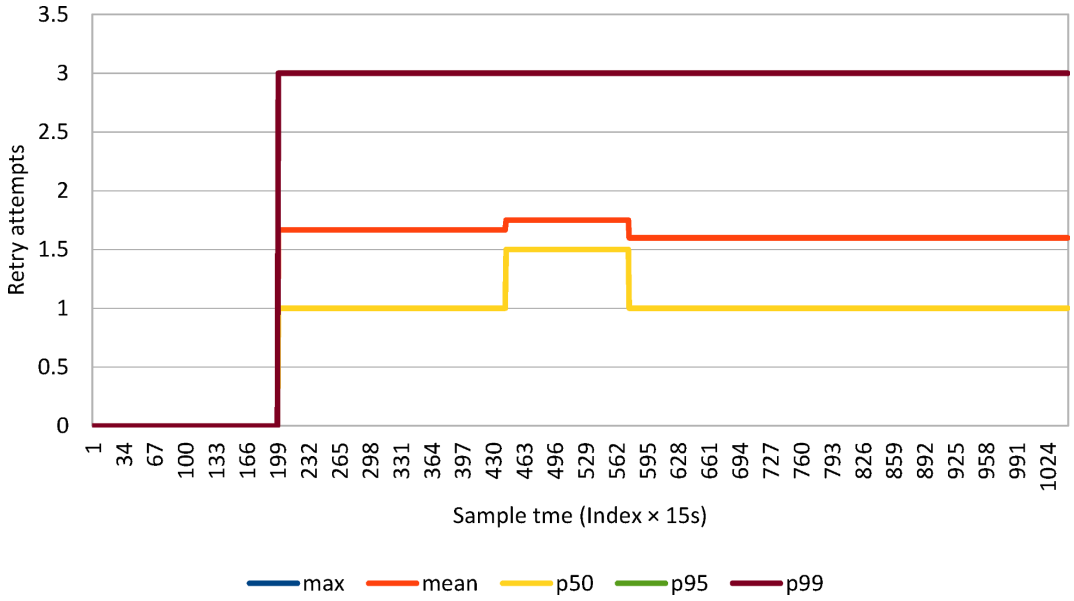


Figure 3.13: Retry attempts

When limiting the connection pool to *5* connections, there are only three retry attempts.

3.4.1.6 Overall connection acquisition time metric

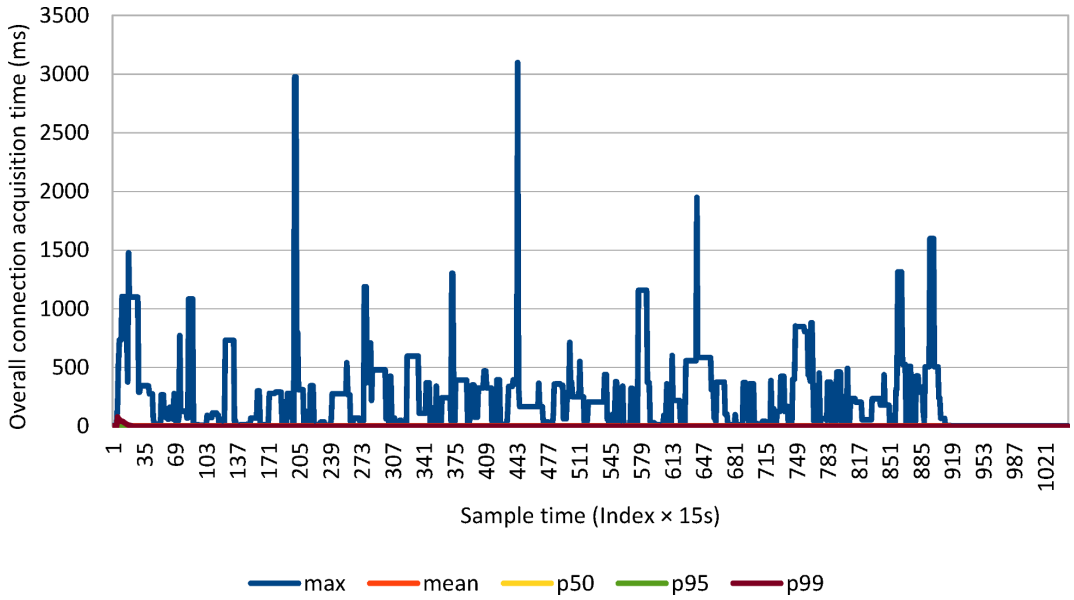


Figure 3.14: Overall connection acquisition Time

While the retry attempts graph only shows how the retry count increases with time, the actual effect of reattempting is visible in the overall connection acquisition time.

3.4.1.7 Connection lease time metric

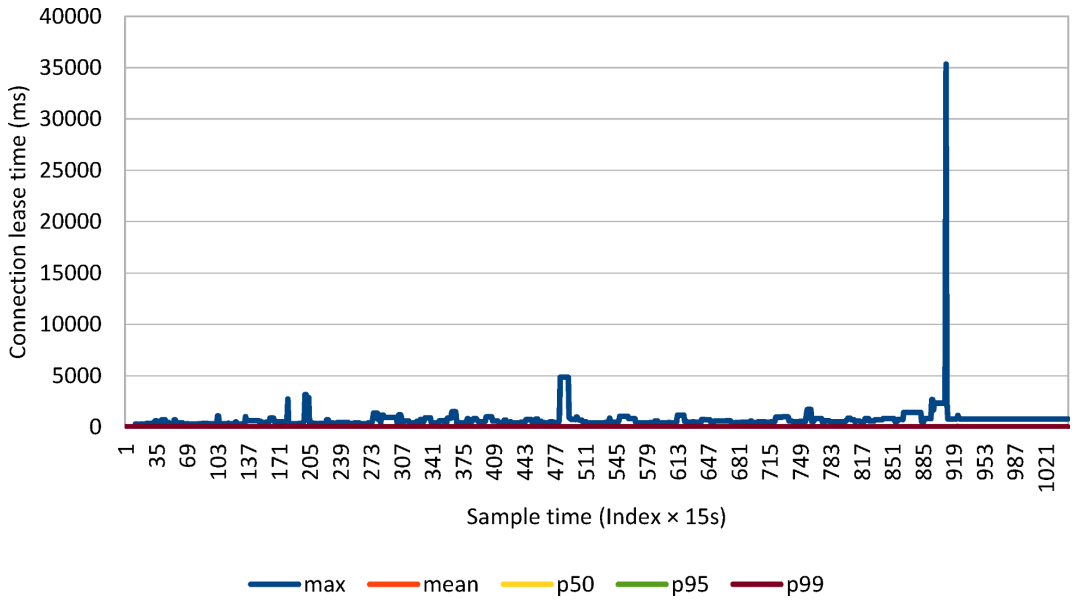


Figure 3.15: Connection lease time

The 99th percentile indicates a rather stable connection lease time throughout the whole job execution. On the other hand, the maximum graph shows a long-running transaction lasting over *35* seconds.

> Holding connections for a long periods of time can increase the connection acquisition time, and fewer resources will be available to other incoming clients.
>
> Most often, connections are leased for the whole duration of a database transaction. Long-running transactions might hold database locks, which, in turn, might lead to increasing the serial portion of the current execution context, therefore hindering parallelism.
>
> Long-running transactions can be addressed by properly indexing slow queries or by splitting the application-level transaction over multiple database transactions, like it's the case in many ETL (Extract, Transform and Load) systems.

4. 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 used 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.

4.1 Batching Statements

For executing static SQL statements, JDBC defines the `Statement` interface and batching multiple DML statements is as straightforward as the following code snippet:

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

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

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] doesn't 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:

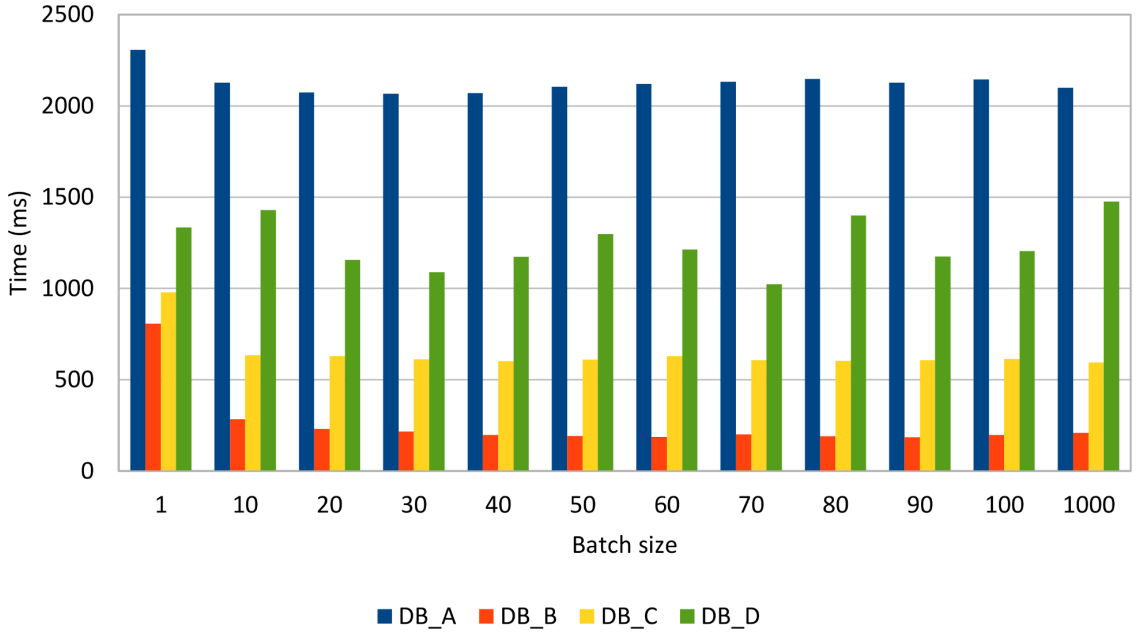


Figure 4.1: Statement batching

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

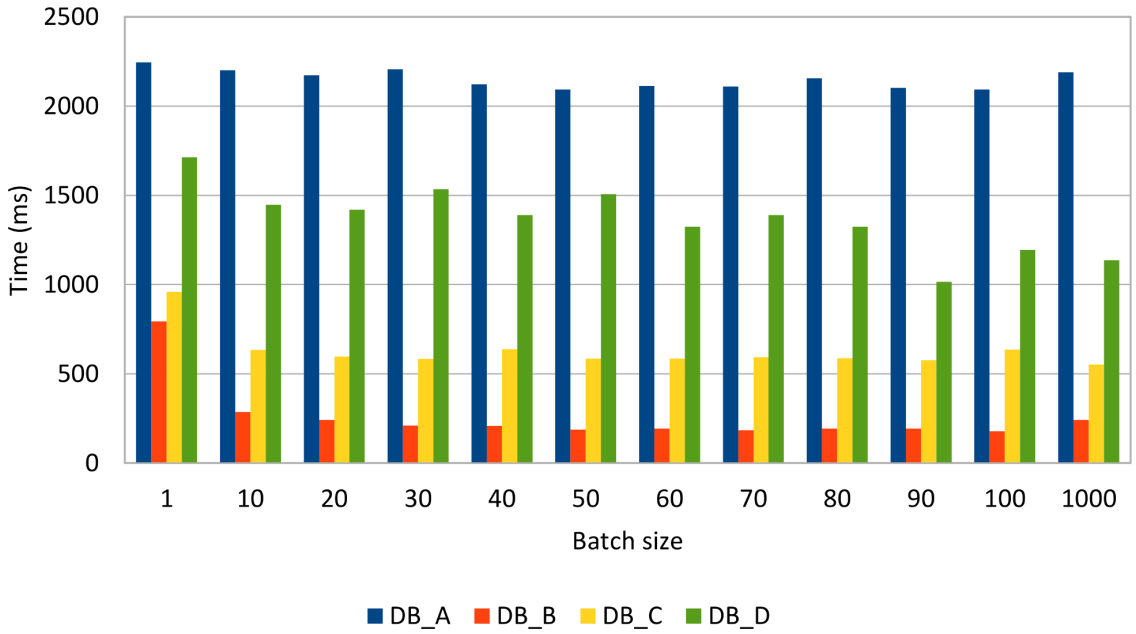


Figure 4.2: Reordered statement batching

Reordering statements doesn't seem to noticeably improve performance, 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 doesn't 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 will rewrite the batched insert statements into a multi-value insert. Unfortunately, the driver won't be able to use server-side prepared statements when enabling rewriting.

Without setting this property, the MySQL driver will simply execute 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 demonstrate how statement rewriting performs against the default behavior of the MySQL JDBC driver:

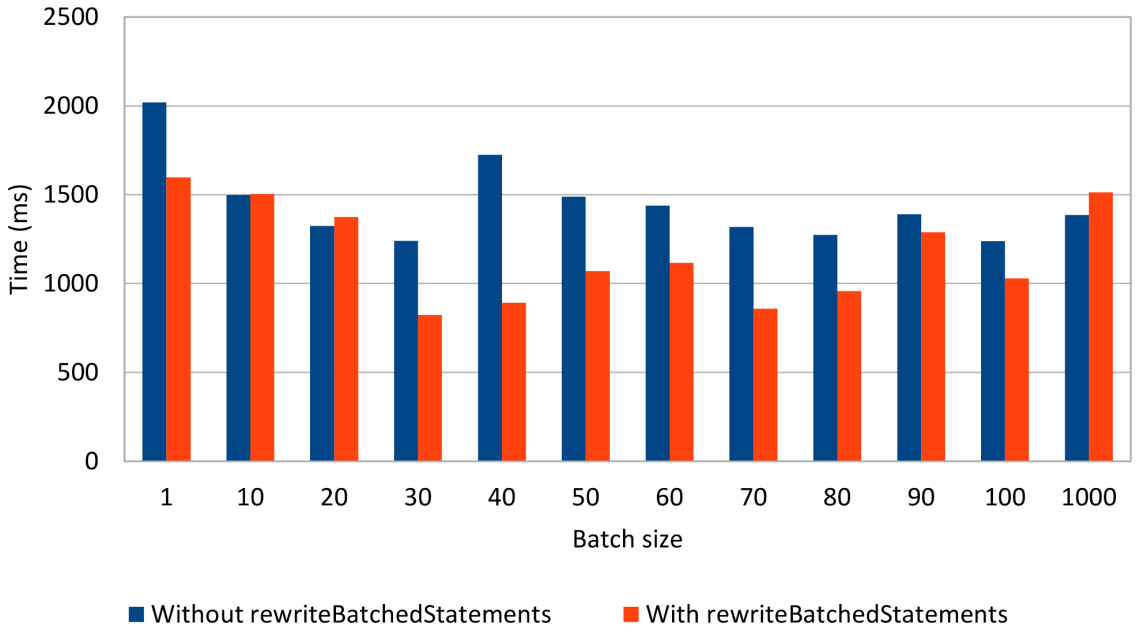


Figure 4.3: MySQL Statement Batching

Rewriting static statements seems to make a difference, as long as the batch size is not too large. In practice, it's 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.

4.2 Batching PreparedStatements

For dynamic 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 safely 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 don't sanitize the incoming request parameters.

When input parameters are passed to the underlying database statement without a proper validation, a *rogue* attacker might inject a malicious SQL routine.

This 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*) will be inserted, updated and deleted while varying the batch size.

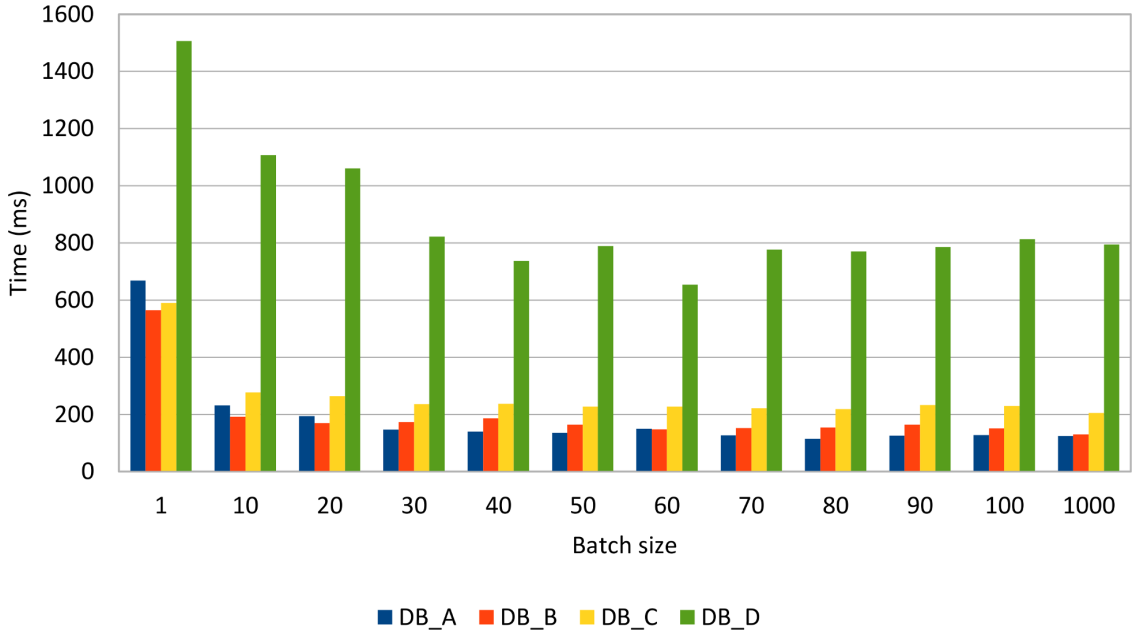


Figure 4.4: Insert PreparedStatement batch size

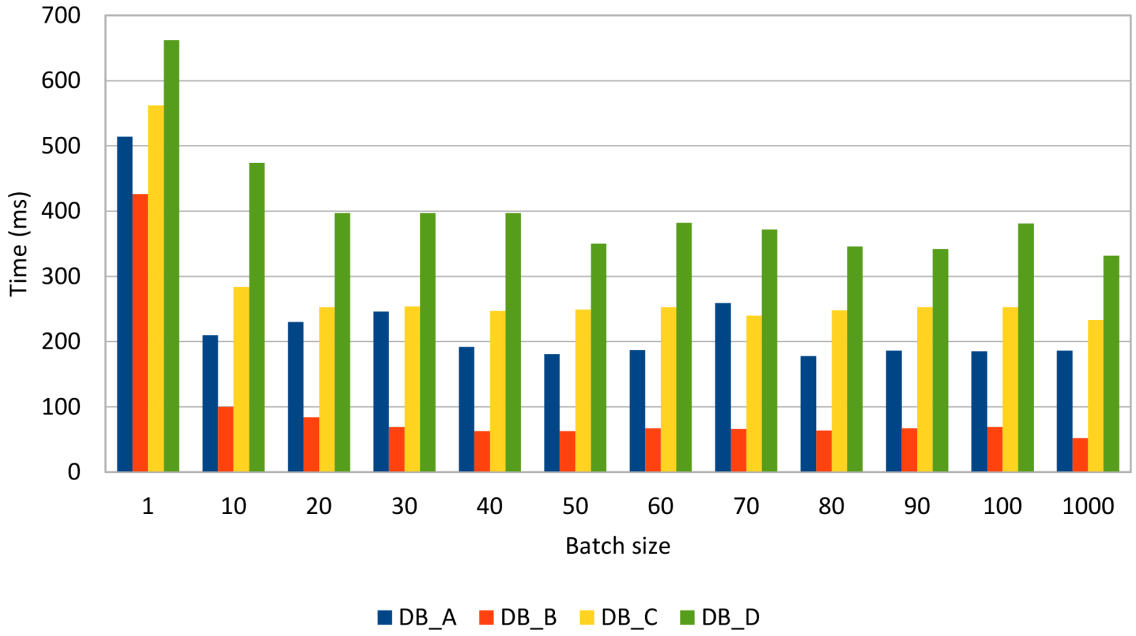


Figure 4.5: Update PreparedStatement batch size

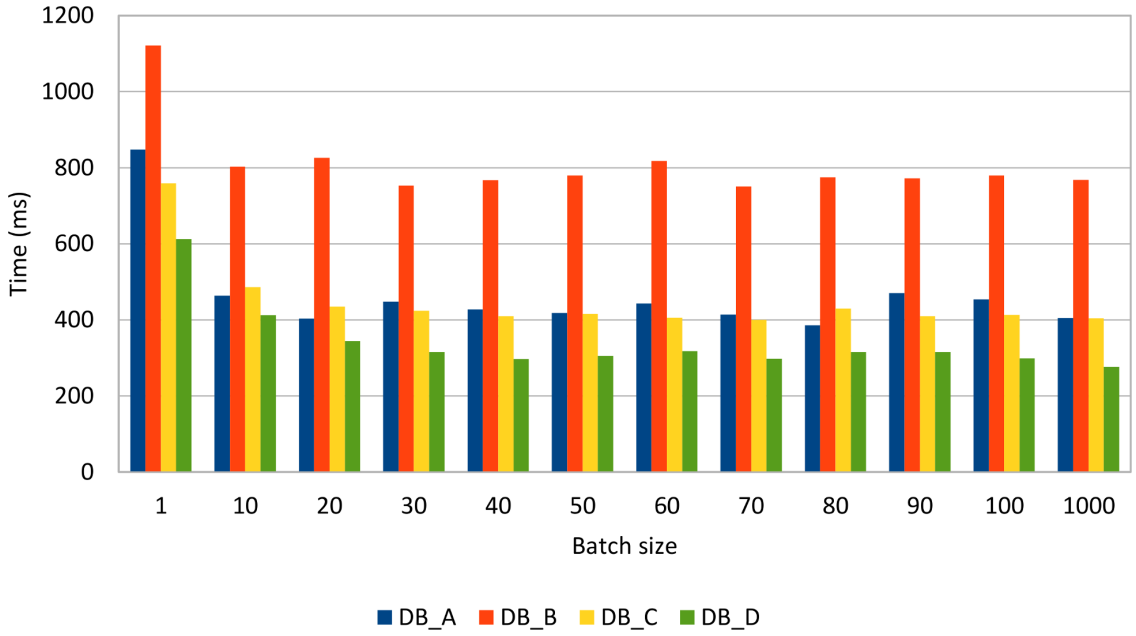


Figure 4.6: Delete PreparedStatement batch size

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's clear that, for the same data load, the `PreparedStatement` use case performs just better.

For dynamic statements, `PreparedStatement` provides better performance (when enabling batching) and stronger security guarantees. Most ORM tools use prepared statements, and since entities are inserted/update/deleted individually, they can take advantage of batching.

4.2.1 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 doesn't grow linearly with batch size. Although a larger batch value can save more database roundtrips, the overall performance gain remains relatively flat and can even drop for larger batch sizes.

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.

4.2.2 Bulk operations

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 batching, but, even so, batch updates are more flexible since each row can take a different update logic. Batch updates can also prevent *lost updates* if the data access logic employs an optimistic locking mechanism.

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

Bulk processing caveats

Processing too much data in a single transaction can degrade application performance, especially in a highly concurrent environment. Whether if using locks (two-phase locking) or MVCC (Multiversion Concurrency Control), writes always block other conflicting writes. In case the bulk updated records conflict with other concurrent transactions, then either the bulk update transaction might have to wait for some row-level locks to be released or other transactions might wait for the bulk updated rows to be committed.

4.3 Retrieving auto-generated keys

It's common practice to delegate the row identifier generation to the database system. This way, the developer doesn't 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's 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 but basically 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 doesn't 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 doesn't 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's 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 actual 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 doesn't 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.*

4.3.1 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) is insignificant.

For batch processors inserting large amounts of data, the extra sequence calls can add up. As an optimization, the identifier generation process can be split among the database and the data access logic. The database sequences can be incremented in steps (for a step of *N* the sequence numbers are *1, N + 1, 2N + 1, 3N + 1, ...*). The data access logic can assign identifiers in-between the database sequence calls (e.g. *2, 3, 4, ..., N -1, N*), and so it mitigates the extra network roundtrips penalty.

5. Statement Caching

Being a declarative language, SQL describes the *what* and not the *how*. The actual database structures and the algorithms used for fetching and preparing the desired result set are hidden away from the database client, which only has to focus on properly defining the SQL statement. This way, to deliver the most efficient data access plan, the database can attempt various execution strategies.

5.1 Statement lifecycle

The main database modules responsible for processing an SQL statement are the *Parser*, the *Optimizer* and the *Executor*.

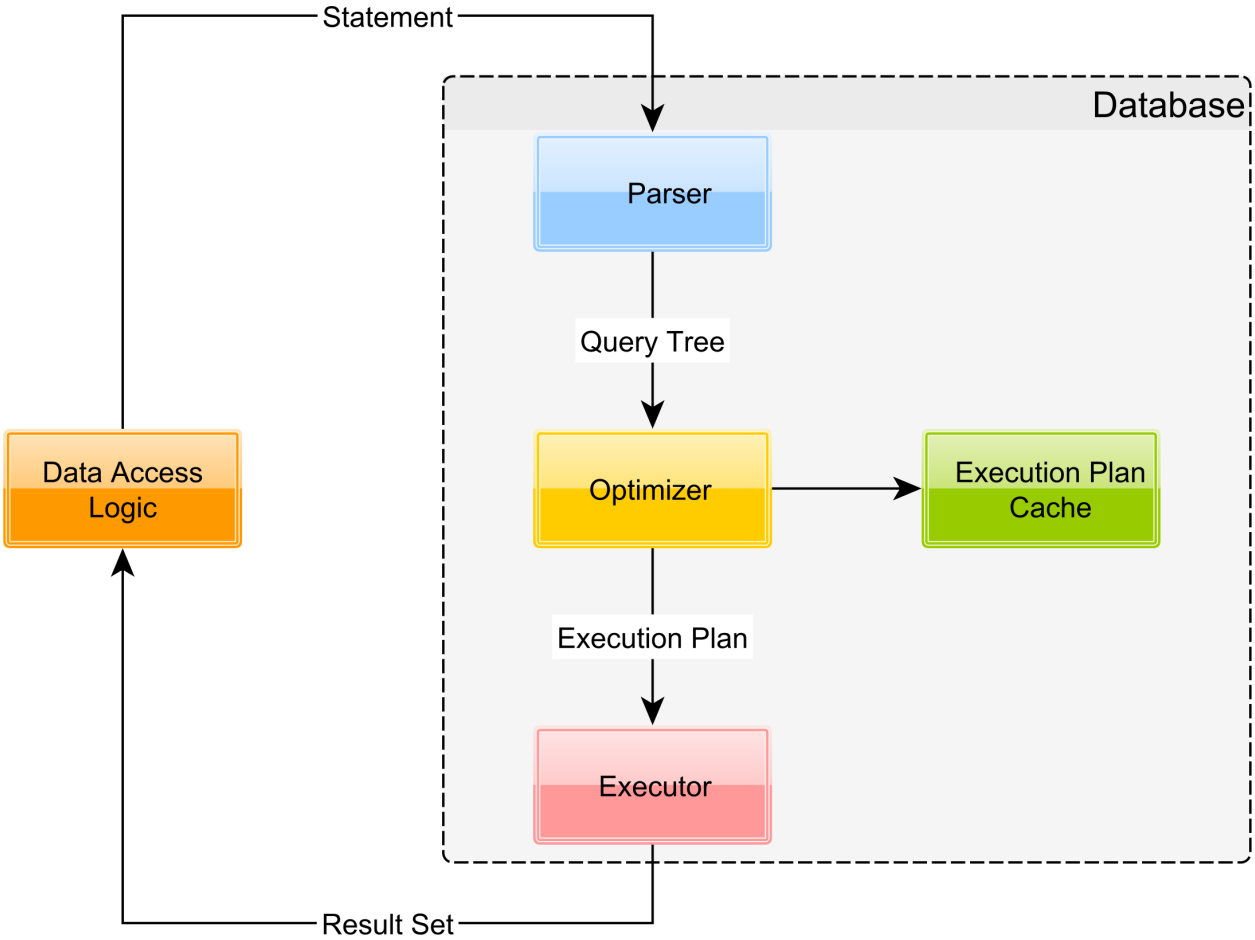


Figure 5.1: Statement lifecycle

5.1.1 Parser

The Parser checks the SQL statement and ensures its validity. The statements are verified both syntactically (the statement keywords must be properly spelled and following the SQL language guidelines) and semantically (the referenced tables and column do exist in the database).

During parsing, the SQL statement is transformed into a database internal representation, called the *syntax tree* (also known as *parse tree* or *query tree*). If the SQL statement is a high-level representation (being more meaningful from a human perspective), the syntax tree is the logical representation of the database objects required for fulfilling the current statement.

5.1.2 Optimizer

For a given syntax tree, the database must decide the most efficient data fetching algorithm. Data is retrieved by following an *access path*, and the Optimizer needs to evaluate multiple data traversing options like:

- the access method for each referencing table (table scan or index scan)
- for index scans, it must decide which index is better suited for fetching this result set
- for each joining relation (e.g. table, views or Common Table Expression), it must choose the best performing join type (e.g. Nested Loops Joins, Hash Joins, Sort Merge Joins)
- the joining order becomes very important especially for Nested Loops Joins.

The list of access path, chosen by the Optimizer, is assembled into an execution plan.

Because of the large number of all the possible action plan combinations, finding a good execution plan is not a trivial task. The more time is spent on finding the best possible execution plan, the higher the transaction response time will get, so the Optimizer has a fixed time budget for finding a reasonable plan.

The most common decision-making algorithm is CBO (Cost-Based Optimizer). Each access method translates to a physical database operation, and its associated cost in resources can be estimated. The database stores various statistics like table sizes and data cardinality (how much the column values differ from one row to the other) to evaluate the cost of a given database operation. Time is the most common unit of cost, and the database estimates it based on the number of CPU cycles and I/O operations required by a particular execution.

When finding an optimal execution plan, the Optimizer might evaluate multiple options, and, based on their overall cost, it will choose the one requiring the least amount of time to execute.

By now, it's clear that finding a proper execution plan is resource intensive, and, for this purpose, some database vendors offer execution plan caching (to eliminate the time spent on finding the optimal plan). While caching can speed up statement execution, it also incurs some additional challenges (making sure the plan is still optimal across multiple executions).

Each execution plan has a given memory footprint, and most database systems use a fixed-size cache (discarding the least used plans to make room for newer ones). DDL (Data Definition Language) statements might corrupt execution plans, making them obsolete, so the database must use a separate process for validating the existing execution plans relevancy.

But the most challenging aspect of caching is to ensure that only a good execution plan goes in the cache, since a bad plan, getting reused over and over, can really hurt application performance.

5.1.2.1 Execution plan visualization

Database tuning would not be possible without knowing the actual execution plan, a database employs for any given SQL statement. Because the output may exceed the length of a page, some execution plan columns were removed for brevity sake.

Oracle

Oracle uses the `EXPLAIN PLAN FOR` syntax, and the output goes into the `dbms_xplan` package:

```
SQL> EXPLAIN PLAN FOR SELECT COUNT(*) FROM post;
SQL> SELECT plan_table_output FROM table(dbms_xplan.display());

-------------------------------------------------------------------------------
| Id  | Operation             | Name        | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |             |     1 |     5   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |             |     1 |            |          |
|   2 |   INDEX FAST FULL SCAN| SYS_C007093 |  5000 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

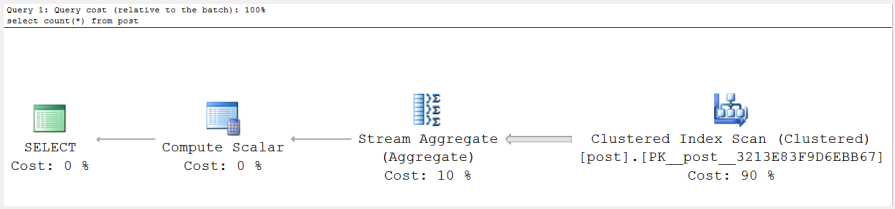
PostgreSQL

PostgreSQL reserves the `EXPLAIN` keyword for displaying execution plans:

```
# EXPLAIN SELECT COUNT(*) FROM post;
QUERY PLAN
-----------------------------------------------------------------
 Aggregate  (cost=99.50..99.51 rows=1 width=0)
   ->  Seq Scan on post  (cost=0.00..87.00 rows=5000 width=0)
```

SQL Server

The SQL Server Management Studio provides an execution plan viewer:



Another option is to enable the `SHOWPLAN_ALL` setting prior to running a statement:

```
SET SHOWPLAN_ALL ON;
GO
SELECT COUNT(*) FROM post;
GO
SET SHOWPLAN_ALL OFF;
GO
```

```
| Stmt Text                  | Est. Rows | Est. IO | Est. CPU | Subtree Cost |
-------------------------------------------------------------------------------
| select count(*) from post; | 1         | NULL    | NULL     | 0.0288       |
|   Compute Scalar           | 1         | 0       | 0.003    | 0.0288       |
|     Stream Aggregate       | 1         | 0       | 0.003    | 0.0288       |
|       Clustered Index Scan | 5000      | 0.020   | 0.005    | 0.0258       |
```

MySQL

The plan is displayed using `EXPLAIN` or `EXPLAIN EXTENDED`:

```
mysql> EXPLAIN EXTENDED SELECT COUNT(*) FROM post;
```

```
+----+--------+-------+-------+---------+-----+------+----------+-------------+
| id | select | table | type  | key     | key | rows | filtered | Extra       |
|    | type   | table | type  |         | len |      |          |             |
+----+--------+-------+-------+---------+-----+------+----------+-------------+
|  1 | SIMPLE | post  | index | PRIMARY | 8   | 5000 |   100.00 | Using index |
+----+--------+-------+-------+---------+-----+------+----------+-------------+
```

5.1.3 Executor

From the Optimizer, the execution plan goes to the Executor where it is used to fetch the associated data and build the result set. The Executor makes use of the Storage Engine (for loading data according to the current execution plan) and the Transaction Engine (to enforce the current transaction data integrity guarantees).

Having a reasonably large in-memory buffer allows the database to reduce the I/O contention, therefore reducing transaction response time. The consistency model also has an impact on the overall transaction performance since locks may be acquired to ensure data integrity, and the more locking, the less the chance for parallel execution.

5.2 Caching performance gain

Before jumping into more details about *server-side* and *client-side* statement caching, it's better to visualize the net effect of reusing statements on the overall application performance. The following test calculates the number of queries a database engine can execute within one minute time span. To better emulate a non-trivial execution plan, the test executes a statement combining both table joining as well as query nesting.

```
SELECT p.title, pd.created_on
FROM   post p
LEFT JOIN post_details pd ON p.id = pd.id
WHERE  EXISTS (
    SELECT 1
    FROM   post_comment
    WHERE  post_id = p.id AND version = ?
)
```

Running it on four different database systems, the following throughput numbers are collected.

Table 5.1: Statement caching performance gain

Database System	No Caching Throughput (Statements Per Minute)	Caching Throughput (Statements Per Minute)	Percentage Gain
DB_A	419 833	507 286	20.83%
DB_B	194 837	303 100	55.56%
DB_C	116 708	166 443	42.61%
DB_D	15 522	15 550	0.18%

Most database systems can clearly benefit from reusing statements and, in some particular use cases,

the performance gain is quite substantial.

Statement caching plays a very important role in optimizing high-performance OLTP (Online transaction processing) systems.

5.3 Server-side statement caching

Because statement parsing and the execution plan generation are resource intensive operations, some database providers offer an execution plan cache. The statement string value is used as input to a hashing function, and the resulting value becomes the execution plan cache entry key. If the statement string value changes from one execution to the other, the database cannot reuse an already generated execution plan. For this purpose, dynamic-generated JDBC `Statement(s)` are not suitable for reusing execution plans.

Forced Parameterization

Some database systems offer the possibility of intercepting SQL statements at runtime, so that all value literals are replaced with bind variables. This way, the newly parametrized statement can reuse an already cached execution plan.

To enable this feature, each database system offers a vendor-specific syntax.

Oracle

```
ALTER SESSION SET cursor_sharing=force;
```

SQL Server

```
ALTER DATABASE high_performance_java_persistence SET PARAMETERIZATION FORCED;
```

Server-side prepared statements allow the data access logic to reuse the same execution plan for multiple executions. A `PreparedStatement` is always associated with a single SQL statement, and bind parameters are used to vary the runtime execution context. Because `PreparedStatement(s)` take the SQL query at creation time, the database can precompile the associated SQL statement prior to executing it.

During the precompilation phase, the database validates the SQL statement and parses it into a syntax tree. When it comes to executing the `PreparedStatement`, the driver sends the actual parameter values, and the database can jump to compiling and running the actual execution plan.

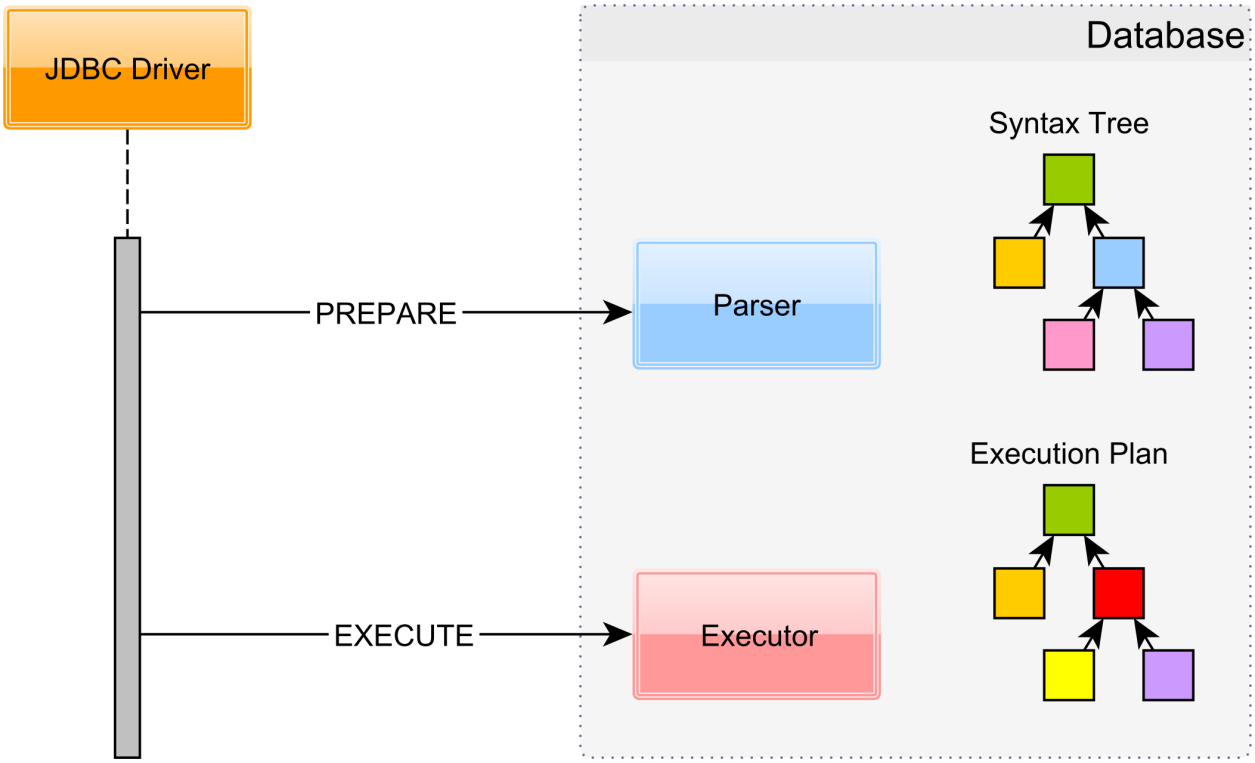


Figure 5.2: Server-Side prepared statement workflow

Because of index selectivity, in the absence of the actual bind parameter values, the Optimizer cannot compile the syntax tree into an execution plan. Since a disk access is required for fetching every additional row-level data, indexing is suitable when selecting only a fraction of the whole table data. Most database systems take this decision based on the index selectivity of the current bind parameter values.

> A covering index bypasses disk access if all requesting columns are scanned by the index.

Because each disk access requires reading a whole block of data, accessing to many disparate blocks can actually perform worse than scanning the whole table (random access is slower than sequential scans).

For prepared statements, the execution plan can either be compiled on every execution or it can be cached and reused. Recompiling the plan can generate the best data access paths for any given bind variable set while paying the price of additional database resources usage. Reusing a plan can spare database resources, but it might not be suitable for every parameter value combination.

5.3.1 Bind-sensitive execution plans

Assuming a *task* table has a *status* column with three distinct values: *TO_DO*, *DONE* and *FAILED*. The table has *100 000* rows, of which *1000* are *TO_DO* entries, *95 000* are *DONE* and *4000* are *FAILED* records.

In database terminology, the number of rows returned by a given predicate is called cardinality and, for the *status* column, the cardinality varies from *1000* to *95 000*.

$$C = \{1000, 4000, 95\ 000\}$$

By dividing cardinality with the total number of rows, the predicate selectivity is obtained:

$$S = \frac{C}{N} \times 100 = \{1\%, 4\%, 95\%\}$$

The lower the selectivity, the less rows will be matched for a given bind value, and the more selective the predicate gets. The Optimizer tends to prefer sequential scans over index lookups for high selectivity percentages, to reduce the total number of disk-access roundtrips (especially when data is scattered among multiple data blocks).

When searching for *DONE* entries, the Optimizer chooses a sequential scan access path (the estimated number of selected rows is *95 080*):

```
SQL> EXPLAIN SELECT * FROM task WHERE status = 'DONE' LIMIT 100;

Limit  (cost=0.00..1.88 rows=100 width=13)
->  Seq Scan on task  (cost=0.00..1791.00 rows=95080 width=13)
     Filter: ((status)::text = 'DONE'::text)
```

Otherwise, the search for *TO_DO* or *FAILED* entries is done through an index lookup:

```
SQL> EXPLAIN SELECT * FROM task WHERE status = 'TO_DO' LIMIT 100;

Limit  (cost=0.29..4.25 rows=100 width=13)
->  Index Scan using task_status_idx on task  (cost=0.29..36.16 rows=907)
     Index Cond: ((status)::text = 'TO_DO'::text)

SQL> EXPLAIN SELECT * FROM task WHERE status = 'FAILED' LIMIT 100;

Limit  (cost=0.29..3.86 rows=100 width=13)
->  Index Scan using task_status_idx on task  (cost=0.29..143.52 rows=4013)
     Index Cond: ((status)::text = 'FAILED'::text)
```

So, the execution plan depends on bind parameter value selectivity. If the selectivity is constant across the whole bind value domain, the execution plan is no longer sensitive to parameter values. A *generic* execution plan is much easier to reuse than a bind-sensitive one.

The following section describes how some well-known database systems implement server-side prepared statements in relation to their associated execution plans.

Oracle

Every SQL statement goes through the Parser, where it is validated both syntactically and semantically. Next, a hashing function takes the SQL statement, and the resulting hash key is used for searching the Shared Pool for an existing execution plan.

In Oracle terminology, reusing an execution plan is called a *soft parse*. To reuse a plan, the SQL statement must be identical with a previously processed one (even the case sensitivity and whitespaces are taken into consideration).

If no execution plan is found, the statement undergoes a *hard parse*[a]. The Optimizer evaluates multiple execution plans and chooses the one with the lowest associated cost, which is further compiled into a *source tree* by the *Row Source Generator*. Whether reused (*soft parse*) or generated (*hard parse*), the source tree goes to the Executor, which fetches the associated result set.

Bind peeking

As previously mentioned, the Optimizer cannot determine an optimal access path in the absence of the actual bind values. For this reason, Oracle uses *bind peeking*[b] during the *hard parse* phase.

The first set of bind parameter values determines the selectivity of the cached execution plan. By now it's clear that this strategy is feasible for uniformly distributed data sets, and a single execution plan cannot perform consistently for bind-sensitive predicates.

As of 11g, Oracle has introduced *adaptive cursor sharing*, so a statement can utilize multiple execution plans. The behavior is reactive to execution times, and bad plans are substituted with optimal ones for certain bind value combinations.

Both the execution plan cache and the *adaptive cursor sharing* are enabled by default, and, for highly concurrent OLTP systems, *hard parsing* should be avoided as much as possible. During execution plan generation, the database uses a *latch* to avoid multiple concurrent statements from accessing the same database objects. Latches introduce a serial execution, which, in turn, increases contention and decreases concurrency and scalability.

`PreparedStatement(s)` optimize the execution plan cache hit rate and are therefore preferred over plain JDBC `Statement(s)`.

[a] https://docs.oracle.com/database/121/TGSQL/tgsql_sqlproc.htm#TGSQL175
[b] https://docs.oracle.com/database/121/TGSQL/tgsql_cursor.htm#TGSQL848

SQL Server

SQL Server always caches execution plans[a] for both JDBC `Statement(s)` and `PreparedStatement(s)`. The execution plans are stored in the *procedure cache* region and they are evicted only when the in-memory storage starts running out of space.

Even if SQL Server supports plain statements *forced parametrization*, preparing statements remains the most effective way to increase the likelihood of an execution plan cache hit.

The catch is that all prepared statements should use the qualified object name, thus the schema must always precede the table name.

So, instead of a query like this:

```
SELECT * FROM task WHERE status = ?;
```

the data access layer should always append the schema to all table names:

```
SELECT * FROM etl.task WHERE status = ?;
```

Without specifying the database object schema, the cache cannot determine which statistics to consider when analyzing the effectiveness of a given execution plan.

SQL Server inspects the actual parameter values during the first execution of a prepared statement. This process is called *parameter sniffing*, and its effectiveness is relative to predicate value distribution.

The database engine monitors statement execution times, and if the existing cached plan doesn't perform efficiently or if the underlying table structure or data distribution statistics undergo a conflicting change, then the database recompiles the execution plan according to the new parameter values.

For skewed data, reusing plans might be suboptimal, and recompiling plans on every execution could be a better alternative. To address the *parameter sniffing* limitations, SQL Server offers the `OPTION (RECOMPILE)` query hint[b], so the statement can bypass the cache and generate a fresh plan on every execution.

```
SELECT * FROM task WHERE status = ? OPTION(RECOMPILE);
```

[a] https://technet.microsoft.com/en-us/library/ms181055%28v=sql.100%29.aspx
[b] https://msdn.microsoft.com/en-us/library/ms181714.aspx

PostgreSQL

Prior to 9.2, a prepared statement was planned and compiled entirely during the prepare phase, so the execution plan was generated in the absence of the actual bind parameter values. Although it attempted to spare database resources, this strategy was very sensitive to skewed data. Since PostgreSQL 9.2, the prepare phase only parses and rewrites a statement, while the optimization and the planning phase are deferred until execution time. This way, the rewritten syntax tree is optimized according to the actual bind parameter values, and an optimal execution plan is generated.

For a singular execution, a plain statement requires only a one database roundtrip, while a prepared statement needs two (a prepare request and an execution call). To avoid the networking overhead, by default, JDBC `PreparedStatement(s)` do both the prepare and the execute phases over a single database request.

A client-side prepared statement must run at least *5* times for the driver to turn it into a server-side statement. The default execution count value is given by the `prepareThreshold` parameter, which is configurable as a connection property or through a driver-specific API[a].

After several executions, if the performance is not sensitive to bind parameter values, the Optimizer might choose to turn the plan into a generic one and cache it for reuse.

[a] https://jdbc.postgresql.org/documentation/publicapi/org/postgresql/PGStatement.html

MySQL

When preparing a statement, the MySQL Parser generates a syntax tree which is further validated and pre-optimized by a *resolution* mechanism. The syntax tree undergoes several data-insensitive transformations and the final output is a *permanent tree*.

Since MySQL 5.7.4[a], all *permanent transformations* (rejoining orders or subquery optimizations) are done in the prepare phase, so the execution phase only applies data-sensitive transformations. MySQL doesn't cache execution plans, so every statement execution is optimized for the current bind parameter values, therefore avoiding data skew issues.

Because of some unresolved issues, since version 5.0.5[b], the MySQL JDBC driver only emulates server-side prepared statements. To switch to server-side prepared statements, both the `useServerPrepStmts` and the `cachePrepStmts` connection properties must be set to *true*.

Before activating this feature, it's better to check the latest Connector/J release notes and validate this feature is safe for using.

[a] http://mysqlserverteam.com/mysql-performance-schema-prepared-statements-instrumentation/
[b] http://dev.mysql.com/doc/relnotes/connector-j/en/news-5-0-5.html

5.4 Client-side statement caching

Not only the database side can benefit from caching statements, but also the JDBC driver can reuse already constructed statement objects. The main goals of the client-side statement caching can be summarized as follows:

- reducing client-side statement processing, which, in turn, lowers transaction response time
- sparing application resources by recycling statement objects along with their associated database-specific metadata.

In high-performance OLTP applications, transactions tend to be very short, so even a minor response time reduction can make a difference on the overall transaction throughput.

Oracle implicit statement caching

Unlike server-side plan cache, the client one is confined to a database connection only. Since the SQL `String` becomes the cache entry key, `PreparedStatement(s)` and `CallableStatement(s)` have a better chance of getting reused, therefore the Oracle JDBC driver supports caching only for these two statement types. When enabling caching (it is disabled by default), the driver returns a logical statement, so when the client closes it, the logical statement goes back to the cache.

From a development point of view, there is an *implicit* statement caching mechanism as well as an *explicit* one. Both caching options share the same driver storage, which needs to be configured according to the current application requirements.

The implicit cache can only store statement *metadata*, which doesn't change from one execution to the other. Although it can be set for each individual `Connection`, it's convenient to configure it at the `DataSource` level (all connections inheriting the same caching properties):

```
connectionProperties.put("oracle.jdbc.implicitStatementCacheSize",
    Integer.toString(cacheSize));
dataSource.setConnectionProperties(connectionProperties);
```

Setting the `implicitStatementCacheSize` also enables the cache. By default, all executing statements are being implicitly cached, and this might not be desirable (some occasional queries might evict other frequently executed statements). To control the statement caching policy, JDBC defines the `isPoolable()` and `setPoolable(boolean poolable)` `Statement` methods:

```
if (statement.isPoolable()) {
    statement.setPoolable(false);
}
```

Oracle explicit statement caching

The explicit cache is configurable and managed through an Oracle-specific API. Prior to using it, it must be enabled and resized using the underlying `OracleConnection` reference.

```
OracleConnection oracleConnection = (OracleConnection) connection;
oracleConnection.setExplicitCachingEnabled(true);
oracleConnection.setStatementCacheSize(cacheSize);
```

When using the explicit cache, the data access controls which statements are cacheable, so there is no need for using the `setPoolable(boolean poolable)` method anymore. The following example demonstrates how to make use of the explicit caching mechanism.

```
PreparedStatement statement = oracleConnection
    .getStatementWithKey(SELECT_POST_REVIEWS_KEY);
if (statement == null)
    statement = connection.prepareStatement(SELECT_POST_REVIEWS);
try {
    statement.setInt(1, 10);
    statement.execute();
} finally {
    ((OraclePreparedStatement) statement).closeWithKey(SELECT_POST_REVIEWS_KEY);
}
```

The explicit caching relies on two main operations, which can be summarized as follows:

1. the `getStatementWithKey(String key)` method loads a statement from the cache. If no entry is found, the `PreparedStatement` must be manually created using standard JDBC API
2. the `closeWithKey(String key)` method pushes the statement back into the pool.

The vendor-specific API couples the data access code to the Oracle-specific API which hinders portability and it require a more complex data access logic (when accommodating multiple database systems).

Aside from caching *metadata*, the explicit cache also stores execution *state* and *data*. Although reusing more client-side constructs might improve performance even further, this strategy poses the risk of mixing previous and current execution contexts, so caution is advised.

SQL Server

Although the Microsoft SQL Server JDBC driver defines a `disableStatementPooling` property, as of writing (the 4.2 version), the statement cache cannot be enabled[a].

On the other hand, jTDS (the open source JDBC 3.0 implementation) offers statement caching on a per connection basis. Being a JDBC 4.0 specific API, The `setPoolable(boolean poolable)` `Statement` method is not implemented in the 1.3.1 jTDS release. The cache has a default size of *500* entries which is also adjustable.

```
((JtdsDataSource) dataSource).setMaxStatements(cacheSize);
```

Even if jTDS has always focused on performance, the lack of a steady release schedule is a major drawback compared to the Microsoft driver.

[a] https://msdn.microsoft.com/en-us/library/ms378988%28v=sql.110%29.aspx

PostgreSQL

Since the PostgreSQL JDBC driver 9.4-1202[a] version, the client-side statements are cached, and their associated server-side statement keys are retained even after the initial `PreparedStatement(s)` is closed. As long as the current connection cache contains a given SQL statement, both the client-side `PreparedStatement` and the server-side object can be reused. The `setPoolable(boolean poolable)` method has no effect, and caching cannot be disabled on a per statement basis.

The statement cache is controlled by the following connection properties:

- `preparedStatementCacheQueries` - the number of statements cached for each database connection. A value of *0* disables the cache, and server-side prepared statements are no longer available after the `PreparedStatement` is closed. The default value is *256*.
- `preparedStatementCacheSizeMiB` - the statement cache has an upper memory bound, and the default value is *5* MB. A value of *0* disables the cache.

These properties can be set both as connection parameters[b] or as `DataSource` properties:

```
((PGSimpleDataSource) dataSource).setPreparedStatementCacheQueries(cacheSize);
((PGSimpleDataSource) dataSource).setPreparedStatementCacheSizeMiB(cacheSizeMb);
```

[a] https://jdbc.postgresql.org/documentation/changelog.html#version_9.4-1202
[b] https://jdbc.postgresql.org/documentation/head/connect.html#connection-parameters

MySQL

The statement caching is associated with a database connection, and it applies to all executing statements. In the 5.1.36 Connector/J driver version, the `setPoolable(boolean poolable)` method can disable caching for server-side statements only, the client-side ones being unaffected by this setting.

The client-side statement cache is configured using the following properties:

- `cachePrepStmts` - enables the client-side statement cache as well as the server-side statement validity checking. By default, the statement cache is disabled.
- `prepStmtCacheSize` - the number of statements cached for each database connection. The default cache size is *25*.
- `prepStmtCacheSqlLimit` - the maximum length of an SQL statement allowed to be cached. The default maximum value is *256*.

These properties can be set both as connection parameters[a] or at `DataSource` level:

```
((MysqlDataSource) dataSource).setCachePrepStmts(true);
((MysqlDataSource) dataSource).setPreparedStatementCacheSize(cacheSize);
((MysqlDataSource) dataSource).setPreparedStatementCacheSqlLimit(maxLength);
```

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

6. ResultSet Fetching

Having discussed the SQL statement optimizations (batching and caching), it's time to move on to the response part of a query processing. Unlike the insert, update and delete statements, which only return the affected row count, a JDBC select query returns a `ResultSet` instead.

The database Executor takes an execution plan and fetches data into a result set. Rows may be either extracted at once or upon being requested by the database client.

The SQL Standard defines both the result set and the cursor descriptor through the following properties:

- scrollability (the direction in which the result set can be iterated)
- sensitivity (when should data be fetched)
- updatability (available for cursors, it allows the client to modify records while traversing the result set)
- holdability (the result set scope in regard to a transaction lifecycle).

Following the standard specification, the JDBC `ResultSet` offers support for all the aforementioned properties.

Table 6.1: JDBC ResultSet properties

Property Name	Description
TYPE_FORWARD_ONLY	The result set can only be iterated from the first to the last element. This is the default *scrollability* value.
TYPE_SCROLL_INSENSITIVE	The result set takes a loading time snapshot which can be iterated both forward and backwards.
TYPE_SCROLL_SENSITIVE	The result set is fetched on demand, while being iterated without any direction restriction.
CONCUR_READ_ONLY	The result set is just a static data projection which doesn't allow row-level manipulation. This is the default *changeability* value.
CONCUR_UPDATABLE	The cursor position can be used to update or delete records, or even insert a new one.
CLOSE_CURSORS_AT_COMMIT	The result set is closed when the current transaction ends.
HOLD_CURSORS_OVER_COMMIT	The result set remains open even after the current transaction is committed.

6.1 ResultSet scrollability

The JDBC `ResultSet` can be traversed using an application-level cursor. The fetching mechanism is therefore hidden behind an iterator API which decouples the application code from the data retrieval strategy. Some database drivers prefetch the whole result set on the client-side, while other implementations retrieve batches of data on a demand basis.

By default, the `ResultSet` uses a *forward-only* application-level cursor, which can be traversed only once, from the first position to last one. Although this is sufficient for most applications, JDBC also offers *scrollable* cursors, therefore allowing the row-level pointer to be positioned freely (in any direction and on every record).

The main difference between the two scrollable result sets lays in their *selectivity*. An *insensitive* cursor offers a static view over the current result set, so the data needs to be fetched entirely prior to being iterated. A *sensitive* cursor allows the result set to be fetched dynamically, so it can reflect concurrent changes.

Oracle

Since the database engine doesn't offer support for scrollable result sets, the JDBC driver emulates it on top of a client-side caching mechanism[a]. As a consequence, the result set shouldn't be too large as otherwise it can easily fill the client application memory.

A sensitive scrollable result set is limited to selecting data from a single table only.

[a] https://docs.oracle.com/database/121/JJDBC/resltset.htm#JJDBC28615

SQL Server

All three cursor types are supported. An insensitive scroll generates a server-side database snapshot, which the client fetches in batches. The sensitive scroll uses a server-side updatable window and changes are synchronized only for the current processing window.

The driver suggests using *read-only* cursors when there is no intent on updating the result set. The forward-only scroll delivers the best performance for small result sets[a].

PostgreSQL

By default, the result set is fetched entirely and cached on the client-side. Only the forward-only and the insensitive scroll are supported. For large result sets, fetching all records at once can put a lot of pressure on both the database server resources and the client-side memory. For this purpose, PostgreSQL allows associating a result set to a database cursor[a], so records can be fetched on demand.

```
PreparedStatement statement = connection.prepareStatement(
    "SELECT title FROM post WHERE id BETWEEN ? AND ?"
);
statement.setFetchSize(100);
```

Only the forward-only result set type can benefit from database-side cursors, and the statement fetch size must be set to a positive integer value.

[a] https://jdbc.postgresql.org/documentation/head/query.html

MySQL

Only the insensitive scroll type is supported, even when explicitly specifying a forward-only result set. Because MySQL doesn't support database cursors, the driver retrieves the whole result set and caches it on the client-side. Large result sets can be streamed[a] only if the statement type is both forward-only and read-only and the fetch size value is set to the lowest `java.lang.Integer` value.

```
PreparedStatement statement = connection.prepareStatement(
    "SELECT title FROM post WHERE id BETWEEN ? AND ?",
    ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY
);
statement.setFetchSize(Integer.MIN_VALUE);
```

Streaming requires fetching one row at a time, which might incur multiple database roundtrips. Until the stream is closed, the connection cannot execute any other statement.

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

6.2 ResultSet changeability

By default, the result set is just a read-only view of the underlying data projection. Inspired by database cursors, the JDBC standard offers updatable result sets, so the data access logic can modify records while iterating the application-level cursor.

Mixing reading and writing logic into a single database transaction reminds of two-tier architectures, where holding the result set, even in the user think time, was both common and acceptable.

For web applications, requests should be as short as possible, and most application-level transactions span over multiple web requests. The former request may use a read-only database transaction to fetch data and render it to the user, while the latter might use a read-write transaction to apply data modifications. In such scenario, an updatable result set is of little use, especially because holding it open (along with the underlying database connection) over multiple requests can really hurt application scalability.

The following test case verifies if a forward-only and read-only cursor performs better than a sensitive and updatable one. The test executes *10 000* statements, fetching *100* posts along with *details* and their associated *1000* comments.

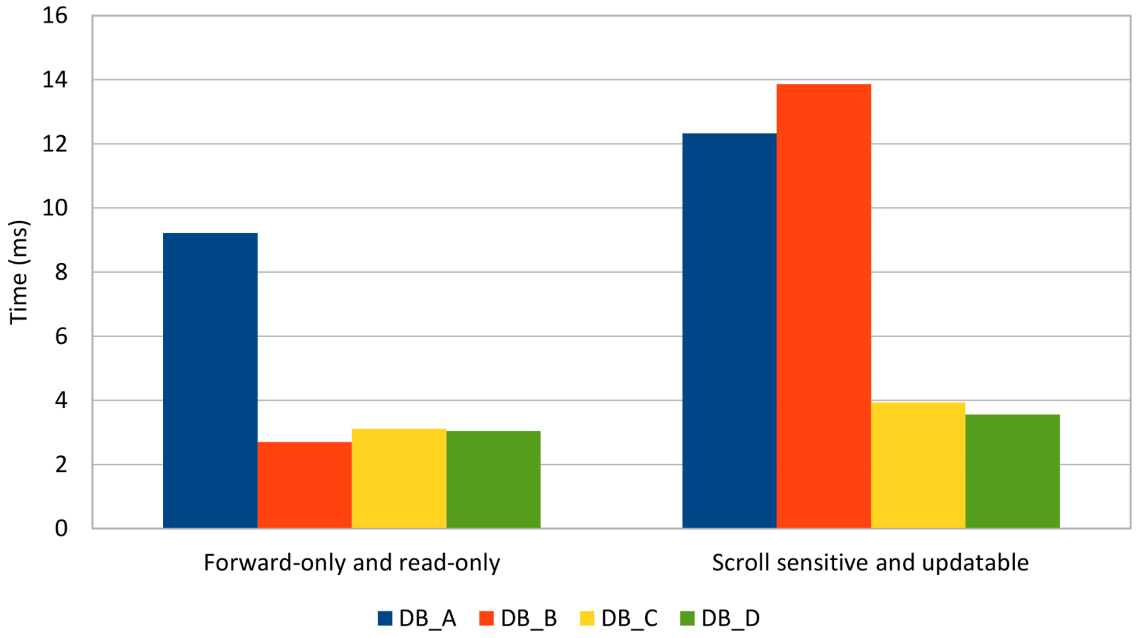


Figure 6.1: ResultSet cursor type

Every database system under test showed a slight improvement when using forward-only and read-only result sets.

As a rule of thumb, if the current transaction doesn't require updating selected records, the forward-only and read-only default result set type is the most efficient option. Even if it is a reminiscence from JDBC 1.0, the default result set is still the right choice in most situations.

6.3 ResultSet holdability

The JDBC 3.0 version added support for result set holdability and, unlike scrollability and updatability, the default value is implementation specific.

Oracle

The default and the only supported holdability value is `HOLD_CURSORS_OVER_COMMIT`. An exception is thrown when trying to change this setting to any other value.

SQL Server

By default, the result set is kept open even after the current transaction is committed or rolled back. SQL Server supports the `CLOSE_CURSORS_AT_COMMIT` setting as well.

PostgreSQL

Unlike other database systems, the default holdability value is `CLOSE_CURSORS_AT_COMMIT`, but the driver also supports the `HOLD_CURSORS_OVER_COMMIT` setting.

MySQL

The default and the only supported holdability value is `HOLD_CURSORS_OVER_COMMIT`.

In a typical enterprise application, database connections are reused from one transaction to another, so holding a result set after a transaction ends is risky. Depending on the underlying database system and on the cursor type, a result set might allocate system resources, which, for scalability reasons, need to be released as soon as possible.

Although the `CLOSE_CURSORS_AT_COMMIT` holdability option is not supported by all database engines, the same effect can be achieved by simply closing all acquired `ResultSet(s)` and their associated `Statement` objects.

6.4 Fetching size

The JDBC `ResultSet` acts as an application-level cursor, so whenever the statement is traversed, the result must be transferred from the database to the client. The transfer rate is controlled by the `Statement` fetch size.

```
statement.setFetchSize(fetchSize);
```

A custom fetch size gives the driver a hint as to the number of rows needed to be retrieved in a single database roundtrip. The default value of *0* leaves each database choose its own driver-specific fetching policy.

Oracle

The default fetch size is set to *10* records, as a consequence of the JDBC driver memory model.

The Oracle 10i and 11g drivers pre-allocate a `byte[]` and a `char[]` buffers at statement creation time, whose lengths are given by the multiplication of the fetch size by the maximum memory footprint of each selected column. A `VARCHAR2(N)` column can accommodate at most *N* characters (or *2N* bytes). Storing a field with a maximum size of *5* characters into a `VARCHAR2(4000)` column would pre-allocate *8000* bytes on the client-side, which is definitely a waste of memory.

Avoiding memory allocation, by reusing existing buffers, is a very solid reason for employing statement caching. Only when using the implicit statement cache, the 10i and 11g drivers can benefit from recycling client-side memory buffers.

The 12c implementation[a] defers the buffer allocation until the result set is ready for fetching. This driver version uses two `byte[]` arrays instead, which are allocated lazily. Compared to the previous versions, the 12c memory footprint is greatly reduced since, instead of allocating the maximum possible data storage, the driver uses the actual extracted data size.

Although the optimal fetch size is application-specific, being influenced by the data size and the runtime environment concurrency topology, the Oracle JDBC driver specification recommends limiting the fetch size to at most *100* records.

Like with any other performance optimization, these indications are suggestions at best, and measuring application performance is the only viable way of finding the right fetch size.

[a] http://www.oracle.com/technetwork/database/application-development/jdbc-memory-management-12c-1964666.pdf

SQL Server

The SQL Server JDBC driver uses *adaptive buffering*[a], so the result set is fetched in batches, as needed. The size of a batch is therefore automatically controlled by the driver.

Although enabled by default, adaptive buffering is limited to forward-only and read-only cursors. Both scrollable and updatable result sets operate on a single block of data, whose length is determined by the current statement fetch size.

[a] https://msdn.microsoft.com/en-us/library/bb879937%28v=sql.110%29.aspx

PostgreSQL

The entire result set is fetched at once[a] into client memory. The default fetch size requires only one database roundtrip, at the price of increasing the driver memory consumption. By changing fetch size, the result set is associated with a database cursor, allowing data to be fetched on demand.

[a] https://jdbc.postgresql.org/documentation/head/query.html

MySQL

Because of the network protocol design consideration[a], fetching the whole result set is the most efficient data retrieval strategy. The only streaming option requires processing one row at a time, which, for large result sets, it implies many database roundtrips.

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

The following graph captures the response time of four database systems when fetching *10 000* rows while varying the fetch size of the forward-only and read-only `ResultSet`.

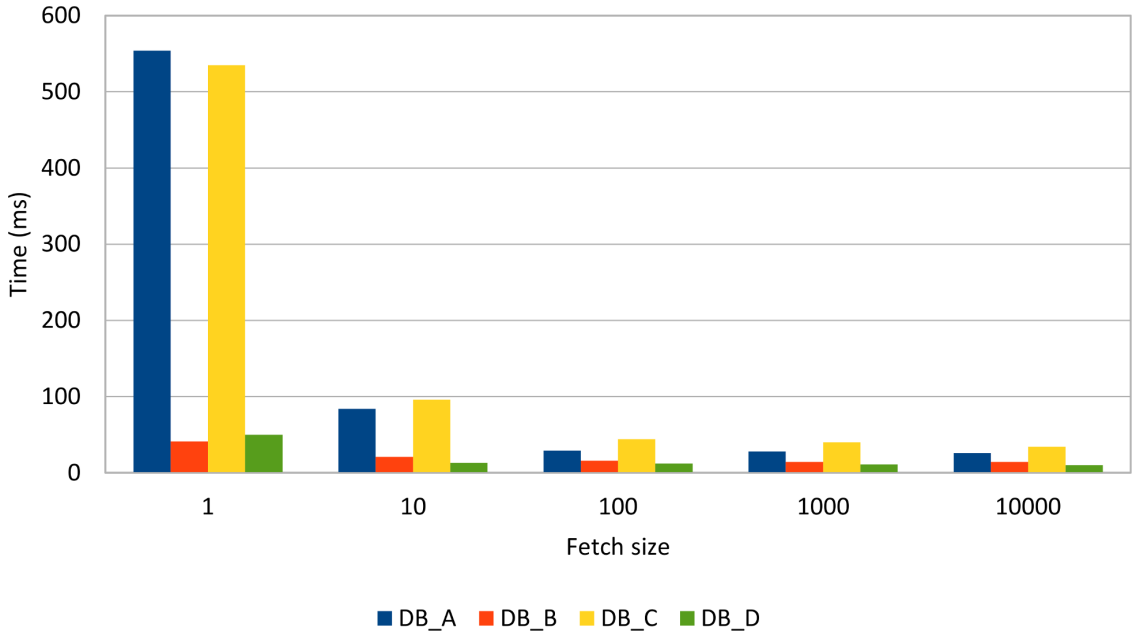


Figure 6.2: ResultSet fetch size

Fetching one row at a time requires *10 000* roundtrips, and the networking overhead impacts response time. Up to *100* rows, the fetch size plays an important role in lowering retrieval time (only *100*

roundtrips), but beyond this point the gain becomes less noticeable.

6.5 ResultSet size

Setting the appropriate fetching size can undoubtedly speed up the result set retrieval, as long as a statement fetches only the data required by the current business logic. All too often, unfortunately, especially with the widespread of ORM tools, the statement might select more data than necessary. This issue might be caused by selecting too many rows or too many columns, which are later discarded in the data access or the business layer.

6.5.1 Too many rows

Tables tend to grow in size (especially if the application gains more traction), and, with time, a moderate result set might easily turn into a performance bottleneck. This issues are often discovered in production systems, long after the application code was shipped.

A user interface can accommodate just as much info as the view allows displaying. For this reason, it's inefficient to fetch a result set entirely if it cannot fit into the user interface. Pagination or dynamic scrolling are common ways of addressing this issue, and partitioning data sets becomes unavoidable.

Limiting result sets is common in batch processing as well. To avoid long-running transactions (which might put pressure on the database undo/redo logs), and to also benefit from parallel execution, a batch processor divides the current work load into smaller jobs. This way, a batch job can take only a subset of the whole processing data.

When the result set size is limited by external factors, it makes no sense to select more data than necessary.

Without placing upper bounds, the result sets grow proportionally with the underlying table data. A large result set requires more time to be extracted and to be sent over the wire too.

Limiting queries can therefore ensure predictable response times and database resource utilization. The shorter the query processing time, the quicker the row-level locks are released, and the more scalable the data access layer becomes.

There are basically two ways of limiting a result set.

The former and the most efficient strategy is to include the row restriction clause in the SQL statement. This way, the Optimizer can better come up with an execution plan that's optimal for the current result set size (like selecting an index scan instead of a full scan).

The latter is to configure a maximum row count at the JDBC `Statement` level. Ideally, the driver can adjust the statement to include the equivalent result set size restriction as an SQL clause, but, most often, it only hints the database engine to use a database cursor instead.

6.5.1.1 SQL limit clause

SQL:2008

Although the SQL:2008 added support for limiting result sets, only starting from Oracle 12c[a], SQL Server 2012[b] and PostgreSQL 8.4[c], the standard syntax started being supported.

```
SELECT pc.id AS pc_id, p.title AS p_title
FROM post_comment pc
INNER JOIN post p ON p.id = pc.post_id
ORDER BY pc_id
OFFSET ? ROWS
FETCH FIRST (?) ROWS ONLY
```

Surrounding the *row count* placeholder with parentheses is a workaround for a PostgreSQL database issue[d]. On SQL Server it works with or without the enclosing parentheses.

Older database versions or other database systems (e.g. MySQL 5.7) still rely on a vendor-specific syntax to restrict the result set size.

[a] https://docs.oracle.com/database/121/SQLRF/statements_10002.htm#SQLRF01702
[b] https://technet.microsoft.com/en-us/library/gg699618%28v=sql.110%29.aspx
[c] http://www.postgresql.org/docs/current/static/sql-select.html#SQL-LIMIT

Oracle

Unlike other relational databases, Oracle doesn't have a reserved keyword for restricting a query result set, but because each record is attributed a result set entry order number (given by the `ROWNUM` virtual column), the syntax for limiting a result set becomes:

```
SELECT *
FROM (
        SELECT pc.id AS pc_id, p.title AS p_title
        FROM post_comment pc
        INNER JOIN post p ON p.id = pc.post_id
        ORDER BY pc_id
)
WHERE ROWNUM <= ?
```

SQL Server

The `TOP` keyword has been the *de facto* way of restricting the result set size:

```
SELECT TOP (?) pc.id AS pc_id, p.title AS p_title
FROM post_comment pc
INNER JOIN post p ON p.id = pc.post_id
ORDER BY pc_id
```

PostgreSQL and MySQL

The `LIMIT` keyword places an upper bound on the result set size:

```
SELECT pc.id AS pc_id, p.title AS p_title
FROM post_comment pc
INNER JOIN post p ON p.id = pc.post_id
ORDER BY pc_id
LIMIT ?
```

6.5.1.2 JDBC max rows

The JDBC specification defines the maxRows[1] attribute which limits all `ResultSet(s)` for the current statement.

```
statement.setMaxRows(maxRows);
```

Unlike the SQL construct, the JDBC alternative is portable across all driver implementations. This can be very handy especially when the application needs to support multiple database systems.

According to the JDBC documentation, the driver is expected to discard the extra rows when the maximum threshold is reached.

From a data access performance perspective, dropping extra rows is a poor strategy because it wastes both database resources (CPU, I/O, Memory) as well as networking bandwidth.

[1] http://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html#setMaxRows-int-

Oracle

When a `ResultSet` is being traversed, the client-side cursor fetches data in chunks (the fetch size attribute controlling the number of records in a chunk).

After each new batch retrieval, the total number of records is checked against the `maxRows` upper bound, and if the threshold is reached, the driver closes the networking stream.

The `maxRows` upper bound can therefore prevent the database and the client-side driver from wasting resources on fetching records the client doesn't even need.

SQL Server

When the `Statement.setMaxRows(int maxRows)`[a] method is called, the driver calls the `SET ROWCOUNT` SQL command:

```
SET ROWCOUNT N
```

Unlike the `TOP` or `FETCH` SQL directives, the `ROWCOUNT` command is taken into consideration only during the execution phase, and it doesn't influence the plan generation. Because of this, the execution plan might not be optimized for the given result set size, so a table scan might be chosen over an index.

The SQL Server documentation[b] recommends using the SQL directives over the `SET ROWCOUNT` command.

[a] https://msdn.microsoft.com/en-us/library/ms378838%28v=sql.110%29.aspx

PostgreSQL

The JDBC driver takes the `maxRows` statement attribute and sends it along with the query being executed. With this info, the Optimizer can choose an execution plan that is tailored for the given result set size, and it might even avoid some expensive operations like sorting the whole projection. The Extractor can also close the database cursor right after it fetched the desired number of records, therefore sparing both database and networking resources.

MySQL

The `maxRows` attribute is not sent to the database server, so neither the Optimizer nor the Extractor can benefit from this hint. While the JDBC driver would normally fetch all rows, by placing an upper bound on the result set size, the client-side can spare some networking overhead.

6.5.1.3 Less is more

The following test is going to demonstrate the performance improvement of limiting the result set size. The test data set consists of *100 000 post* and *1 000 000 comments* entries. In the first round, the entire result set is being fetched, and the response time is going to be proportional to the projection size. By limiting the result set to *100* records, either by using SQL or the JDBC `maxSize` setting, the response time is going to drop significantly.

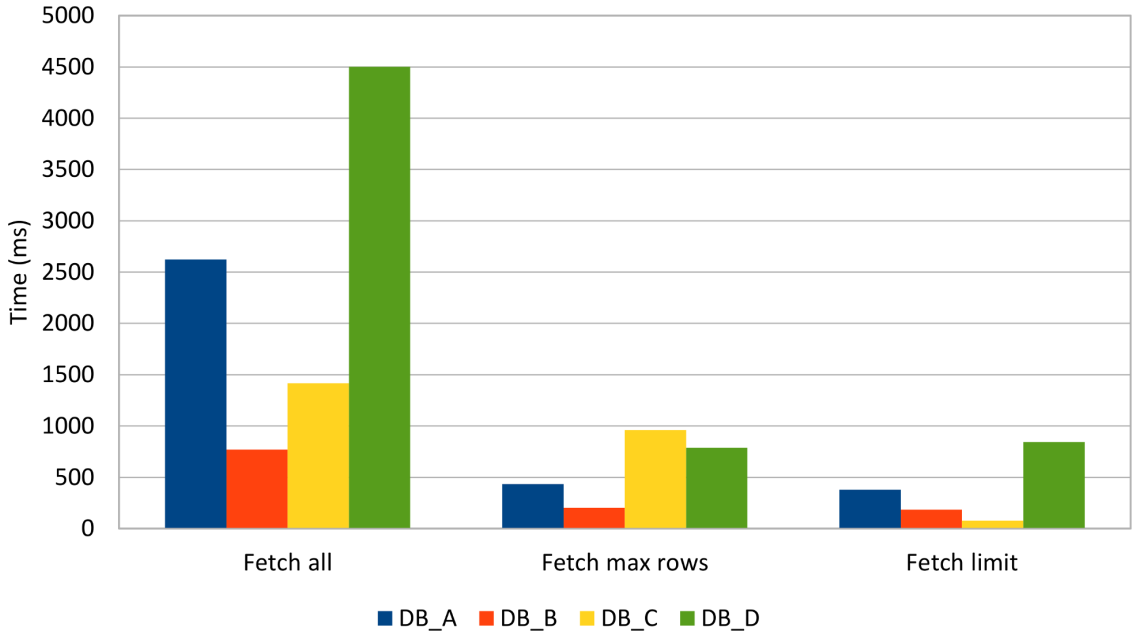


Figure 6.3: ResultSet size

The test results confirm the previous assumptions and the SQL level restriction proves to be the optimal strategy for limiting a result set. The `maxRows` driver implementation yields a surprisingly good result, especially when taking into consideration the JDBC specification on dropping extra records. Fetching a large result set puts a lot of pressure on database resources, which doesn't only affect the current processing unit of work. Other concurrent transactions can also exhibit longer processing times, as a consequence of database resources shortage.

6.5.2 Too many columns

Not only fetching too many rows can cause performance issues, but even extracting too many columns can increase the result set processing response time. The next test case is going to select *100 posts* with *details* and their associated *1000 comments*, using one of following two statements:

```
SELECT *
FROM post_comment pc
INNER JOIN post p ON p.id = pc.post_id
INNER JOIN post_details pd ON p.id = pd.id

SELECT pc.version
FROM post_comment pc
INNER JOIN post p ON p.id = pc.post_id
INNER JOIN post_details pd ON p.id = pd.id
```

The following graph depicts the execution times of fetching all columns, as opposed to extracting only a subset of the whole column projection.

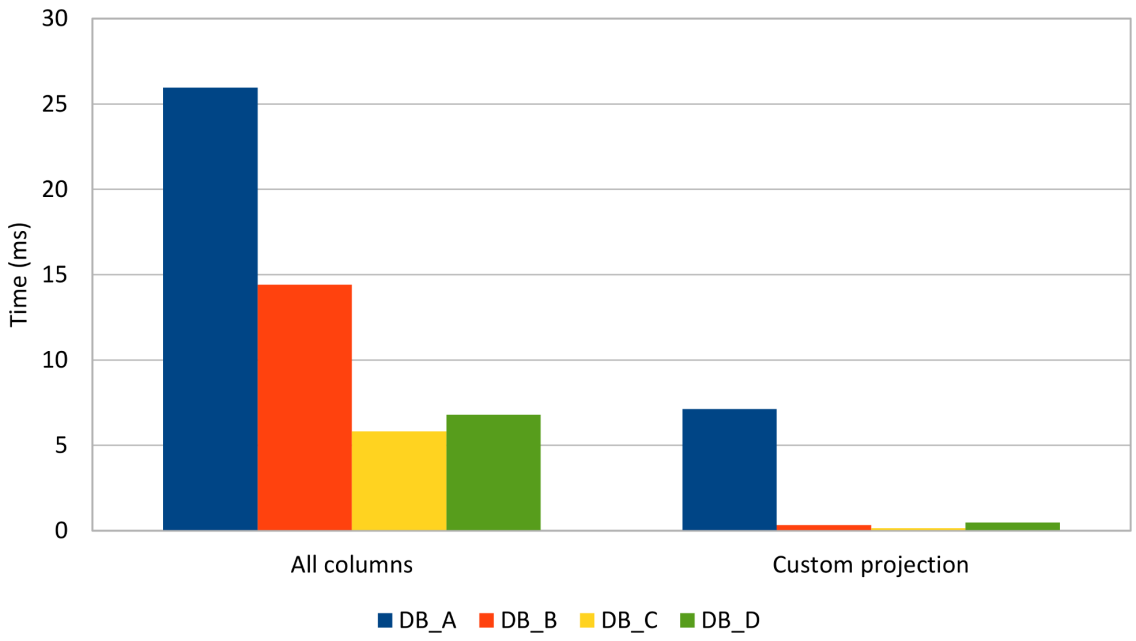


Figure 6.4: ResultSet projection size

This situation is more prevalent among ORM tools, as for populating entities entirely, all columns are needed to be selected. This might pass unnoticed when selecting just a few entities, but, for large result sets, this can turn into a noticeable performance issue.

If a business case requires only a subset of all entity properties, fetching extra columns becomes a waste of database and application resources (CPU, Memory, I/O, Networking).

7. Transactions

A database system must allow concurrent access to the underlying data. But shared data means that read and write operations must be synchronized to ensure that data integrity is not compromised.

To control concurrent modifications, the Java programming language defines the `synchronized` keyword for two purposes:

- it can restrict access to a shared `Object` (to preserve invariants), so only a `Thread` can execute a routine at any given time
- it propagates changes from the current `Thread` local memory to the global memory, that's available to all running threads of executions.

This behavior is typical for other concurrent programming environments and database systems are no different. In a relational database, the mechanism for ensuring data integrity is implemented on top of transactions.

A transaction is a collection of read and write operations that can either succeed or fail together, as a unit. All database statements must execute within a transactional context, even when the database client doesn't explicitly define its boundaries.

In 1981, Jim Gray has first defined the properties of a database transaction in his famous paper: The transaction concept: virtues and limitations[1]. Both this paper and the first versions of the SQL standard (SQL-86 and SQL-89) only used three properties for defining a database transaction: *Atomicity*, *Consistency* and Durability*.

Along with other relation database topics, the transaction research has continued ever since, and so the SQL-92 version introduced the concept of *Isolation Levels*. These four properties have been assembled in the well-known *ACID (Atomicity, Consistency, Isolation and Durability)* acronym that soon became synonym with relation database transactions.

Knowing how database transactions work is very important for two main reasons:

- effective data access (data integrity shouldn't be compromised when aiming for high-performance)
- efficient data access (reducing contention can minimize transaction response time which, in turn, increases throughput).

The next sections will detail each transaction property in relation to high-performance data processing.

[1] http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf

7.1 Atomicity

Atomicity is the property of grouping multiple operations into an all-or-nothing unit of work, which can succeed only if all individual operations succeed. For this reason, the database must be able to roll back all actions associated with every executed statement.

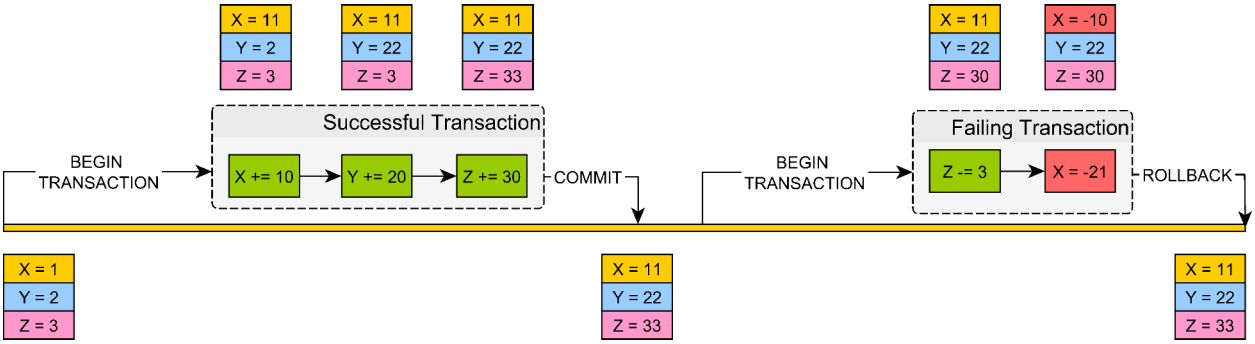


Figure 7.1: Atomic units of work

Write-write conflicts

Ideally, every transaction would have a completely isolated branch which could be easily discarded in case of a rollback. This scenario would be similar to how a *Version Control System* (e.g. *git*) implements branching. In case of conflicts, the Virtual Control System aborts the commit operation, and the client has to manually resolve the conflict. But unlike VCS tools, the relational database engine must manage conflicts without any human intervention.

For this reason, the database prevents write-write conflict situations and only one transaction can write a record at any given time.

All statements are executed against the actual data structures (tables, indexes, in-memory buffers), only to be materialized at commit time. In case of rollback, the database must revert any pending changed datum to its previous state.

Oracle

The *undo tablespace*[a] stores the previous data versions in *undo segments*. Upon rolling back, the database engine searches the associated *undo segments* that can recreate the *before image* of every datum that was changed by the current running transaction.

[a] https://docs.oracle.com/database/121/ADMIN/undo.htm#ADMIN11460

SQL Server

The *transaction log*[a] stores details about the current running transactions and their associated modifications. The rollback process scans the *transaction log* backward to find the associated undo records. When the record is found, the database engine restores the before image of the affected datum.

To prevent the transaction log from filling up, the log must be truncated on a regular basis. Long-running transactions can delay the truncation process, so that's another reason to avoid them as much as possible.

[a] https://msdn.microsoft.com/en-us/library/ms190925.aspx

PostgreSQL

Unlike other database systems, PostgreSQL doesn't use a dedicated append-only undo log. Because of its multi-version nature, every database object maintains its own version history. In the absence of the log seek-up phase, the rollback process becomes much lighter as it only requires to switch from one version to the other.

The downside is that the previous version space is limited in size, and so it must be reused. The process of reclaiming the storage occupied by old versions is called *VACUUMING*.

Each transaction has an associated XID[a] and newer transactions must have a greater XID number than all previous ones.

The transaction XID is a 32-bit number, so it can accommodate over 4 billion transactions. In a high-performance application, the transaction lifespan is very short, and if the *VACUUM* process is disabled, this threshold can be reached. When the XID counter reaches its maximum value, it wraps around and start again from zero.

The transactions issued prior to the XID wraparound will have their identifiers greater than newer transactions started after the XID counter reset. This anomaly can cause the system to perceive older transactions as they were started in the future, which can lead to very serious data integrity issues.

MySQL

The *undo log* is stored in the *rollback segment*[a] of the system tablespace.

Each *undo log* is split into two sections, one responsible for rolling back purposes and the other for reconstructing the before image. The first section can be wiped out right after the transaction is ended, while the other needs to linger for as long as any current running query or other concurrent transactions need to see a previous version of the records in question.

Behind the scenes, MySQL runs a *purge* process that cleans up the storage occupied by deleted records, and it also reclaims the undo log segments that are no longer required.

Long-running transactions delays the *purge* process execution, causing the undo log to grow very large, especially in write-heavy data access scenarios.

[a] https://dev.mysql.com/doc/refman/5.7/en/innodb-multi-versioning.html

7.2 Consistency

A modifying transaction can be seen like a state transformation, moving the database from one valid state to another. The relational database schema ensures that all primary modifications (insert/update/delete statements), as well as secondary ones (issued by triggers), obey certain rules on the underlying data structures:

- column types
- column length
- column nullability
- foreign key constraints
- unique key constraints
- custom *check* constraints.

Consistency is about validating the transaction state change, so that all committed transactions leave the database in a proper state. If only one constraint gets violated, the entire transaction is rolled back and all modifications are going to be reverted.

Although the application must validate user input prior to crafting database statements, the application-level checks cannot span over other concurrent requests, possibly coming from different web servers. When the database is the primary integration point, the advantages of a strict schema become even more apparent.

MySQL

Traditionally, MySQL constraints are not strictly enforced[a], and the database engine replaces invalid values with predefined defaults:

- out of range numeric values are set to either *0* or the maximum possible value
- `String` values are trimmed to the maximum length
- Incorrect data values are permitted (e.g. 2015-02-30)
- NOT NULL constraints are only enforced for single INSERT statements. For multi-row inserts, *0* replaces a null numeric values, and the empty value is used for a null `String`.

Since the 5.0.2 version, *strict* constraints are possible if the database engine is configured to use a custom *sql mode*[b]:

```
SET GLOBAL sql_mode='POSTGRESQL,STRICT_ALL_TABLES';
```

Because the `sql_mode` resets on server startup, it's better to set it up in the MySQL configuration file:

```
[mysqld]
sql_mode = POSTGRESQL,STRICT_ALL_TABLES
```

[a] https://dev.mysql.com/doc/refman/5.7/en/constraint-invalid-data.html
[b] https://dev.mysql.com/doc/refman/5.7/en/sql-mode.html

Consistency as in CAP Theorem

According to the CAP theorem[a], when a distributed system encounters a network partition, the system must choose either *Consistency* (all changes are instantaneously applied to all nodes) or *Availability* (any node can accept a request), but not both. While in the definition of ACID, consistency is about obeying constraints, in the CAP theorem context, consistency refers to *linearizability*[b], which is an isolation guarantee instead.

[a] https://en.wikipedia.org/wiki/CAP_theorem
[b] http://www.bailis.org/blog/linearizability-versus-serializability/

7.3 Isolation

If there were only one user accessing the database, there wouldn't be any risk of data conflicts. According to the Universal Scalability Law, if the sequential fraction of the data access patterns is less than *100%*, the database system can benefit from parallelization.

By offering multiple concurrent connections, the transaction throughput can increase and the database system can accommodate more traffic. But parallelization imposes additional challenges as the database must interleave transactions in such a way that conflicts don't compromise data integrity. The execution order of all the current running transaction operations is said to be *serializable* when its outcome is the same as if the underlying transactions were executed one after the other.

The serializable execution is therefore the only transaction isolation level that doesn't compromise data integrity, while allowing a certain degree of parallelization. In 1981, Jim Gray described the largest airlines and banks as having *10 000* terminals and *100* active transactions, which explains why, up until SQL-92, serializable was the *de facto* transaction isolation level.

7.3.1 Concurrency control

To manage data conflicts, several concurrency control mechanisms have been developed throughout the years. There are basically two strategies for handling data collisions:

- avoiding conflicts (e.g. *two-phase locking*) requires locking to control access to shared resources
- detecting conflicts (e.g. *Multi-Version Concurrency Control*) provides better concurrency, at the price of relaxing serializability and possibly accepting various data anomalies.

7.3.1.1 Two-phase locking

In 1976, Kapali Eswaran and Jim Gray (et al.) published The Notions of Consistency and Predicate Locks in a Database System[2] paper, which demonstrated that serializability can be obtained if all transactions use the *two-phase locking (2PL)* protocol.

Initially all database systems employed 2PL for implementing serializable transactions, but, with time, many vendors have moved towards an MVCC (Multi-Version Concurrency Control) architecture. By default, SQL Server still uses locking for implementing the Serializability isolation level.

Because 2PL guarantees transaction serializability, it's very important to understand the price of maintaining strict data integrity on the overall application scalability and transaction performance.

But locking isn't used only in 2PL implementations, and, to address both DML and DDL statement interaction and to minimize contention on shared resources, relational database systems use Multiple granularity locking[3].

[2] http://research.microsoft.com/en-us/um/people/gray/papers/On%20the%20Notions%20of%20Consistency%20and%20Predicate%20Locks%20in%20a%20Database%20System%20CACM.pdf

[3] https://en.wikipedia.org/wiki/Multiple_granularity_locking

Database objects are hierarchical in nature, a logical tablespace being mapped to multiple database files, which are built of data pages, each page containing multiple rows. For this reason, locks can be acquired on different database object types.

Locking on lower-levels (e.g. rows) can offer better concurrency as it reduces the likelihood of contention. Because each lock takes resources, holding multiple lower-level locks can add up, so the database might decide to substitute multiple lower-level locks into a single upper-level one. This process is called *lock escalation* and it trades off concurrency for database resources.

Each database system comes with its own lock hierarchy but the most common types (even mentioned by the 2PL initial paper) remain the following ones:

- shared (read) lock, preventing a record from being written while allowing concurrent reads
- exclusive (write) lock, disallowing both read and write operations.

Locks alone are not sufficient for preventing conflicts. A concurrency control strategy must define how locks are being acquired and released because this also has an impact on transaction interleaving.

For this purpose, the 2PL protocol defines a lock management strategy for ensuring serializability. The 2PL protocol splits a transaction in two sections:

- expanding phase (locks are acquired and no lock is released)
- shrinking phase (all locks are released and no other lock is further acquired).

In a locking-based concurrency control implementation, all currently interleaved transactions must follow the 2PL protocol as otherwise serializability might be compromised, resulting in data anomalies.

Transaction schedule

To provide recovery from failures, the transaction schedule (the sequence of all interleaved operations) must be strict. If a write operation, in a first transaction, happens before a conflict occurring in a subsequent transaction, in order to achieve transaction *strictness*, the first transaction commit event must also happen before the conflict.

Because operations are properly ordered, strictness can prevent *cascading aborts (one transaction rollback triggering a chain of other transaction aborts, to preserve data consistency)*. Releasing all locks only after the transaction has ended (either commit or rollback) is a requirement for having a strict schedule.

The following diagram shows how transaction interleaving is coordinated by 2PL:

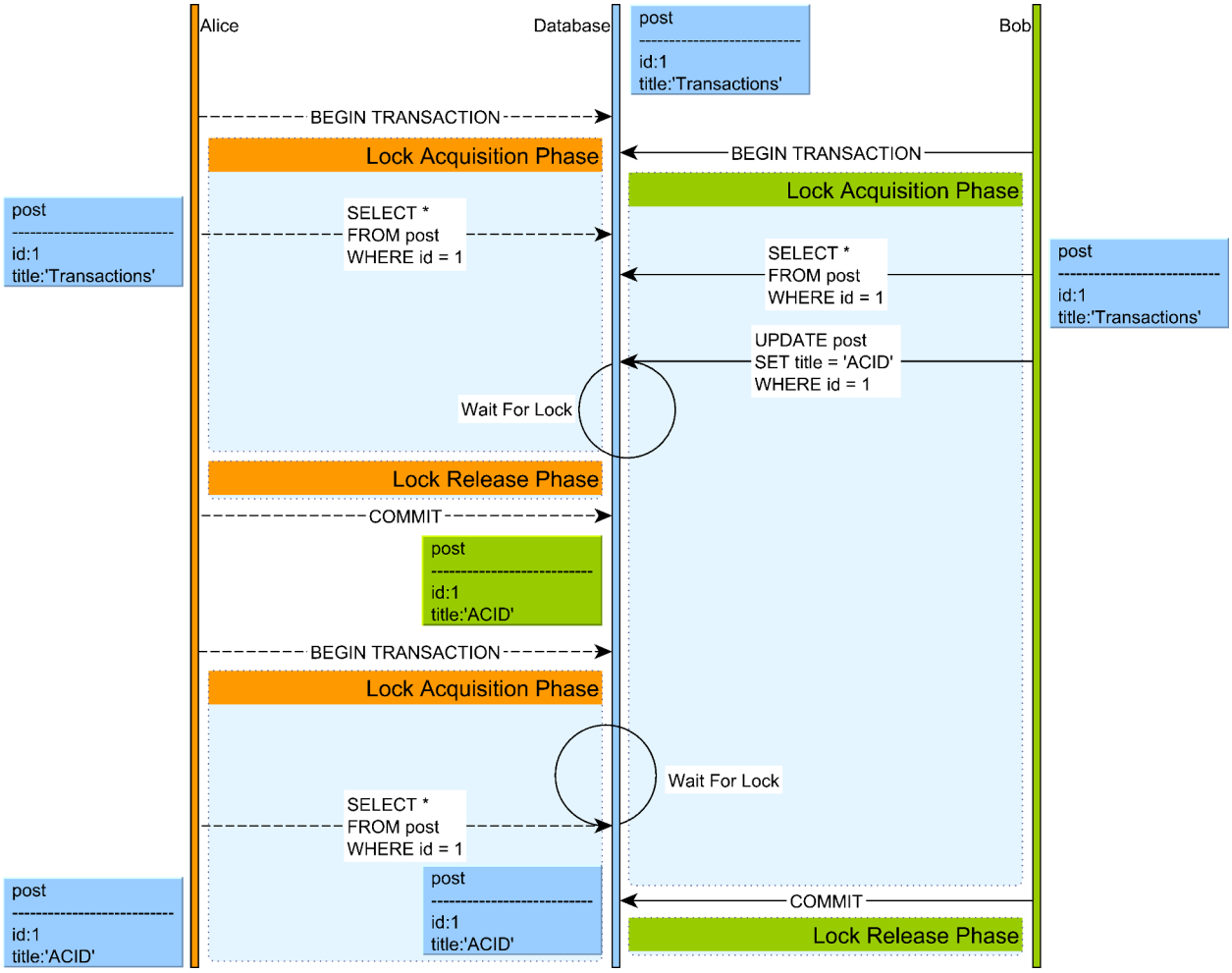


Figure 7.2: Two-phase locking

- both *Alice* and *Bob* select a *post* record, both acquiring a shared lock on this record
- when *Bob* attempts to update the *post* entry, his statement is blocked by the Lock Manager because *Alice* is still holding a shared lock on this database row
- only after *Alice's* transaction ends and all locks are being released, *Bob* can resume his update operation
- *Bob's* update will generate a lock upgrade, so the shared lock is replaced by an exclusive lock, which will prevent any other concurrent read or write operation
- *Alice* starts a new transaction and issues a select query for the same *post* entry, but the statement is blocked by the Lock Manager since *Bob* owns an exclusive lock on this record
- after *Bob's* transaction is committed, all locks are released and *Alice's* query can be resumed, so she will get the latest value of this database record.

Deadlocks

Using locking for controlling access to shared resources is prone to deadlocks, and the transaction scheduler alone cannot prevent their occurrences.

A deadlock happens when two concurrent transactions cannot make progress because each one waits for the other to release a lock. Because both transactions are in the lock acquisition phase, neither one will release a lock prior to acquiring the next one.

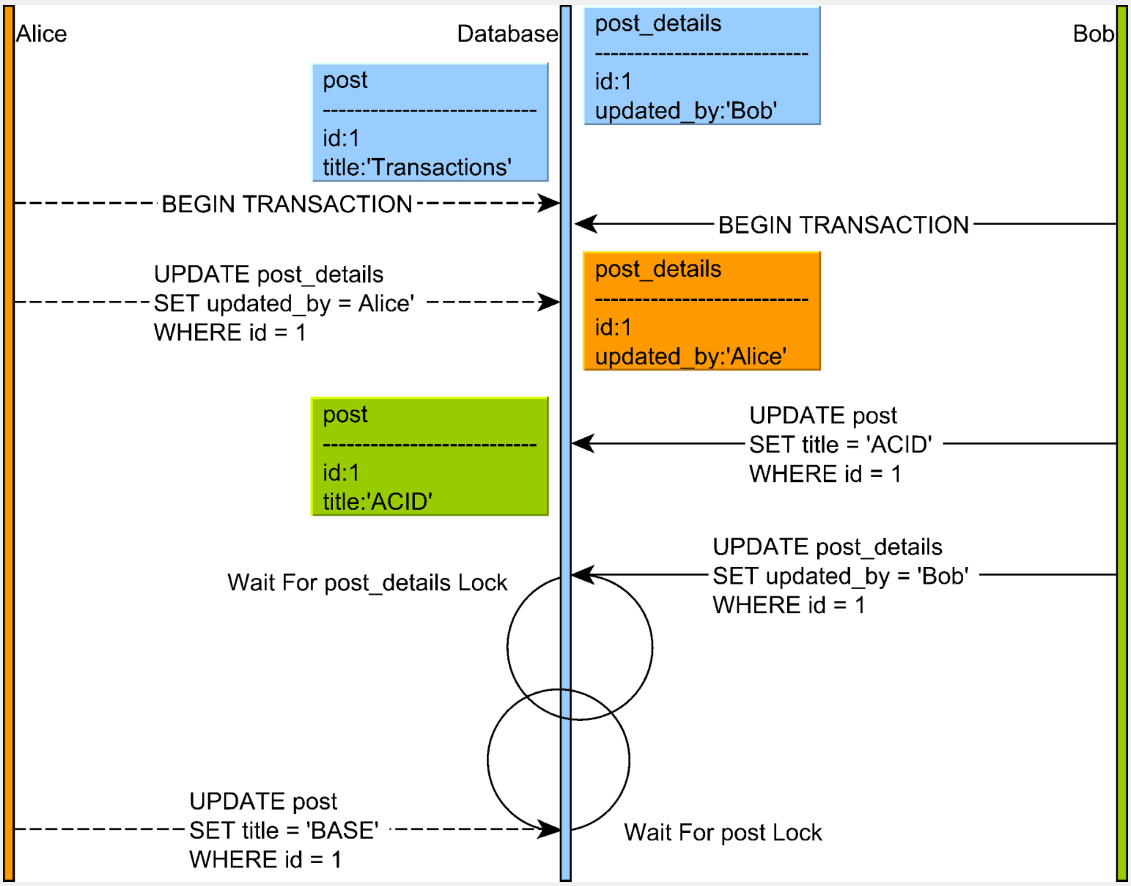


Figure 7.3: Dead lock

Preserving the lock order becomes the responsibility of the data access layer, and the database can only assist in recovering from a deadlock situation.

The database engine runs a separate process that scans the current conflict graph for lock-wait cycles (which are caused by deadlocks). When a cycle is detected, the database engine picks one transaction and aborts it, causing its locks to be released, so the other transaction can make progress.

7.3.1.2 Multi-Version Concurrency Control

Although locking can provide a serializable transaction schedule, the cost of lock contention can undermine both transaction response time and scalability. The response time can increase because transactions must wait for locks to be released, and long-running transactions can slow down the progress of other concurrent transactions as well. According to both Amdahl's Law and the Universal Scalability Law, concurrency is also affected by contention.

To address these shortcomings, the database vendors have opted for optimistic concurrency control mechanisms. If 2PL prevents conflicts, Multi-Version Concurrency Control (MVCC) uses a conflict detection strategy instead.

The promise of MVCC is that readers don't block writers and writers don't block readers. The only source of contention comes from writers blocking other concurrent writers, which otherwise would compromise transaction rollback and atomicity.

To prevent blocking, the database can rebuild previous versions of a database record, so an uncommitted change can be hidden away from incoming concurrent readers. The lack of locking makes it more difficult to implement a serializable schedule, so the database engine must analyze the current interleaving operations and detect anomalies that would compromise serializability.

Oracle

Oracle doesn't implement 2PL at all, relying on MVCC mechanism for managing concurrent data access. Every query gets a point in time data snapshot and, depending on the isolation level, the timestamp reference can be relative to the current statement or to the current transaction start time.

To rebuild previous record versions, Oracle uses the undo segments[a], which already contain all the necessary data required for rolling back an uncommitted change. The point in time is based on the System Change Number (SCN), which is a logical timestamp reference and, unlike physical time, it is guaranteed to be incremented monotonically.

Apart from MVCC, Oracle also supports explicit locking as well, using the `SELECT FOR UPDATE` SQL syntax.

[a] https://docs.oracle.com/database/121/CNCPT/consist.htm#CNCPT221

SQL Server

By default, SQL Server uses locks for implementing all the isolation levels stipulated by the SQL standard.

For the *Read Committed* isolation level to take advantage of the MVCC model, the following configuration must be first set:

```
ALTER DATABASE high_performance_java_persistence
SET READ_COMMITTED_SNAPSHOT ON;
```

For a higher-level isolation, SQL Server offers the *Snapshot* isolation mode, which must be activated at the database level:

```
ALTER DATABASE high_performance_java_persistence
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Because Snapshot is a custom isolation level, it must also be set at the connection level prior to starting a new transaction:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO
BEGIN TRANSACTION;
GO
COMMIT TRANSACTION;
GO
```

After enabling row versioning, the database can track record changes in the *tempdb* database.

When a row is either updated or deleted, the current row entry will hold a reference back to the previous version, which is recorded in the version store, in the tempdb database. Rows are not deleted right away but only marked for deletion, the actual removal being done by the *Ghost cleanup task*. Old versions must be kept for as long as a current running transaction might need them, which is specified by the transaction isolation level.

The Ghost cleanup task runs periodically and reclaims storage from old versions that are no longer necessary.

A long-running transaction would require the database engine to keep some old version for a very long time, and, because version changes are chained in a linked list structure, restoring previous version might becomes resource intensive.

PostgreSQL

Unlike all other database systems, PostgreSQL stores both the current rows and their previous versions (even the ones for the aborted transactions) in the actual database table. Like Oracle, PostgreSQL embraces the MVCC data access model, and it doesn't offer a 2PL transaction isolation implementation at all.

Each table row has two additional columns (*xmin* and *xmax*), which are used to control the visibility of various row versions. When a row is inserted, the current transaction identifier is stored in the xmin column.

Both the update and the delete operations end up creating a new row entry with an xmax column storing the current transaction identifier.

The *Vacuum* cleaner process runs regularly and reclaims storage occupied by deleted entries (and successfully committed) or by previous versions that are no longer required by the current running transactions.

Although PostgreSQL is seen as a pure MVCC model, locking is still required to prevent write-write conflicts or for explicit locking[a]. `SELECT FOR UPDATE` is used to acquire an exclusive row-level lock, while `SELECT FOR SHARE` is for applying a shared lock instead.

MySQL

The InnoDB storage engine offers support for ACID transactions and uses MVCC for controlling access to shared resources. The InnoDB MVCC implementation is very similar to Oracle, and previous versions of database rows are stored in the rollback segment as well.

When a transaction demands a previous row version, MySQL must reconstruct it from rollback segments. Delete operations just mark an entry as being ready for deletion, and the *purge* thread is going to do the actual physical cleanup.

Both the transaction rollback and the previous row version restoring processes (required by a given transaction visibility guarantees) are very much the same thing.

Like other database systems, MySQL also offers explicit locking[a] for when MVCC is no longer satisfactory. A shared lock is acquired using `SELECT LOCK IN SHARE MODE`, while for exclusive locks the much more common `SELECT FOR UPDATE` syntax is being used.

7.3.2 Phenomena

For reasonable transaction throughput values, it makes sense to imply transaction *serializability*. As the incoming traffic grows, the price for strict data integrity becomes too high, and this is the primary reason for having multiple isolation levels. Relaxing serializability guarantees may generate data integrity anomalies, which are also referred as *phenomena*.

The SQL-92 standard introduced three phenomena that can occur when moving away from a serializable transaction schedule:

- *dirty read*
- *non-repeatable read*
- *phantom read.*

In reality, there are other phenomena that can occur due to transaction interleaving, as the famous paper *A Critique of ANSI SQL Isolation Levels*[4] describes:

- *dirty write*
- *read skew*
- *write skew*
- *lost update.*

> Choosing a certain isolation level is a trade-off between increasing concurrency and acknowledging the possible anomalies that might occur.
>
> Scalability is undermined by contention and coherency costs. The lower the isolation level, the less locking (or multi-version transaction abortions), and the more scalable the application will get.
>
> But a lower isolation level allows more phenomena, and the data integrity responsibility is shifted from the database side to the application logic, which must ensure that it takes all measures to prevent or mitigate any such data anomaly.

Before jumping to isolation levels, it's better to understand what's behind each particular phenomenon and how it can affect data integrity. When choosing a given transaction isolation level, understanding phenomena becomes fundamental to taking the right decision,

[4] http://research.microsoft.com/apps/pubs/default.aspx?id=69541

7.3.2.1 Dirty write

A dirty write happens when two concurrent transactions are allowed to modify the same row at the same time. As previously mentioned, all changes are applied against the actual database object structures, which means that the second transaction simply overwrites the first transaction pending change.

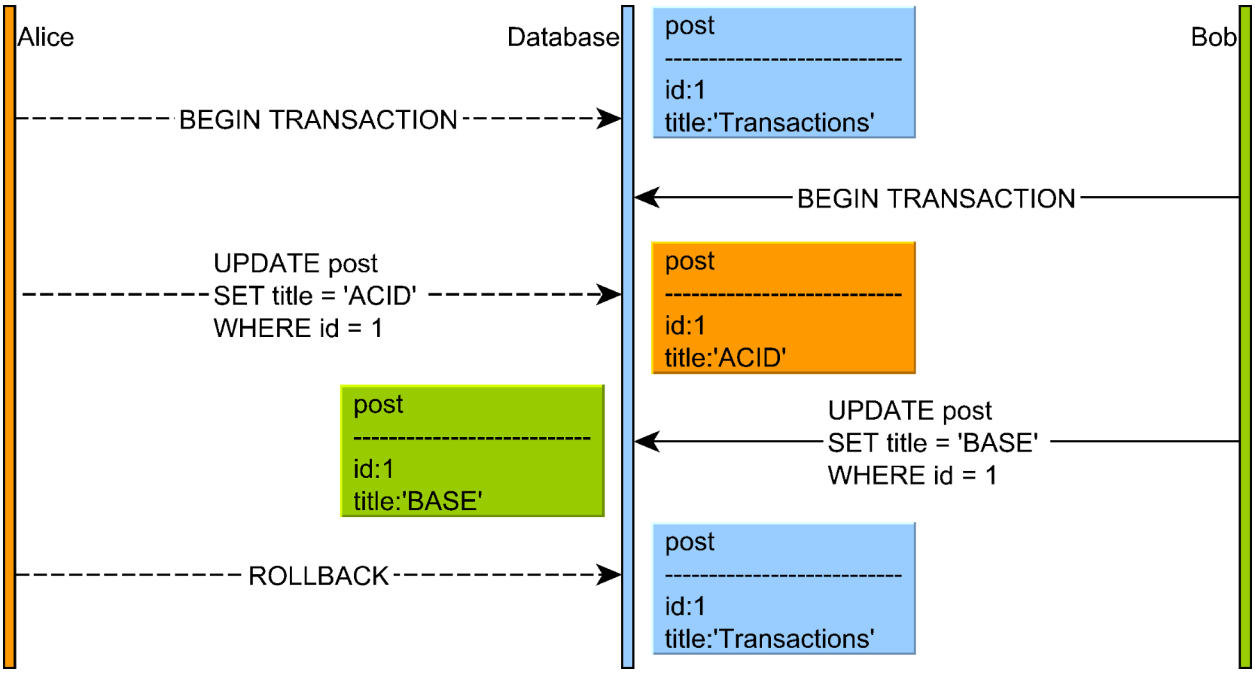


Figure 7.4: Dirty write

If the two transactions commit, one transaction will silently overwrite the other transaction, causing a *lost update*. Another problem arises when the first transaction wants to roll back. The database engine would have to choose one of the following action paths:

- it can restore the row to its previous version (as it was before the first transaction changed it), but then it will overwrite the second transaction uncommitted change
- it can acknowledge the existence of a newer version (issued by the second transaction), but then, if the second transaction has to roll back, it's previous version becomes the uncommitted change of the first transaction.

If the database engine didn't prevent dirty writes, guaranteeing rollbacks would not be possible. Because atomicity cannot be implemented in the absence of reliable rollbacks, all database systems must therefore prevent dirty writes.

Although the SQL standard doesn't mention this phenomenon, even the lowest isolation level (*Read Uncommitted*) is able to prevent it.

7.3.2.2 Dirty read

As previously mentioned, all database changes are applied against the actual data structures (memory buffers, data blocks, indexes). A dirty read happens when a transaction is allowed to read the uncommitted changes of some other concurrent transaction. Taking a business decision on a value that hasn't been committed is risky because uncommitted changes might get rolled back.

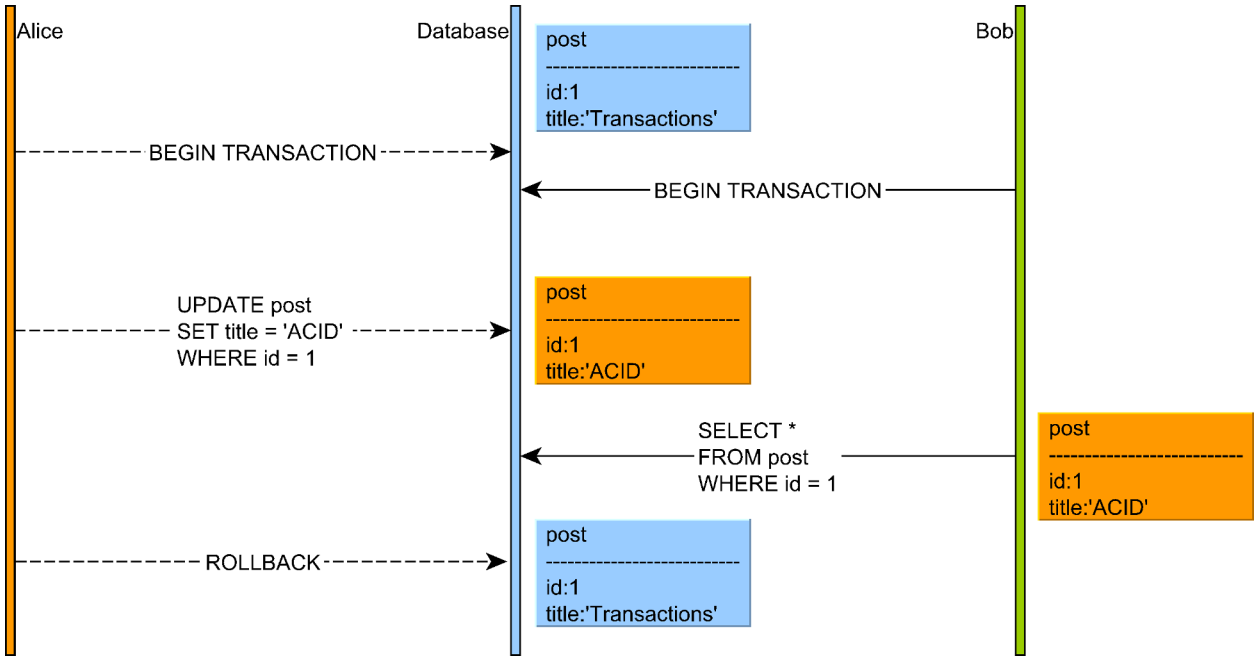


Figure 7.5: Dirty read

This anomaly is only permitted by the Read Uncommitted isolation level, and, because of the serious impact on data integrity, most database systems offer a higher default isolation level.

To prevent dirty reads, the database engine must hide uncommitted changes from all the concurrent transactions (but the one that authored the change). Each transaction is allowed to see its own changes because otherwise the read-your-own-writes consistency guarantee is compromised.

A naive approach would be to lock uncommitted rows but this wouldn't be practical at all (if a long-running transaction acquired such a lock, no other transaction would be able to read that record until the lock is released). Locks incur contention and contention affects scalability.

Since the undo log already captures the previous version of every uncommitted record, the database engine can use it to restore the previous value in other concurrent transaction queries. Because this mechanism is used by all other isolation levels (Read Committed, Repeatable Read, Serializable), most database systems optimize the before image restoring process (lowering its overhead on the overall application performance).

Cases for using Read Uncommitted are seldom (non-strict reporting queries where dirty reads are acceptable), so Read Committed is usually the lowest practical isolation level.

7.3.2.3 Non-repeatable read

If one transaction reads a database row without applying a shared lock on the newly fetched record, then a concurrent transaction might change this row before the first transaction has ended.

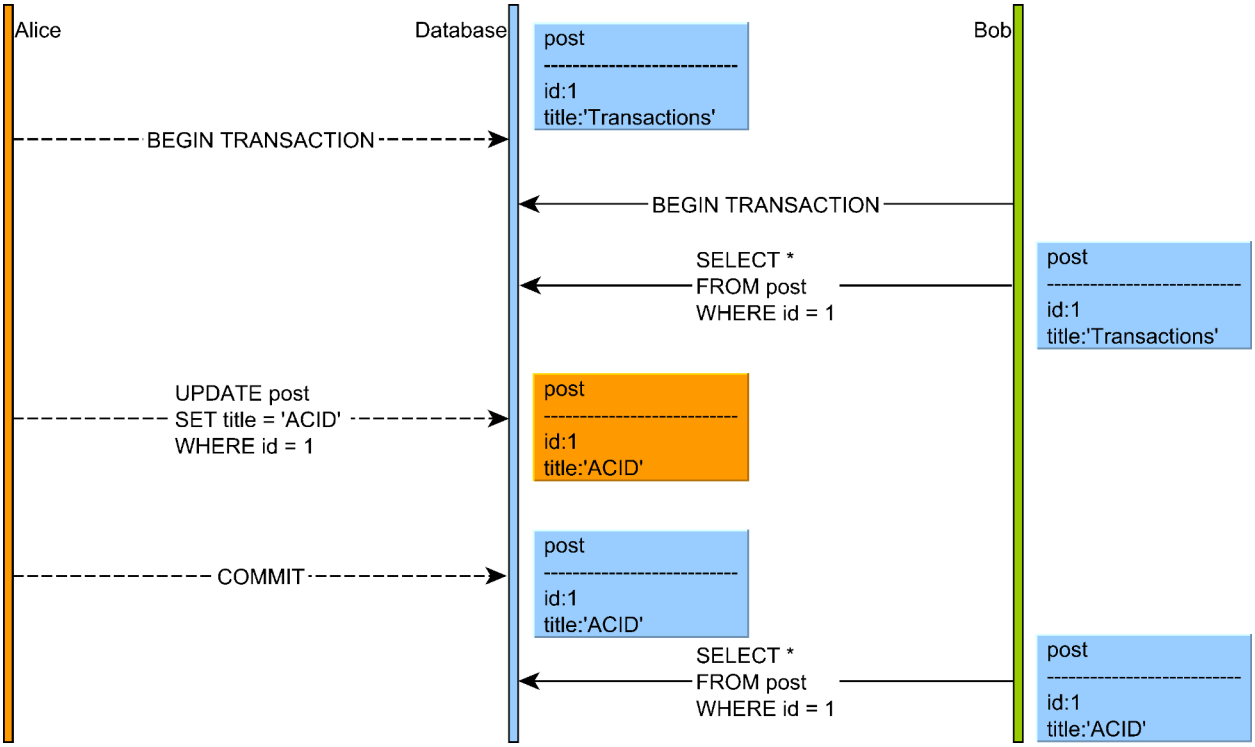


Figure 7.6: Non-repeatable read

This phenomenon is problematic when the current transaction makes a business decision based on the first value of the given database row (a client might order a product based on a stock quantity value that's no longer a positive integer).

Most database systems have moved to a Multi-Version Concurrency Control model, and shared locks are no longer mandatory for preventing non-repeatable reads. By verifying the current row version, a transaction can be aborted if a previously fetched record has changed in the meanwhile.

Repeatable Read and Serializable prevent this anomaly by default. With Read Committed, it's possible to avoid non-repeatable (fuzzy) reads if the shared locks are acquired explicitly (e.g. `SELECT FOR SHARE`).

Some ORM frameworks (e.g. JPA/Hibernate) offer application-level repeatable reads. The first snapshot of any retrieved entity is cached in the current running *Persistence Context*. Any successive query returning the same database row is going to use the very same object that was previously cached. This way, the fuzzy reads may be prevented even in Read Committed isolation level.

7.3.2.4 Phantom read

If a transaction makes a business decision based on a set of rows satisfying a given predicate, without predicate locking, a concurrent transaction might insert a record matching that particular predicate.

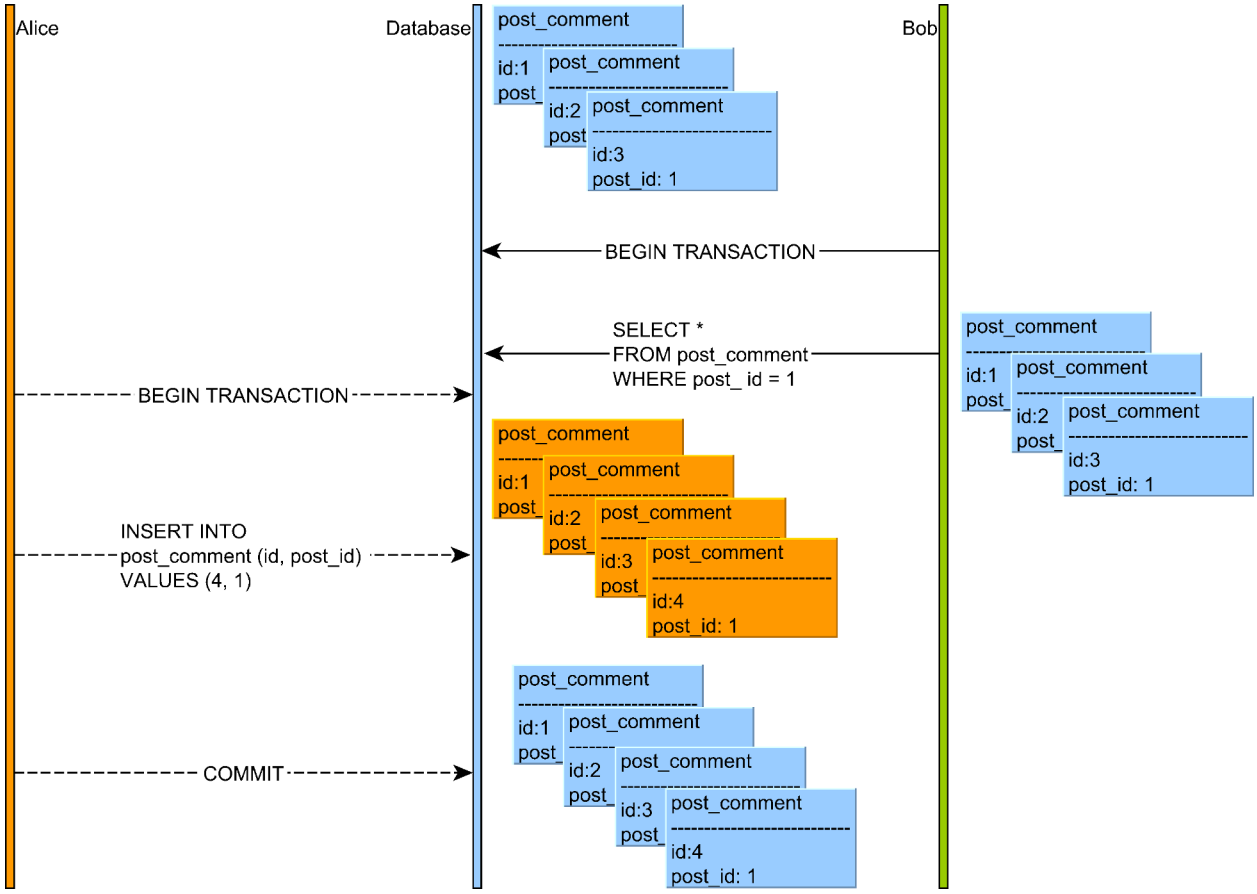


Figure 7.7: Phantom read

Phantom rows can lead a buyer into purchasing a product without being aware of a better offer that was added right after the user has finished fetching the offer list.

Traditionally, the Serializable isolation prevented phantom reads through predicate locking. Other MVCC implementations can detect phantom rows by introspecting the transaction schedule and aborting any transaction whose serializability guarantees were violated.

If explicit locking could prevent non-repeatable reads even in Read Committed isolation level, there isn't such SQL construct to permit range locks to be acquired explicitly.

7.3.2.5 Read skew

Read skew is a lesser known anomaly that involves a constraint on more than one database tables. In the following example, the application requires the *post* and the *post_details* be updated in sync. Whenever a *post* record changes, its associated *post_details* must register the user who made the current modification.

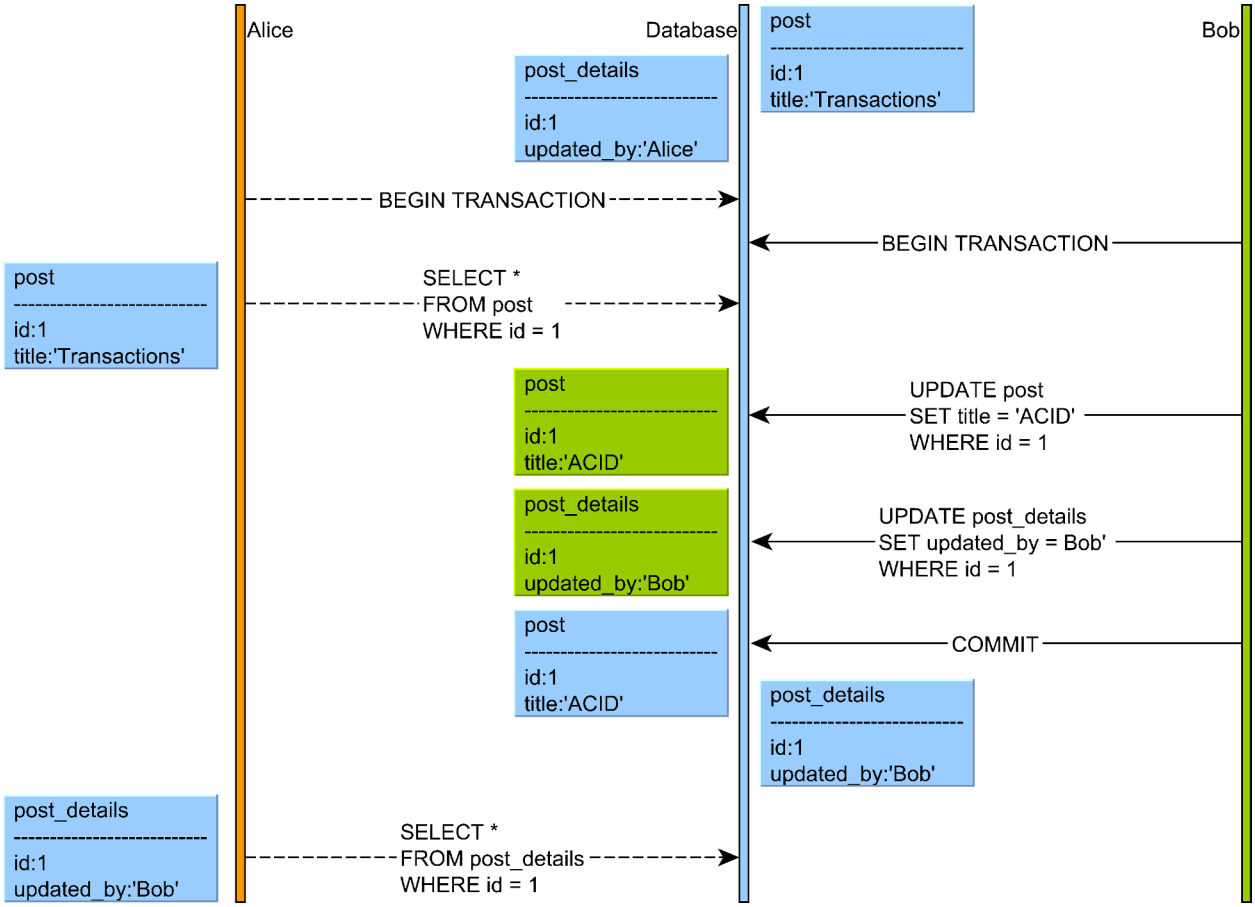


Figure 7.8: Read skew

In between selecting the *post* and the *post_details* rows, a second transaction sneaks in and manages to update both records. The first transaction will see an older version of the *post* row and the latest version of the associated *post_details*. Because of this read skew, the first transaction will assume that this particular *post* was updated by *Bob*, although, in fact, it is an older version updated by *Alice*.

Like with non-repeatable reads, there are two ways to avoid this phenomenon:

- the first transaction can acquire shared locks on every read, therefore preventing the second transaction from updating these records
- the first transaction can be aborted upon validating the commit constraints (when using an MVCC implementation of the Repeatable Read or Serializable isolation levels).

7.3.2.6 Write skew

Like read skew, this phenomenon involves disjoint writes over two different tables that are constrained to be updated as a unit. Whenever the *post* row changes, the client must update the *post_details* with the user making the change.

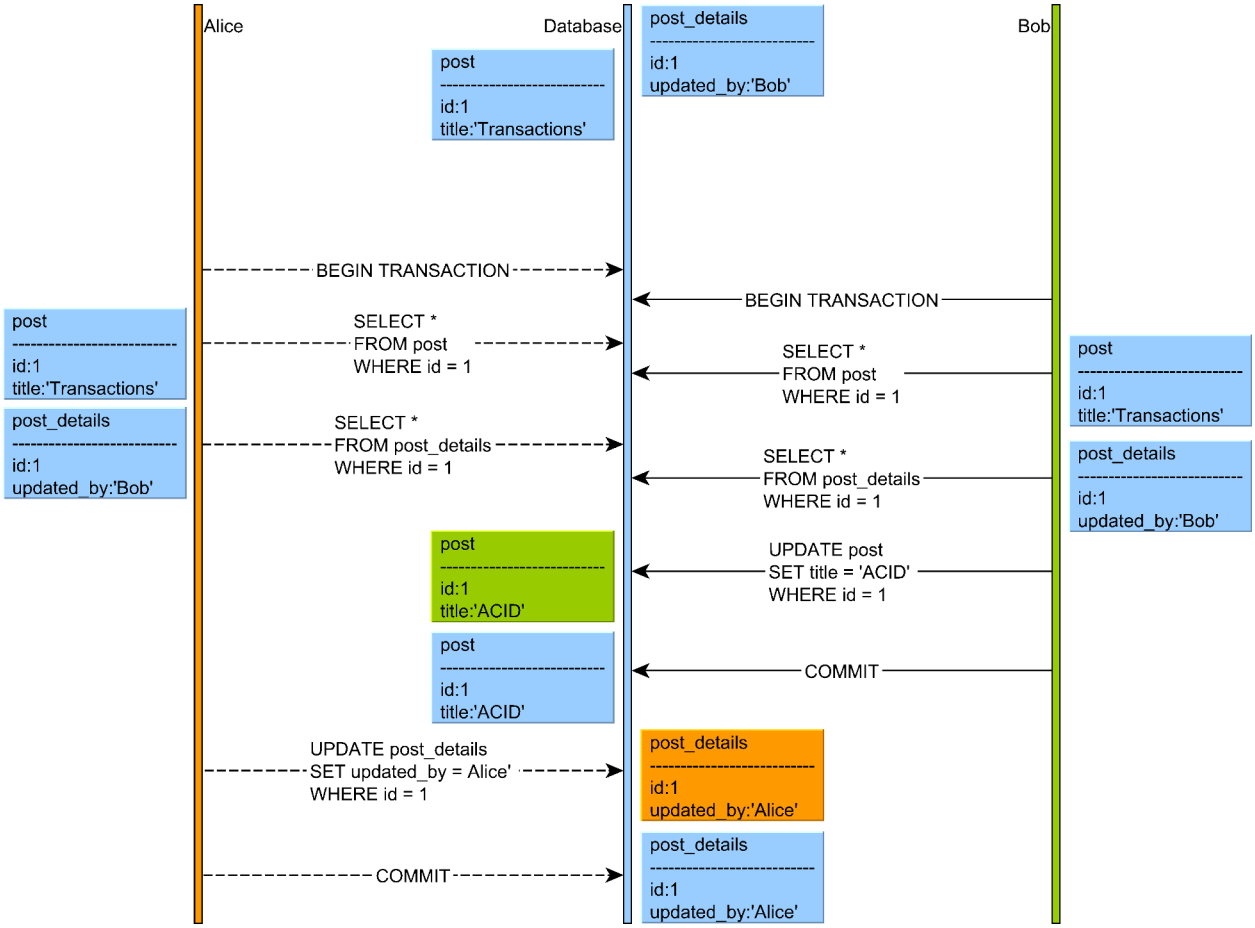


Figure 7.9: Write skew

Both *Alice* and *Bob* will select the *post* and its associated *post_details* record. If write skew is allowed, *Alice* and *Bob* can update these two records separately, therefore breaking the constraint.

Like with non-repeatable reads, there are two ways to avoid this phenomenon:

- the first transaction can acquire shared locks on both entries, therefore preventing the second transaction from updating these records
- the database engine can detect that another transaction has changed these records, and so it can force the first transaction to roll back (under an MVCC implementation of Repeatable Read or Serializable).

7.3.2.7 Lost update

This phenomenon happens when a transaction reads a row while another transaction modifies it prior to the first transaction to finish. In the following example, *Bob's* update is silently overwritten by *Alice*, who is not aware of the record update.

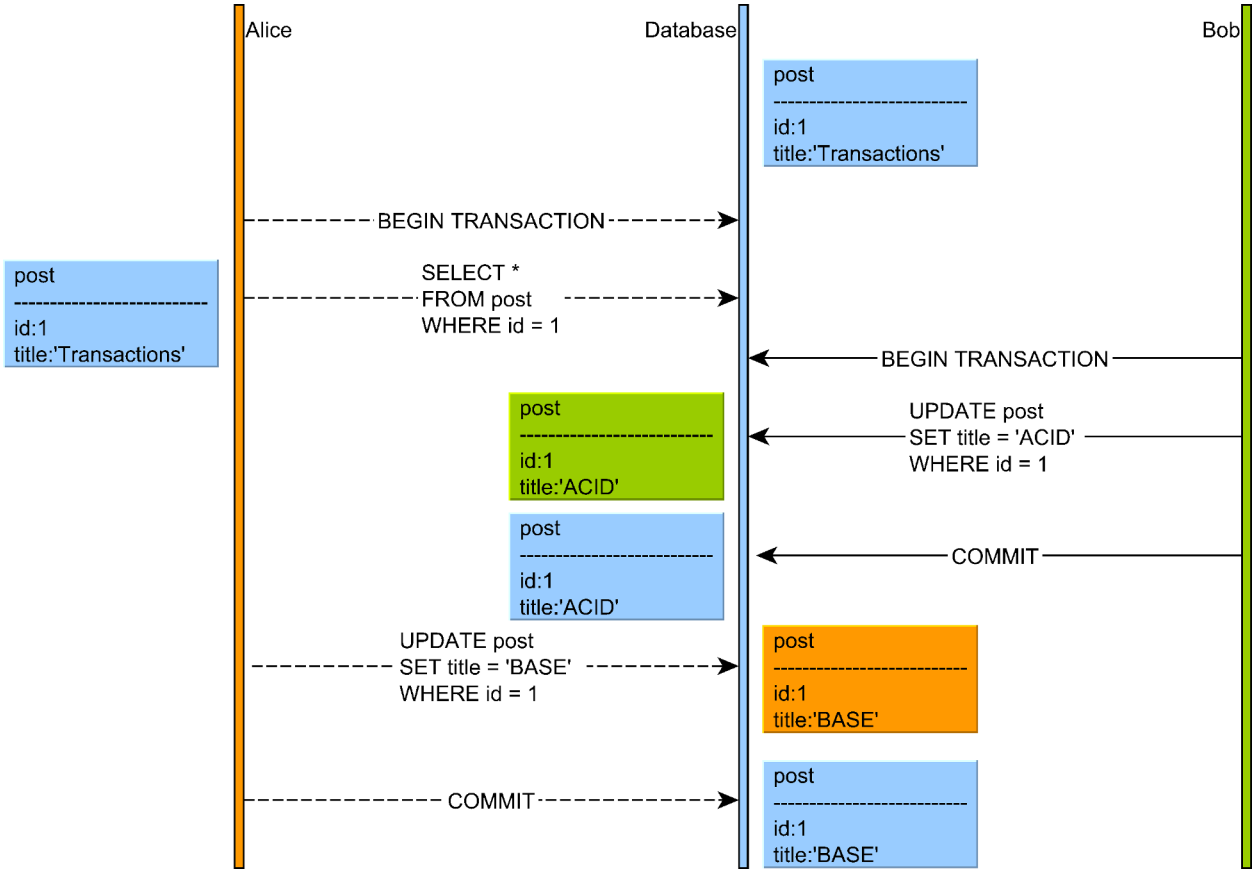


Figure 7.10: Lost update

This anomaly can have serious consequences on data integrity (a buyer might purchase a product without knowing the price has just changed), especially because it affects Read Committed, the default isolation level in many database systems.

Traditionally, Repeatable Read protected against lost updates since the shared locks could prevent a concurrent transaction from modifying an already fetched record. With MVCC, the second transaction is allowed to make the change, while the first transaction is aborted when the database engine detects the row version mismatch (during the first transaction commit).

Most ORM tools, such as Hibernate, offer application-level optimistic locking, which automatically integrates the row version whenever a record modification is issued. On a row version mismatch, the update count is going to be zero, so the application can roll back the current transaction, as the current data snapshot is stale.

7.3.3 Isolation levels

As previously stated, Serializable is the only isolation level to provide a truly ACID transaction interleaving. But serializability comes at a price as locking introduces contention, which, in turn, limits concurrency and scalability. Even in multi-version concurrency models, serializability may require aborting too many transactions that are affected by phenomena.

For this purpose, the SQL-92 version introduced multiple isolation levels, and the database client has the option of balancing concurrency against data correctness. Each isolation level is defined in terms of the minimum number of phenomena that it must prevent, and so the SQL standard introduces the following transaction isolation levels:

Table 7.1: Standard isolation levels

Isolation Level	Dirty read	Non-repeatable read	Phantom read
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

Without an explicit setting, the JDBC driver uses the default isolation level, which can be introspected using the `getDefaultTransactionIsolation()`[5] method of the `DatabaseMetaData` object:

```
int level = connection.getMetaData().getDefaultTransactionIsolation();
```

The default isolation level can be changed using the `setTransactionIsolation(int level)`[6] `Connection` method.

```
connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

> Even if ACID properties imply a serializable schedule, most relational database systems use a lower default isolation level instead:
>
> - Read Committed (Oracle, SQL Server, PostgreSQL)
> - Repeatable Read (MySQL).

The following sections will go through each particular transaction isolation level and demonstrate the actual list of phenomena that are prevented by a given database system.

[5] http://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html#getDefaultTransactionIsolation--

[6] http://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#setTransactionIsolation-int-

7.3.3.1 Read Uncommitted

Table 7.2: Read Uncommitted phenomena occurrence

Phenomena	SQL Server	PostgreSQL	MySQL
Dirty Write	No	No	No
Dirty Read	Yes	No	Yes
Non-Repeatable Read	Yes	Yes	Yes
Phantom Read	Yes	Yes	Yes
Read Skew	Yes	Yes	Yes
Write Skew	Yes	Yes	Yes
Lost Update	Yes	Yes	Yes

Oracle

Dirty reads are not allowed, and so the lowest isolation level is Read Committed.

The JDBC driver even throws an exception if the client tries to set the Read Uncommitted isolation on the current `Connection`.

SQL Server

Read Uncommitted only protects against dirty writes, all other phenomena being allowed. When using Read Uncommitted, there is no exclusive lock associated with a given SQL modification, so uncommitted changes are available to other concurrent transactions even before they get committed. If the risk of dirty reads can be assumed, avoiding exclusive locks can speed up reporting queries, especially when scanning large amounts of data.

For locking-based concurrency control mechanisms, Read Uncommitted is worth considering if the risk of dirty reads is a much smaller issue than locking a large portion of a database table. Because MVCC avoids reader-writer and writer-reader locking, it might not exhibit a considerable performance enhancement from permitting dirty reads.

PostgreSQL

Like Oracle, PostgreSQL doesn't allow dirty reads, the lowest isolation level being Read Committed.

When choosing Read Uncommitted, the JDBC driver silently falls back to Read Committed.

MySQL

Although it uses MVCC, InnoDB implements Read Uncommitted so that dirty reads are permitted. As an optimization, each query is spared from rebuilding the previous committed versions (using the rollback segments) of the currently scanned records (in case they have been recently modified).

7.3.3.2 Read Committed

Read Committed is one of the most common isolation level, and it behaves consistently across multiple relational database systems or various concurrency control models.

Many database systems choose Read Committed as the default isolation level because it delivers the best performance, while preventing fatal anomalies such as dirty writes and dirty reads. But performance has its price as Read Committed permits many anomalies that might lead to data corruption.

Table 7.3: Read Committed phenomena occurrence

Phenomena	Oracle	SQL Server	SQL Server MVCC	PostgreSQL	MySQL
Dirty Write	No	No	No	No	No
Dirty Read	No	No	No	No	No
Non-Repeatable Read	Yes	Yes	Yes	Yes	Yes
Phantom Read	Yes	Yes	Yes	Yes	Yes
Read Skew	Yes	Yes	Yes	Yes	Yes
Write Skew	Yes	Yes	Yes	Yes	Yes
Lost Update	Yes	Yes	Yes	Yes	Yes

Oracle

Every statement has a start timestamp, which is used to create a database snapshot relative to this particular point in time. This way, writers can still update the current selected records, and the database can simply reconstruct the previous versions that were available when the query started. Subsequent query executions can return different row versions, so non-repeatable reads are permitted.

When two transactions attempt to update the same record, the first one will lock the record to prevent dirty writes. The second transaction must wait until the first transaction releases the lock (either commit or rollback), and the statement filtering criteria is reevaluated against latest data.

PostgreSQL

Like Oracle, every query sees a database snapshot as of the beginning of the current running query. Because shared locks are not used to protect previously read records from being modified, Read Committed allows a large spectrum of data anomalies.

Exclusive locks prevent write-write conflicts, so when two transactions update the same record, the second one waits for the first transaction to release its locks. When the second transaction resumes its execution, if the filtering criteria is still relevant, it might overwrite the first transaction modifications, therefore causing lost updates.

MySQL

Query-time snapshots are used to isolate statements from other concurrent transactions. When explicitly acquiring shared or exclusive locks or when issuing update or delete statements (which acquire exclusive locks to prevent dirty writes), if the selected rows are filtered by unique search criteria (e.g. primary key), the locks can be applied to the associated index entries.

Prior to 5.7[a], if the modifying statements used a range filter and the search criteria took advantage of a unique index scan, then the database could use a gap or a next-key lock (therefore protecting against phantom reads as well). Statement-based replication is not available for Read Committed, so the application must use the row-based binary logging instead.

SQL Server

By default, SQL statements use shared locks to prevent other transactions from modifying the currently fetched records. The locks are released by the time the query finishes executing. When activating Read Committed Snapshot Isolation, the database will not use shared locks anymore, and each query will select the row version as it was when the query started.

7.3.3.3 Repeatable Read

One of the least compliant isolation levels, Repeatable Read implementation details leak into its phenomena prevention spectrum:

Table 7.4: Repeatable Read phenomena occurrence

Phenomena	SQL Server	PostgreSQL	MySQL
Dirty Write	No	No	No
Dirty Read	No	No	No
Non-Repeatable Read	No	No	No
Phantom Read	Yes	No	No
Read Skew	No	No	No
Write Skew	No	Yes	Yes
Lost Update	No	No	Yes

Oracle

The Repeatable Read isolation is not supported at all, and the JDBC driver throws an exception if the client tries to set it explicitly.

SQL Server

For every row the client reads, the current transaction acquires a shared lock that prevents any other transaction from concurrently modifying it. The shared locks are released when the transaction either commits or rolls back.

PostgreSQL

The Repeatable Read is implemented using Snapshot Isolation[a], so not only fuzzy reads are prevented but even phantom reads are prohibited as well. Instead of using locking, the PostgreSQL MVCC implementation allows conflicts to occur, but it will abort any transaction whose guarantees don't hold anymore.

[a] https://en.wikipedia.org/wiki/Snapshot_isolation

MySQL

Every transaction can only see rows as if they were when the current transaction started. This prevents non-repeatable reads, but it still allows lost updates and write skews.

7.3.3.4 Serializable

Serializable is supposed to provide a transaction schedule, whose outcome, even in spite of statement interleaving, is equivalent to a serial execution.

Even if the concurrency control mechanism is locking-based or it manages multiple record versions, it must prevent all phenomena to ensure serializable transactions. Preventing all phenomena mentioned by the SQL standard (dirty reads, non-repeatable reads and phantom reads) is not enough, and Serializable must protect against lost update, read skew and write skew as well.

In practice, the concurrency control implementation details leak, and not all relational database systems provide a truly Serializable isolation level (some data integrity anomalies might still occur).

Table 7.5: Serializable phenomena occurrence

Phenomena	Oracle	SQL Server	SQL Server MVCC	PostgreSQL	MySQL
Dirty Write	No	No	No	No	No
Dirty Read	No	No	No	No	No
Non-Repeatable Read	No	No	No	No	No
Phantom Read	No	No	No	No	No
Read Skew	No	No	No	No	No
Write Skew	Yes	No	Yes	No	No
Lost Update	No	No	No	No	No

Oracle

The Serializable isolation level is in fact an MVCC implementation of the Snapshot Isolation concurrency control mechanism. Like the Repeatable Read isolation in PostgreSQL, Oracle cannot prevent write skews, meaning it cannot provide a truly serializable transaction.

SQL Server

The Serializable isolation level is based on 2PL, and all phenomena are therefore prevented. The MVCC-based Snapshot isolation is close to Oracle Serializable and PostgreSQL Repeatable Read, and so it allows write skews.

PostgreSQL

To overcome the Snapshot Isolation limitations, PostgreSQL has developed the Serializable Snapshot Isolation (SSI), which provides true serializable transactions. Because SSI is still an MVCC implementation, PostgreSQL monitors the transaction schedule and detects *possible* serializability anomalies.

The current implementation may detect *false positives*[a], and some transactions might get aborted even if they didn't really break transaction serializability. Only the Precisely Serializable Snapshot Isolation (PSSI) model can eliminate all false positives, but the performance penalty being too high, the database implementers stuck to SSI instead.

MySQL

The Serializable isolation builds on top of Repeatable Read with the difference that every record that gets selected is protected with a shared lock as well. The locking-based approach allows MySQL to prevent the write skew phenomena, which is prevalent among many Snapshot Isolation implementations.

7.4 Durability

When purchasing an airline ticket, the money is withdrawn from the bank account and a seat is reserved for the given buyer. Assuming that, right after the ticket is purchased, the airline reservation system crashes, all the previously processed transactions must hold true even after the system restarts. If the system doesn't enforce this requirement, the registered ticket might vanish, and the buyer is possibly left with a debited account and no ticket at all.

> Durability ensures that all committed transaction changes become permanent.

Durability allows system recoverability, and, to some extent, it's similar to the rolling back mechanism.

What about undo logs?

To support transaction rollbacks and to rebuild previous versions in MVCC systems, the database system already records the current modifications (including uncommitted changes) in the undo log. But recoverability needs committed changes only, and, because the obsolete undo segments might be frequently recycled, the undo log alone is not suitable for recoverability.

When a transaction is committed, the database persists all current changes in an append-only, sequential data structure commonly known as the *redo log*.

Oracle

The *redo log*[a] consists of multiple redo records, each one containing *change vectors*, which capture the actual data block changes. For performance reasons, the redo records are stored in a buffer and the Log Writer flushes the in-memory records to the current active redo log file. At any given time, Oracle has at least two redo files, but only one of them is active and available for collecting the log buffer entries. When a transaction is committed, the database flushes the buffer, and changes become persisted.

If the buffer fills, Oracle flushes it along with any uncommitted changes, which can be removed if their associated transaction is rolled back.

[a] https://docs.oracle.com/database/121/ADMIN/onlineredo.htm#ADMIN11302

SQL Server

Unlike Oracle, SQL Server combines both the undo log and the redo long into a single data structure (*transaction log*). By default, when a transaction is committed, all the associated transaction log entries are flushed to the disk before returning the control back to the client.

SQL Server 2014 added support for configurable durability[a]. The log entry flushing can be delayed, which can provide better I/O utilization and lower transaction response times. If the system crashes, all the unflushed log entries are wiped out from memory. Asynchronous flushing is therefore appropriate only when data loss is tolerated.

PostgreSQL

Statement changes are captured in the *Write-Ahead Log (WAL)*[a]. The log entries are buffered in memory and flushed on every transaction commit.

The cached data pages and index entries need not be flushed for every transaction (therefore optimizing I/O utilization), because their state can be restored from the WAL during recovery. Ever since 9.1[b], PostgreSQL supports configurable durability, so the WAL can also be flushed asynchronously.

[a] http://www.postgresql.org/docs/9.5/static/wal-intro.html

MySQL

All the *redo log* entries associated with a single transaction are stored in the *mini transaction buffer* and flushed at once into the *global redo buffer*. The global buffer is flushed to disk during commit. By default, there are two log files which are used alternatively.

Flushing is done synchronously by default, but it can be switched to an asynchronous mode via the innodb_flush_log_at_trx_commit[a] parameter.

Since durability is very important for business operations, it's better to stick to the synchronous flushing mechanism.

Delaying durability guarantees becomes a valid option only when data loss is tolerated by business requirements and the redo log flushing is a real performance bottleneck.

7.5 Read-only transactions

The JDBC `Connection` defines the `setReadOnly(boolean readOnly)`[7] method which can be used to *hint* the driver to apply some database optimizations for the upcoming read-only transactions. This method shouldn't be called in the middle of a transaction because the database system cannot turn a read-write transaction into a read-only one (a transaction must start as read-only from the very beginning).

Oracle

According to the JDBC driver documentation[a] the database server does not support read-only transaction optimizations. Even when the read-only `Connection` status is set to `true`, modifying statements are still permitted, and the only way to restrict such statements is to execute the following SQL command:

```
connection.setAutoCommit(false);
try(CallableStatement statement = connection.prepareCall(
   "BEGIN SET TRANSACTION READ ONLY; END;")) {
    statement.execute();
}
```

The `SET TRANSACTION READ ONLY` command must run after disabling the auto-commit status as otherwise it will only be applied for this particular statement only.

[a] https://docs.oracle.com/database/121/JJDBC/apxtips.htm#JJDBC28956

[7] http://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#setReadOnly%28boolean%29

SQL Server

Like Oracle, the read-only `Connection` doesn't propagate to the database engine, and the only way to disable SQL modifications is to use a separate account, restricted to viewing data only.

Setting the `ApplicationIntent=ReadOnly`[a] connection property doesn't prevent the JDBC driver from executing modifying statements on a read-only `Connection`. This property has the purpose of routing read-write and read-only connections to replica nodes instead.

[a] https://msdn.microsoft.com/en-us/library/gg471494.aspx

PostgreSQL

An exception is thrown when executing a modifying statement on a `Connection` whose read-only status was set to `true`.

The database engine optimizes read-only transactions so the false-positives anomaly rate is reduced for the Serializable isolation level, and it allows *deferrable* serializable snapshots[a]. A deferrable snapshot is activated when executing `SET TRANSACTION SERIALIZABLE READ ONLY DEFERRABLE`. The current transaction must wait for a safe snapshot to become available, which can be executed without the risk of being aborted by a non-serializable anomaly. If the default read-write Serializable transactions are problematic when accessing large volumes of data, the deferrable snapshots might be a better alternative for long-running transactions.

[a] http://arxiv.org/pdf/1208.4179.pdf

MySQL

If a modifying statement is executed when the `Connection` is set to read-only, the JDBC driver throws an exception.

InnoDB can optimize read-only transactions[a] because it can skip the transaction ID generation as it's not required for read-only transactions.

[a] https://dev.mysql.com/doc/refman/5.7/en/innodb-performance-ro-txn.html

7.5.1 Read-only transaction routing

Setting up a database replication environment is useful for both high-availability (a Slave can replace a crashing Master) and traffic splitting. In a Master-Slave replication topology, the Master node accepts both read-write and read-only transactions, while Slave nodes only take read-only traffic.

Oracle

The Oracle ADG (Active Data Guard) allows an enterprise application to distribute read-write traffic to the Primary node and read-only transactions to a Standby database. WebLogic Server GridLink Data Source[a] provides failover and load balancing capabilities over Oracle ADG.

[a] http://www.oracle.com/technetwork/middleware/weblogic/learnmore/1534212

SQL Server

The database Availability Group must be configured to use read-only routing, in which case the redirection is based on the `ApplicationIntent` connection property. This means that the application requires separate `DataSource(s)` for read-write and read-only connections, and transaction routing must initiate in the application service layer.

PostgreSQL

The JDBC driver defines two connection properties[a] for load balancing purposes: `loadBalanceHosts` (which is disabled by default) and `targetServerType` (`master` or `preferSlave`). To enable transaction routing, the application must do the routing itself using separate `DataSource(s)`.

[a] https://jdbc.postgresql.org/documentation/head/connect.html

MySQL

The `com.mysql.jdbc.ReplicationDriver`[a] supports transaction routing on a Master-Slave topology, the decision being made on the `Connection` read-only status basis.

[a] https://dev.mysql.com/doc/connector-j/en/connector-j-master-slave-replication-connection.html

Even if the JDBC driver doesn't support Master-Slave routing, the application can do it using multiple `DataSource` instances. This design cannot rely on the read-only status of the underlying `Connection` since the routing must take place before a database connection is fetched.

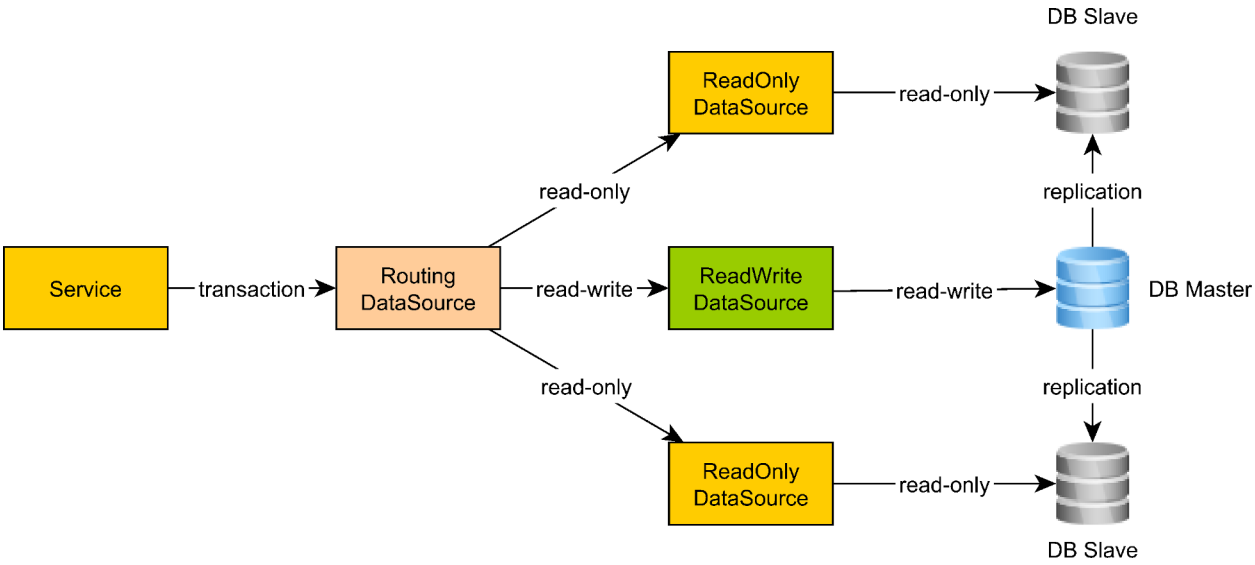


Figure 7.11: Transaction routing

If the transaction manager supports declarative read-only transactions, the routing decision can be taken based on the current transaction read-only preference. Otherwise, the routing must be done manually in each service layer component, and so a read-only transaction uses a read-only `DataSource` or a read-only JPA `PersistenceContext`.

7.6 Transaction boundaries

Every database statement executes in the context of a database transaction, even if the client doesn't explicitly set transaction boundaries. While there might be single statement transactions (usually a read-only query), when the unit of work consists of multiple SQL statements, the database should wrap them all in a single unit of work.

By default, every `Connection` starts in *auto-commit* mode, each statement being executed in a separate transaction. Unfortunately, it doesn't work for multi-statement transactions as it moves atomicity boundaries from the logical unit of work to each individual statement.

Auto-commit should be avoided as much as possible, and, even for single statement transactions, it's good practice to mark the transaction boundaries explicitly.

In the following example, a sum of money is transferred between two bank accounts. The balance must always be consistent, so if an account gets debited, the other one must always be credited with the same amount of money.

```
try(Connection connection = dataSource.getConnection();
    PreparedStatement transferStatement = connection.prepareStatement(
        "UPDATE account SET balance = ? WHERE id = ?"
    )) {
    transferStatement.setLong(1, Math.negateExact(cents));
    transferStatement.setLong(2, fromAccountId);
    transferStatement.executeUpdate();

    transferStatement.setLong(1, cents);
    transferStatement.setLong(2, toAccountId);
    transferStatement.executeUpdate();
}
```

Because of the auto-commit mode, if the second statement failed, only those particular changes can be rolled back, the first statement being already committed cannot be reverted anymore.

The default auto-commit mode must be disabled and the transaction will have to be managed manually. The transaction is committed if every statement runs successfully and a rollback is triggered on a failure basis. With this in mind, the previous example should be rewritten as follows:

```
try(Connection connection = dataSource.getConnection()) {
    connection.setAutoCommit(false);
    try(PreparedStatement transferStatement = connection.prepareStatement(
        "UPDATE account SET balance = ? WHERE id = ?"
    )) {
        transferStatement.setLong(1, Math.negateExact(cents));
        transferStatement.setLong(2, fromAccountId);
        transferStatement.executeUpdate();

        transferStatement.setLong(1, cents);
        transferStatement.setLong(2, toAccountId);
        transferStatement.executeUpdate();

        connection.commit();
    } catch (SQLException e) {
        connection.rollback();
        throw e;
    }
}
```

The astute reader will notice that the previous example breaks the *Single responsibility principle* since the Data Access Object (DAO) method mixes both transaction management and data access logic. Transaction management is a cross-cutting concern, making it a good candidate for being moved to a separate common library. This way, the transaction management logic will reside in one place, and a lot of duplicated code can be removed from the DAO methods. One way to extract the transaction management logic is to use the *Template method pattern*:

```
public void transact(Consumer<Connection> callback) {
    Connection connection = null;
    try {
        connection = dataSource.getConnection();
        callback.accept(connection);
        connection.commit();
    } catch (Exception e) {
        if (connection != null) {
            try {
                connection.rollback();
            } catch (SQLException ex) {
                throw new DataAccessException(e);
            }
        }
        throw (e instanceof DataAccessException ?
            (DataAccessException) e : new DataAccessException(e));
    } finally {
        if(connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                throw new DataAccessException(e);
            }
        }
    }
}
```

Transactions should never be abandoned on failure, and it's mandatory to initiate a transaction rollback (to allow the database to revert any uncommitted changes and release any lock as soon as possible).

With this utility in hand, the previous example can be simplified to:

```
transact((Connection connection) -> {
    try(PreparedStatement transferStatement = connection.prepareStatement(
        "UPDATE account SET balance = ? WHERE id = ?"
    )) {
        transferStatement.setLong(1, Math.negateExact(cents));
        transferStatement.setLong(2, fromAccountId);
        transferStatement.executeUpdate();

        transferStatement.setLong(1, cents);
        transferStatement.setLong(2, toAccountId);
        transferStatement.executeUpdate();
    } catch (SQLException e) {
        throw new DataAccessException(e);
    }
});
```

Although better than the first code snippet, separating data access logic and transaction management is not sufficient.

The transaction boundaries are still rigid, and, to include multiple data access method in a single database transaction, the `Connection` object has to be carried out as a parameter to every single DAO method.

Declarative transactions can better address this issue by breaking the strong coupling between the data access logic and the transaction management code. Transaction boundaries are marked with metadata (e.g. annotations) and a separate transaction manager abstraction is in charge of coordinating transaction logic.

Java EE and JTA

Declarative transactions become a necessity for distributed transactions. When Java EE (Enterprise Edition) first emerged, application servers hosted both web applications and middleware integration services, which meant that the Java EE container needed to coordinate multiple `DataSource(s)` or even JMS (Java Messaging) queues.

Following the X/Open XA architecture, JTA (Java Transaction API) powers the Java EE distributed transactions requirements.

7.6.1 Distributed transactions

The difference between local and global transactions is that the former uses a single resource manager, while the latter operates on multiple heterogeneous resource managers. The ACID guarantees are still enforced on each individual resource, but a global transaction manager is mandatory to orchestrate the distributed transaction outcome.

All transactional resource adapters are registered by the global transaction manager, which decides when a resource is allowed to commit or rollback. The Java EE managed resources become accessible through JNDI (Java Naming and Directory Interface) or CDI (Contexts and Dependency Injection).

Spring provides a transaction management abstraction layer which can be configured to either use local transactions (JDBC or RESOURCE_LOCAL[8] JPA) or global transactions through a stand-alone JTA transaction manager. The dependency injection mechanism auto-wires managed resources into Spring beans.

7.6.1.1 Two-phase commit

JTA makes use of the two-phase commit (2PC) protocol to coordinate the atomic resource commitment in two steps: a *prepare* and a *commit* phase.

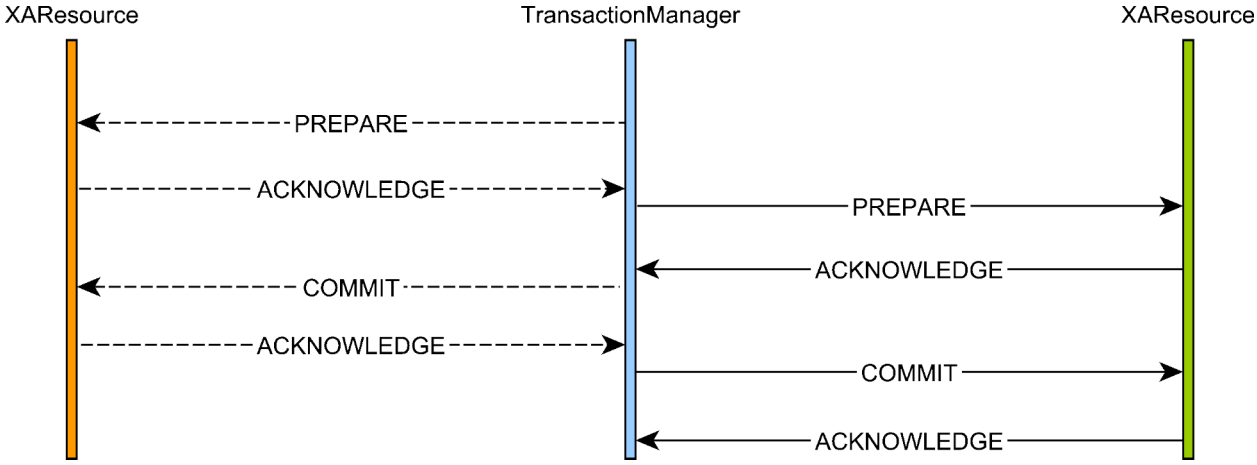


Figure 7.12: Transaction boundaries

In the former phase, a resource manager takes all the necessary actions to prepare the transaction for the upcoming commit. Only if all resource managers successfully acknowledge the preparation step, the transaction manager will proceed with the commit phase. If one resource doesn't acknowledge the prepare phase, the transaction manager proceeds to rolling back all current participants.

If all resource managers acknowledge the commit phase, the global transaction ends successfully. If one resource fails to commit (or times out), the transaction manager will have to retry this operation in a background thread until it succeeds or reports the incident for manual intervention.

[8] http://docs.oracle.com/javaee/7/api/javax/persistence/spi/PersistenceUnitTransactionType.html#RESOURCE_LOCAL

The one-phase commit (1PC) optimization

Because Java EE uses JTA transactions exclusively, the extra coordination overhead of the additional database roundtrip may hurt performance in a high-throughput application environment. When a transaction enlists only one resource adapter (designating a single resource manager), the transaction manager can skip the prepare phase, and either execute the commit or the rollback phase. With this optimization, the distributed transaction behaves similarly to how JDBC `Connection(s)` manage local transactions.

The `XAResource.commit(Xid xid, boolean onePhase`[a] method takes a `boolean` flag, which the transaction manager will set to *true* to hint the associated resource adapter to initiate the 1PC optimization.

7.6.2 Declarative transactions

Transaction boundaries are usually associated with a Service layer, which uses one or more DAO to fulfil the business logic. The transaction *propagates* from one component to the other within the service-layer transaction boundaries.

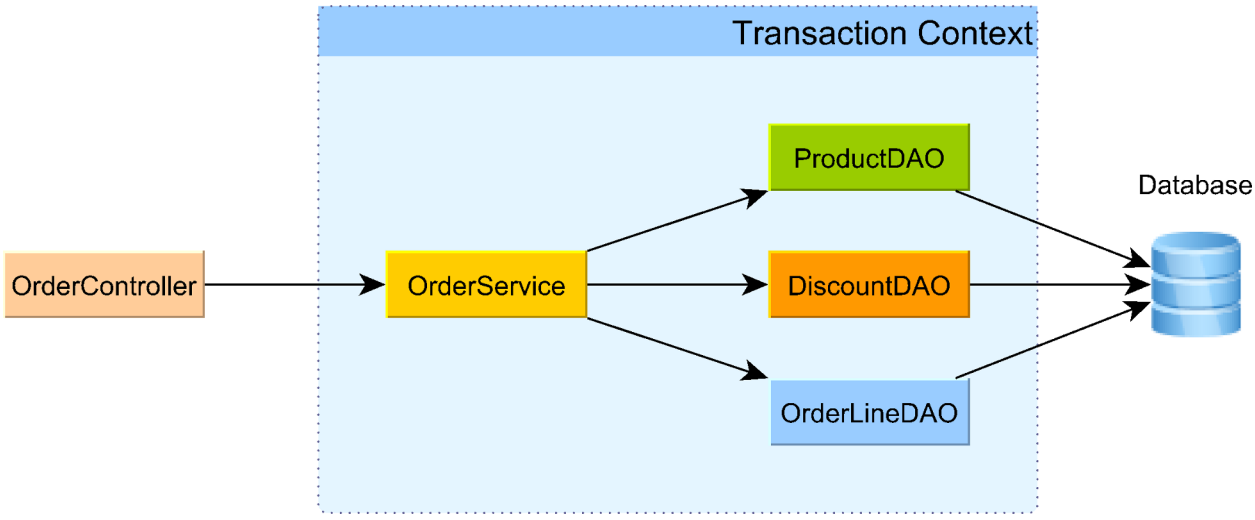


Figure 7.13: Transaction boundaries

The declarative transaction model is supported by both Java EE and Spring. Transaction boundaries are controlled through similar propagation strategies, which define how boundaries are inherited or disrupted at the borderline between the outermost component (in the current call stack) and the current one (waiting to be invoked).

Propagation

To configure the transaction propagation strategy for EJB components, Java EE defines the `@TransactionAttribute`[a] annotation. Since Java EE 7, even non-EJB components can now be enrolled in a transactional context if they are augmented with the `@Transactional`[b] annotation.

In Spring, transaction propagation (like any other transaction properties) is configurable via the `@Transactional`[c] annotation.

[a] http://docs.oracle.com/javaee/7/api/javax/ejb/TransactionAttribute.html
[b] http://docs.oracle.com/javaee/7/api/javax/transaction/Transactional.html
[c] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#propagation--

Table 7.6: Transaction propagation strategies

Propagation	Java EE	Spring	Description
REQUIRED	Yes	Yes	this is the default propagation strategy, and it only starts a transaction unless the current thread isn't already associated with a transaction context
REQUIRES_NEW	Yes	Yes	any current running transaction context is suspended and replaced by a new transaction
SUPPORTS	Yes	Yes	if the current thread already runs inside a transaction, this method will use it, otherwise it will execute outside of a transaction context
NOT_SUPPORTED	Yes	Yes	any current running transaction context is suspended, and the current method is run outside of a transaction context
MANDATORY	Yes	Yes	the current method runs only if the current thread is already associated with a transaction context
NESTED	No	Yes	the current method is executed within a nested transaction if the current thread is already associated with a transaction, otherwise a new transaction is started.
NEVER	No	Yes	the current method must always run outside of a transaction context, and, if the current thread is associated with a transaction, an exception is thrown.

Declarative exception handling

Since the transaction logic wraps around the underlying service and data access logic call chain, the exception handling must also be configured declaratively. By default, both Java EE and Spring roll back on system exceptions (any `RuntimeException`) and commit on application exceptions (checked exceptions).

In Java EE, the rollback policy can be customized using the `@ApplicationException`[a] annotation.

Spring allows each transaction to customize the rolling back policy[b] by listing the exception types triggering a transaction failure.

[a] http://docs.oracle.com/javaee/7/api/javax/ejb/ApplicationException.html

[b] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#rollbackFor--

Declarative read-only transactions

Java EE doesn't support read-only transactions to be marked declaratively.

Spring offers the transactional read-only attribute[a], which can propagate to the underlying JPA provider (to optimize the `EntityManager` flushing mechanism) and to the current associated JDBC `Connection`.

[a] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#readOnly--

Declarative isolation levels

The Java EE doesn't offer support for configurable isolation levels, so it's up to the underlying `DataSource` to define it for all database connections.

Spring supports transaction-level isolation levels[a] when using the `JPATransactionManager`[b]. For JTA transactions, the `JTATransactionManager`[c] follows the Java EE standard and disallows overriding the default isolation level. As a workaround, the Spring framework provides extension points, so the application developer can customize the default behavior and implement a mechanism to set isolation levels on a transaction basis.

[a] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html#isolation--

[b] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/JpaTransactionManager.html

[c] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/jta/JtaTransactionManager.html

7.7 Application-level transactions

So far, the book focused on database transactions to enforce ACID properties. But from the application perspective, a business workflow might span over multiple physical database transactions, in which case the database ACID guarantees will not be sufficient anymore.

A logical transaction may be composed of multiple web requests, including user think time, for which reason it can be visualized as a *long conversation*.

In the following example, both *Alice* and a background batch process are concurrently modifying the same database record.

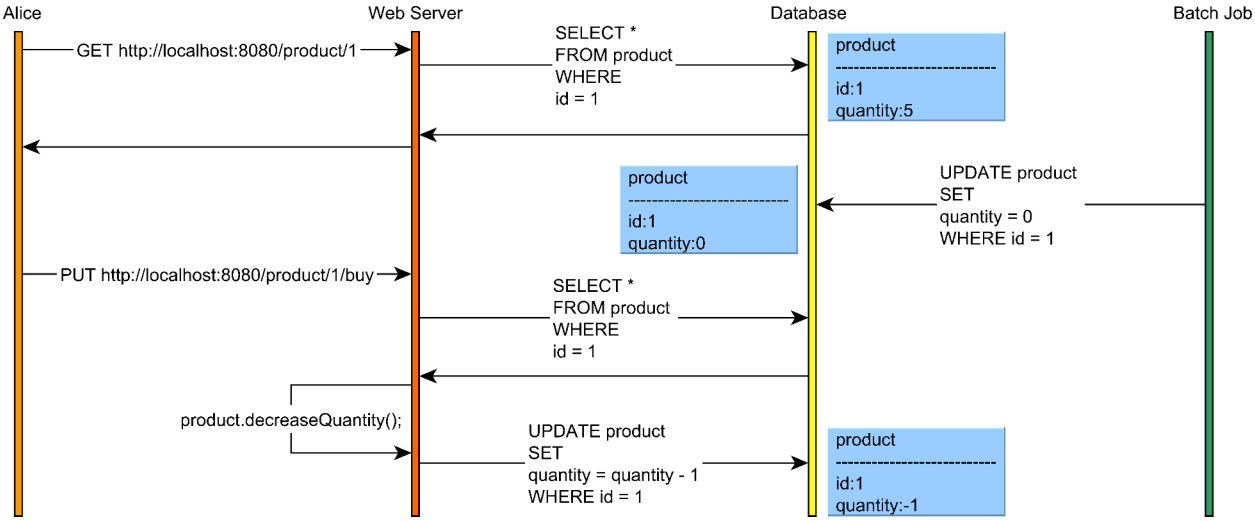


Figure 7.14: Stateless conversation loosing updates

Because *Alice* logical transaction encloses two separate web requests, each one associated with a separate database transaction, without an additional concurrency control mechanism, even the strongest isolation level cannot prevent the lost update phenomena.

Spanning a database transaction over multiple web requests is prohibitive since locks would be held during user think time, therefore hurting scalability. Even with MVCC, the cost of maintaining previous versions (that can lead to a large version graph) can escalate and affect both performance and concurrency.

In a highly concurrent environment, database transactions are bound to be as short as possible. Application-level transactions require application-level concurrency control mechanisms.

HTTP is stateless by nature and, for very good reasons, stateless applications are easier to scale than stateful ones. But application-level transactions cannot be stateless as otherwise newer requests would not continue from where the previous request was left. Preserving state across multiple web requests allows building a conversational context, providing application-level repeatable reads guarantees.

In the next diagram, *Alice* uses a stateful conversational context, but, in the absence of a record versioning system, it's still possible to lose updates.

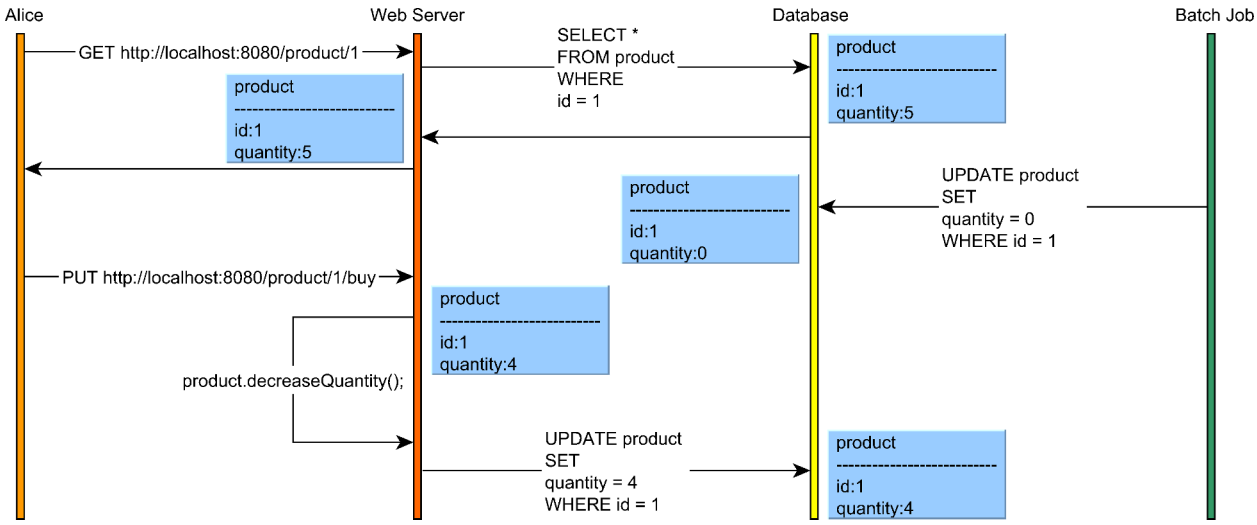


Figure 7.15: Stateful conversation loosing updates

Without *Alice* to notice, the batch process resets the product quantity. Thinking the product version hasn't changed, *Alice* attempts to purchase one item which decreases the previous product quantity by one. In the end, *Alice* has simply overwritten the batch processor modification and data integrity has been compromised.

So the application-level repeatable reads are not self-sufficient (this argument is true for database isolation levels as well). To prevent lost updates, a concurrency control mechanism becomes mandatory.

7.7.1 Pessimistic and optimistic locking

Isolation levels entail implicit locking, whether it involves physical locks (like 2PL) or data anomaly detection (MVCC). To coordinate state changes, application-level concurrency control makes use of explicit locking, which comes in two flavors: pessimistic and optimistic locking.

7.7.1.1 Pessimistic locking

As previously explained, most database systems already offer the possibility of manually requesting shared or exclusive locks. This concurrency control is said to be *pessimistic* because it assumes that conflicts are bound to happen, and so they must be prevented accordingly.

As locks can be released in a timely fashion, exclusive locking is appropriate during the last database transaction of a given long conversation. This way, the application can guarantee that, once locks are acquired, no other transaction can interfere with the currently locked resources.

Acquiring locks on critical records can prevent non-repeatable reads, lost updates, as well as read and write skew phenomena.

7.7.1.2 Optimistic locking

Undoubtedly a misnomer (albeit rather widespread), optimistic locking doesn't incur any locking at all. A much better name would be optimistic concurrency control since it uses a totally different approach to managing conflicts than pessimistic locking.

MVCC is an optimistic concurrency control strategy since it assumes that contention is unlikely to happen, and so it doesn't rely on locking for controlling access to shared resources. The optimistic concurrency mechanisms detect anomalies and resort to aborting transactions whose invariants no longer hold.

While the database knows exactly which row versions have been issued in a given time interval, the application is left to maintaining a *happens-before* event ordering. Each database row must have an associated version, which is locally incremented by the logical transaction. Every modifying SQL statement (update or delete) uses the previously loaded version as an assumption that the row hasn't been changed in the meanwhile.

Because even the lowest isolation level can prevent write-write conflicts, only one transaction is allowed to update a row version at any given time. Since the database already offers monotonic updates, the row versions can also be incremented monotonically, and the application can detect when an updating record has become stale. The optimistic locking concurrency algorithm looks like this:

- when a client reads a particular row, its version comes along with the other fields
- upon updating a row, the client filters the current record by the version it has previously loaded.

```
UPDATE product
SET (quantity, version) = (4, 2)
WHERE id = 1 AND version = 1
```

- if the statement update count is zero, the version was incremented in the meanwhile, and the current transaction now operates on a stale record version.

The previous example can be adjusted to take advantage of this optimistic concurrency control mechanism. This time, the product is versioned, and both the web application and the batch processor data access logic are using the row versions to coordinate the *happens before* update ordering.

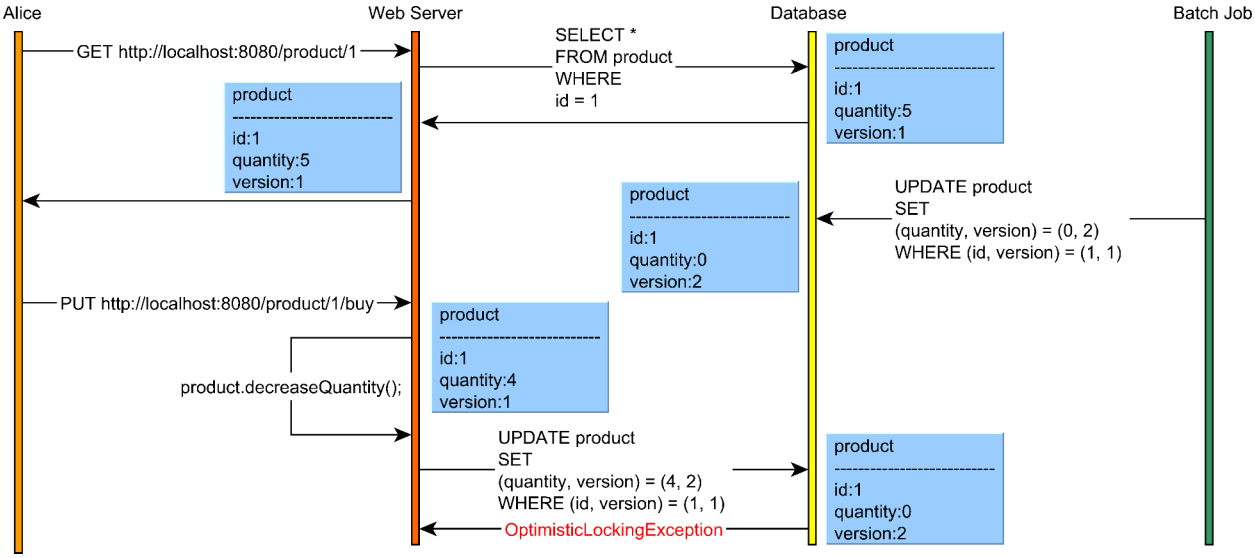


Figure 7.16: Stateful conversation preventing lost updates

Both *Alice* and the batch processor try to increment the product version optimistically. The batch processor can successfully update the product quantity since the SQL statement filtering criteria matches the actual database record version. When *Alice* tries to update the product, the database returns a zero update count, and, this way, she is notified about the concurrent update that happened in the meanwhile. The lost update can be prevented because the application can abort the current transaction when being notified of the stale record versions.

Using timestamps to order events is rarely a good idea. System time is not always monotonically incremented, and it can even go backwards, due to network time synchronization. Time accuracy across different database systems varies from nanoseconds (Oracle[a]), to 100 nanoseconds (SQL Server[b]), to microseconds (PostgreSQL[c] and MySQL 5.6.4[d] and even to seconds (previous versions of MySQL). In distributed systems, logical clocks (e.g. vector clocks or Lamport timestamps) are always preferred to physical timestamp (wall clocks) when it comes to ordering events.

For this reason, employing a numerical record version is more appropriate than assigning timestamps to row updates.

III JPA and Hibernate

8. 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 will 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.

8.1 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 object to have both state and behavior, and bidirectional associations are permitted.

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

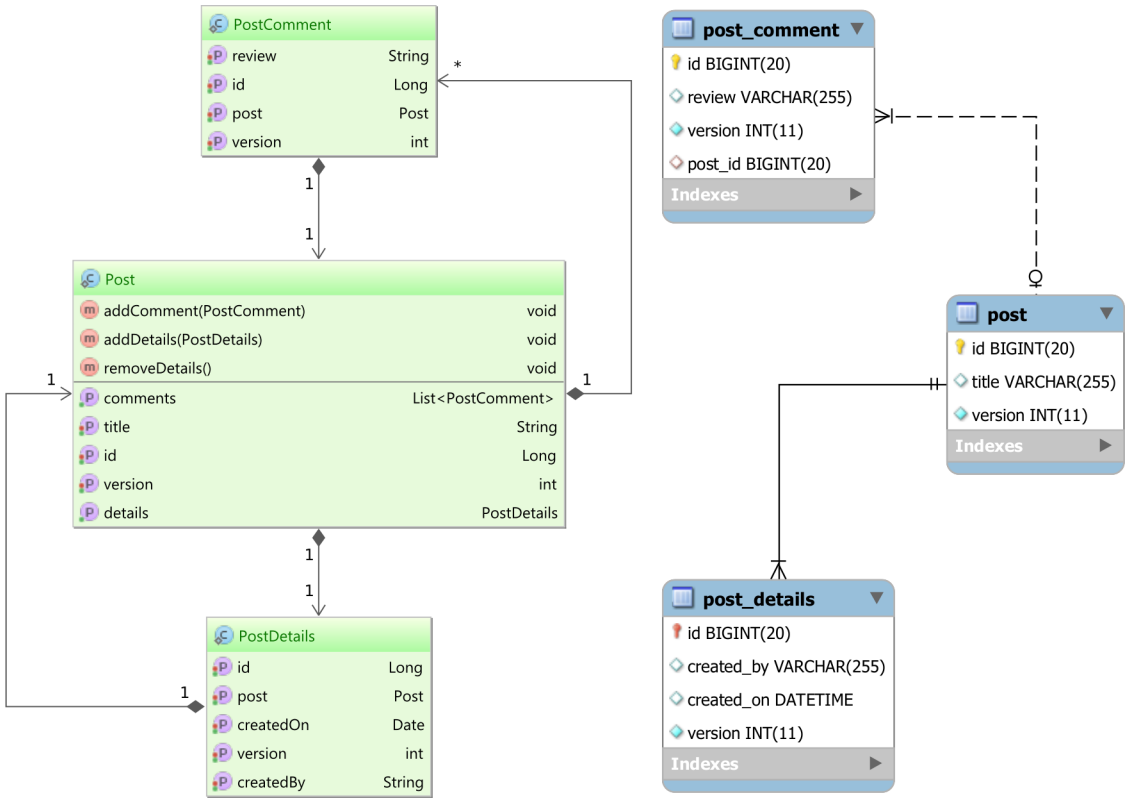


Figure 8.1: Object/Relational Mapping

The above diagram portrays the two different schemas, 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 Modelling 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 bridging 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's much easier to manipulate Domain Model relationships rather than visualizing the business logic through its underlying SQL statements.

8.2 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's 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, implementing a HiLo optimizer that's interoperable with other database clients
- 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)`)
- version-less optimistic locking (e.g. `OptimisticLockType.ALL`, `OptimisticLockType.DIRTY`)
- support for skipping (without waiting) pessimistic lock requests.

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's familiar with the JPA specification only:

- the FlushModeType.AUTO[1] doesn't trigger a flush for native SQL queries, like it does for JPQL or Criteria API
- the FetchType.EAGER[2] might choose an 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's why this book is focused on how Hibernate manages to implement both the JPA specification and its native non-standard 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's 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. SQL-2003 introduced `MERGE`, but PostgreSQL 9.4.5 doesn't implement it, and 9.5 will support `UPSERT` instead. By adhering to an 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's already much easier to swap JPA providers, if there's 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

8.3 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's 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.

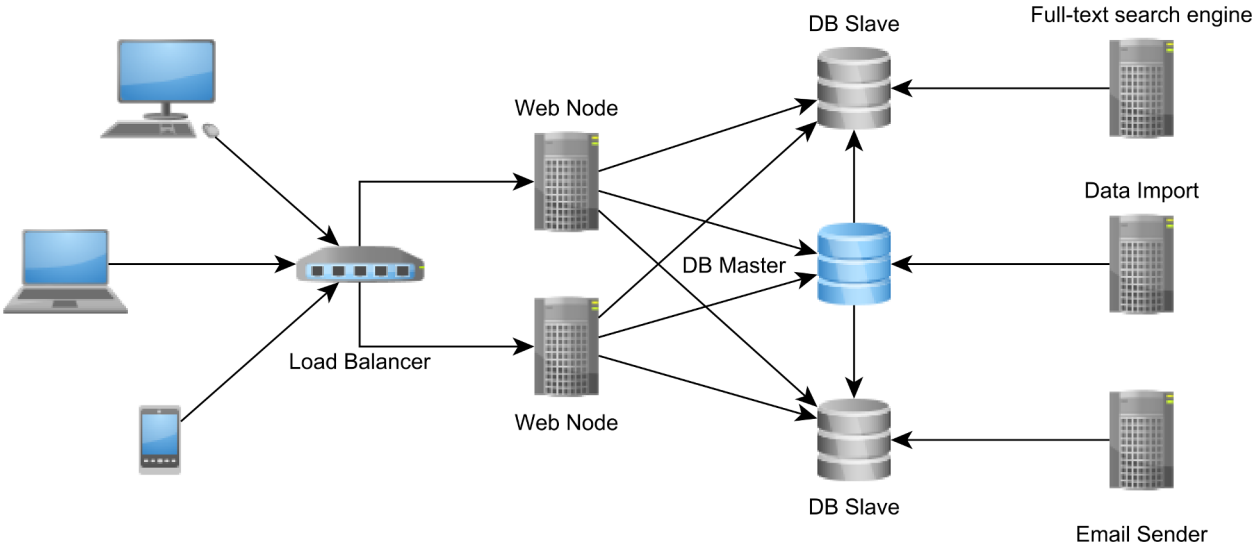


Figure 8.2: Database-centric integration

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 doesn't provide transactional semantics, it's 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, 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's 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.

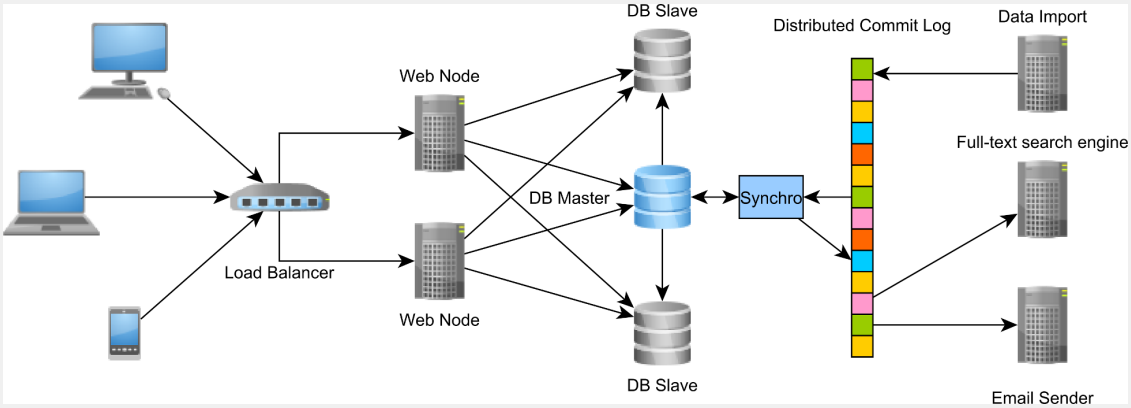


Figure 8.3: Distributed commit log integration

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 don't require random access) and read performance (concurrent reads don't require blocking) as well.

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

No matter what architecture style is chosen, there is still a 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's 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.

8.4 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, that is not mapped to any database row, is considered to be in the New or Transient state. Once it becomes managed, the Persistence Context will issue an insert statement at flush-time.
Managed (Persistent)	A Persistent entity is associated with a database row and it's being managed by the current 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 current running Persistence Context is closed, all the previously managed entities become detached. Successive changes will no longer be 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 to 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/stable/orm/javadocs/org/hibernate/Session.html

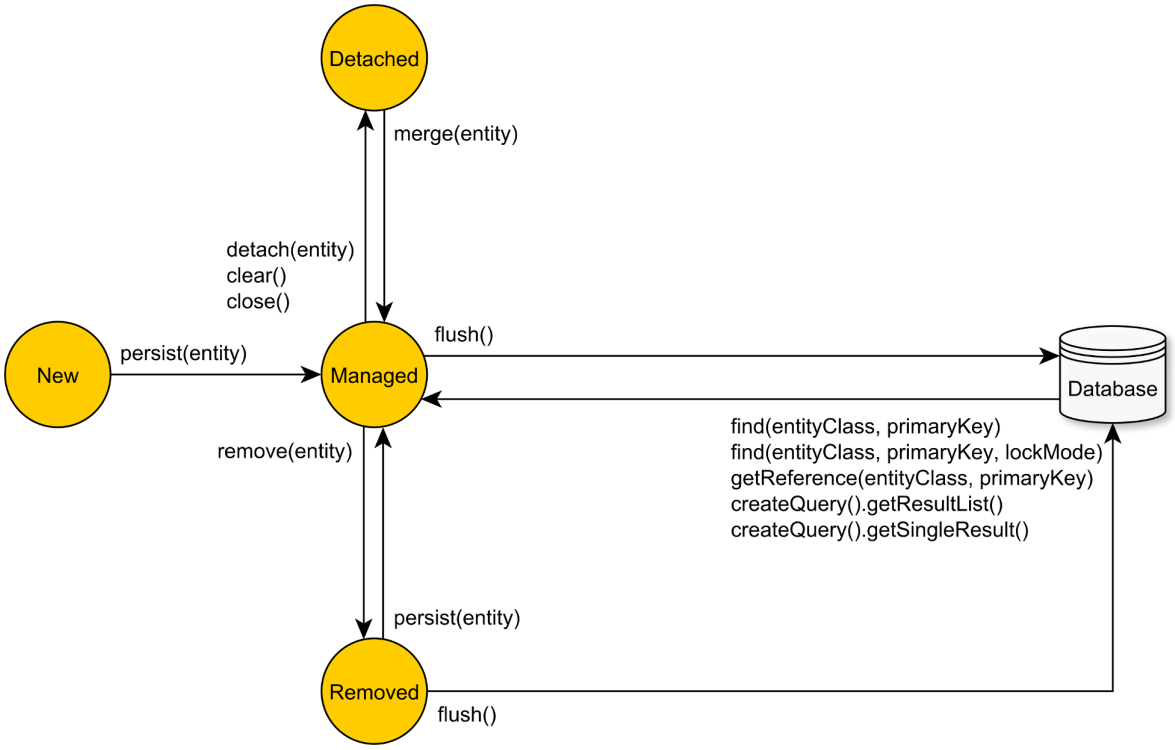


Figure 8.4: JPA entity state transitions

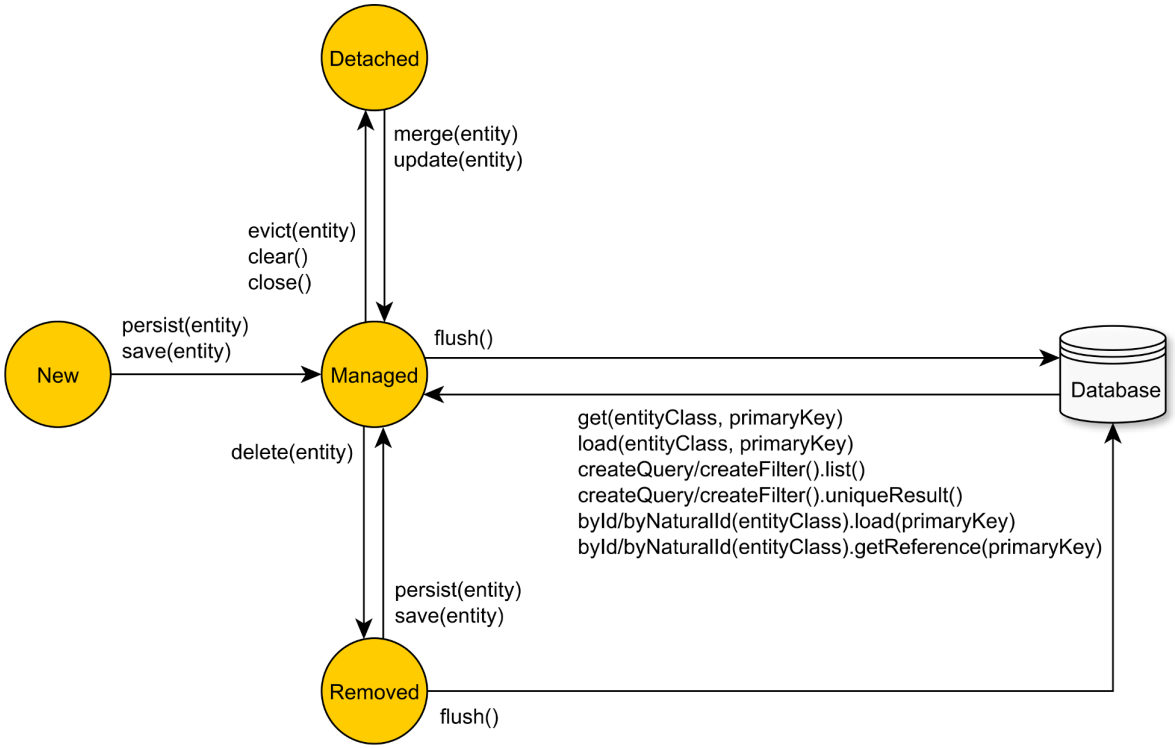


Figure 8.5: Hibernate entity state transitions

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 rigour is advised.

Hibernate uses `PreparedStatement(s)` exclusively, so not only it protect 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 needs 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.

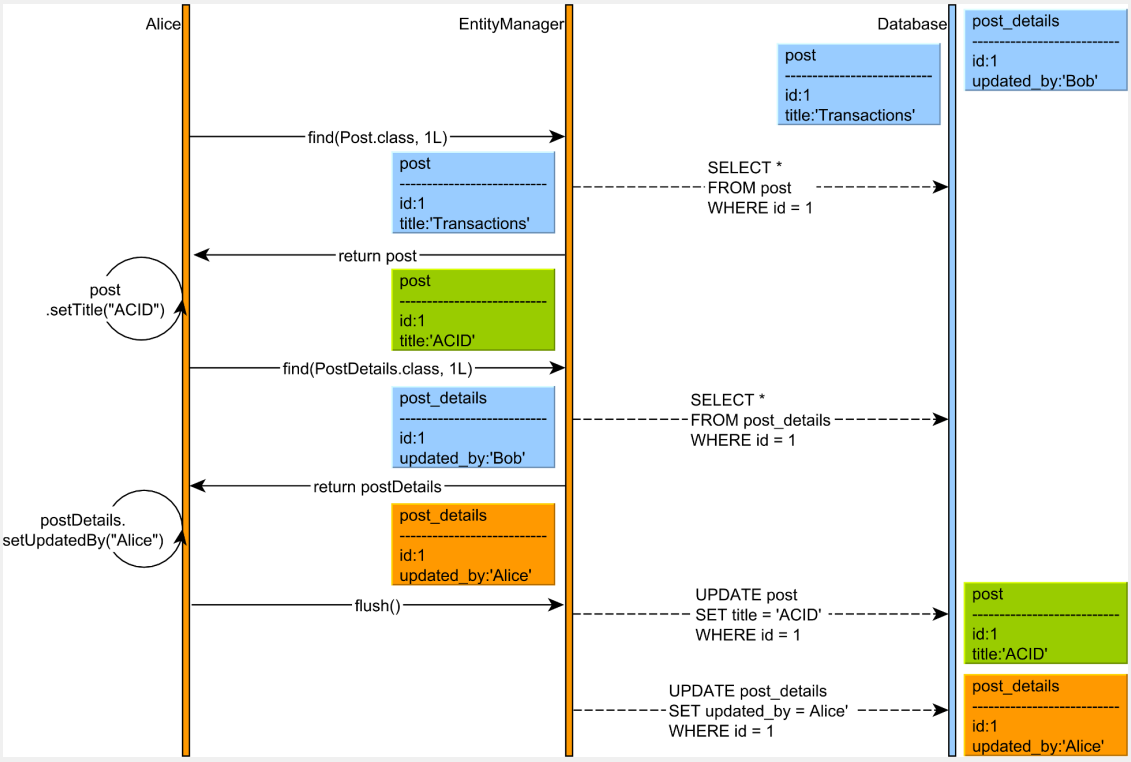


Figure 8.6: Persistence context

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.

But 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 it will be detailed in the following chapters, Hibernate doesn't 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. But 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. But 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 a 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.

8.5 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's 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's mandatory to keep in mind the SQL statements associated with every fetching operation.

In the following example, the *posts* records will be 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` will contain 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's 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 anyway, 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 will only happen 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 is 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 doesn't 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 through 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.

8.6 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 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 than what the reader might have been previously used to. There is no need to worry if some topics are not entirely clear because the upcoming chapters will analyze all these concepts in greater detail.

9. Connection Management and Monitoring

As previously explained in the JDBC Connection Management chapter, for performance reasons, database connections are better off reused. Because JPA providers generate SQL statements on behalf of users, it's very important to monitor this process and acknowledge its outcome. This chapter will explain the Hibernate connection provider mechanism and ways to monitor statement execution.

9.1 JPA connection management

Like the whole Java EE suite, the JPA 1.0 specification was very much tied to enterprise application servers. In a Java EE container, all database connections are managed by the application server which provides connection pooling, monitoring and JTA capabilities.

Once configured, the Java EE `DataSource` can be located through JNDI. In the `persistence.xml` configuration file, the application developer must supply the JNDI name of the associated JTA or RESOURCE_LOCAL `DataSource`. The `transaction-type` attribute must also match the data source transaction capabilities.

```
<persistence-unit name="persistenceUnit" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <jta-data-source>java:global/jdbc/flexypool</jta-data-source>
</persistence-unit>
```

A `RESOURCE_LOCAL` transaction must use a `non-jta-data-source` `DataSource`.

```
<persistence-unit name="persistenceUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <non-jta-data-source>java:/comp/env/jdbc/hsqldb</non-jta-data-source>
</persistence-unit>
```

While for a Java EE application it's perfectly fine to rely on the application server for providing a full-featured `DataSource` reference, stand-alone applications are usually configured using dependency injection rather than JNDI.

From the JPA implementation perspective the `DataSource` can be either configured externally or by the JPA provider itself. Most often, configuring an external `DataSource` is still the preferred

alternative as it gives more flexibility in decorating the connection providing mechanism (e.g. logging, monitoring).

JPA providers can fetch connections through the underlying JDBC `Driver` since JPA 2.0 has standardized the database connection configuration properties:

Table 9.1: JPA connection properties

Property	Description
javax.persistence.jdbc.driver	Driver full class name (e.g. `org.hsqldb.jdbc.JDBCDriver`)
javax.persistence.jdbc.url	Driver Url (e.g. `jdbc:hsqldb:mem:test`)
javax.persistence.jdbc.user	Database user's name
javax.persistence.jdbc.password	Database user's password

Unfortunately, these properties alone are not sufficient because most enterprise applications need connection pooling and monitoring capabilities anyway. For this reason, JPA connection management is still an implementation specific topic, and the upcoming sections will dive into the connection provider mechanism employed by Hibernate.

9.2 Hibernate connection providers

Hibernate needs to operate both in Java EE and stand-alone environments, and the database connectivity configuration can be done either declaratively or programmatically. To accommodate JDBC `Driver` connections as well as RESOURCE_LOCAL and JTA `DataSource` configurations, Hibernate defines its own connection factory abstraction, represented by the `org.hibernate.engine.jdbc.connections.spi.` interface:

```
public interface ConnectionProvider extends Service, Wrapped {

    public Connection getConnection() throws SQLException;

    public void closeConnection(Connection connection) throws SQLException;

    public boolean supportsAggressiveRelease();
}
```

Because the connection provider might influence transaction response time, each provider will be analyzed from a high-performance OLTP system perspective.

9.2.1 DriverManagerConnectionProvider

Hibernate picks this provider when being given the aforementioned JPA 2.0 connection properties or the Hibernate-specific configuration counterpart:

- `hibernate.connection.driver_class`
- `hibernate.connection.url`
- `hibernate.connection.username`
- `hibernate.connection.password`.

Although it fetches database connections through the underlying `DriverManager`, this provider tries to avoid the connection acquisition overhead by using a trivial pooling implementation. The Hibernate documentation doesn't recommend using the `DriverManagerConnectionProvider` in a production setup.

9.2.2 C3P0ConnectionProvider

C3p0[1] is a mature connection pooling solution that has proven itself in many production environments, and, using the underlying JDBC connection properties, Hibernate can replace the built-in connection pool with a c3p0 `DataSource`. To activate this provider, the application developer must supply at least one configuration property starting with the `hibernate.c3p0` prefix:

```
<property name="hibernate.c3p0.max_size" value="5"/>
```

C3p0 (released in 2001) and Apache DBCP[a] (released in 2002) are the oldest and the most deployed stand-alone Java connection pooling solutions. Later in 2010, BoneCP[b] emerged as a high-performance alternative for c3p0 and Apache DBCP. Nowadays, the BoneCP GitHub page says it's been deprecated in favor of HikariCP.

[1] http://www.mchange.com/projects/c3p0/

As of writing, the most attractive Java connection pools are HikariCP[a], Vibur DBCP[b] and Apache DBCP2. HikariCP and Vibur DBCP offer built-in Hibernate connection providers.

9.2.3 HikariConnectionProvider

HikariCP is one of the fastest Java connection pool, and, although not natively supported by Hibernate, it also comes with its own `ConnectionProvider` implementation. By specifying the `hibernate.connection.provider_class` property, the application developer can override the default connection provider mechanism:

```
<property
    name="hibernate.connection.provider_class"
    value="com.zaxxer.hikari.hibernate.HikariConnectionProvider"/>
```

Unlike `DriverManagerConnectionProvider` or `C3P0ConnectionProvider`, HikariCP doesn't recognize the JPA or Hibernate-specific connection properties. The `HikariConnectionProvider` requires framework-specific properties[2] like the following ones:

Table 9.2: HikariCP connection properties

Property	Description
hibernate.hikari.dataSourceClassName	Driver full class name
hibernate.hikari.dataSource.url	Driver Url
hibernate.hikari.dataSource.user	Database user's name
hibernate.hikari.dataSource.password	Database user's password
hibernate.hikari.maximumPoolSize	Maximum pool size

[2] https://github.com/brettwooldridge/HikariCP

9.2.4 DatasourceConnectionProvider

This provider is chosen when the JPA configuration file defines a `non-jta-data-source` or a `jta-data-source` element, or when supplying a `hibernate.connection.datasource` configuration property.

Unlike other providers, this one is compatible with JTA transactions, which are mandatory in Java EE.

Spring works with both stand-alone JTA transaction managers (e.g. Bitronix or Atomikos) and Java EE `DataSource(s)`, and, because it offers the best control over the actual `DataSource` configuration, the `DatasourceConnectionProvider` is the preferred choice (even for HikariCP) .

9.2.5 Connection release modes

Hibernate defers the database connection acquisition until the current transaction has to execute its first SQL statement (either triggered by a read or a write operation). This optimization allows Hibernate to reduce the physical transaction interval, therefore increasing the chance of getting a connection from the pool.

The connection release strategy is controlled through the `hibernate.connection.release_mode` property which can take the following values:

Table 9.3: Connection release modes

Value	Description
`after_transaction`	Once acquired, the database connection is release only after the current transaction either commits or rolls back.
`after_statement`	The connection is released after each statement execution and reacquired prior to running the next statement. Although not required by either JDBC or JTA specifications, this strategy is meant to prevent application servers from mistakenly detecting[3] a connection leak between successive *EJB* (Enterprise Java Beans) calls
`auto`	This is the default value, and for RESOURCE_LOCAL transactions it uses the `after_transaction` mode, while for JTA transactions it falls back to `after_statement`.

[3] http://lists.jboss.org/pipermail/hibernate-dev/2006-December/000903.html

For JTA transactions, the default mode might be too strict since not all Java EE application servers exhibit the same behavior for managing transactional resources. This way, it's important to check if database connections can be closed outside of the EJB component that triggered the connection acquisition event. Spring-based enterprise systems don't use Enterprise Java Beans, and, even when using a stand-alone JTA transaction manager, the `after_transaction` connection release mode might be just fine.

It's somehow intuitive that the `after_statement` mode incurs some performance penalty associated with the frequent acquisition/releasing connection cycles. For this reason, the following test measures the connection acquisition overhead when using Bitronix in a Spring application context. Each transaction executes the same statement (fetching the current timestamp) for a given number of times (represented on the x-axis). The y-axis captures the recorded transaction response times for both `after_statement` and `after_transaction` connection release modes.

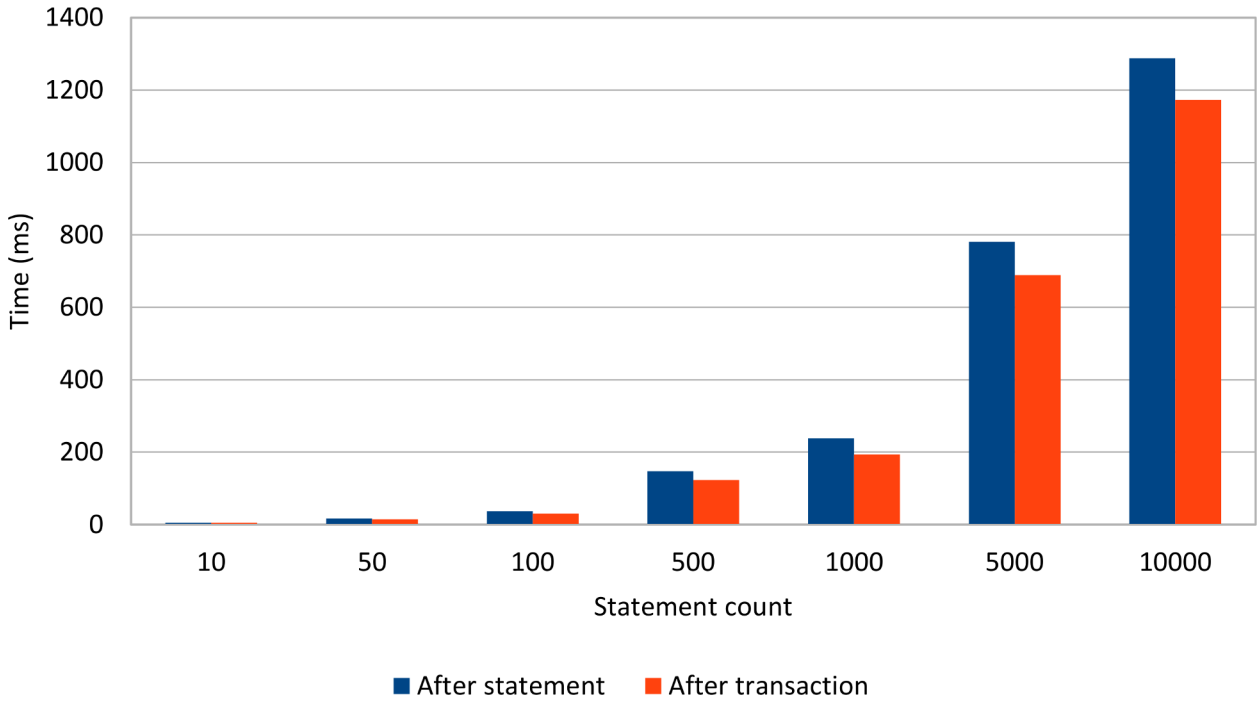


Figure 9.1: Connection release mode

The more statements a transaction will execute, the greater the penalty of reacquiring the associated database connection from the underlying connection pool. To better visualize the connection acquisition overhead, the test runs up to *10 000* statements, even if this number is probably too high for the typical OLTP transaction.

Ideally, database transactions should be as short as possible, and the number of statements shouldn't be too high either. This requirement stems from the fact that the number of pooled connections is limited and locks are better released sooner than later.

The `after_transaction` connection release mode is more efficient than the default JTA `after_statement` strategy, and so it should be used if the JTA transaction resource management logic doesn't interfere with this connection releasing strategy.

9.3 Monitoring connections

As previously concluded, using an externally configured `DataSource` is preferred because the actual `DataSource` can be decorated with connection pooling, monitoring and logging capabilities. Because that's exactly how FlexyPool[4] works too, the following diagram captures the `DataSource` proxying mechanism:

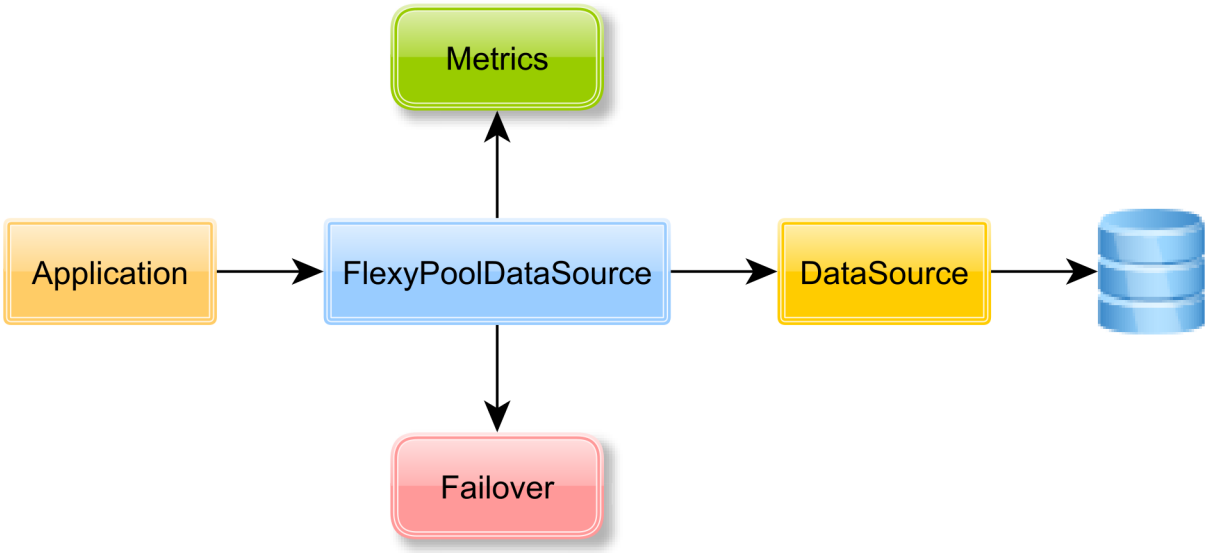


Figure 9.2: DataSource proxy

Instead of getting the actual `DataSource` instance, the data access layer gets a proxy reference. The proxy intercepts connection acquisition and releasing requests, and, this way, it can monitor its usage.

When using Spring, setting up FlexyPool is fairly easy because the application has total control over the `DataSource` configuration.

In Java EE, database connections should always be fetched from a managed `DataSource`, and one simple way of integrating FlexyPool is to extend the default `DatasourceConnectionProviderImpl` and substitute the original `DataSource` with the `FlexyPoolDataSource`.

[4] https://github.com/vladmihalcea/flexy-pool

For this reason, FlexyPool comes with the following Hibernate connection provider:

```
public class FlexyPoolHibernateConnectionProvider
    extends DatasourceConnectionProviderImpl {

    private transient FlexyPoolDataSource<DataSource> flexyPoolDataSource;

    @Override
    public void configure(Map props) {
        super.configure(props);
        flexyPoolDataSource = new FlexyPoolDataSource<>(getDataSource());
    }

    @Override
    public Connection getConnection() throws SQLException {
        return flexyPoolDataSource.getConnection();
    }

    @Override
    public boolean isUnwrappableAs(Class unwrapType) {
        return super.isUnwrappableAs(unwrapType) ||
            getClass().isAssignableFrom(unwrapType);
    }

    @Override
    public void stop() {
        flexyPoolDataSource.stop();
        super.stop();
    }
}
```

To use the `FlexyPoolHibernateConnectionProvider`, the application must configure the `hibernate.connection.provider_class` property:

```
<property
 name="hibernate.connection.provider_class"
 value="com.vladmihalcea.flexypool.adaptor.FlexyPoolHibernateConnectionProvider"
/>
```

9.3.1 Hibernate statistics

Hibernate has a built-in statistics collector which gathers notifications related to database connections, `Session` transactions and even second-level caching usage. The `StatisticsImplementor` interface defines the contract for intercepting various Hibernate internal events:

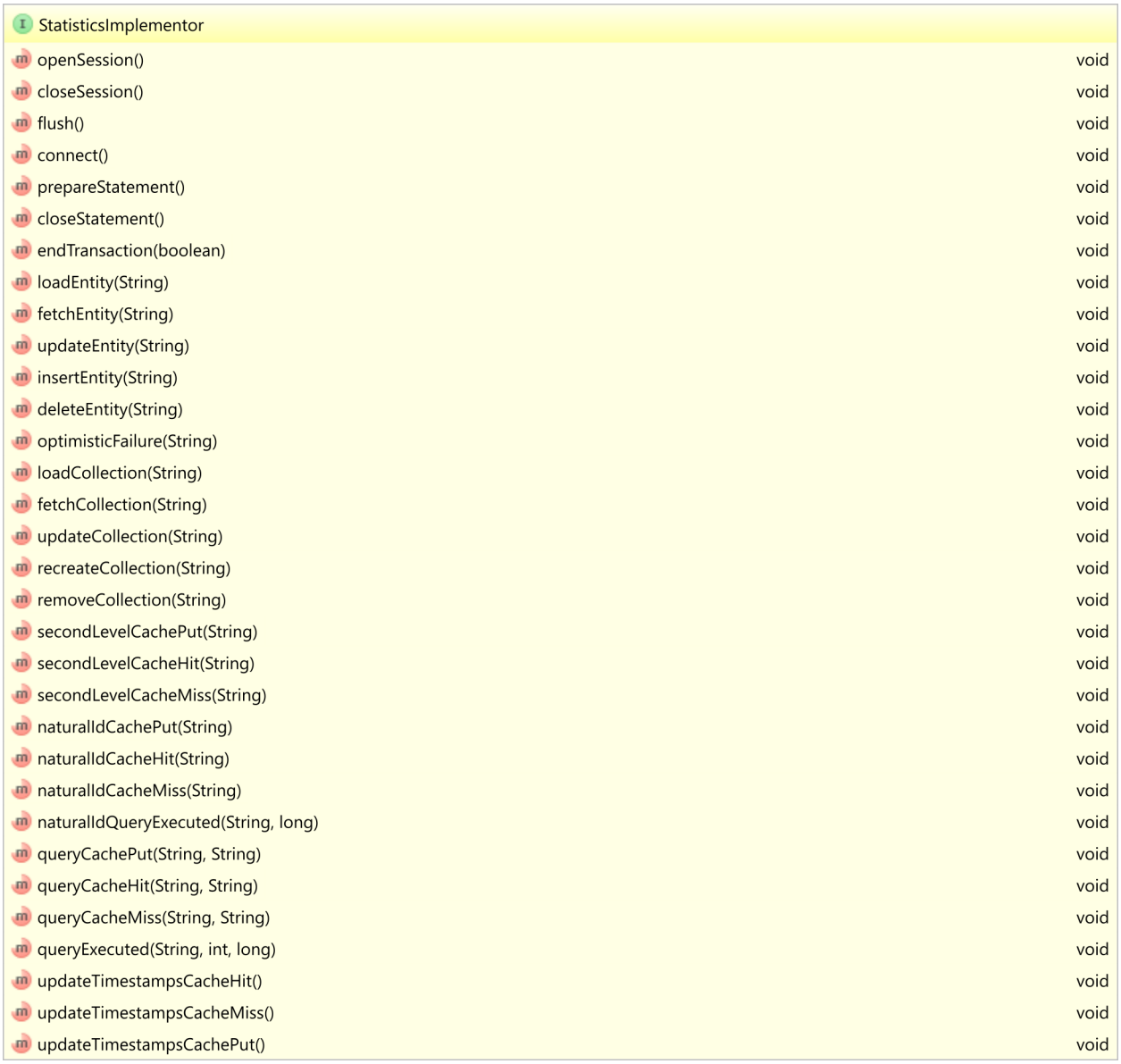


Figure 9.3: Hibernate `StatisticsImplementor` interface

There is a great variety of metrics Hibernate can collect on user's behalf, but, for performance reasons, the statistics mechanism is disabled by default.

To enable the statistics gathering mechanism, the following property must be configured first:

```
<property name="hibernate.generate_statistics" value="true"/>
```

Once statistics are being collected, in order to print them into the current application log, the following logger configuration must be set up:

```
<logger
    name="org.hibernate.engine.internal.StatisticalLoggingSessionEventListener"
    level="info" />
```

With these two settings in place, whenever a Hibernate `Session` (Persistence Context) ends, the following report will be displayed in the current running log.

```
37125102 nanoseconds spent acquiring 10000 JDBC connections;
25521714 nanoseconds spent releasing 10000 JDBC connections;
95242323 nanoseconds spent preparing 10000 JDBC statements;
923615040 nanoseconds spent executing 10000 JDBC statements;
```

The default statistics collector just counts the number of times a certain callback method was called, and, if that's not satisfactory, the application developer can supply its own custom `StatisticsImplementor` implementation.

Dropwizard Metrics

In a high-throughput transaction system, the amount of metric data needed to be recorded can be overwhelming, so storing all these values into memory is not really practical at all.

To reduce the memory footprint, the Dropwizard Metrics[a] project uses various *reservoir sampling*[b] strategies, that either employ a fixed-size sampler or a time-based sampling window.

Not only it supports a great variety of metrics (e.g. timers, histograms, gauges), but Dropwizard Metrics can use multiple reporting channels as well (e.g. SLF4J, JMX, Ganglia, Graphite).

For all these reasons, it's better to use a mature framework such as Dropwizard Metrics instead of building a custom implementation from scratch.

[a] https://github.com/dropwizard/metrics
[b] https://en.wikipedia.org/wiki/Reservoir_sampling

9.3.1.1 Customizing statistics

Although the built-in metrics are rather informative, Hibernate is not limited to the default statistics collector mechanism which can be completely customized.

In the upcoming example, the statistics collector will also provide the following metrics:

- the distribution of physical transaction time (the interval between the moment a connection is first acquired and when it gets released)
- a histogram of the number of connections acquisition requests during the lifespan of any given transaction (due to the `after_statement` release mode).

The `StatisticsReport` class provides metric storage and report generation features on top of Dropwizard Metrics:

```
public class StatisticsReport {

    private final Logger LOGGER = LoggerFactory.getLogger(getClass());

    private MetricRegistry metricRegistry = new MetricRegistry();

    private Histogram connectionCountHistogram = metricRegistry.
        histogram("connectionCountHistogram");

    private Timer transactionTimer = metricRegistry.
        timer("transactionTimer");

    private Slf4jReporter logReporter = Slf4jReporter
            .forRegistry(metricRegistry)
            .outputTo(LOGGER)
            .build();

    public void transactionTime(long nanos) {
        transactionTimer.update(nanos, TimeUnit.NANOSECONDS);
    }

    public void connectionsCount(long count) {
        connectionCountHistogram.update(count);
    }

    public void generate() {
        logReporter.report();
    }
}
```

The `StatisticsImplementor` interface defines the contract between the Hibernate internal API and the various custom statistics gathering implementations. For simplicity sake, the following `StatisticsImplementor` interface implementation extends the default `ConcurrentStatisticsImpl` class, as it only needs to override the `connect()` and the `endTransaction(boolean success)` callback methods.

```
public class TransactionStatistics extends ConcurrentStatisticsImpl {

    private ConcurrentMap<Long, Long> transactionStartNanos =
        new ConcurrentHashMap<>();
    private ConcurrentMap<Long, AtomicLong> connectionCounter =
        new ConcurrentHashMap<>();
    private StatisticsReport report = new StatisticsReport();

    @Override public void connect() {
        long threadId = Thread.currentThread().getId();
        AtomicLong counter = connectionCounter.get(threadId);
        if(counter == null) {
            counter = new AtomicLong();
            connectionCounter.put(threadId, counter);
        }
        counter.incrementAndGet();
        transactionStartNanos.putIfAbsent(threadId, System.nanoTime());
        super.connect();
    }

    @Override public void endTransaction(boolean success) {
        long threadId = Thread.currentThread().getId();
        Long startNanos = transactionStartNanos.remove(threadId);
        if (startNanos != null)
            report.transactionTime(System.nanoTime() - startNanos);
        AtomicLong connectionCounter = this.connectionCounter.remove(threadId);
        if (connectionCounter != null)
            report.connectionsCount(connectionCounter.longValue());
        report.generate();
        super.endTransaction(success);
    }
}
```

Because the `StatisticsImplementor` is a singleton instance, therefore being accessed by multiple concurrent running `Sessions(s)`, the context correlation is done based on the current running `Thread` identifier. When a transaction ends, the report is generated, and both the physical transaction time and the number of connections (issued during a particular transaction) are flushed to the log.

The report is not generated at the end of each transaction, and not in the `logSummary()` callback which only gets called when the Hibernate `Session` gets closed, because a Persistence Context can run multiple successive transactions.

To use a custom `StatisticsImplementor` instance, Hibernate requires a `StatisticsFactory` supplied as a configuration property. Taking a `SessionFactoryImplementor` parameter, the `StatisticsImplementor` building process has access to the Hibernate configuration data as well.

```
public class TransactionStatisticsFactory implements StatisticsFactory {
    @Override public StatisticsImplementor buildStatistics(
        SessionFactoryImplementor sessionFactory) {
        return new TransactionStatistics();
    }
}
```

The `hibernate.stats.factory` configuration property must contain the fully qualified name of the `StatisticsFactory` implementation class:

```
<property name="hibernate.stats.factory" value="com.vladmihalcea.book.hpjp.hiber\
nate.statistics.TransactionStatisticsFactory" />
```

When running the previous JTA connection release mode example along with this custom statistics collector, the following output is being displayed:

```
type=HISTOGRAM, name=connectionCounterHistogram, count=107,
min=1, max=10000, mean=162.41, stddev=1096.69,
median=1.0, p75=1.0, p95=50.0, p98=1000.0, p99=5000.0, p999=10000.0

type=TIMER, name=transactionTimer, count=107,
min=0.557524, max=1272.75, mean=27.16, stddev=152.57,
median=0.85, p75=1.24, p95=41.25, p98=283.50, p99=856.19, p999=1272.75,
mean_rate=36.32, rate_unit=events/second, duration_unit=milliseconds
```

For a high-performance data access layer, statistics and metrics becomes mandatory requirements. The Hibernate statistics mechanism is a very powerful tool, allowing the development team to get a better insight into Hibernate inner workings.

9.4 Statement logging

An ORM tool can automatically generate DML statements, and it is the application developer responsibility to validate both their effectiveness as well as their overall performance impact. Deferring the SQL statement validation until the data access layer starts showing performance issues is risky, and it can even impact development cost. For this reason, SQL statement logging becomes relevant from the early stages of application development.

When a business logic is implemented, the Definition of Done should include a review of all the associated data access layer operations. Following this rule can save a lot of hassle when the enterprise system is deployed into production.

Although the JPA 2.1 doesn't feature a standard configuration property for logging SQL statements, most JPA implementations support this feature through framework-specific setups. For this purpose, Hibernate defines the following configuration properties:

Table 9.4: Connection release modes

Property	Description
`hibernate.show_sql`	Prints SQL statements to the console
`hibernate.format_sql`	Formats SQL statements before being logged or printed to the console
`hibernate.use_sql_comments`	Adds comments to the automatically generated SQL statement

Using the `System` console for logging is bad practice, a logging framework (e.g. Logback or Log4j) being a better alternative for it supports configurable appenders and logging levels.

Because it prints to the console, the `hibernate.show_sql` property should be avoided.

Hibernate logs all SQL statements on a `debug` level in the `org.hibernate.SQL` logging hierarchy.

To enable statement logging, the following Logback logger must be added in the associated configuration file:

```
<logger name="org.hibernate.SQL" level="debug"/>
```

Because Hibernate uses `PreparedStatement(s)` exclusively, the bind parameter values are not available when the statement gets printed into the log:

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

Although bind parameters might be logged separately (e.g. `org.hibernate.type.descriptor.sql`), the most straight-forward way of logging SQL statements along with their runtime bind parameter values is to use an external `DataSource` proxy. Because the proxy intercepts all statement executions, the bind parameter values can be introspected and printed as well.

9.4.1 Statement formatting

By default, every SQL statement, no matter how long, is written as a single line of text. To increase readability, Hibernate can transform SQL statements in a human-readable format that spans over multiple log lines. This feature can be activated by setting the following configuration property:

```
<property name="hibernate.format_sql" value="true" />
```

With this setting in place, the previous statement can be formatted as follows:

```
insert
    into
        post
        (title, version, id)
    values
        (?, ?, ?)
```

Although formatting statements can improve readability, this setting is only suitable during the development phase. In a production system, logs are often parsed and aggregated in a centralized system, and the multi-line statement format can impact the log parsing mechanism. Once aggregated, logged queries can be formatted prior to being displayed in the application performance monitoring user interface.

The `hibernate.format_sql` property applies to logged statements only and it doesn't propagate to the underlying JDBC `Driver` (SQL statements are still sent as single lines of text).

This way, the statement formatting doesn't have any effect when statements are logged through an external `DataSource` proxy.

9.4.2 Statement-level comments

Besides formatting, Hibernate can explain the statement generation process by appending SQL-level comments into the statement body. This feature allows the application developer to get a better understanding of the following processes:

- the entity state transition that triggered the current executing statement
- the reason for choosing a join when fetching a given result set
- the explicit locking mechanism employed by the current statement.

By default, Hibernate doesn't append any SQL comment in the automatically generated statements, and, to enable this mechanism, the following Hibernate property must be configured:

```
<property name="hibernate.use_sql_comments" value="true" />
```

When persisting a `Post` entity, Hibernate explains the entity state transition associated with this particular statement through the following comment:

```
/* insert com.vladmihalcea.book.hpjp.util.providers.BlogEntityProvider$Post */
INSERT INTO post (title, version, id) VALUES (?, ?, ?)
```

As opposed to SQL statement formatting, SQL comments are not only generated during logging, and they propagate to the underlying `Driver` as well.

Although it might be a useful technique for debugging purposes, in a production environment, it's better to leave it disabled, to reduce the database request networking overhead.

9.4.3 Logging parameters

Either the JDBC `Driver` or the `DataSource` must be proxied to intercept statement executions and log them along with the actual parameter values. Besides statement logging, a JDBC proxy can provide other cross-cutting features like long-running query detection or custom statement execution listeners.

9.4.3.1 DataSource-proxy

A lesser-known JDBC logging framework, datasource-proxy[5] provides support for custom JDBC statement execution listeners. In Java EE, not all application servers allow configuring an external `DataSource`, as they rely on their own custom implementations that bind the user supplied JDBC `Driver`. Because it can only decorate a `DataSource`, datasource-proxy might not be suitable in all Java EE environments.

Figure 9.4: DataSource-Proxy Architecture

On the other hand, the programmatic configuration support fits the Java-based configuration approach taken by most modern Spring applications:

```
@Bean
public DataSource dataSource() {
    SLF4JQueryLoggingListener loggingListener = new SLF4JQueryLoggingListener();
    loggingListener.setQueryLogEntryCreator(new InlineQueryLogEntryCreator());
    return ProxyDataSourceBuilder
        .create(actualDataSource())
        .name(DATA_SOURCE_PROXY_NAME)
        .listener(loggingListener)
        .build();
}
```

[5] https://github.com/ttddyy/datasource-proxy

In the following example datasource-proxy is used to log a batch insert of three `PreparedStatement(s)`. Although normally a batch is printed in a single line of log, the output was split into multiple lines to fit the current page layout.

```
Name:DATA_SOURCE_PROXY, Time:6, Success:True,
Type:Prepared, Batch:True, QuerySize:1, BatchSize:3,
Query:["insert into post (title, version, id) values (?, ?, ?)"],
Params:[(Post no. 0, 0, 0), (Post no. 1, 0, 1), (Post no. 2, 0, 2)]
```

Not only the bind parameter values are now present, but, because they are grouped altogether, it's very easy to visualize the batching mechanism too.

With the custom statement listener support, datasource-proxy allows building a query count validator to assert the auto-generated statement count, and therefore prevent N+1 query problems during development phase.

9.4.3.2 P6Spy

P6Spy[6] was released in 2002, in an era when J2EE application servers were ruling the world of enterprise systems. Because Java EE application servers don't allow programmatic `DataSource` configuration, P6Spy supports a declarative configuration approach (through a `spy.properties` file).

P6Spy offers support for proxying both a JDBC `Driver` (which is suitable for Java EE applications) or a JDBC `DataSource` (supported by some Java EE containers and common practice for Spring enterprise applications).

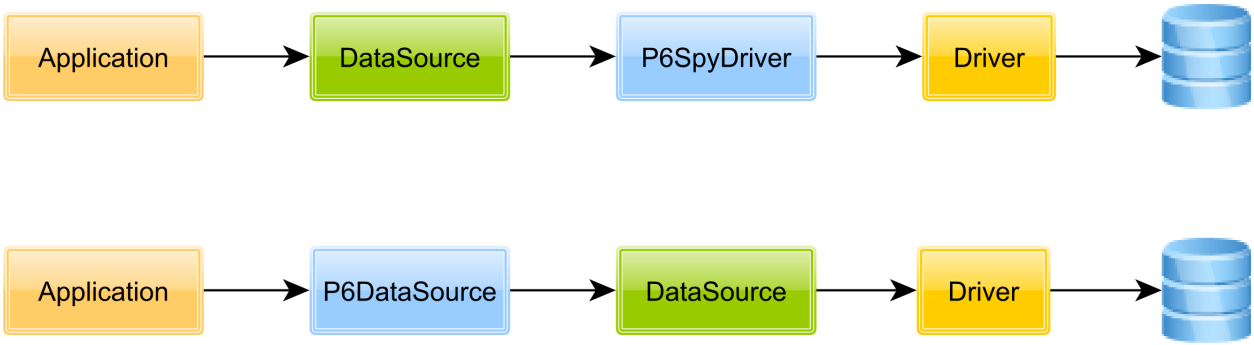


Figure 9.5: P6Spy Architecture

[6] https://github.com/p6spy/p6spy

Running the previous example gives the following output (formatting was also applied):

```
p6spy - 1448122491807|0|batch|connection 7|
        insert into post (title, version, id) values (?, ?, ?)|
        insert into post (title, version, id) values ('Post no. 0', 0, 0)
p6spy - 1448122491807|0|batch|connection 7|
        insert into post (title, version, id) values (?, ?, ?)|
        insert into post (title, version, id) values ('Post no. 1', 0, 1)
p6spy - 1448122491807|0|batch|connection 7|
        insert into post (title, version, id) values (?, ?, ?)|
        insert into post (title, version, id) values ('Post no. 2', 0, 2)
p6spy - 1448122491812|5|statement|connection 7|
        insert into post (title, version, id) values (?, ?, ?)|
        insert into post (title, version, id) values ('Post no. 2', 0, 2)
```

In the order of their occurrence, the output is built out of the following columns:

Table 9.5: P6Spy output

Field	Description
Timestamp	The statement execution timestamp
Execution time	The statement execution duration (in milliseconds)
Category	The current statement category (e.g. statement, batch)
Connection	The database connection identifier (as assigned by P6Spy)
Original statement	The original statement that was intercepted by P6Spy
Formatted statement	The statement with all parameter placeholders replaced with the actual bind values

The first three lines are associated with adding statements to the batch, while the fourth line is logging the actual batch execution (which also explains the *execution time* column value).

One very useful configuration is the `outagedetection` property, which can detect long-running statements.

10. Mapping Types and Identifiers

JPA addresses the Object/Relational mismatch by associating Java object types to database structures Assuming there is a `task` database table having four columns (e.g. `id`,`created_by`, `created_on` and `status`), the JPA provider must map it to Domain Model consisting of two Java class (e.g. `Task` and `Change`).

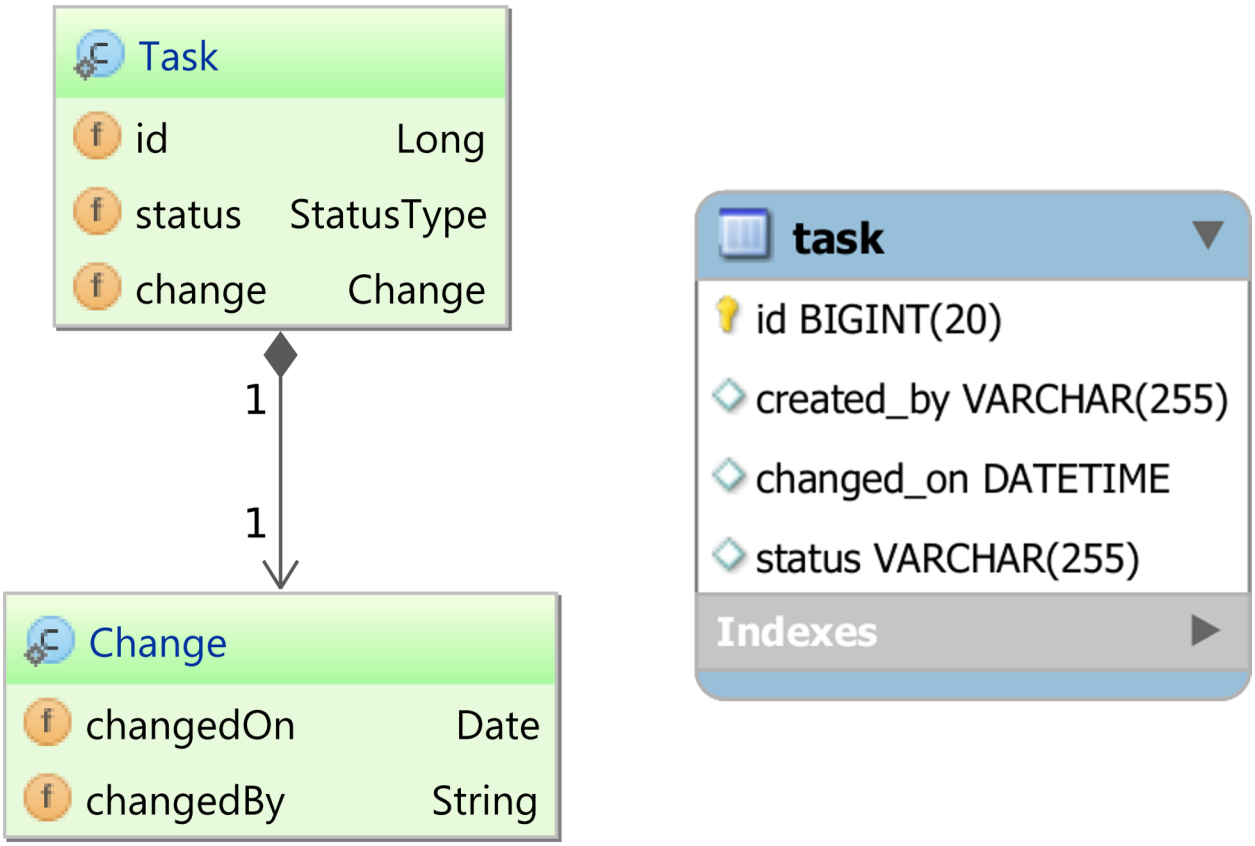


Figure 10.1: Type Mapping

JPA uses three main Object-Relational mapping elements: *type*, *embeddable* and *entity*. In the previous diagram, the `Task` object is an entity, while the `Change` object is an embeddable type.

Both the entity and the embeddable group multiple Domain Model properties by relying on Hibernate `Type(s)` to associate database column types with Java value objects (e.g. `String`, `Integer`, `Date`). The major difference between an entity and an embeddable is the presence of an identifier, which is used to associate a database table unique key (usually the primary key) with a Domain Model object property.

Although it's common practice to map all database columns, this is not a strict requirement. Sometimes it's more practical to use a root entity and several sub-entities, so each business case fetches just as much info as needed (while still benefiting from entity state management).

Identifiers are mandatory for entity elements, and an embeddable type is forbidden to have an identity of its own. Knowing the database table and the column that uniquely identifies any given row, Hibernate can correlate database rows with Domain Model entities.

An embeddable type groups multiple properties in a single reusable component.

```
@Embeddable
public class Change {
    @Column(name = "changed_on")
    private Date changedOn;

    @Column(name = "created_by")
    private String changedBy;
}
```

The Domain Model can share state between multiple entities either by using inheritance or composition. Embeddable types can reuse state through composition.

The composition association[1], defined by UML, is the perfect analogy for the relationship between an entity and an embeddable. When an entity includes an embeddable type, all its properties become part of the owner entity.

The embeddable object cannot define its own identifier because otherwise the entity will have more than one identities. Lacking an identifier, the embeddable object cannot be managed by a Persistence Context, and its state is controlled by its parent entity.

Because they can have a significant impact on the overall application performance, the rest of the chapter discusses types and identifiers in great detail.

[1] https://en.wikipedia.org/wiki/Object_composition

10.1 Types

For every supported database type, JDBC defines a `java.sql.JDBCType` enumeration. Since it builds on top of JDBC, Hibernate does the mapping between JDBC types and their associated Java counterparts (primitives or Objects).

10.1.1 Primitive types

Table 10.1: Primitive Types

Hibernate type	JDBC type	Java type
`BooleanType`	`BIT`	`boolean`, `Boolean`
`NumericBooleanType`	`INTEGER` (e.g. `0`, `1`)	`boolean`, `Boolean`
`TrueFalseType`	`CHAR` (e.g. `'F'`, `'f'`, `'T'`, `'t'`)	`boolean`, `Boolean`
`YesNoType`	`CHAR` (e.g. `'N'`, `'n'`, `'Y'`, `'y'`)	`boolean`, `Boolean`
`ByteType`	`TINYINT`	`byte`, `Byte`
`ShortType`	`SMALLINT`	`short`, `Short`
`CharacterType`	`CHAR`	`char`, `Character`
`CharacterNCharType`	`NCHAR`	`char`, `Character`
`IntegerType`	`INTEGER`	`int`, `Integer`
`LongType`	`BIGINT`	`long`, `Long`
`FloatType`	`FLOAT`	`float`, `Float`
`DoubleType`	`DOUBLE`	`double`, `Double`
`CharArrayType`	`VARCHAR`	`char[]`, `Character[]`

From one database system to another, the boolean type can be represented either as a `BIT`, `BYTE`, `BOOLEAN` or `CHAR` database type, so defines four `Type(s)` to resolve the boolean primitive type.

Only non-nullable database columns can be mapped to Java primitives (`boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`). For mapping nullable columns, it's better to use the primitive wrappers instead (`Boolean`, `Byte`, `Short`, `Char`, `Integer`, `Long`, `Float`, `Double`).

10.1.2 String types

A Java `String` can consume as much memory as the Java Heap has available. On the other hand, database systems define both limited-size types (`VARCHAR` and `NVARCHAR`) and unlimited ones (`TEXT`,

NTEXT, BLOB and NCLOB). To accommodate this mapping discrepancy, Hibernate defines the following Type(s):

Table 10.2: String Types

Hibernate type	JDBC type	Java type
StringType	VARCHAR	String
StringNVarcharType	NVARCHAR	String
TextType	LONGVARCHAR	String
NTextType	LONGNVARCHAR	String
MaterializedClobType	CLOB	String
MaterializedNClobType	NCLOB	String

10.1.3 Date and Time types

When it comes to time, there are multiple Java or database representations, which explains the vast number of time-related Hibernate Type(s).

Table 10.3: Date and Time Types

Hibernate type	JDBC type	Java type
DateType	DATE	Date
TimeType	TIME	Time
TimestampType	TIMESTAMP	Timestamp, Date
DbTimestampType	TIMESTAMP	Timestamp, Date
CalendarType	TIMESTAMP	Calendar, GregorianCalendar
CalendarDateType	DATE	Calendar, GregorianCalendar
CalendarTimeType	TIME	Calendar, GregorianCalendar
TimeZoneType	VARCHAR	TimeZone

Handling time is tricky because of various time zones, leap seconds and day-light saving conventions. Storing timestamps in UTC (Coordinated Universal Time) and doing time zone transformations in the data layer is common practice.

10.1.4 Numeric types

Oracle can represent numbers up to *38* digits, therefore only fitting in a `BigInteger` or a `BigDecimal` (`Long` and `Double` can only store up to *8* bytes).

Table 10.4: Numeric Types

Hibernate type	JDBC type	Java type
`BigIntegerType`	`NUMERIC`	`BigInteger`
`BigDecimalType`	`NUMERIC`	`BigDecimal`

10.1.5 Binary types

For binary types, most database systems offer multiple storage choices (e.g. `RAW`, `VARBINARY`, `BYTEA`, `BLOB`, `CLOB`). In Java, the data access layer can use an array of `byte(s)`, a JDBC `Blob` or `Clob`, or even a `Serializable` type, if the Java object was marshaled prior to being saved to the database.

Table 10.5: Binary Types

Hibernate type	JDBC type	Java type
`BinaryType`	`VARBINARY`	`byte[]`, `Byte[]`
`BlobType`	`BLOB`	`Blob`
`ClobType`	`CLOB`	`Clob`
`NClobType`	`NCLOB`	`Clob`
`MaterializedBlobType`	`BLOB`	`byte[]`, `Byte[]`
`ImageType`	`LONGVARBINARY`	`byte[]`, `Byte[]`
`SerializableType`	`VARBINARY`	`Serializable`
`SerializableToBlobType`	`BLOB`	`Serializable`

10.1.6 UUID types

There are various ways of persisting a Java `UUID` (Universally Unique Identifier), and, based on the memory footprint, the most efficient storage types are the database-specific UUID column types.

Table 10.6: UUID Types

Hibernate type	JDBC type	Java type
`UUIDBinaryType`	`BINARY`	`UUID`
`UUIDCharType`	`VARCHAR`	`UUID`
`PostgresUUIDType`	`OTHER`	`UUID`

When not natively supported, a `BINARY` type requires less bytes than a `VARCHAR`, so the associated index will have a smaller memory footprint too.

10.1.7 Other types

Hibernate can also map Java `Enum(s)`, `Class`, `URL`, `Locale` and `Currency` too.

Table 10.7: Other Types

Hibernate type	JDBC type	Java type
`EnumType`	`CHAR`, `LONGVARCHAR`, `VARCHAR` `INTEGER`, `NUMERIC`, `SMALLINT`, `TINYINT`, `BIGINT`, `DECIMAL`, `DOUBLE`, `FLOAT`	`Enum`
`ClassType`	`VARCHAR`	`Class`
`CurrencyType`	`VARCHAR`	`Currency`
`LocaleType`	`VARCHAR`	`Locale`
`UrlType`	`VARCHAR`	`URL`

10.1.8 Custom types

Not only that it has a very rich set of data types, but PostgreSQL allows adding custom types as well (using the `CREATE DOMAIN`[2] DDL statement). Choosing the appropriate database type for each Domain Model field can really make a difference in terms of data access performance. Although there is a great variety of built-in `Type(s)`, the application developer is not limited to the off-the-shelf ones only, and new `Type(s)` can be added without too much effort.

In the following example, the business logic requires monitoring access to an enterprise application. For this purpose, the data access layer stores the IP (Internet Protocol) addresses of each logged-in user.

Assuming this internal application uses the IPv4 protocol only, the IP addresses are stored in the Classless Inter-Domain Routing format (e.g. `192.168.123.231/24`). PostgreSQL can store IPv4 addresses either in a `cidr` or `inet` type, or it can use a `VARCHAR(18)` column type.

The `VARCHAR(18)` column requires *18* characters, and, assuming a UTF-8 encoding, each IPv4

[2] http://www.postgresql.org/docs/9.5/static/sql-createdomain.html

address needs at most *18* bytes. The smallest size address (e.g. `0.0.0.0/0`) taking *9* characters, the `VARCHAR(18)` approach requires between *9* and *18* characters for each IPv4 address.

The `inet` type is specially designed for IPv4 and IPv6 network addresses, and it also supports various network address specific operators (e.g. `<`, `>`, `&&`), as well as other address transforming functions (e.g. `host(inet)`, `netmask(inet)`). As opposed to the `VARCHAR(18)` approach, the `inet` type requires only *7* bytes for each IPv4 address.

For it has a more compact size (the index can better fit into memory) and supporting many specific operators, the `inet` type is a much more attractive choice. Although, by default, Hibernate doesn't support `inet` types, adding a custom Hibernate `Type` is a straightforward task. The IPv4 address is encapsulated in its own wrapper, which can also define various address manipulation functions too.

```
public class IPv4 implements Serializable {

    private final String address;

    public IPv4(String address) {
        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        return Objects.equals(address, IPv4.class.cast(o).address);
    }

    @Override public int hashCode() {
        return Objects.hash(address);
    }

    public InetAddress toInetAddress() throws UnknownHostException {
        return Inet4Address.getByName(address);
    }
}
```

When an entity wants to change an `IPv4` field, it must provide a new object instance. An immutable type is much easier to handle since its internal state doesn't change throughout the current running Persistence Context.

All custom types must implement the `UserType` interface, and, since the `ImmutableType` takes care of most `UserType` implementation details, the `IPv4Type` can focus on type specific conversation logic.

```
public class IPv4Type extends ImmutableType<IPv4> {

    public IPv4Type() {
        super(IPv4.class);
    }

    @Override public int[] sqlTypes() { return new int[]{ Types.OTHER}; }

    @Override public IPv4 get(ResultSet rs, String[] names,
        SessionImplementor session, Object owner) throws SQLException {
        String ip = rs.getString(names[0]);
        return (ip != null) ? new IPv4(ip) : null;
    }

    @Override public void set(PreparedStatement st, IPv4 value, int index,
        SessionImplementor session) throws SQLException {
        if (value == null) {
            st.setNull(index, Types.OTHER);
        } else {
            PGobject holder = new PGobject();
            holder.setType("inet");
            holder.setValue(value.getAddress());
            st.setObject(index, holder);
        }
    }
}
```

The `get()` method is used to map the `inet` field to an `IPv4` object instance, while the `set()` is used for transforming the `IPv4` object to the PostgreSQL JDBC driver `inet` equivalent.

`Types.OTHER` is used for mapping database types not supported by JDBC.

```
public abstract class ImmutableType<T> implements UserType {
    private final Class<T> clazz;

    protected ImmutableType(Class<T> clazz) { this.clazz = clazz; }

    @Override public Object nullSafeGet(ResultSet rs, String[] names,
        SessionImplementor session, Object owner) throws SQLException {
        return get(rs, names, session, owner);
    }

    @Override public void nullSafeSet(PreparedStatement st, Object value,
        int index, SessionImplementor session) throws SQLException {
        set(st, clazz.cast(value), index, session);
    }

    @Override public Class<T> returnedClass() { return clazz; }

    @Override public boolean equals(Object x, Object y) {
        return Objects.equals(x, y);
    }

    @Override public int hashCode(Object x) { return x.hashCode(); }

    @Override public Object deepCopy(Object o) { return o; }

    @Override public boolean isMutable() { return false; }

    @Override public Serializable disassemble(Object o) {
        return (Serializable) o;
    }

    @Override public Object assemble(Serializable o, Object owner) { return o; }

    @Override
    public Object replace(Object o, Object target, Object owner) { return o; }

    protected abstract T get(ResultSet rs, String[] names,
        SessionImplementor session, Object owner) throws SQLException;

    protected abstract void set(PreparedStatement st, T value, int index,
        SessionImplementor session) throws SQLException;
}
```

The `@Type` annotation instructs Hibernates to use the `IPv4Type` for mapping the `IPv4` field.

```
@Entity
public class Event {

    @Id @GeneratedValue
    private Long id;

    @Type(type = "com.vladmihalcea.book.hpjp.hibernate.type.IPv4Type")
    @Column(name = "ip", columnDefinition = "inet")
    private IPv4 ip;

    public Event() {}

    public Event(String address) {
        this.ip = new IPv4(address);
    }

    public Long getId() {
        return id;
    }

    public IPv4 getIp() {
        return ip;
    }

    public void setIp(String address) {
        this.ip = new IPv4(address);
    }
}
```

GiST operators

PostgreSQL 9.4 added GiST operator support[a] for `inet` and `cidr` column types. To enable this feature, a GiST index with the `inet_ops` operator class must be created on the associated `inet` columns.

```
CREATE INDEX ON event USING gist (ip inet_ops)
```

[a] http://www.postgresql.org/docs/9.4/static/gist-builtin-opclasses.html

Managing `Event(s)` is easy when Hibernate takes care of the underlying type conversation.

```
final AtomicReference<Event> eventHolder = new AtomicReference<>();

doInJPA(entityManager -> {
    entityManager.persist(new Event());
    Event event = new Event("192.168.0.231");
    entityManager.persist(event);
    eventHolder.set(event);
});

doInJPA(entityManager -> {
    Event event = entityManager.find(Event.class, eventHolder.get().getId());
    event.setIp("192.168.0.123");
});
```

Running the previous example generates the following SQL statements:

```
INSERT INTO event (ip, id) VALUES (NULL(OTHER), 1)
INSERT INTO event (ip, id) VALUES (`192.168.0.231`, 2)

SELECT e0_.id as id1_0_0_, e0_.ip as ip2_0_0_
FROM event e0_
WHERE e0_.id = 2

UPDATE event SET ip=`192.168.0.123` WHERE id = 2
```

One of the best aspects of using database-specific types is getting access to advanced querying capabilities. Because the GiST index allows `inet_ops` operators, the following query can be used to check if an `Event` was generated for a given subnetwork:

```
Event matchingEvent = (Event) entityManager.
        createNativeQuery(
                "SELECT {e.*} " +
                "FROM event e " +
                "WHERE " +
                "   e.ip && CAST(:network AS inet) = TRUE", Event.class).
        setParameter("network", "192.168.0.1/24").
        getSingleResult();
assertEquals("192.168.0.123", matchingEvent.getIp().getAddress());
```

10.2 Identifiers

All database tables must have a primary key column, so each row can be uniquely identified (the primary key must be both `UNIQUE` and `NOT NULL`).

Although the SQL standard doesn't impose primary keys to be immutable, it's more practical[a] to avoid changing them.

The primary key can have a meaning in the real world, in which case it's a *natural* key, or it can be generated synthetically, in which case it's called a *surrogate* identifier.

For natural keys, unicity is enforced by a real-world unique sequence generator (e.g. National Identification Numbers, Social Security Numbers, Vehicle Identification Numbers). In reality, natural unique numbers might pose problems when the unique constraints do not hold true anymore. For example, a National Identification Number might yield unique numbers, but if the enterprise system must accommodate users coming from multiple countries, it's possible that two different countries assigned the same identifier.

The natural key can be composed of one or multiple columns. Compound natural keys might incur an additional performance penalty because multi-column joins are slower than single-column ones, and multi-column indexes have a bigger memory footprint too.

Natural keys must be sufficiently long to accommodate as many identifiers as the system needs throughout its lifecycle. Because primary keys are often indexed, the longer the key, the more memory an index entry will require. Each joined table will include a foreign key mirroring the parent primary key, and foreign keys are frequently indexed as well.

Index memory impact

Fixed-size non-numerical keys (e.g. `CHAR`, `VARCHAR`) are less efficient than numerical ones (e.g. `INTEGER`, `BIGINT`) both for joining (a simple key performs better than a compound one) or indexing (the more compact the data type, the less memory space is required by an associated index).

A `CHAR(17)` natural key (e.g. Vehicle Identification Number) requires *17* characters (17 bytes when using ASCII characters and a UTF-8 encoding) as opposed to *4* bytes (*32* bit `INTEGER`) or *8* bytes (*64* bit `BIGINT`).

Surrogate keys are generated independently of the current row data, so table column constraints may evolve with time (changing a user birthday or email address). The surrogate key can be generated by a numerical sequence generator (e.g. a database identity column or a sequence), or it can be constructed by a pseudorandom number generator (e.g GUID[3] or UUID[4]). Both the numerical and UUID keys have both pros and cons.

The UUID is defined by the RFC 4122[5] standard and it is stored as a *128* bit sequence. The GUID term refers to any globally unique identifier, which might comprise other non-standard implementations. For consistency, this chapter will further refer to unique identifiers as UUID.

A UUID takes *128* bits, which is four times more than an `INTEGER` and twice as as `BIGINT`. On the other hand, a UUID number has less chance of a conflict in a Multi-Master database replication topology. To avoid such conflicts, many relational database systems increment the identity or sequence numbers in steps, each node getting its own offset. Because UUIDs are not sequential, they induce fragmentation and that can really affect the performance of clustered indexes.

Requiring less space and being more index-friendly, numerical sequences are preferred over UUID keys.

10.2.1 UUID identifiers

Nevertheless, some enterprise systems use UUID primary keys, so it's worth knowing what Hibernate types work best for this task. The UUID key can either be generated by the application using the `java.util.UUID` class or it can be assigned by the database system.

If the database system doesn't have a built-in UUID type, a `BINARY(16)` column type is preferred. Although a `CHAR(32)` column could also store the UUID textual representation, the additional space overhead makes it a less favorable pick.

[3] http://en.wikipedia.org/wiki/Globally_Unique_Identifier
[4] http://en.wikipedia.org/wiki/Universally_Unique_Identifier
[5] https://www.ietf.org/rfc/rfc4122.txt

Oracle

There is no UUID type in Oracle, so a `RAW(16)` column must be used instead. The `SYS_GUID()`[a] database function can generate a globally unique identifier.

[a] http://docs.oracle.com/database/121/SQLRF/functions202.htm#SQLRF06120

SQL Server

The `uniqueidentifier`[a] data type is used for storing GUID identifiers. The `NEWID()`[b] function can generate a UUID compatible with the RFC 4122 standard.

Because by default SQL Server uses clustered indexes for primary keys, to avoid the fragmentation effect, the `NEWSEQUENTIALID()`[c] function can assign pseudo-sequential UUID numbers (greater than previously generated ones). This guarantee is kept as long as the Windows server is not restarted.

[a] https://msdn.microsoft.com/en-us/library/ms187942.aspx
[b] https://msdn.microsoft.com/en-us/library/ms190348.aspx

PostgreSQL

The `UUID` type[a] can store RFC 4122 compliant unique identifiers. The database doesn't offer a built-in UUID generation function, so the identifier must be generated by the data access layer.

[a] http://www.postgresql.org/docs/9.5/static/datatype-uuid.html

MySQL

The UUID must be stored in a `BINARY(16)` column type. The `UUID()`[a] functions can generate a *128* bit unique identifier. Because the `UUID()` function might cause problems for statement-based replication, passing the generated identifier as a variable[b] is a workaround to this limitation.

[a] http://dev.mysql.com/doc/refman/5.7/en/miscellaneous-functions.html#function_uuid
[b] https://dev.mysql.com/doc/refman/5.7/en/replication-features-functions.html

For generating UUID identifiers, Hibernate offers three generators (*assigned*, *uuid*, *uuid2*), which we'll be discussed in great detail in the following sections.

10.2.1.1 The assigned generator

By simply omitting the `@GeneratedValue`[6] annotation, Hibernate will fall back to the assigned identifier, which allows the data access layer to control the identifier generation process. The following example maps a `java.util.UUID` identifier to a `BINARY(16)` column type:

```
@Entity @Table(name = "post")
public class Post {

    @Id @Column(columnDefinition = "BINARY(16)")
    private UUID id;

    public Post() {}

    public Post(UUID id) {
        this.id = id;
    }
}
```

When persisting a *post*, Hibernate generates the following insert statement:

```
INSERT INTO post (id) VALUES
([86, 10, -104, 26, 60, -115, 79, 78, -118, -45, 64, 94, -64, -40, 66, 100])
```

The `UUIDBinaryType` translates the `java.util.UUID` to an array of `byte(s)` that's stored in the associated `BINARY(16)` column type.

Because the identifier is generated in the data access layer, the database server is freed from this responsibility, and so it can allocate its resources for other data processing tasks.

Hibernate can also generate a UUID identifier on behalf of the application developer, as described in the following two sections.

[6] https://docs.oracle.com/javaee/7/api/javax/persistence/GeneratedValue.html

10.2.2 The legacy UUID generator

The UUID hex generator[7] is registered under the `uuid` name and generates hexadecimal UUID string representations. Using a *8-8-4-8-4* byte layout, the UUID hex generator is not compliant with the RFC 4122 standard, which uses a *8-4-4-4-12* byte format. The following code snippet depicts the `UUIDHexGenerator` mapping and the associated insert statement.

```
@Entity @Table(name = "post")
public class Post {

    @Id @Column(columnDefinition = "CHAR(32)")
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid")
    private String id;
}
```

```
INSERT INTO post (id) VALUES (402880e451724a820151724a83d00000)
```

10.2.2.1 The newer UUID generator

The newer UUID generator[8] is RFC 4122 compliant (variant 2) and is registered under the `uuid2` name (working with `java.lang.UUID`, `byte[]` and `String` Domain Model object types). Compared to the previous use case, the mapping and the test case look as follows:

```
@Entity @Table(name = "post")
public class Post {

    @Id @Column(columnDefinition = "BINARY(16)")
    @GeneratedValue(generator = "uuid2")
    @GenericGenerator(name = "uuid2", strategy = "uuid2")
    private UUID id;
}
```

```
INSERT INTO post (id) VALUES
([77, 2, 31, 83, -45, -98, 70, 40, -65, 40, -50, 30, -47, 16, 30, 124])
```

[7] http://docs.jboss.org/hibernate/stable/orm/javadocs/org/hibernate/id/UUIDHexGenerator.html
[8] http://docs.jboss.org/hibernate/stable/orm/javadocs/org/hibernate/id/UUIDGenerator.html

Being RFC 4122 compliant and able to operate with `BINARY` column type, the `UUIDGenerator` is preferred over the legacy `UUIDHexGenerator`.

10.2.3 Numerical identifiers

As previously explained, a numerical surrogate key is usually preferred since it takes less space and indexes work better with sequential identifiers. To generate numerical identifiers, most database systems offer either identity (or auto_increment) columns or sequence objects.

JPA defines the `GenerationType`[9] enumeration for all supported identifier generator types:

- `IDENTITY` is for mapping the entity identifier to a database identity column
- `SEQUENCE` allocates identifiers by calling a given database sequence
- `TABLE` is for relational databases that don't support sequences (e.g MySQL 5.7), the *table* generator emulating a database sequence by using a separate table
- `AUTO` decides the identifier generation strategy based on the current database dialect.

As explained in the JDBC part, database sequences work better with batch updates and allow various application-side optimization techniques as well.

10.2.3.1 Identity generator

The identity column type (included in the SQL:2003[10] standard) is supported by Oracle 12c[11], SQL Server[12] and MySQL (AUTO_INCREMENT)[13], and it allows an integer or a bigint column to be auto-incremented on demand.

The incrementation process is very efficient since it uses a lightweight locking mechanism, as opposed to the more heavyweight transactional course-grain locks. The only drawback is that the newly assigned value can only be known after executing the actual insert statement.

[9] https://docs.oracle.com/javaee/7/api/javax/persistence/GenerationType.html
[10] http://en.wikipedia.org/wiki/SQL:2003
[11] http://docs.oracle.com/cd/E16655_01/server.121/e17209/statements_7002.htm#CJAHJHJC
[12] http://msdn.microsoft.com/en-us/library/ms186775.aspx
[13] http://dev.mysql.com/doc/refman/5.7/en/example-auto-increment.html

Batch updates

Because Hibernate separates the id generation from the actual entity insert statement, entities using the identity generator may not participate in JDBC batch updates. Hibernate issues the insert statement during the `persist()` method call, therefore breaking the transactional write-behind caching semantic used for entity state transitions.

Even if some JDBC drivers allow fetching the associated generated keys when executing a batch update, Hibernate still needs an improvement in this regard.

The identity generator can be mapped as follows:

```
@Entity @Table(name = "post")
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

The following example demonstrates how the transaction write-behind caching model is circumvented by the identity column semantics. Although disabled by default, JDBC batching was enabled to compare results between identity and sequence generators.

```
doInJPA(entityManager -> {
    for (int i = 0; i < batchSize; i++) {
        entityManager.persist(new Post());
    }
    LOGGER.debug("Flush is triggered at commit-time");
});
```

Executing the previous test case generates the following output.

```
INSERT INTO post (id) VALUES (DEFAULT)
INSERT INTO post (id) VALUES (DEFAULT)

DEBUG - Flush is triggered at commit-time
```

Because the associated entity identifier can only be known after the insert statement is executed, Hibernate will trigger the entity state transition prior to flushing the current running Persistence Context.

10.2.3.2 Sequence generator

A sequence is a database object that generates consecutive numbers. Defined by the SQL:2003 standard, database sequences are supported by Oracle, SQL Server 2012 and PostgreSQL, and, compared to identity columns, sequences offer the following advantages:

- the same sequence can be used to populate multiple columns, even across tables
- values may be preallocated to improve performance
- allowing incremental steps, sequences can benefit from application-level optimization techniques
- because the sequence call can be decoupled from the actual insert statement, Hibernate doesn't disable JDBC batch updates

To demonstrate the difference between the identity and the sequence identifier generators, the previous example will be changed to use a database sequence this time.

```
@Entity @Table(name = "post")
public static class Post {

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    private Long id;
}
```

Running the previous test case generates the following output:

```
CALL NEXT VALUE FOR hibernate_sequence
CALL NEXT VALUE FOR hibernate_sequence

DEBUG - Flush is triggered at commit-time

INSERT INTO post (id) VALUES (1, 2)
```

When executing the `persist()` method, Hibernate calls the associated database sequence and fetches an identifier for the newly persisted entity. The actual insert statement is postponed until flush-time, which allows Hibernate to take advantage of JDBC batching.

10.2.3.3 Table generator

Because of the mismatch between the identifier generator and the transactional write-behind cache, JPA offers an alternative sequence-like generator that works even when sequences are not natively supported.

A database table is used to hold the latest sequence value and row-level locking is employed to prevent two concurrent connections from acquiring the same identifier value.

Escaping transactional row-level locking

A database sequence is a non-transactional object because the sequence value allocation happens outside of the transactional context associated with the database connection requesting a new identifier. Database sequences use dedicated locks to prevent concurrent transactions from acquiring the same value, but locks are released as soon as the counter is incremented. This design ensures minimal contention even when the sequence is used concomitantly by multiple concurrent transactions.

Using a database table as a sequence is challenging, as, to prevent two transactions from getting the same sequence value, row-level locking must be used. But unlike the sequence object locks, the row-level lock is transactional, and, once acquired, it can only be released when the current transaction ends (either committing or rolling back). This would be a terrible scalability issue because a long-running transaction would prevent any other transaction from acquiring a new sequence value.

To cope with this limitation, a separate database transactions is used for fetching a new sequence value. This way, the row-level lock associated with incrementing the sequence counter value can be released as soon as the sequence update transaction ends.

For local transactions, a new transaction means fetching another database connection and committing it after executing the sequence processing logic. This can put additional pressure on the underlying connection pool, especially if there is already a significant contention for database connections.

In a JTA environment, the current running transaction must be suspended, and the sequence value is fetched in a separate transaction. The JTA transaction manager has to do additional work to accommodate the transaction context switch, and that can also have an impact on the overall application performance.

Without any application-level optimization, the row-level locking approach can become a performance bottleneck if the sequence logic is called way too often.

To continue the previous example, the *post* will use the table generator this time:

```
@Entity @Table(name = "post")
public class Post {

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE)
    private Long id;
}
```

The following output is obtained when inserting a new *post*:

```
SELECT tbl.next_val
FROM hibernate_sequences tbl
WHERE tbl.sequence_name=default
FOR UPDATE

INSERT INTO hibernate_sequences (sequence_name, next_val)
VALUES (default, 1)

UPDATE hibernate_sequences SET next_val=2
WHERE next_val=1 AND sequence_name=default

SELECT tbl.next_val
FROM hibernate_sequences tbl
WHERE tbl.sequence_name=default
FOR UPDATE

UPDATE hibernate_sequences SET next_val=3
WHERE next_val=2 AND sequence_name=default

DEBUG - Flush is triggered at commit-time

INSERT INTO post (id) values (1, 2)
```

The table generator benefits from JDBC batching, but every table sequence update incurs three steps:

- the lock statement is executed to ensure that the same sequence value is not allocated for two concurrent transactions
- the current value is incremented in the data access layer
- the new value is saved back to the database and the secondary transaction is committed to release the row-level lock.

Unlike identity columns and sequences, which can increment the sequence in a single request, the table generator entails a significant performance overhead. For this reason, Hibernate comes with a series of optimizers which can improve performance for both database sequences and table generators.

Although it is a portable identifier generation strategy, the table generator introduces a serializable execution (the row-level lock), which can hinder scalability. Compared to this application-level sequence generation technique, identity columns and sequences are highly optimized for high-concurrency scenarios and should be the preferred choice anyway.

10.2.3.4 Optimizers

As previously mentioned, both the sequence and the table identifier generator have multiple implementations which can improve the performance of the identifier generation process. The sequence and table generators can be split into two categories:

- legacy implementations (being deprecated since Hibernate 5.0) like `SequenceGenerator`, `SequenceHiLoGenerator` and `MultipleHiLoPerTableGenerator`
- newer and more efficient implementations such as `SequenceStyleGenerator` and `TableGenerator`.

These two categories are not compatible, and the application developer must either choose the legacy identifiers or the enhanced ones. Prior to Hibernate 5.0, the legacy identifier generators were provided by default and the application developer could switch to the newer ones by setting the following configuration property:

```
<property name="hibernate.id.new_generator_mappings" value="true"/>
```

Hibernate 5 has decided to drop support for the legacy identifiers and to use the enhanced ones by default.

Among the legacy identifier generators, the `SequenceGenerator` didn't offer any optimization, as every new identifier value would require a call to the underlying database sequence. On the other hand, the `SequenceHiLoGenerator` and the `MultipleHiLoPerTableGenerator` offered a *hi/lo* optimization mechanism aimed to reduce the number of calls to a database server. Although these generators are deprecated, the legacy *hi/lo* algorithm is still a valid optimizer even for the newer identifier generators.

10.2.3.4.1 The hi/lo algorithm

The hi/lo algorithms splits the sequences domain into *hi* groups. A *hi* value is assigned synchronously, and every *hi* group is given a maximum number of *lo* entries, that can by assigned off-line without worrying about identifier value conflicts.

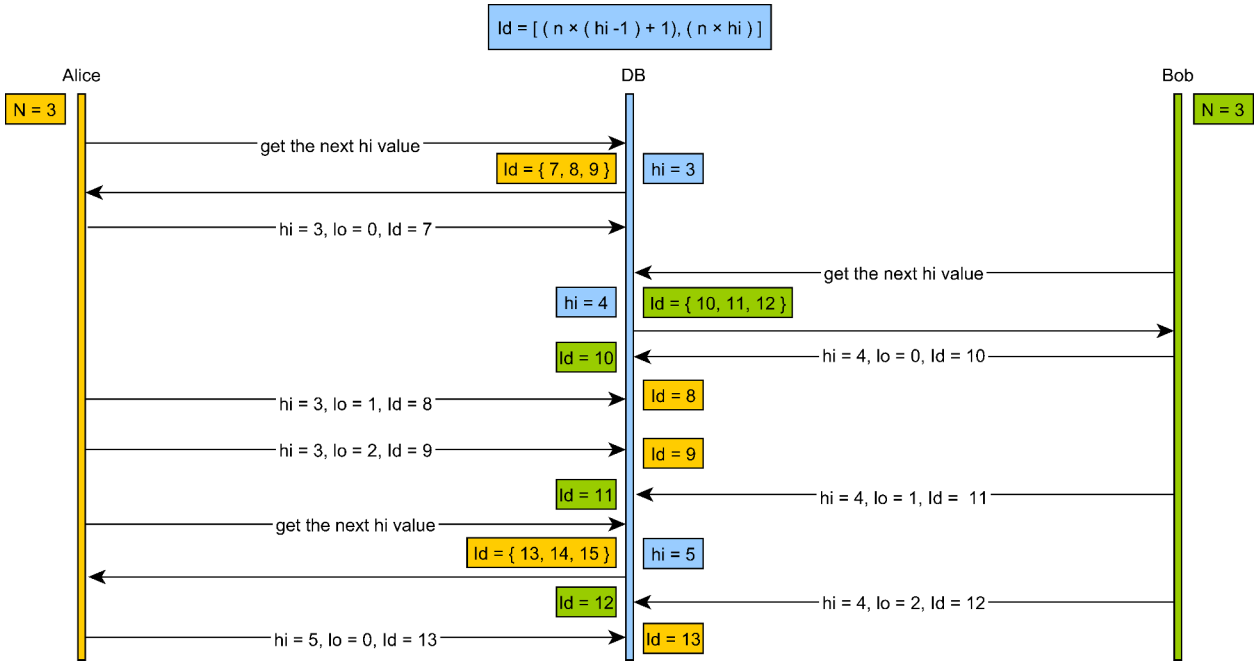


Figure 10.2: The hi/lo algorithm

1. The *hi* token is assigned either by the database sequence or the table generator, so two consecutive calls are guaranteed to see monotonically increasing values
2. Once a *hi* token is retrieved, the *increment size (n - the number of *lo* entries)* defines the range of identifier values a transaction can safely allocate. The identifiers range is bounded by the following interval: $Id = [n \times (hi - 1) + 1, n \times hi]$ and the allocation is done as follows:
   - the current group values start from $n \times (hi - 1) + 1$
   - the *lo* value is taken from the following interval: {$$} [0, n - 1] {$$}
   - by adding the *lo* value to the initial group value, a unique identifier number is obtained.
3. When all *lo* values are used, a new *hi* value is fetched and the cycle continues.

The following example shows how the hi/lo algorithm works in practice. The entity is mapped as follows:

```
@Entity
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "hilo")
    @GenericGenerator(
        name = "hilo",
        strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
        parameters = {
            @Parameter(name = "sequence_name", value = "sequence"),
            @Parameter(name = "initial_value", value = "1"),
            @Parameter(name = "increment_size", value = "3"),
            @Parameter(name = "optimizer", value = "hilo")
        }
    )
    private Long id;
}
```

Because the increment size is *3*, the following test will insert *4* entities to show the number of database sequence calls.

```
doInJPA(entityManager -> {
    for(int i = 0; i < 4; i++) {
        Post post = new Post();
        entityManager.persist(post);
    }
});
```

Running the previous test generates the following output:

```
CALL NEXT VALUE FOR hilo_seqeunce
CALL NEXT VALUE FOR hilo_seqeunce

INSERT INTO Post (id) VALUES (1)
INSERT INTO Post (id) VALUES (2)
INSERT INTO Post (id) VALUES (3)
INSERT INTO Post (id) VALUES (4)
```

The first sequence call is for the first three values, while the second one is generated when reaching the forth entity that needs to be persisted. The more inserts a transaction requires, the better the performance gain from reducing the number of database sequence calls.

Unfortunately, this optimizer has a major limitation. Because the database sequence only assigns group values, all database clients must be aware of this algorithm. If the DBA must insert a row in the aforementioned table, he must use the hi/lo algorithm to determine the range of values that she can safely use.

For this reason, Hibernate offers other optimizer algorithms that are interoperable with external clients, unaware of the application-level optimization technique in-use.

10.2.3.4.2 The default sequence identifier generator

The JPA identifier generator strategy only specifies the identifier type and not the algorithm used for generating such identifiers.

For the sequence generator, considering the following JPA mapping:

```
@Id
@GeneratedValue(generator = "sequence", strategy=GenerationType.SEQUENCE)
@SequenceGenerator(name = "sequence", allocationSize = 3)
private Long id;
```

Hibernate will choose the `SequenceHiLoGenerator` when the `hibernate.id.new_generator_mappings` configuration property is false. This was the default setting for Hibernate 3 and 4. The legacy `SequenceHiLoGenerator` uses the hi/lo algorithm, and, if the allocation size is greater than one, database interoperability could be compromised (every insert must be done according to the hi/lo algorithm rules).

If the aforementioned configuration property is true (the default setting for Hibernate 5), then the JPA mapping above will use the `SequenceStyleGenerator` instead.

Unlike its previous predecessor, the `SequenceStyleGenerator` uses configurable identifier optimizer strategies, and the application developer can even supply its own optimization implementation.

10.2.3.4.3 The default table identifier generator

Just like with sequences, the JPA table generator mapping can use a legacy or an enhanced generator, depending on the current Hibernate configuration settings:

```
@Id
@GeneratedValue(generator = "table", strategy=GenerationType.TABLE)
@TableGenerator(name = "table", allocationSize = 3)
private Long id;
```

If the `hibernate.id.new_generator_mappings` configuration property is false, then Hibernate chooses the `MultipleHiLoPerTableGenerator`. This generator requires a single table for managing multiple identifiers, and just like `SequenceHiLoGenerator`, it also uses the hi/lo algorithm by default.

When the enhanced identifier generators are activated, Hibernate will use the `TableGenerator` instead, which can also take configurable optimizer strategies.

For both the enhanced sequence and the table identifier generator, Hibernate comes with the following built-in optimizers:

Table 10.8: Hibernate identifier optimizers

Optimizer type	Implementation class	Description
`none`	`NoopOptimizer`	Every identifier is fetched using a new roundtrip to the database
`hi/lo`	`HiLoOptimizer`	It allocates identifiers by using the legacy hi/lo algorithm
`pooled`	`PooledOptimizer`	It's an enhanced version of the hi/lo algorithm which is interoperable with other systems unaware of this identifier generator
`pooled-lo`	`PooledLoOptimizer`	It's a variation of the `pooled` optimizer, the database sequence value representing the *lo* value instead of the *hi* one

By default, the `SequenceStyleGenerator` and `TableGenerator` identifier generators will use the `pooled` optimizer. If the `hibernate.id.optimizer.pooled.prefer_lo` configuration property is set to true, Hibernate will use the `pooled-lo` optimizer by default.

Both the `pooled` and the `pooled-lo` encode the database sequence value into the identifier range boundaries, so allocating a new value using the actual database sequence call doesn't interfere with the identifier generator allocation process.

10.2.3.4.4 The pooled optimizer

The pooled optimizer can be configured as follows:

```
@Entity
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "pooled")
    @GenericGenerator(
        name = "pooled",
        strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
        parameters = {
            @Parameter(name = "sequence_name", value = "sequence"),
            @Parameter(name = "initial_value", value = "1"),
            @Parameter(name = "increment_size", value = "3"),
            @Parameter(name = "optimizer", value = "pooled")
        }
    )
    private Long id;
}
```

This increment size gives the range of values that can be allocated by the sequence generator with one database roundtrip. Although it's not efficient to flush the Persistence Context after every `persist()` method call, in this particular test, the flush outlines when the database sequence is called.

```
doInJPA(entityManager -> {
    for (int i = 0; i < 5; i++) {
        entityManager.persist(new Post());
        entityManager.flush();
    }
    entityManager.unwrap(Session.class).doWork(connection -> {
        try(Statement statement = connection.createStatement()) {
            statement.executeUpdate(
                "INSERT INTO post VALUES NEXT VALUE FOR sequence"
            );
        }
    });
    for (int i = 0; i < 3; i++) {
        entityManager.persist(new Post());
        entityManager.flush();
    }
});
```

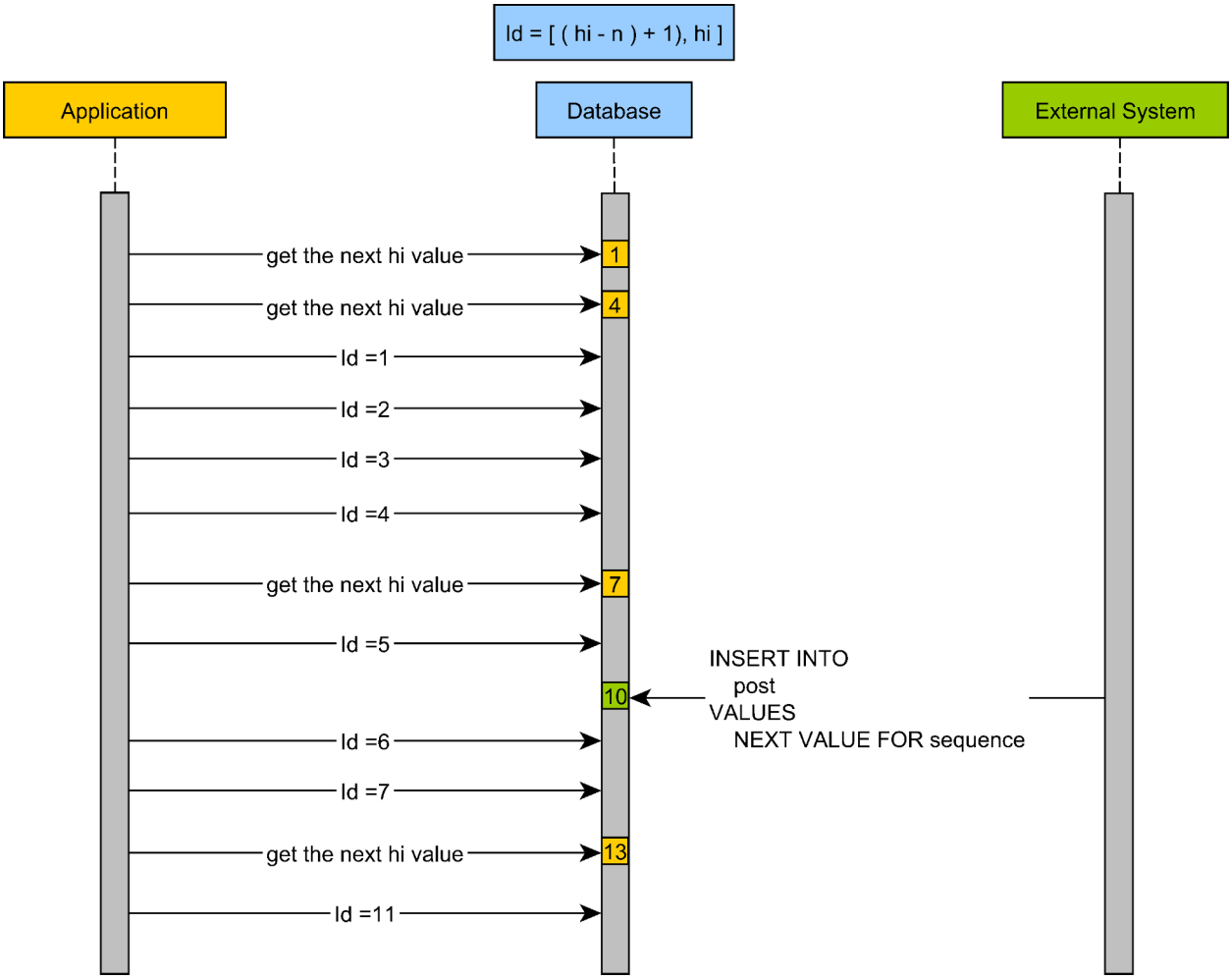


Figure 10.3: The pooled optimizer

If increment size (n) is the number of identifiers within a range, the pooled optimizer generates identifiers with the following formula: $Id = [(hi - n) + 1, hi]$.

- the first sequence call generates the *lo* value and the second one determines the *hi* value, so the first range of identifiers is {*2, 3, 4*}
- when adding the *5th* entity, the pooled optimizer calls the sequence again and obtains the next *hi* value, the next identifier range being {*5, 6, 7*}
- after inserting the *5th* entity, an external system adds a *post* row and assigns the primary key with the value returned by the sequence call
- the Hibernate application thread resumes and inserts the identifiers *6* and *7*
- the *8th* entity requires a new sequence call, and so a new range is allocated {*11, 12, 13*}

10.2.3.4.5 The pooled-lo optimizer

By changing the previous mapping to use the `pooled-lo` optimizer, the identifier generation changes as follows:

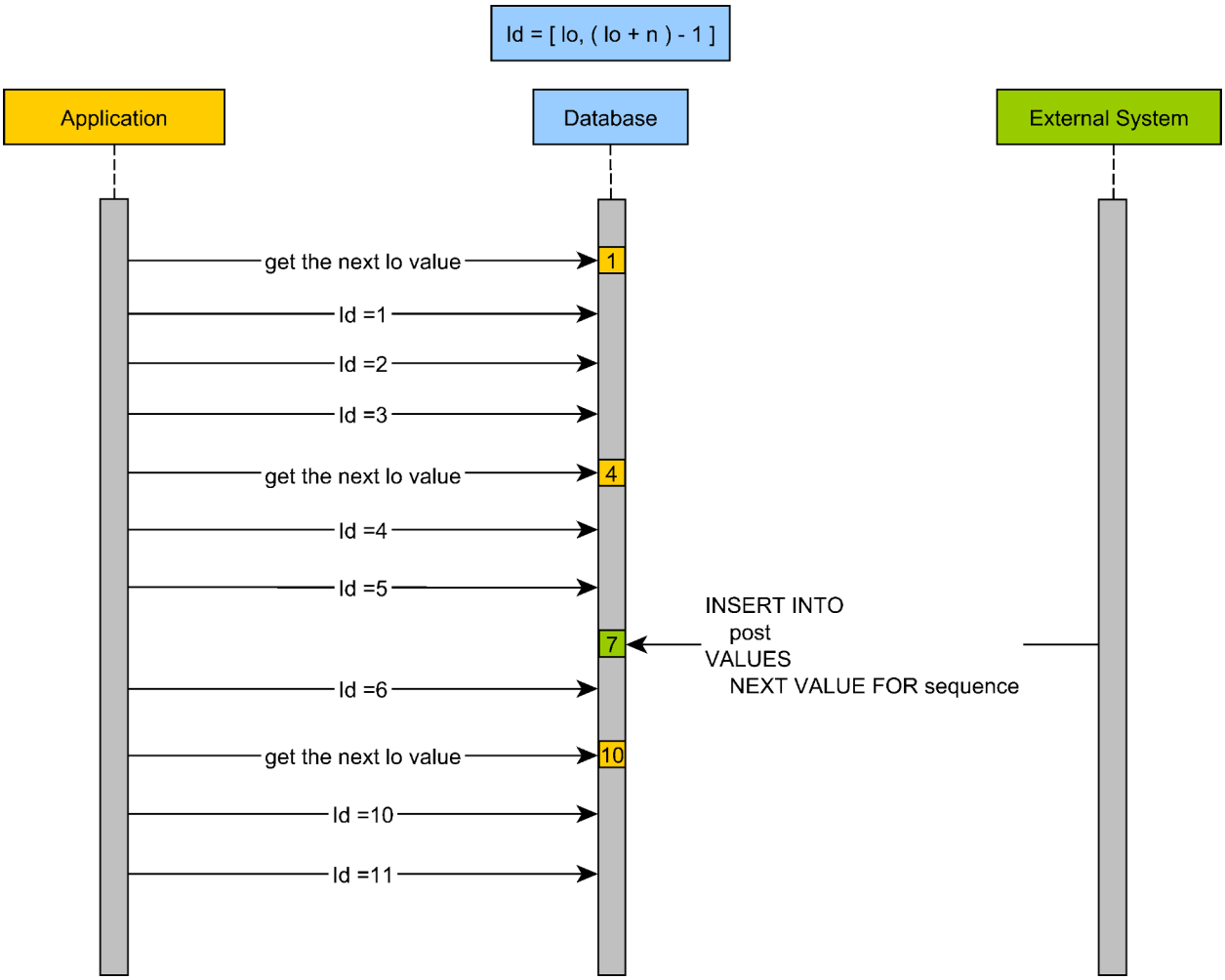


Figure 10.4: The pooled-lo optimizer

If increment size (n) is the number of identifiers within a range, the pooled-lo optimizer generates identifiers with the following formula: $Id = [lo, (lo + n) - 1]$.

- the first sequence call generates the *lo* value, so the first range of identifiers is {*1, 2, 3*}
- when adding the *4th* entity, the pooled-lo optimizer calls the sequence and obtains the next *lo* value, the next identifier range being {*4, 5, 6*}
- after inserting the *5th* entity, an external system adds a *post* row and assigns the primary key with the value returned by the sequence call
- the Hibernate application thread resumes and inserts the identifier *6*
- the *7th* entity requires a new sequence call, and so a new range is allocated {*10, 11, 12*}

10.2.3.5 Optimizer gain

To visualize the performance gain of using sequence and table generator optimizers, the following test will measure the identifier allocation time when inserting *50 post* entities and while varying the increment size (*1*, *5*, *10* and *50*).

10.2.3.5.1 Sequence generator performance gain

When using a sequence generator with the default pooled optimizer, the following 99th percentile is being recorded:

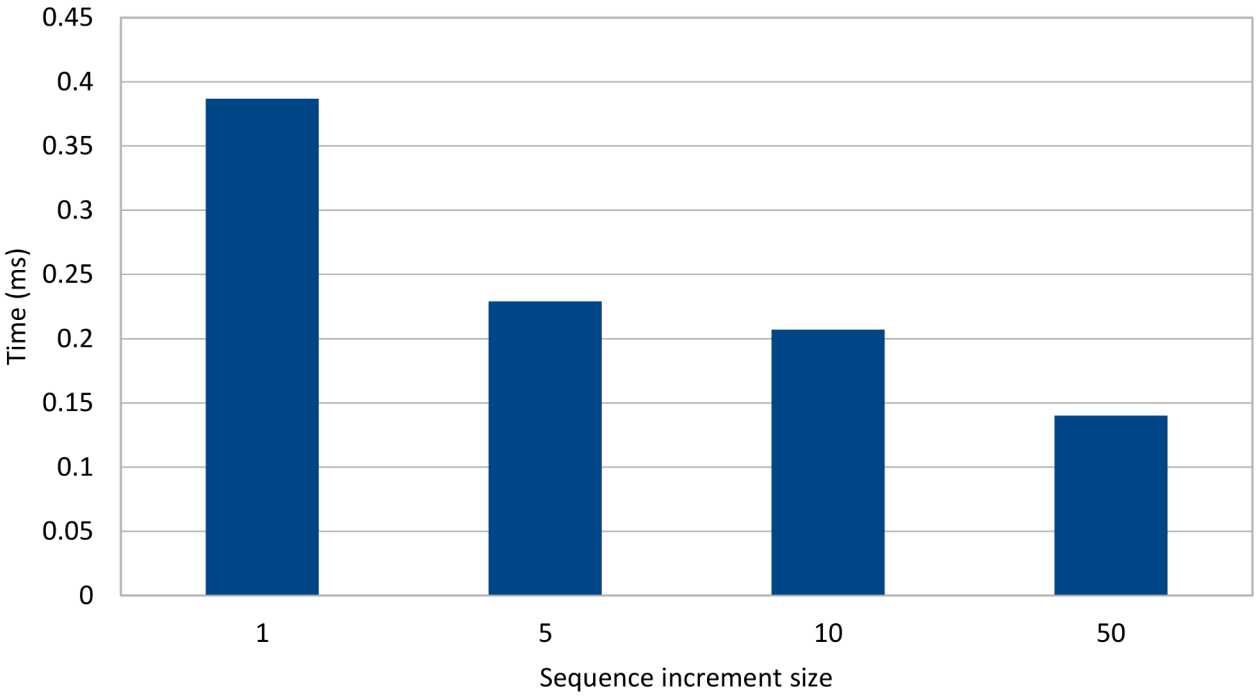


Figure 10.5: Sequence pooled optimizer gain

Database sequences are fast, but, even so, the pooled optimizer manages to reduce the identifier generation time considerably.

For write-intensive application, the increment size needs to be adjusted according to the amount of rows requires to be inserted in one transaction.

10.2.3.5.2 Table generator performance gain

The same test suite is run against a table generator with a pooled optimizer, and the increment size also varies between *1*, *5*, *10* and *50*. Because of the row-level locking and the extra database connection switch overhead, the table generator is less efficient than a database sequence.

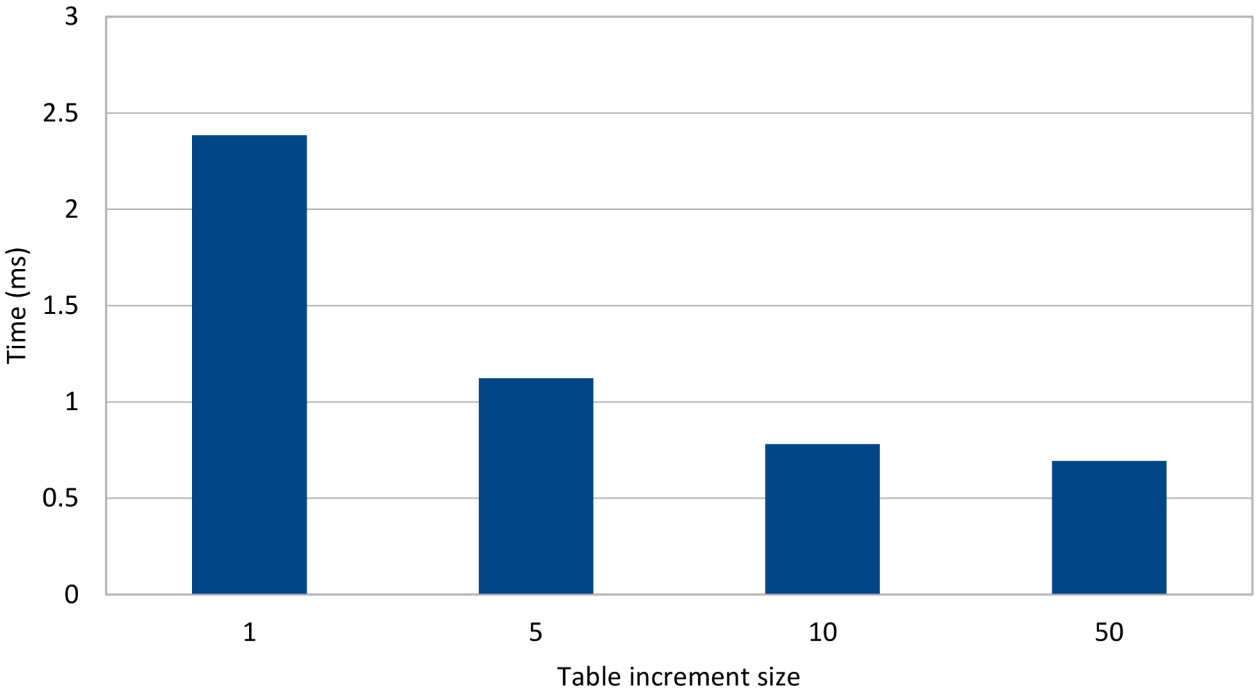


Figure 10.6: Table pooled optimizer gain

Just like with the database sequence, the pooled optimizer managed to reduce the time it took for assigning a new entity identifier.

10.2.3.6 Identifier generator performance

To evaluate the concurrency cost of each identifier generators, the following test will measure the time it takes to insert *100* post entities when multiple running threads are involved. JDBC batching is enabled, and the connection pool is adjusted to accommodate the maximum number of database connection required (e.g. *32*).

In reality, the application might not be configured with so many database connections, and the table generator connection acquisition cost might be even higher.

The first relational database system under test supports identity columns, so it's worth measuring how the identifier and the table generator compete with each other. Unlike the previous test, this one measures the total time taken for inserting all entities, and not just the identifier allocation time interval.

Each test iteration increases contention by allocating more worker threads that need to execute the same database insert load.

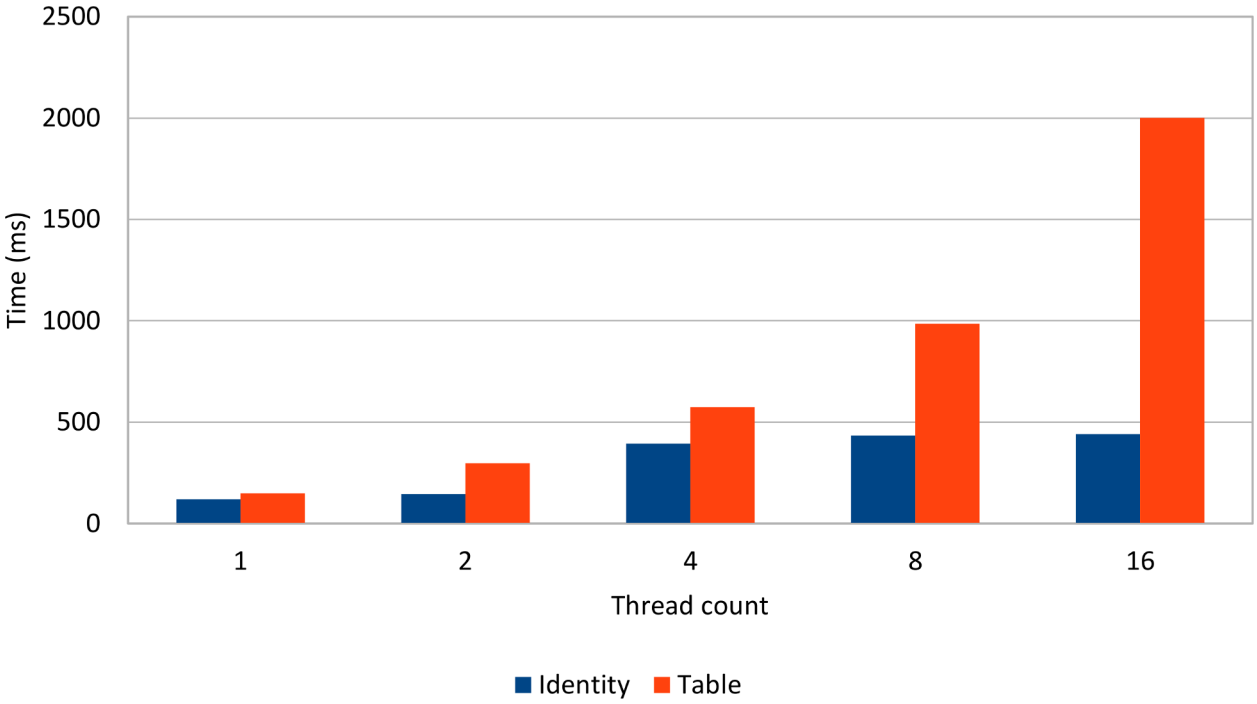


Figure 10.7: Identity vs Table

Even if it cannot benefit from JDBC batching, the identity generator still manages to outperform the table generator, which uses a pooled optimizer with an increment size of *100*.

The more threads are used, the less efficient the table generator becomes. On the other hand, identity columns scale much better with more concurrent transactions.

Even if doesn't support JDBC batching, native identity columns are still a valid choice, and, in future, Hibernate might even support batch inserts for those as well.

If compared to identity columns the table generators could benefit from JDBC batching and the pooled optimizer, the gap between the sequence and the table generator is even higher because sequences can also take advantage of all these optimizations as well.

Running the same test against a relational database supporting sequences, the following results are being recorded:

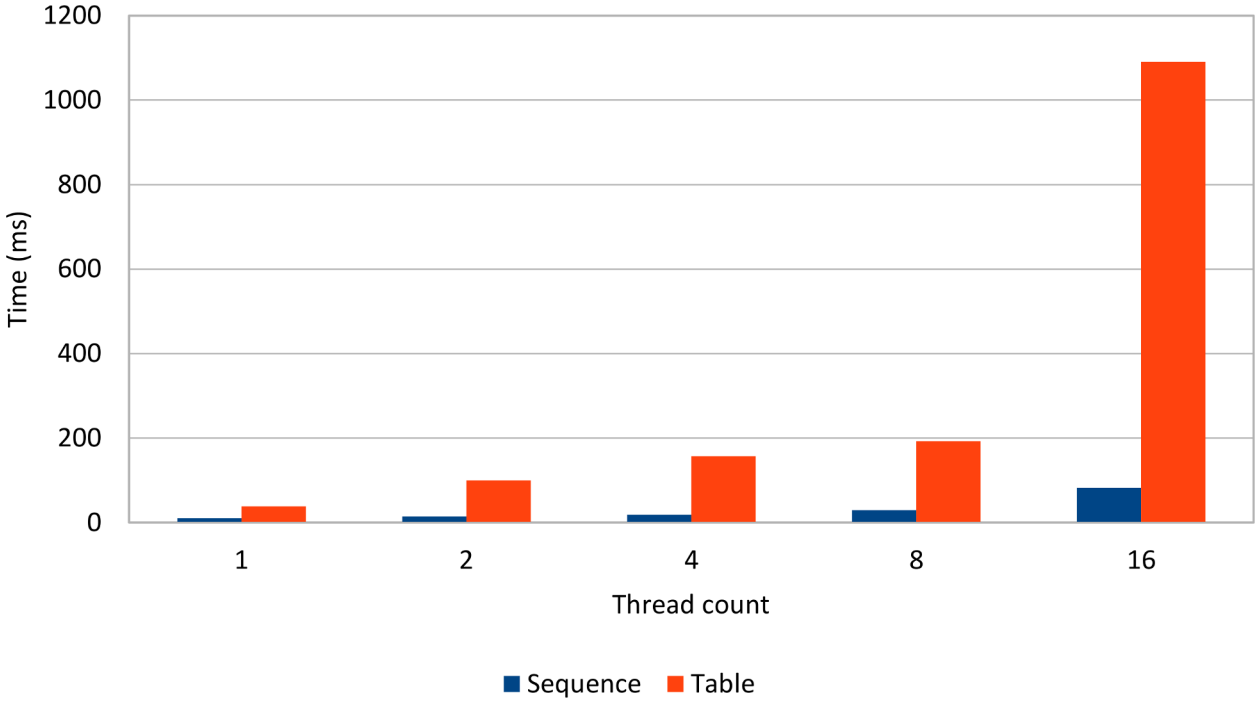


Figure 10.8: Sequence vs Table

The performance impact of the table generator becomes noticeable in high concurrent environments, where the row-level locking and the database connection switch introduces a serial execution.

Because they use lightweight synchronization mechanisms, database sequences scale better than row-level locking concurrency control mechanisms.

Database sequences are the most efficient Hibernate identifier choice, allowing sequence call optimizers and without compromising JDBC batching.

11. Relationships

In a relational database, associations are formed by correlating rows belonging to different tables. A relationship is established when a child table defines a foreign key referencing the primary key of its parent table. Every database association is built on top of foreign keys, resulting three table relationship types:

- *one-to-many* is the most common relationship and it associates a row from a parent table to multiple rows in a child table
- *one-to-one* is a variation of the *one-to-many* relationship with an additional uniqueness constraint on the child-side foreign key
- *many-to-many* requires a junction table containing two foreign keys that reference two different parent tables.

The following diagram depicts all these three table relationships:

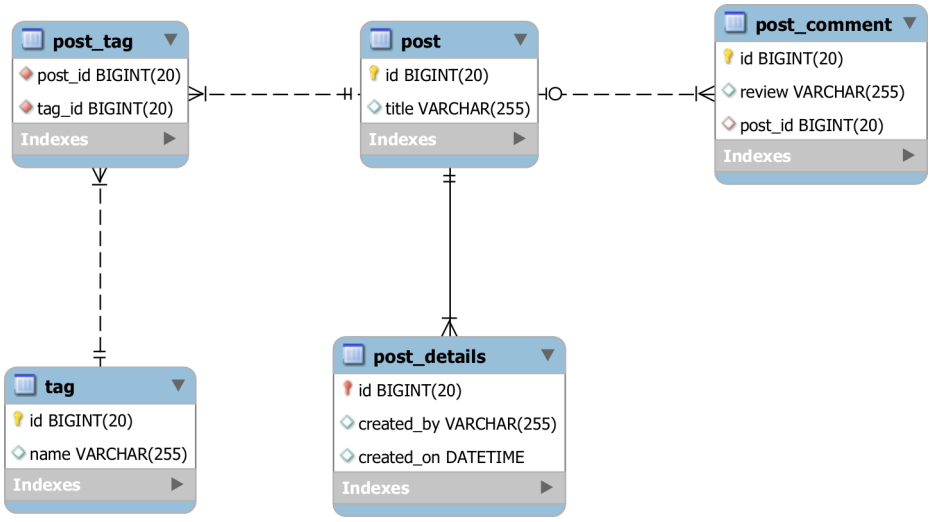


Figure 11.1: Table relationships

The `post` table has a one-to-many relationship with the `post_comment` table because a post row might be referenced by multiple comments. The one-to-many relationship is established through the `post_id` column which has a foreign key referencing the `post` table primary key. Because a `post_comment` cannot exist without a `post`, the `post` is the parent-side while the `post_comment` is the child-side.

The `post` table has a one-to-one relationship with the `post_details`. Like the one-to-many association, the one-to-one relationship involves two tables and a foreign key. The foreign key has a uniqueness constraint, so only one child row can reference a parent record.

The `post` and the `tag` are both independent tables and neither one is a child of the other. A `post` can feature several `tag(s)`, while a `tag` can also be associated with multiple `post(s)`. This is a typical many-to-many association and it requires a junction table to resolve the child-side of these two parent entities. The junction table requires two foreign keys referencing the two parent tables.

The foreign key is therefore the most important construct in building a table relationship, and, in a relation database, the child-side controls a table relationship.

In a relational database, the foreign key is associated with the child-side only. For this reason, the parent-side has no knowledge of any associated child relationships, and, from a mapping perspective, table relationships are always unidirectional (the child foreign key references the parent primary key).

11.1 Relationship types

When mapping a JPA entity, besides the underlying table columns, the application developer can map entity relationships either in one direction or in a bidirectional way. This is another impedance mismatch between the object-oriented Domain Model and relational database system because, when using an ORM tool, the parent and the child-side can reference each other.

A relationship is unidirectional if only one entity side maps the table relationship and is bidirectional if the table relationship can be navigated in both directions (either from the entity parent-side or the child-side).

To properly represent both sides of an entity relationship, JPA defines four association mapping constructs:

- `@ManyToOne` represents the child-side (where the foreign key resides) in a database one-to-many table relationship
- `@OneToMany` is associated with the parent-side of a one-to-many table relationship
- `@ElementCollection` defines a one-to-many association between an entity and multiple value types (basic or embeddable)
- `@OneToOne` is used for both the child-side and the parent-side in a one-to-one table relationship
- `@ManyToMany` mirrors a many-to-many table relationship.

Because the entity relationship choice has a considerable impact on the overall application performance, this chapter analyses the data access operation efficiency of all these JPA associations.

Mapping collections

In a relational database, all table relationships are constructed using foreign keys and navigated through SQL queries. JPA allows mapping both the foreign key side (the child entity has a reference to its parent), as well as the parent side (the parent entity has one or more child entities).

Although `@OneToMany`, `@ManyToMany` or `@ElementCollection` are convenient from a data access perspective (entity state transitions can be cascaded from parent entities to children), they are definitely not free of cost. The price for reducing data access operations is paid in terms of result set fetching flexibility and performance. A JPA collection, either of entities or value types (basic or embeddables), binds a parent entity to a query that usually fetches all the associated child records. Because of this, the entity mapping becomes sensitive to the number of child entries.

If the children count is relatively small, the performance impact of always retrieving all child entities might be unnoticeable. But if the number of child records grows too large, fetching the entire children collection can become a performance bottleneck. Unfortunately, the entity mapping is done during the early phases of a project development, and the development team might be unaware of the number of child records a production system will exhibit.

Not just the mere size can be problematic, but also the number of properties of the child entity. Because entities are usually fetched as a whole, the result set is therefore proportional to the number of columns the child table contains. Even if a collection is fetched lazily, Hibernate might still require to fully load each entity when the collection is accessed for the first time. Although Hibernate supports extra lazy collection fetching, this is only a workaround and doesn't address the root problem.

Alternatively, every collection mapping can be replaced by a data access query, which can use an SQL projection that's tailored by the data requirements of each business use case. This way, the query can take business case specific filtering criteria. Although JPA 2.1 doesn't support dynamic collection filtering, Hibernate offers `Session` (Persistence Context) bound collection `Filters`.

When handling large data sets, it's good practice to limit the result set size, both for UI (to increase responsiveness) or batch processing tasks (to avoid long running transactions). Just because JPA offers supports collection mapping, it doesn't mean they are mandatory for every domain model mapping. Until there's a clear understanding of the number of child records (or if there's even a need to fetch child entities entirely), it's better to post pone the collection mapping decision. For high-performance systems, a data access query is often a much more flexible alternative anyway.

11.2 @ManyToOne

The @ManyToOne relationship is the most common JPA association and it maps exactly to the one-to-many table relationship. When using a @ManyToOne association, the underlying foreign key is controlled by the child-side, no matter the association is unidirectional or bidirectional.

This section focuses on unidirectional @ManyToOne relationship only, the bidirectional case being further discussed with the @OneToMany relationship. In the following example, the Post entity represents the parent-side, while the PostComment is the child-side.

As already mentioned, the JPA entity relationship diagram matches exactly the one-to-many table relationship.

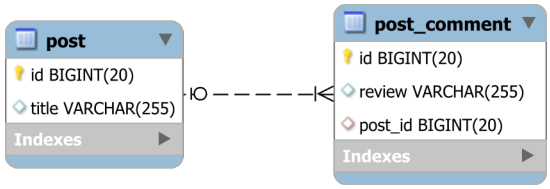


Figure 11.2: The one-to-many table relationship

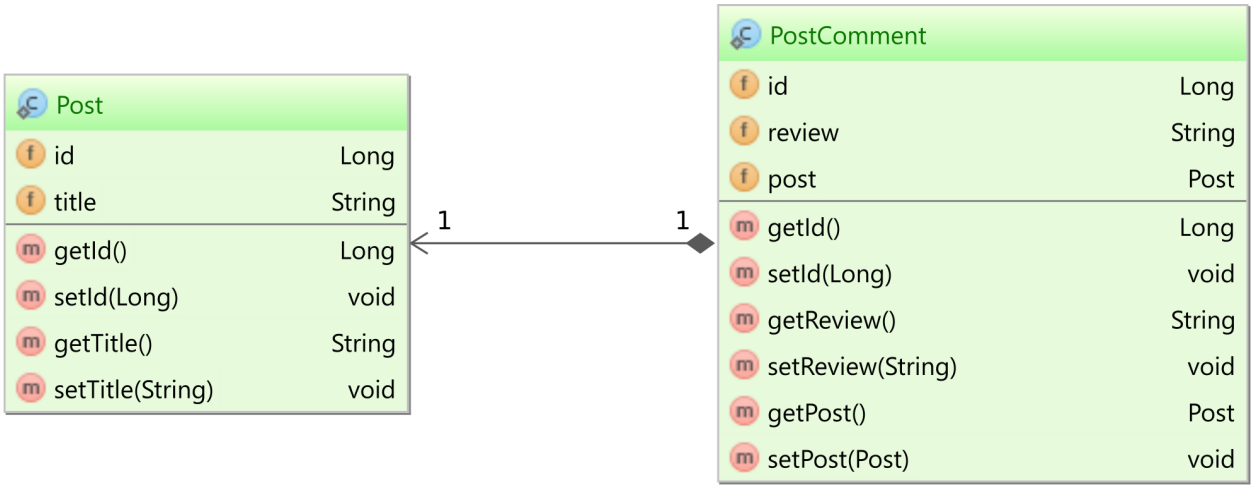


Figure 11.3: @ManyToOne relationship

Instead of mapping the post_id foreign key column, the PostComment uses a @ManyToOne relationship to the parent Post entity. The PostComment can be associated with an existing Post object reference, and the PostComment can also be fetched along with the Post entity.

```
@ManyToOne
@JoinColumn(name = "post_id")
private Post post;
```

Hibernate translates the internal state of the @ManyToOne Post object reference to the post_id foreign key column value.

If the `@ManyToOne` attribute is set to a valid `Post` entity reference:

```
Post post = entityManager.find(Post.class, 1L);
PostComment comment = new PostComment("My review");
comment.setPost(post);
entityManager.persist(comment);
```

Hibernate generates an insert statement populating the `post_id` column with the identifier of the associated `Post` entity.

```
INSERT INTO post_comment (post_id, review, id) VALUES (1, 'My review', 2)
```

If the `Post` attribute is later set to null:

```
comment.setPost(null);
```

The `post_id` column is also updated with a `NULL` value:

```
UPDATE post_comment SET post_id = NULL, review = 'My review' WHERE id = 2
```

Because the `@ManyToOne` association controls the foreign key directly, the automatically generated DML statements are very efficient.

Actually, the best performing JPA associations always rely on the child-side to translate the JPA state to the foreign key column value.

This is one of the most important rule in JPA relationship mapping, and it will be further emphasized for `@OneToMany`, `@OneToOne` and even `@ManyToMany` associations.

11.3 `@OneToMany`

While the `@ManyToOne` association is the most natural mapping of the one-to-many table relationship, the `@OneToMany` association can also mirror this database relationship, but only when being used as a bidirectional mapping. A unidirectional `@OneToMany` association uses an additional junction table, which no longer fits the one-to-many table relationship semantics.

11.3.1 Bidirectional `@OneToMany`

The bidirectional `@OneToMany` association has a matching `@ManyToOne` child-side mapping that controls the underlying one-to-many table relationship. The parent-side is mapped as a collection of child entities.

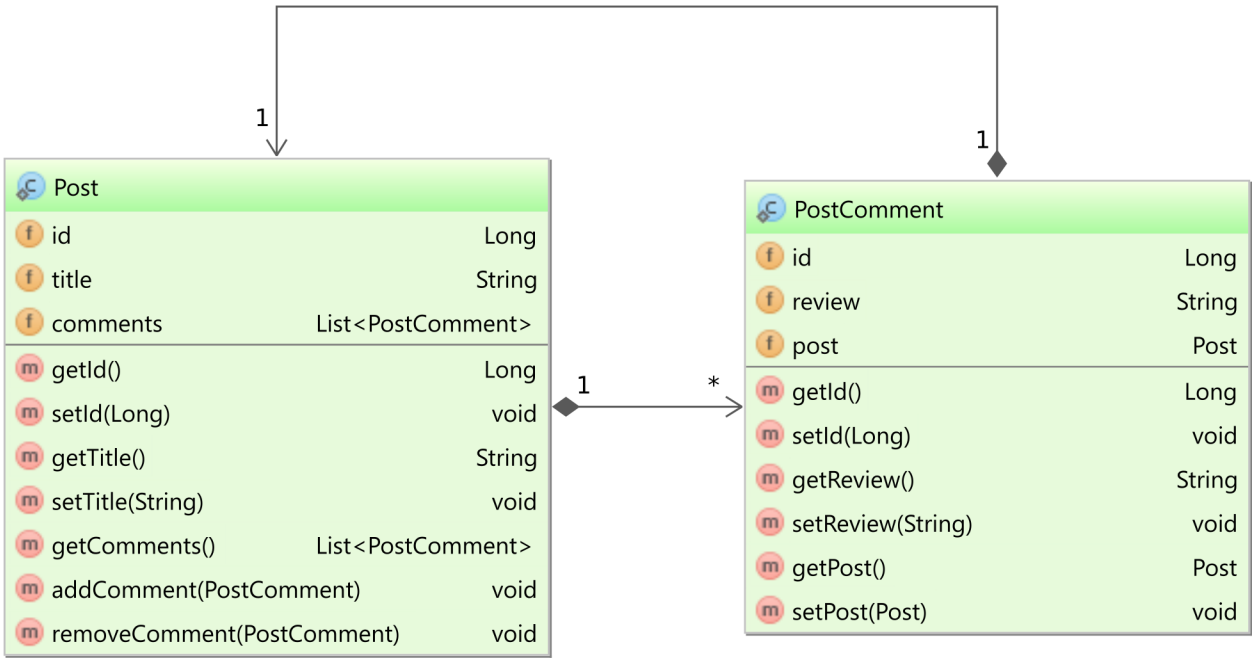


Figure 11.4: Bidirectional `@OneToMany` relationship

In a bidirectional association, only one side can control the underlying table relationship. For the bidirectional `@OneToMany` mapping, it's the child-side `@ManyToOne` association in charge of keeping the foreign key column value in sync with the in-memory Persistence Context. This is the reason why the bidirectional `@OneToMany` relationship must define the `mappedBy` attribute, indicating that it only mirrors the `@ManyToOne` child-side mapping.

```
@OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval = true)
private List<PostComment> comments = new ArrayList<>();
```

Even if the child-side is in charge of synchronizing the entity state changes with and the database foreign key column value, a bidirectional association must always have both the parent-side and the child-side in sync.

To synchronize both ends, it's practical to provide parent-side helper methods that add/remove child entities.

```
public void addComment(PostComment comment) {
    comments.add(comment);
    comment.setPost(this);
}

public void removeComment(PostComment comment) {
    comments.remove(comment);
    comment.setPost(null);
}
```

One of the major advantages of using a bidirectional association is that entity state transitions can be cascaded from the parent entity to its children. In the following example, when persisting the parent `Post` entity, all the `PostComment` child entities are persisted as well.

```
Post post = new Post("First post");
entityManager.persist(post);

PostComment comment1 = new PostComment("My first review");
post.addComment(comment1);
PostComment comment2 = new PostComment("My second review");
post.addComment(comment2);

entityManager.persist(post);
```

```
INSERT INTO post (title, id) VALUES ('First post', 1)
INSERT INTO post_comment (post_id, review, id) VALUES (1, 'My first review', 2)
INSERT INTO post_comment (post_id, review, id) VALUES (1, 'My second review', 3)
```

When removing a comment from the parent-side collection, the orphan removal attribute will instruct Hibernate to generate a delete DML statement on the targeted child entity:

```
post.removeComment(comment1);
```

```
DELETE FROM post_comment WHERE id = 2
```

Equality-based entity removal

The helper method for the child entity removal relies on the underlying child object equality for matching the collection entry that needs to be removed.

If the application developer doesn't choose to override the default `equals` and `hashCode` methods, the `java.lang.Object` identity-based equality is going to be used. The problem with this approach is that the application developer must supply a child entity object reference that's contained in the current child collection.

Sometimes child entities are loaded in one web request and saved in a `HttpSession` or a Stateful Enterprise Java Bean. Once the Persistence Context, which loaded the child entity is closed, the entity becomes detached. If the child entity is sent for removal into a new web request, the child entity must be reattached or merged into the current Persistence Context. This way, if the parent entity is loaded along with its child entities, the removal operation will work properly since the removing child entity is already managed and contained in the children collection.

If the entity hasn't changed, reattaching this child entity is redundant and so the `equals` and the `hashCode` methods must be overridden to express equality in terms of a unique business key. In case the child entity has a `@NaturalId` or a unique property/properties set, the `equals` and the `hashCode` methods can be implemented on top of that. Assuming the `PostComment` entity has the following two columns whose combination render a unique business key, the equality contract can be implemented as follows:

```
private String createdBy;

@Temporal(TemporalType.TIMESTAMP)
private Date createdOn = new Date();

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    PostComment that = (PostComment) o;
    return Objects.equals(createdBy, that.createdBy) &&
           Objects.equals(createdOn, that.createdOn);
}

@Override
public int hashCode() {
    return Objects.hash(createdBy, createdOn);
}
```

The bidirectional `@OneToMany` association generates efficient DML statements because the `@ManyToOne` mapping is in charge of the table relationship. Because it simplifies data access operations as well, the bidirectional `@OneToMany` association is worth considering when the size of the child records is relatively low.

11.3.2 Unidirectional `@OneToMany`

The unidirectional `@OneToMany` association is very tempting because the mapping is simpler than its bidirectional counterpart. Because there is only one side to take into consideration, there's no need for helper methods and the mapping doesn't feature a `mappedBy` attribute either.

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
private List<PostComment> comments = new ArrayList<>();
```

Unfortunately, in spite its simplicity, the unidirectional `@OneToMany` association is less efficient than the unidirectional `@ManyToOne` mapping or the bidirectional `@OneToMany` association.

Against any intuition, the unidirectional `@OneToMany` association doesn't map to a one-to-many table relationship. Because there is no `@ManyToOne` side to control this relationship, Hibernate uses a separate junction table to manage the association between a parent row and its child records.

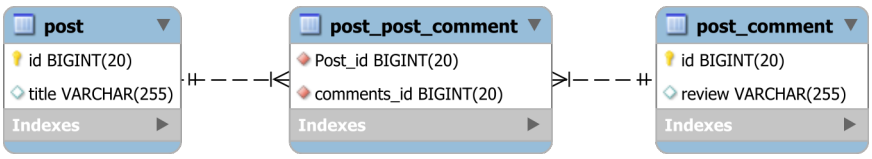


Figure 11.5: The `@OneToMany` table relationship

The table `post_post_comment` has two foreign key columns, which reference both the parent-side row (the `Post_id` column is a foreign key to the `post` table primary key) and the child-side entity (the `comments_id` references the primary key of the `post_comment` table).

Without going into analyzing the associated data access operations, it's obvious that joining three tables is less efficient than joining just two. Because there are two foreign keys, there needs to be two indexes (instead of one), so the indexes memory footprint increases.

But since this is a regular table mapping for a many-to-many relationship, the extra table and the increased memory footprint is not the biggest performance issue. The algorithm for managing the collection state is what makes any unidirectional `@OneToMany` association less attractive.

Considering there is a `Post` entity with two `PostComment` child records, obtained by running the following example:

```
Post post = new Post("First post");

post.getComments().add(new PostComment("My first review"));
post.getComments().add(new PostComment("My second review"));
post.getComments().add(new PostComment("My third review"));

entityManager.persist(post);
```

While for a bidirectional `@OneToMany` association there were three child rows being added, the unidirectional association requires three additional inserts for the junction table records.

```
INSERT INTO post (title, id) VALUES ('First post', 1)
INSERT INTO post_comment (review, id) VALUES ('My first review', 2)
INSERT INTO post_comment (review, id) VALUES ('My second review', 3)
INSERT INTO post_comment (review, id) VALUES ('My third review', 4)
INSERT INTO post_post_comment (Post_id, comments_id) VALUES (1, 2)
INSERT INTO post_post_comment (Post_id, comments_id) VALUES (1, 3)
INSERT INTO post_post_comment (Post_id, comments_id) VALUES (1, 4)
```

When removing the first element of the collection:

```
post.getComments().remove(0);
```

Hibernate generates the following DML statements:

```
DELETE FROM post_post_comment WHERE Post_id = 1
INSERT INTO post_post_comment (Post_id, comments_id) VALUES (1, 3)
INSERT INTO post_post_comment (Post_id, comments_id) VALUES (1, 4)
DELETE FROM post_comment WHERE id = 2
```

First, all junction table rows associated with the parent entity are deleted, and then the remaining in-memory records are added back again. The problem with this approach is that instead of a single junction table remove operation, the database has way more DML statements to execute.

Another problem is related to indexes. If there is an index on each foreign key column (which is the default for many relational databases), the database engine must delete the associated index entries only to add back the remaining ones. The more elements a collection has, the less efficient a remove operation will get.

The unidirectional `@OneToMany` relationship is less efficient both for reading data (three joins are required instead of two), as for adding (two tables must be written instead of one) or removing (entries are removed and added back again) child entries.

11.3.3 Ordered unidirectional `@OneToMany`

If the collection can store the index of every collection element, the unidirectional `@OneToMany` relationship can benefit for some element removal operations. First, an `@OrderColumn` annotation must be defined along the `@OneToMany` relationship mapping:

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@OrderColumn(name = "entry")
private List<PostComment> comments = new ArrayList<>();
```

At the database level, the `entry` column is included in the junction table.

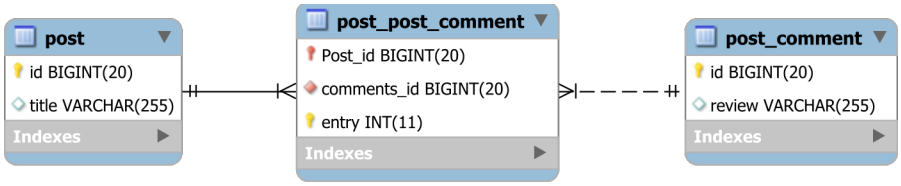


Figure 11.6: The unidirectional `@OneToMany` with an `@OrderColumn`

It's better not to mistake the `@OrderColumn` with the `@OrderBy` JPA annotation. While the former allows the JPA provider to materialize the element index into a dedicated database column so that the collection is sorted using an `ORDER BY` clause, the latter does the sorting at runtime based on the ordering criteria provided by the `@OrderBy` annotation.

Considering there are three `PostComment` entities added for a given `Post` parent entity:

```
post.getComments().add(new PostComment("My first review"));
post.getComments().add(new PostComment("My second review"));
post.getComments().add(new PostComment("My third review"));
```

The index of every collection element is going to be stored in the `entry` column of the junction table:

```
INSERT INTO post_comment (review, id) VALUES ('My first review', 2)
INSERT INTO post_comment (review, id) VALUES ('My second review', 3)
INSERT INTO post_comment (review, id) VALUES ('My third review', 4)
INSERT INTO post_post_comment (Post_id, entry, comments_id) VALUES (1, 0, 2)
INSERT INTO post_post_comment (Post_id, entry, comments_id) VALUES (1, 1, 3)
INSERT INTO post_post_comment (Post_id, entry, comments_id) VALUES (1, 2, 4)
```

When removing elements from the tail of the collection:

```
post.getComments().remove(2);
```

Hibernate only requires a single junction table delete statement:

```
DELETE FROM post_post_comment WHERE Post_id = 1 and entry = 2
DELETE FROM post_comment WHERE id = 4
```

Unfortunately, this optimization doesn't hold for entries that are not located towards the head of the collection, so when deleting the first element:

```
post.getComments().remove(0);
```

Hibernate deletes all child elements associated with this parent entity, and then it updates the remaining database entries to preserve the same element ordering as the in-memory collection snapshot:

```
DELETE FROM post_post_comment WHERE Post_id=1 and entry=1
UPDATE post_post_comment set comments_id = 3 WHERE Post_id = 1 and entry = 0
DELETE FROM post_comment WHERE id = 2
```

If the unidirectional `@OneToMany` collection is used like a stack and elements are always removed from the collection tail, the remove operations are more efficient when using an `@OrderColumn`. But the closer an element is to the head of the list, the more update statements must be issued, and the additional updates have an associated performance overhead.

11.3.3.1 `@ElementCollection`

Although it's not an entity association type, the `@ElementCollection` is very similar to the unidirectional `@OneToMany` relationship. To represent collections of basic types (e.g. `String`, `int`, `BigDecimal`) or embeddable types, the `@ElementCollection` must be used instead. If the previous associations involved multiple entities, this time there's only a single `Post` entity with a collection of `String` comments.

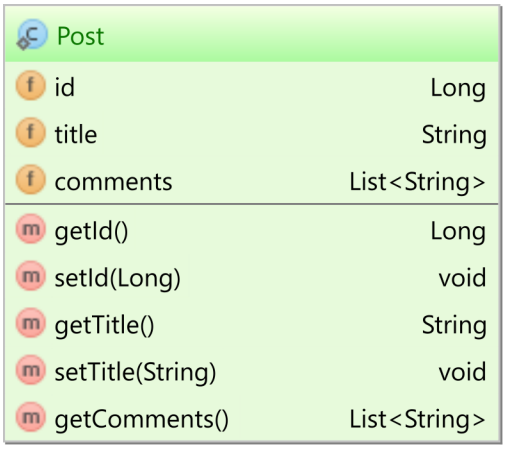


Figure 11.7: The `@ElementCollection` relationship

The mapping for the comments collection looks as follows:

```
@ElementCollection
private List<String> comments = new ArrayList<>();
```

Value types inherit the persistent state from their parent entities, so their lifecycle is also bound to the owner entity. Any operation against the entity collection is going to be automatically materialized into a DML statement.

When it comes to adding or removing child records, the `@ElementCollection` behaves like a unidirectional `@OneToMany` relationship, annotated with `CascadeType.ALL` and `orphanRemoval`.

From a database perspective, there's one child table holding both the foreign key column and the collection element value.

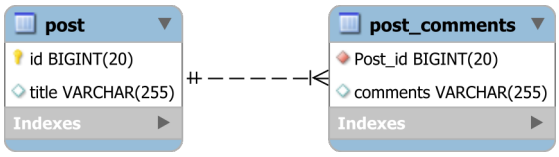


Figure 11.8: The `@ElementCollection` table relationship

To persist three comments, the data access layer only has to add them to the parent entity collection:

```
post.getComments().add("My first review");
post.getComments().add("My second review");
post.getComments().add("My third review");
```

Hibernate will issue the insert statements during Persistence Context flushing:

```
INSERT INTO Post_comments (Post_id, comments) VALUES (1, 'My first review')
INSERT INTO Post_comments (Post_id, comments) VALUES (1, 'My second review')
INSERT INTO Post_comments (Post_id, comments) VALUES (1, 'My third review')
```

Unfortunately, the remove operation uses the same logic as the unidirectional `@OneToMany` association, so when removing the first collection element:

```
post.getComments().remove(0);}
```

Hibernate deletes all the associated child-side records and re-inserts the in-memory ones back into the database table:

```
DELETE FROM Post_comments WHERE Post_id = 1
INSERT INTO Post_comments (Post_id, comments) VALUES (1, 'My second review')
INSERT INTO Post_comments (Post_id, comments) VALUES (1, 'My third review')
```

In spite its simplicity, the `@ElementCollection` is not very efficient for element removal. Just like unidirectional `@OneToMany` collections, the `@OrderColumn` can optimize the removal operation for entries located near the collection tail.

11.3.4 `@OneToMany` with `@JoinColumn`

JPA 2.0 added support for mapping the `@OneToMany` association with a `@JoinColumn` so that it can map the one-to-many table relationship. With the `@JoinColumn`, the `@OneToMany` association controls the child table foreign key so there is no need for a junction table.

On the JPA side, the class diagram is identical to the aforementioned unidirectional `@OneToMany` relationship, and the only difference is the JPA mapping which takes the additional `@JoinColumn`:

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "post_id")
private List<PostComment> comments = new ArrayList<>();
```

When adding three `PostComment` entities, Hibernate generates the following SQL statements:

```
post.getComments().add(new PostComment("My first review"));
post.getComments().add(new PostComment("My second review"));
post.getComments().add(new PostComment("My third review"));
```

```
INSERT INTO post_comment (review, id) VALUES ('My first review', 2)
INSERT INTO post_comment (review, id) VALUES ('My second review', 3)
INSERT INTO post_comment (review, id) VALUES ('My third review', 4)
UPDATE post_comment SET post_id = 1 WHERE id = 2
UPDATE post_comment SET post_id = 1 WHERE id = 3
UPDATE post_comment SET post_id = 1 WHERE id = 4
```

Besides the regular insert statements, Hibernate issues three update statements for setting the `post_id` column on the newly inserted child records. The child entity doesn't own the `post_id` foreign key column, so the child entities are inserted without any knowledge of their parent entity. After inserting the child entities, the parent collection can update each particular child entity foreign key column with the parent entity identifier.

Although it's an improvement over the regular `@OneToMany` mapping, in practice, it's still not as efficient as a regular bidirectional `@OneToMany` association.

When deleting the last element of the collection:

```
post.getComments().remove(2);
```

Hibernate generates the following SQL statements:

```
UPDATE post_comment SET post_id = null WHERE post_id = 1 AND id = 4
DELETE from post_comment WHERE id = 4
```

Again, there is an additional update statement associated with the child removal operation. When a child entity is removed from the parent-side collection, Hibernate will set the child table foreign key column to null. Afterwards, the orphan removal logic kicks in and it triggers a delete statement against the disassociated child entity.

Unlike the regular `@OneToMany` association, the `@JoinColumn` alternative is consistent in regard to the collection entry position that's being removed. So, when removing the first element of the collection:

```
post.getComments().remove(0);
```

Hibernate still generates an additional update statement:

```
UPDATE post_comment SET post_id = null WHERE post_id = 1 AND id = 2
DELETE from post_comment WHERE id = 2
```

Bidirectional `@OneToMany` with `@JoinColumn` relationship

The `@OneToMany` with `@JoinColumn` association can also be turned into a bidirectional relationship, but it requires instructing the child-side to avoid any insert and update synchronization:

```
@ManyToOne
@JoinColumn(name = "post_id", insertable = false, updatable = false)
private Post post;
```

The redundant update statements are generated for both the unidirectional and the bidirectional association, so the most efficient foreign key mapping is the `@ManyToOne` association.

11.4 @OneToOne

From a database perspective, the one-to-one association is based on a foreign key that's constrained to be unique. This way, a parent row can be referenced by at most one child record only.

In JPA, the `@OneToOne` relationship can be either unidirectional or bidirectional.

11.4.1 Unidirectional @OneToOne

In the following example, the `Post` entity represents the parent-side, while the `PostDetails` is the child-side of the one-to-one association.

As already mentioned, the JPA entity relationship diagram matches exactly the one-to-one table relationship.

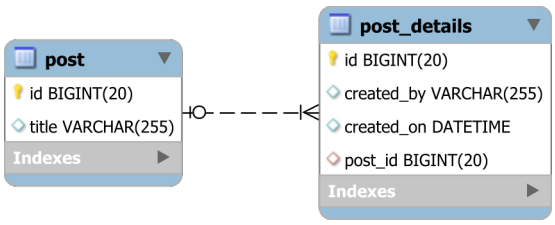


Figure 11.9: The one-to-one table relationship

Even from the Domain Model side, the unidirectional `@OneToOne` relationship is strikingly similar to the unidirectional `@ManyToOne` association.

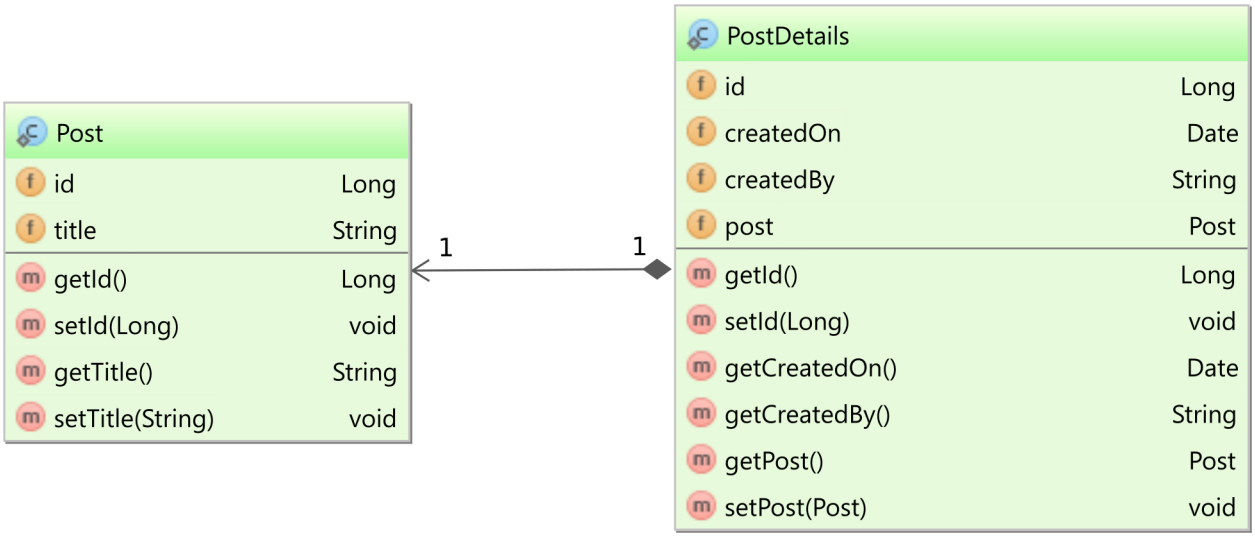


Figure 11.10: The unidirectional `@OneToOne` relationship

The mapping is done through the `@OneToOne` annotation, which, just like the `@ManyToOne` mapping, might also take a `@JoinColumn` as well.

```
@OneToOne
@JoinColumn(name = "post_id")
private Post post;
```

The unidirectional `@OneToOne` association controls the associated foreign key, so, when the `post` property is set:

```
Post post = entityManager.find(Post.class, 1L);
PostDetails details = new PostDetails("John Doe");
details.setPost(post);
entityManager.persist(details);
```

Hibernate populate the foreign key column with the associated `post` identifier:

```
INSERT INTO post_details (created_by, created_on, post_id, id)
VALUES ('John Doe', '2016-01-08 11:28:21.317', 1, 2)
```

Even if this is a unidirectional association, the `Post` entity is still the parent-side of this relationship. To fetch the associated `PostDetails`, a JPQL query is needed:

```
PostDetails details = entityManager.createQuery(
    "select pd " +
    "from PostDetails pd " +
    "where pd.post = :post", PostDetails.class)
.setParameter("post", post)
.getSingleResult();
```

If the `Post` entity always need its `PostDetails`, a separate query is undesirable. To overcome this limitation, it's important to know the `PostDetails` identifier prior to loading the entity.

One workaround would be to use a `@NaturalId`, which might not require a database access if the entity is stored in the second-level cache. Fortunately, there's even a simpler approach which is also portable across JPA providers as well. The JPA 2.0 specification added support for derived identifiers, making possible to link the `PostDetails` identifier to the `post` table primary key.

This way, the `post_details` table primary key can also be a foreign key referencing the `post` table identifier.

The `PostDetails` `@OneToOne` mapping is changed as follows:

```
@OneToOne
@MapsId
private Post post;
```

This time, the table relationship doesn't feature any additional foreign key column since the `post_details` table primary key references the `post` table primary key:

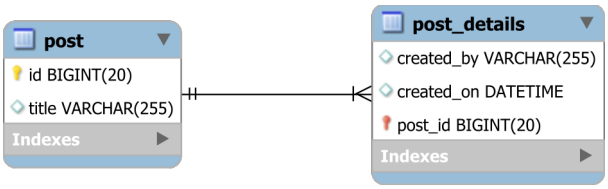


Figure 11.11: The shared key one-to-one

Because `PostDetails` has the same identifier as the parent `Post` entity, it can be fetched without having to write a JPQL query.

```
PostDetails details = entityManager.find(PostDetails.class, post.getId());
```

The shared primary key efficiency

First of all, the shared primary key approach reduces the memory footprint of the child-side table indexes since it requires a single indexed column instead of two. The more records a child table has, the better the improvement gain for reducing the number of indexed columns.

More, the child entity can now be simply retrieved from the second-level cache, therefore preventing a database hit. In the previous example, because the child entity identifier was not known, a query was inevitable. To optimize the previous use case, the *query cache* would be required as well, but the query cache is not without issues either.

Because of the reduced memory footprint and enabling the second-level cache direct retrieval, the JPA 2.0 *derived identifier* is the preferred `@OneToOne` mapping strategy. The shared primary key is not limited to unidirectional associations, being available for bidirectional `@OneToOne` relationships as well.

11.4.2 Bidirectional `@OneToOne`

A bidirectional `@OneToOne` association allows the parent entity to map the child-side as well:

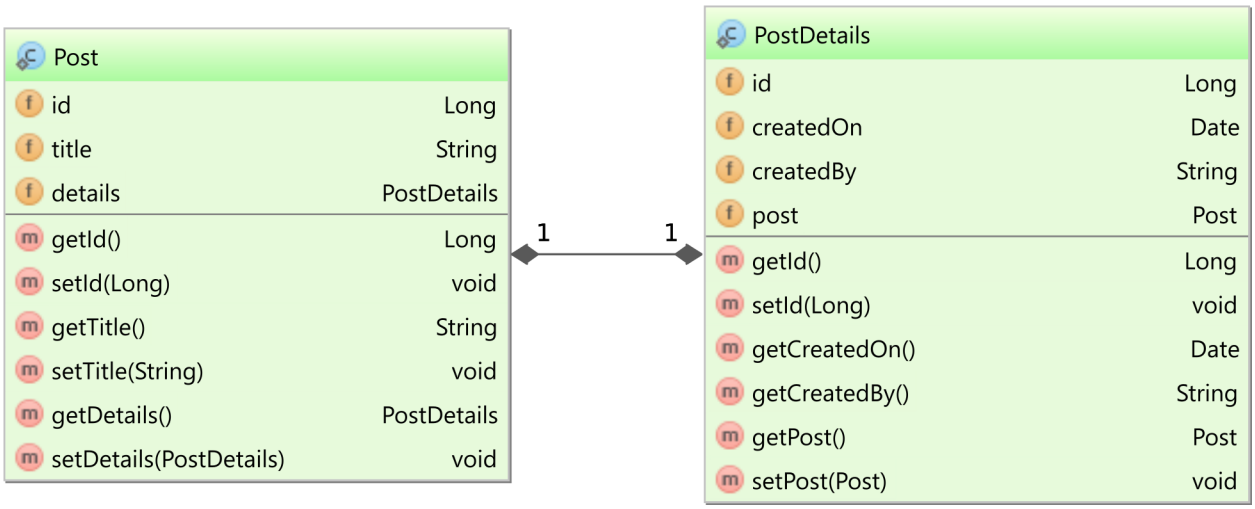


Figure 11.12: The bidirectional `@OneToOne` relationship

The parent-side defines a `mappedBy` attribute because the child-side (which can still share the primary key with its parent) is still in charge of this JPA relationship:

```
@OneToOne(mappedBy = "post", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
private PostDetails details;
```

Because this is a bidirectional relationship, the `Post` entity must ensure that both sides of this relationship are set, upon associating a `PostDetails` entity:

```
public void setDetails(PostDetails details) {
    this.details = details;
    details.setPost(this);
}
```

Even if the association is lazy, when fetching a `Post` entity, Hibernate fetches the child entity as well:

```
Post post = entityManager.find(Post.class, 1L);
```

```
SELECT p.id AS id1_0_0_,
       p.title AS title2_0_0_
FROM  post p
WHERE p.id = 1

SELECT pd.post_id AS post_id3_1_0_,
       pd.created_by AS created_1_1_0_,
       pd.created_on AS created_2_1_0_
FROM  post_details pd
WHERE pd.post_id = 1
```

So instead of only one query, Hibernate requires two queries.

Unlike the parent-side `@OneToMany` relationship where Hibernate can simply assign a proxy even if the child collection is empty, the `@OneToOne` relationship must decide if to assign the child reference to null or to an `Object`, be it the actual entity object type or a runtime Proxy.

This is an issue that affects the parent-side `@OneToOne` association, while the child-side, which has an associated foreign key column, knows whether the parent reference should be null or not. For this reason, the parent-side must execute a secondary query to know if there's a mirroring foreign key reference on the child-side.

If the application developer only needs parent entities, the additional child-side secondary queries are executed unnecessarily and this might affect application performance. The more parent entities are needed to be retrieved, the more obvious the secondary queries performance impact will be.

Bytecode enhancement

Even if the foreign key is `NOT NULL` and the parent-side is aware about its non-nullability through the `optional` attribute (e.g. `@OneToOne(mappedBy = "post", fetch = FetchType.LAZY, optional = false)`), Hibernate still generates a secondary select statement. Knowing that the child entity is never null, Hibernate can assign it to a Proxy instead of an actual child entity object reference. But the Persistence Context requires both the entity type and the identifier of every managed entity, so the child entity identifier must be known upon loading the parent entity, and the only way to find the associated `post_details` primary key is to execute a secondary query.

Bytecode enhancement is the only viable workaround; the parent entity Proxy being instrumented to only fetch the child entity upon first access. Because it's much simpler and performs well even without bytecode enhancement, the unidirectional `@OneToOne` relationship is often preferred.

11.5 @ManyToMany

The @ManyToMany relationship is the trickiest of all JPA relationships as the remaining of this chapter will demonstrate. Like the @OneToOne relationship, the @ManyToMany association can be either unidirectional or bidirectional. From a database perspective, the @ManyToMany association mirrors a many-to-many table relationship:

Figure 11.13: The many-to-many table relationship

11.5.1 Unidirectional @ManyToMany

In the following example, it makes sense to have the Post entity map the @ManyToMany relationship since there isn't much need for navigating this association from the Tag relationship side (although we can still do it with a JPQL query).

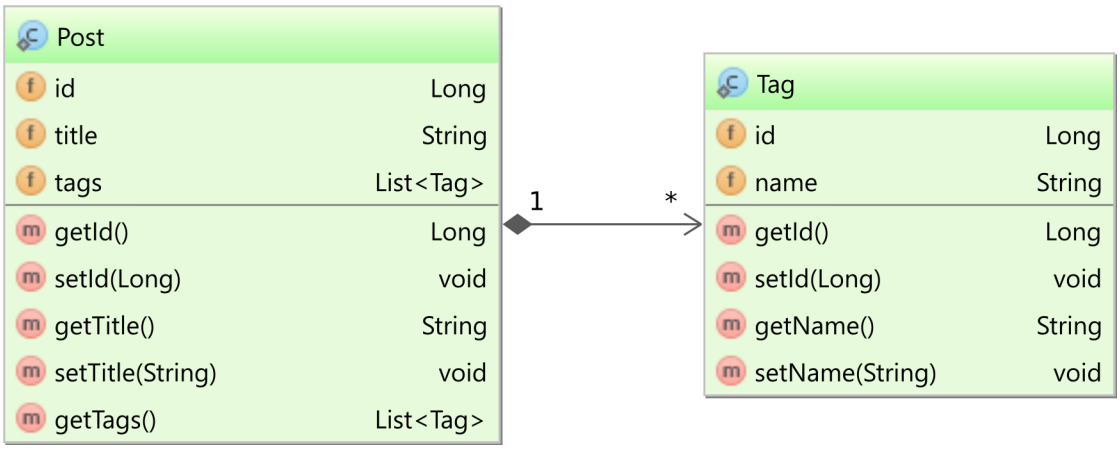


Figure 11.14: The unidirectional @ManyToMany relationship

In the Post entity, the @ManyToMany unidirectional association is mapped as follows:

```
@ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE } )
@JoinTable(name = "post_tag",
    joinColumns = @JoinColumn(name = "post_id"),
    inverseJoinColumns = @JoinColumn(name = "tag_id")
)
private List<Tag> tags = new ArrayList<>();
```

When adding several entities:

```
Post post1 = new Post("JPA with Hibernate");
Post post2 = new Post("Native Hibernate");

Tag tag1 = new Tag("Java");
Tag tag2 = new Tag("Hibernate");

post1.getTags().add(tag1);
post1.getTags().add(tag2);
post2.getTags().add(tag1);

entityManager.persist(post1);
entityManager.persist(post2);
```

Hibernate manages to persist both the `Post` and the `Tag` entities along with their junction records.

```
INSERT INTO post (title, id) VALUES ('JPA with Hibernate', 1)
INSERT INTO post (title, id) VALUES ('Native Hibernate', 4)

INSERT INTO tag (name, id) VALUES ('Java', 2)
INSERT INTO tag (name, id) VALUES ('Hibernate', 3)

INSERT INTO post_tag (post_id, tag_id) VALUES (1, 2)
INSERT INTO post_tag (post_id, tag_id) VALUES (1, 3)
INSERT INTO post_tag (post_id, tag_id) VALUES (4, 2)
```

Cascading

For `@ManyToMany` associations, `CascadeType.REMOVE` doesn't make too much sense when both sides represent independent entities. In this case, removing a `Post` entity shouldn't trigger a `Tag` removal because the `Tag` can be referenced by other *posts* as well. The same arguments apply to orphan removal since removing an entry from the `tags` collection should only delete the junction record and not the target `Tag` entity.

For both unidirectional and bidirectional associations, it's better to avoid the `CascadeType.REMOVE` mapping. Instead of `CascadeType.ALL`, the cascade attributes should be declared explicitly (e.g. `CascadeType.PERSIST`, `CascadeType.MERGE`).

But just like the unidirectional `@OneToMany` association, problems arise when it comes to removing the junction records:

```
post1.getTags().remove(tag1);
```

Hibernate deletes all junction rows associated with the `Post` entity whose `Tag` association is being removed and inserts back the remaining ones:

```
DELETE FROM post_tag WHERE post_id = 1

INSERT INTO post_tag (post_id, tag_id) VALUES (1, 3)
```

11.5.2 Bidirectional `@ManyToMany`

The bidirectional `@ManyToMany` relationship can be navigated from both the `Post` and the `Tag` side.

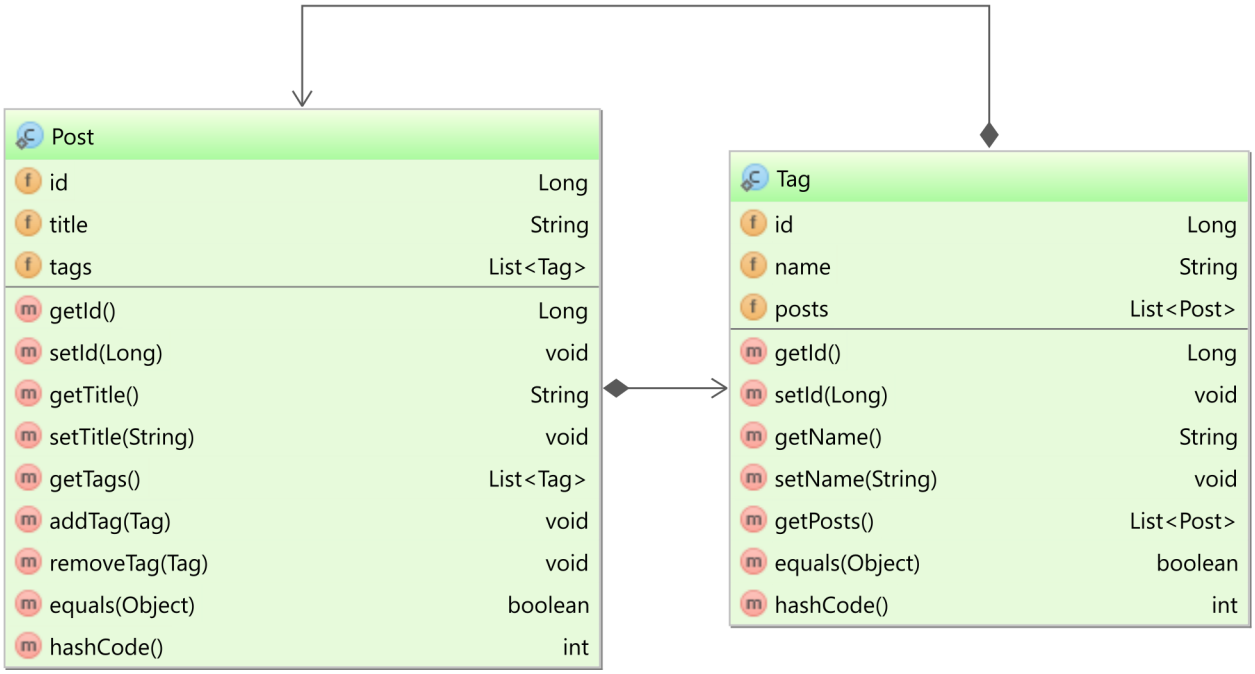


Figure 11.15: The unidirectional `@ManyToMany` relationship

While in the one-to-many and many-to-many association the child-side is the one holding the foreign key, for a many-to-many table relationship both ends are actually parent-sides and the junction table is the child-side.

Because the junction table is hidden when using the default `@ManyToMany` mapping, the application developer must choose an owning and a `mappedBy` side.

In this example, the `Post` retains the same mapping as shown in the unidirectional `@ManyToMany` section, while the `Tag` entity adds a `mappedBy` side:

```
@ManyToMany(mappedBy = "tags")
private List<Post> posts = new ArrayList<>();
```

Like any other bidirectional associations, both sides must in sync, so the helper methods are being added here as well. For a `@ManyToMany` association, the helper methods must be added to the entity that's more likely to interact with. In this example, the business logic manages `Post(s)` rather than `Tag(s)`, so the helper methods are added to the `Post` entity:

```
public void addTag(Tag tag) {
    tags.add(tag);
    tag.getPosts().add(this);
}

public void removeTag(Tag tag) {
    tags.remove(tag);
    tag.getPosts().remove(this);
}
```

Both `Post` and `Tag` entities have unique properties which can simplify the entity removal operation even when mixing detached and managed entities.

While adding an entity into the `@ManyToMany` collection is efficient since it requires a single SQL insert into the junction table, disassociating entities suffers from the same issue as the unidirectional `@ManyToMany` does.

When changing the order of the elements:

```
post1.getTags().sort(Collections.reverseOrder(Comparator.comparing(Tag::getId)));
```

Hibernate will delete all associated junction entries and reinsert them back again, as imposed by the unidirectional *bag* semantics:

```
DELETE FROM post_tag WHERE post_id = 1

INSERT INTO post_tag (post_id, tag_id) VALUES (1, 3)
INSERT INTO post_tag (post_id, tag_id) VALUES (1, 2)
```

Hibernate manages each side of a `@ManyToMany` relationship like a unidirectional `@OneToMany` association between the parent-side (e.g. `Post` or the `Tag`) and the hidden child-side (e.g. the `post_tag` table `post_id` or `tag_id` foreign keys). This is the reason why the entity removal or changing their order resulted in deleting all junction entries and reinserting them by mirroring the in-memory Persistence Context.

11.5.3 The `@OneToMany` alternative

Just like the unidirectional `@OneToMany` relationship can be optimized by allowing the child-side to control this association, the `@ManyToMany` mapping can be transformed so that the junction table is mapped to an entity.

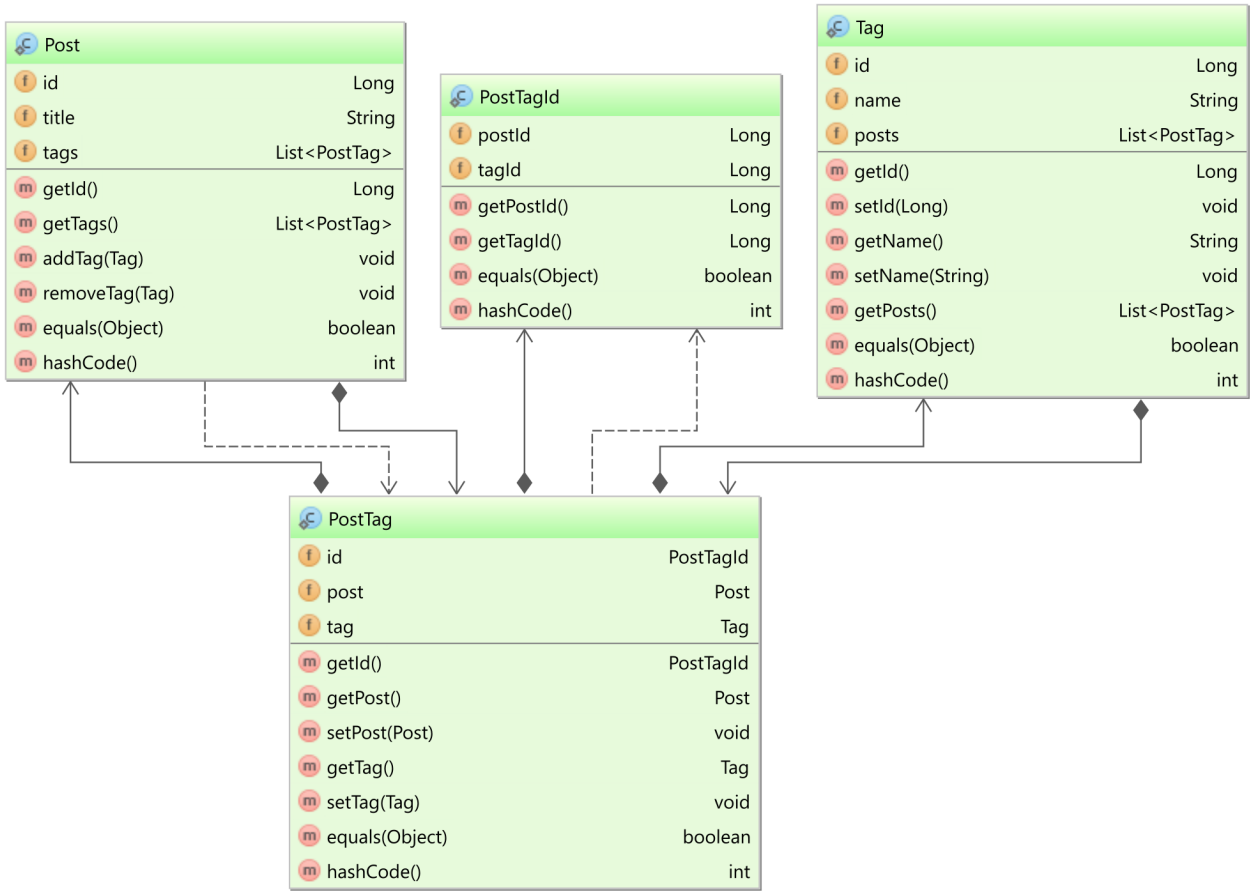


Figure 11.16: The `@OneToMany` many-to-many table relationship

The `PostTag` entity has a composed identifier made out of the `post_id` and `tag_id` columns.

```
@Embeddable
public static class PostTagId implements Serializable {

    private Long postId;

    private Long tagId;

    public PostTagId() {}

    public PostTagId(Long postId, Long tagId) {
        this.postId = postId;
        this.tagId = tagId;
    }

    public Long getPostId() {
        return postId;
    }

    public Long getTagId() {
        return tagId;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PostTagId that = (PostTagId) o;
        return Objects.equals(postId, that.postId) &&
                Objects.equals(tagId, that.tagId);
    }

    @Override
    public int hashCode() {
        return Objects.hash(postId, tagId);
    }
}
```

Using these columns, the `PostTag` entity can map the `@ManyToOne` sides as well:

```
@Entity(name = "PostTag")
@Table(name = "post_tag")
public static class PostTag {

    @EmbeddedId
    private PostTagId id;

    @ManyToOne
    @MapsId("postId")
    private Post post;

    @ManyToOne
    @MapsId("tagId")
    private Tag tag;

    private PostTag() {}

    public PostTag(Post post, Tag tag) {
        this.post = post;
        this.tag = tag;
        this.id = new PostTagId(post.getId(), tag.getId());
    }

    //Getters and setters omitted for brevity

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PostTag that = (PostTag) o;
        return Objects.equals(post, that.post) &&
                Objects.equals(tag, that.tag);
    }

    @Override
    public int hashCode() {
        return Objects.hash(post, tag);
    }
}
```

The `Post` entity maps the bidirectional `@OneToMany` side of the `post` `@ManyToOne` association:

```
@OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval = true)
private List<PostTag> tags = new ArrayList<>();
```

The `Tag` entity maps the bidirectional `@OneToMany` side of the `tag` `@ManyToOne` association:

```
@OneToMany(mappedBy = "tag", cascade = CascadeType.ALL, orphanRemoval = true)
private List<PostTag> posts = new ArrayList<>();
```

This way, the bidirectional `@ManyToMany` relationship is transformed in two bidirectional `@OneToMany` associations.

The `removeTag` helper method is much more complex because it needs to locate the `PostTag` associated with the current `Post` entity and the `Tag` that's being disassociated.

```
public void removeTag(Tag tag) {
    for (Iterator<PostTag> iterator = tags.iterator(); iterator.hasNext(); ) {
        PostTag postTag = iterator.next();
        if (postTag.getPost().equals(this) && postTag.getTag().equals(tag)) {
            iterator.remove();
            postTag.getTag().getPosts().remove(postTag);
            postTag.setPost(null);
            postTag.setTag(null);
            break;
        }
    }
}
```

The `PostTag` `equals` and `hashCode` methods rely on the `Post` and `Tag` equality semantics. The `Post` entity uses the `title` as a business key, while the `Tag` relies on its `name` column uniqueness constraint.

When rerunning the entity removal example featured in the unidirectional `@ManyToMany` section:

```
post1.removeTag(tag1);
```

Hibernate issues a single delete statement, therefore targeting a single `PostTag` junction record:

```
DELETE FROM post_tag WHERE post_id = 1 AND tag_id = 3
```

Changing the junction elements order has not effect this time:

```
post1.getTags().sort((postTag1, postTag2) ->
    postTag2.getId().getTagId().compareTo(postTag1.getId().getTagId())
)
```

This is because the `@ManyToOne` side only monitors the foreign key column changes and the internal collection state is not taken into consideration. To materialize the order of elements, the `@OrderColumn` must be used instead:

```
@OneToMany(mappedBy = "post", cascade = CascadeType.ALL, orphanRemoval = true)
@OrderColumn(name = "entry")
private List<PostTag> tags = new ArrayList<>();
```

The `post_tag` junction table will feature an `entry` column storing the collection element order. When reversing the element order, Hibernate will update the `entry` column:

```
UPDATE post_tag SET entry = 0 WHERE post_id = 1 AND tag_id = 4
UPDATE post_tag SET entry = 1 WHERE post_id = 1 AND tag_id = 3
```

The most efficient JPA relationships are the ones where the foreign key side is controlled by a child-side `@ManyToOne` or `@OneToOne` association. For this reason, the many-to-many table relationship is best mapped with two bidirectional `@OneToMany` associations. The entity removal and the element order changes are more efficient than the default `@ManyToMany` relationship and the junction entity can also map additional columns (e.g. `created_on`, `created_by`).

12. Inheritance

Java, like any other object-oriented programming language, makes heavy use of inheritance and polymorphism. Inheritance allows defining class hierarchies that offer different implementations of a common interface.

Conceptually, the Domain Model defines both data (e.g. persisted entities) and behavior (business logic). Nevertheless, inheritance is more useful for varying behavior rather than data (composition is much more suitable for sharing structures). Even if the data (persisted entities) and the business logic (transactional services) are decoupled, inheritance can still help varying business logic (e.g. Visitor pattern[1]).

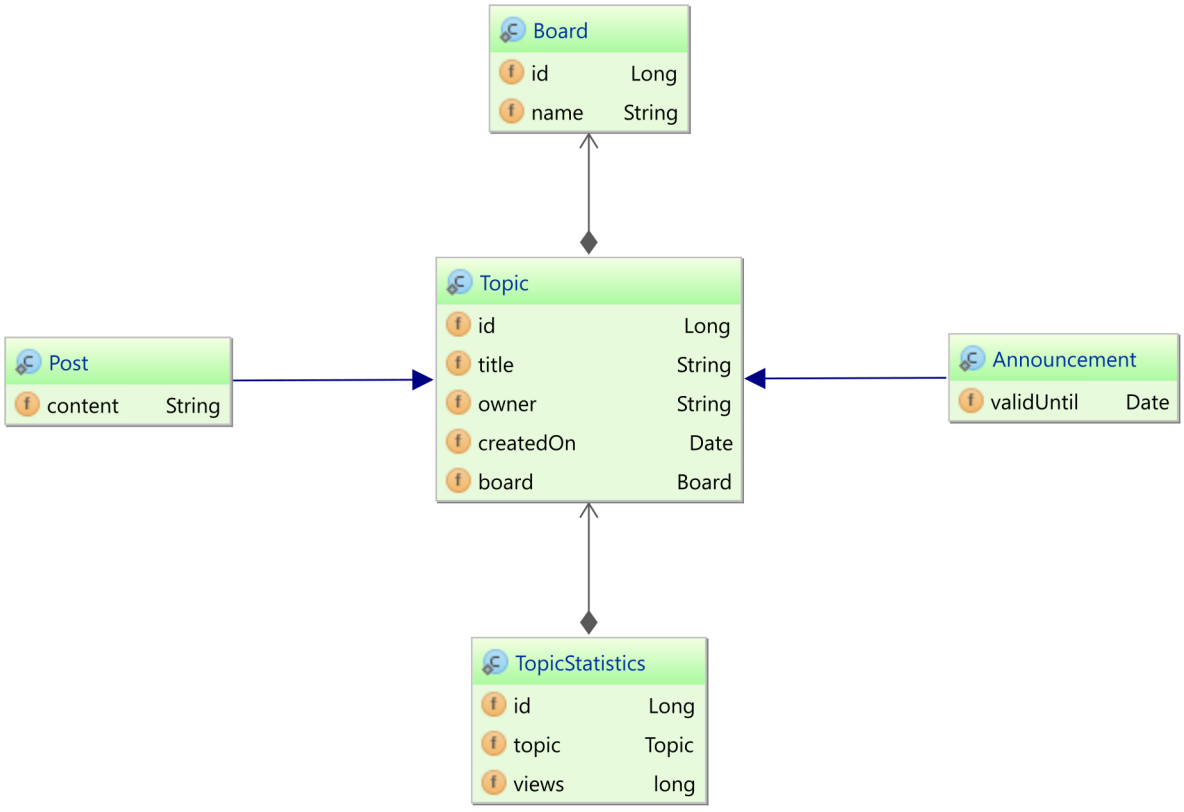


Figure 12.1: Domain Model Inheritance

[1] https://en.wikipedia.org/wiki/Visitor_pattern

The root entity of this Domain Model is the `Board` entity because, either directly or indirectly, all the other entities are associated with a `Board`

```
@Entity(name = "Board") @Table(name = "board")
public class Board {

    @Id @GeneratedValue
    private Long id;

    private String name;

    //Getters and setters omitted for brevity
}
```

The end user can submit either a `Post` or an `Announcement` on a particular `Board`. Because the `Post` and the `Announcement` share the same functionality (differing only in data), they both inherit from a `Topic` base class.

The `Topic` class defines a relationship to a `Board` entity, hence the `Post` and the `Announcement` entities can also be associated with a `Board` instance.

```
@Entity @Table(name = "topic")
public class Topic {

    @Id @GeneratedValue
    private Long id;

    private String title;

    private String owner;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdOn = new Date();

    @ManyToOne(fetch = FetchType.LAZY)
    private Board board;

    //Getters and setters omitted for brevity
}
```

Both the `Post` and the `Announcement` entities extend the `Topic` class and define their own specific properties.

```
@Entity @Table(name = "post")
public class Post extends Topic {

    private String content;

    //Getters and setters omitted for brevity
}

@Entity @Table(name = "announcement")
public class Announcement extends Topic {

    @Temporal(TemporalType.TIMESTAMP)
    private Date validUntil;

    //Getters and setters omitted for brevity
}
```

The `TopicStatistics` is at the bottom of this Domain Model as it's only needed for monitoring purposes, without being directly associated with the main business logic. Because statistics are needed for both `Post` and `Announcement` entities, the `TopicStatistics` defines a `Topic` entity association.

```
@Entity @Table(name = "topic_statistics")
public class TopicStatistics {

    @Id @GeneratedValue
    private Long id;

    @OneToOne @JoinColumn(name = "id") @MapsId
    private Topic topic;

    private long views;

    //Getters and setters omitted for brevity
}
```

Another approach would be to add this relationship into the `Topic` class, and the `topic` rows would then reference the `topic_statistics` records. For the sake of demonstrating how entity association polymorphism works, the `TopicStatistics` was chosen to be the child-side.

As natural as this Domain Model may be represented in an object-oriented programming language, transposing it to a relational database is anything but straightforward. Relational database systems do not support inheritance, relying on tuples and relational algebra for representing and manipulating data. For this reason, mapping inheritance in a relational database is one of the most obvious object-relational impedance mismatch.

Without native support from the database system, inheritance can only be emulated through table relationships. In the *Patterns of Enterprise Application Architecture* book, Martin Fowler defines three ways of mapping inheritance into a relational database:

- Single Table Inheritance[2], which uses a single database table to represent all classes in a given inheritance hierarchy
- Class Table Inheritance[3], which maps each class to a table, and the inheritance association is resolved through table joins
- Concrete Table Inheritance[4], where each table defines all fields that are either defined in the subclass or inherited from a super class.

The JPA specification defines all these three inheritance mapping models through the following strategies:

- `InheritanceType.SINGLE_TABLE`
- `InheritanceType.JOINED`
- `InheritanceType.TABLE_PER_CLASS`

JPA also covers the case when inheritance is only available in the Domain Model, without being mirrored into the database (e.g. `@MappedSuperclass`).

Whenever the data access layer implements a functionality without support from the underlying database system, care must be taken to ensure that application performance is not compromised. This chapter aims to analyze what trade-offs are required for employing inheritance as well as its impact on application performance.

[2] http://martinfowler.com/eaaCatalog/singleTableInheritance.html
[3] http://martinfowler.com/eaaCatalog/classTableInheritance.html
[4] http://martinfowler.com/eaaCatalog/concreteTableInheritance.html

12.1 Single table

The single table inheritance is the default JPA strategy, funneling a whole inheritance Domain Model hierarchy into a single database table.

To employ this strategy, the `Topic` entity class must be mapped with one of the following annotations:

- `@Inheritance` (being the default inheritance model, it's not mandatory to supply the strategy when using single table inheritance)
- `@Inheritance(strategy = InheritanceType.SINGLE_TABLE)`.

The `Post` and the `Announcement` entities don't need any extra mapping (the Java inheritance semantics being sufficient).

Preserving the same layout as depicted in the Domain Model class diagram, the table relationships associated with this inheritance strategy look like this:

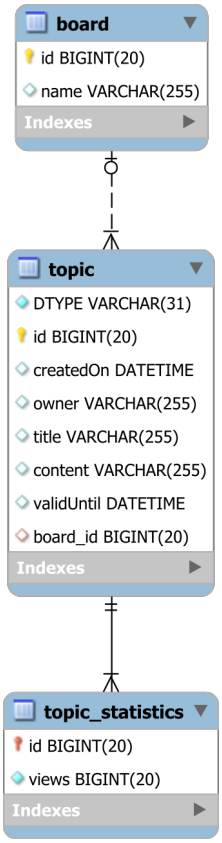


Figure 12.2: Single table

The `topic` table contains columns associated with the `Topic` base class as well as columns related to properties from `Post` and `Announcement` entities.

In the following example, one `Post` and one `Announcement` entities are going to be persisted along with their associated `@OneToOne` `TopicStatistics` relations.

```
Post post = new Post();
post.setOwner("John Doe");
post.setTitle("Inheritance");
post.setContent("Best practices");
post.setBoard(board);

entityManager.persist(post);

Announcement announcement = new Announcement();
announcement.setOwner("John Doe");
announcement.setTitle("Release x.y.z.Final");
announcement.setValidUntil(Timestamp.valueOf(LocalDateTime.now().plusMonths(1)));
announcement.setBoard(board);

entityManager.persist(announcement);

TopicStatistics postStatistics = new TopicStatistics(post);
postStatistics.incrementViews();
entityManager.persist(postStatistics);

TopicStatistics announcementStatistics = new TopicStatistics(announcement);
announcementStatistics.incrementViews();
entityManager.persist(announcementStatistics);
```

Both the `Post` and the `Announcement` entities are saved in the `topic` table whose primary key is shared with the `topic_statistics` table.

```
INSERT INTO topic (board_id, createdOn, owner, title, content, DTYPE, id)
VALUES (1, '2016-01-17 09:22:22.11', 'John Doe', 'Inheritance',
    'Best practices', 'Post', 1)

INSERT INTO topic (board_id, createdOn, owner, title, validUntil, DTYPE, id)
VALUES (1, '2016-01-17 09:22:22.11', 'John Doe', 'Release x.y.z.Final',
    '2016-02-17 09:22:22.114', 'Announcement', 2)

INSERT INTO topic_statistics (views, id) VALUES (1, 2)

INSERT INTO topic_statistics (views, id) VALUES (1, 3)
```

One advantage of using inheritance in the Domain Model is the support for polymorphic queries. When the application developer issues a select query against the `Topic` entity:

```
List<Topic> topics = entityManager.createQuery(
    "select t from Topic t where t.board.id = :boardId", Topic.class)
    .setParameter("boardId", 1L)
    .getResultList();
```

Hibernate goes to the `topic` table, and, after fetching the result set, it maps every row to its associated subclass instance (e.g. `Post` or `Announcement`) by analyzing the discriminator column (e.g. `DTYPE`) value.

```
SELECT t.id AS id2_1_, t.board_id AS board_id8_1_, t.createdOn AS createdO3_1_,
    t.owner AS owner4_1_, t.title AS title5_1_, t.content AS content6_1_,
    t.validUntil AS validUnt7_1_, t.DTYPE AS DTYPE1_1_
FROM topic t
WHERE t.board_id = 1
```

Domain Model inheritance allows base class entity associations to be automatically resolved upon being retrieved. When loading a `TopicStatistics` along with its `Topic` relation:

```
TopicStatistics statistics = entityManager.createQuery(
    "select s from TopicStatistics s join fetch s.topic t where t.id = :topicId"
    , TopicStatistics.class)
    .setParameter("topicId", topicId)
    .getSingleResult();
```

Hibernate joins the `topic_statistics` and the `topic` tables so that it can create a `TopicStatistics` entity with an actual `Post` or `Announcement` property object reference.

```
SELECT
    ts.id AS id1_2_0_, t.id AS id2_1_1_, ts.views AS views2_2_0_,
    t.board_id AS board_id8_1_1_, t.createdOn AS createdO3_1_1_,
    t.owner AS owner4_1_1_, t.title AS title5_1_1_, t.content AS content6_1_1_,
    t.validUntil AS validUnt7_1_1_, t.DTYPE AS DTYPE1_1_1_
FROM topic_statistics ts
INNER JOIN topic t ON ts.id = t.id
WHERE t.id = 2
```

Even if not practical in this particular example, `@OneToMany` associations are also possible.

The `Board` entity can map a bidirectional `@OneToMany` relationship as follows:

```
@OneToMany(mappedBy = "board")
private List<Topic> topics = new ArrayList<>();
```

Fetching the collection lazily generates a separate select statement, identical to the aforementioned `Topic` entity query. When fetching the collection eagerly, Hibernate requires a single table join.

```
Board board = entityManager.createQuery(
    "select b from Board b join fetch b.topics where b.id = :id", Board.class)
    .setParameter("id", id)
    .getSingleResult();
```

```
SELECT b.id AS id1_0_0_, t.id AS id2_1_1_, b.name AS name2_0_0_,
    t.board_id AS board_id8_1_1_, t.createdOn AS createdO3_1_1_,
    t.owner AS owner4_1_1_, t.title AS title5_1_1_, t.content AS content6_1_1_,
    t.validUntil AS validUnt7_1_1_, t.DTYPE AS DTYPE1_1_1_,
    t.board_id AS board_id8_1_0__, t.id AS id2_1_0__
FROM board b
INNER JOIN topic t ON b.id = t.board_id
WHERE b.id = 1
```

Performance and data integrity considerations

Since only one table is used for storing entities, both read and write operations are fast. Even when using a `@ManyToOne` or a `@OneToOne` base class association, Hibernate needs a single join between the child-side table (e.g. `topic_statistics`) and the parent-side one (e.g. `topic`). The `@OneToMany` base class entity relationship is also efficient since it either generates a secondary select or a single parent-child table join.

Because all subclass properties are collocated in a single table, `NOT NULL` constraints are not allowed for columns belonging to subclasses. Being automatically inherited by all subclasses, the base class properties may be non-nullable. From a data integrity perspective, this limitation defeats the purpose of Consistency (guaranteed by the ACID properties).

Nevertheless, the data integrity rules can be enforced through database trigger procedures (a column non-nullability is accounted based on the class discriminator value). Another approach is to move the check into the data access layer. Bean Validation can validate `@NotNull` properties at runtime. JPA also defines callback methods (e.g. `@PreUpdate`, `@PreUpdate`) as well as entity listeners (e.g. `@EntityListeners`) which can throw an exception when a non-null constraint is violated.

12.2 Join table

The join table inheritance resembles the Domain Model class diagram since each class is mapped to an individual table. The subclass tables have a foreign key column referencing the base class table primary key.

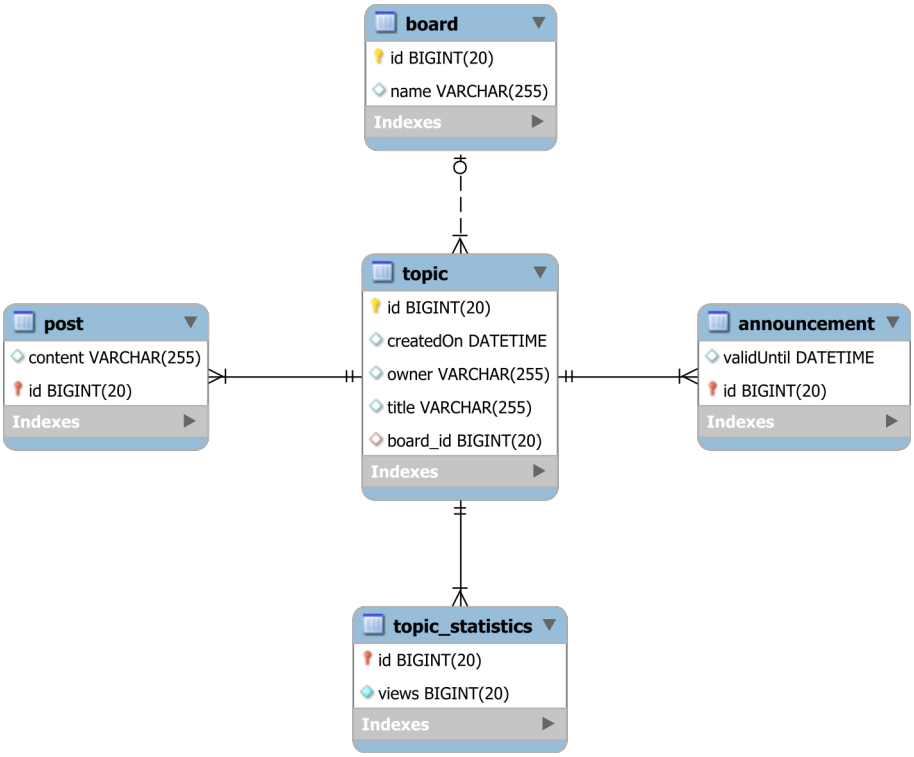


Figure 12.3: Join table

To use this inheritance strategy, the `Topic` entity must be annotated with:

```
@Inheritance(strategy = InheritanceType.JOINED)
```

The `Post` and the `Announcement` entities can use a `@PrimaryKeyJoinColumn` mapping to define the base class foreign key column.

By default, the subclass table primary key column is used as a foreign key as well.

When persisting the same entities defined in the single table section, Hibernate generates the following SQL statements:

```
INSERT INTO topic (board_id, createdOn, owner, title, id)
VALUES (1, '2016-01-17 09:27:10.694', 'John Doe', 'Inheritance', 1)

INSERT INTO post (content, id) VALUES ('Best practices', 1)

INSERT INTO topic (board_id, createdOn, owner, title, id)
VALUES (1, '2016-01-17 09:27:10.694', 'John Doe', 'Release x.y.z.Final', 2)

INSERT INTO announcement (validUntil, id) VALUES ('2016-02-17 09:27:10.698', 2)

INSERT INTO topic_statistics (views, id) VALUES (1, 2)

INSERT INTO topic_statistics (views, id) VALUES (1, 3)
```

The base class information goes into the `topic` table while the subclass content goes in the `post` or the `announcement` tables. When fetching all `Topic` entities associated with a specific `Board`:

```
List<Topic> topics = entityManager.createQuery(
    "select t from Topic t where t.board.id = :boardId", Topic.class)
    .setParameter("boardId", 1L)
    .getResultList();
```

Hibernate must join the base class with each individual subclass table.

```
SELECT
    t.id AS id1_3_,
    t.board_id AS board_id5_3_,
    t.createdOn AS createdO2_3_,
    t.owner AS owner3_3_,
    t.title AS title4_3_,
    t1_.content AS content1_2_,
    t2_.validUntil AS validUnt1_0_,
    CASE WHEN t1_.id IS NOT NULL THEN 1
        WHEN t2_.id IS NOT NULL THEN 2
        WHEN t.id IS NOT NULL THEN 0
    END AS clazz_
FROM topic t
LEFT OUTER JOIN post t1_ ON t.id = t1_.id
LEFT OUTER JOIN announcement t2_ ON t.id = t2_.id
WHERE t.board_id = 1
```

When loading a `TopicStatistics` entity along with its `Topic` association:

```
TopicStatistics statistics = entityManager.createQuery(
    "select s from TopicStatistics s join fetch s.topic t where t.id = :topicId"
    , TopicStatistics.class)
    .setParameter("topicId", topicId)
    .getSingleResult();
```

Hibernate must join four tables to construct the desired result set:

```
SELECT
    ts.id AS id1_4_0_,
    t.id AS id1_3_1_,
    ts.views AS views2_4_0_,
    t.board_id AS board_id5_3_1_,
    t.createdOn AS createdO2_3_1_,
    t.owner AS owner3_3_1_,
    t.title AS title4_3_1_,
    t1_.content AS content1_2_1_,
    t2_.validUntil AS validUnt1_0_1_,
    CASE WHEN t1_.id IS NOT NULL THEN 1
        WHEN t2_.id IS NOT NULL THEN 2
        WHEN t.id IS NOT NULL THEN 0
    END AS clazz_1_
FROM topic_statistics ts
INNER JOIN topic t ON ts.id = t.id
LEFT OUTER JOIN post t1_ ON t.id = t1_.id
LEFT OUTER JOIN announcement t2_ ON t.id = t2_.id
WHERE t.id = 2
```

Considering that the `Board` entity defines a `@OneToMany` `Topic` association:

```
@OneToMany(mappedBy = "board")
private List<Topic> topics = new ArrayList<>();
```

Fetching the collection lazily generates a separate select statement, identical to the previous `Topic` entity query.

When fetching the collection eagerly:

```
Board board = entityManager.createQuery(
    "select b from Board b join fetch b.topics where b.id = :id", Board.class)
    .setParameter("id", id)
    .getSingleResult();
```

Hibernate requires three joins to fetch all topic-related information.

```
SELECT
    b.id AS id1_1_0_, t.id AS id1_3_1_, b.name AS name2_1_0_,
    t.board_id AS board_id5_3_1_, t.createdOn AS createdO2_3_1_,
    t.owner AS owner3_3_1_, t.title AS title4_3_1_,
    t1_.content AS content1_2_1_, t2_.validUntil AS validUnt1_0_1_,
    CASE WHEN t1_.id IS NOT NULL THEN 1
         WHEN t2_.id IS NOT NULL THEN 2
         WHEN t.id IS NOT NULL THEN 0
    END AS clazz_1_,
    t.board_id AS board_id5_3_0__, t.id AS id1_3_0__
FROM board b
INNER JOIN topic t ON b.id = t.board_id
LEFT OUTER JOIN post t1_ ON t.id = t1_.id
LEFT OUTER JOIN announcement t2_ ON t.id = t2_.id
WHERE b.id = 1
```

Performance considerations

Unlike single table inheritance, the joined table strategy allows nullable subclass property columns.

When writing data, Hibernate requires two insert statements for each subclass entity, so there's a performance impact compared to single table inheritance. The index memory footprint also increases because instead of a single table primary key, the database must index the base class and all subclasses primary keys.

When reading data, polymorphic queries require joining the base class with all subclass tables, so, if there are `n` subclasses, Hibernate will need `n + 1` joins. The more joins, the more difficult it is for the database to calculate the most efficient execution plan.

12.3 Table-per-class

The table-per-class inheritance model has a table layout similar to the joined table strategy, but, instead of storing base class columns in the `topic` table, each subclass table stores also columns from the `topic` table. There is no foreign key between the `topic` and the `post` or `announcement` subclass tables, and there is no foreign key in the `topic_statistics` table either.

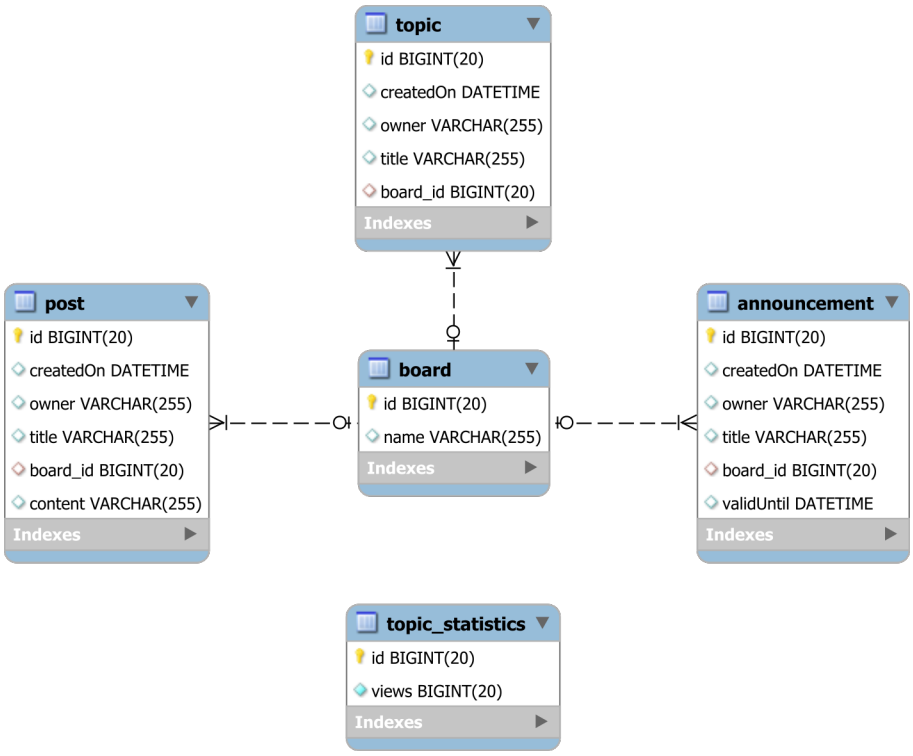


Figure 12.4: Table-per-class

To use this inheritance strategy, the `Topic` must be annotated with `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`.

Inserting the `Post` and the `Announcement` entities defined in the single table inheritance section generates the following SQL statements:

```
INSERT INTO post (board_id, createdOn, owner, title, content, id)
VALUES (1, '2016-01-17 09:31:12.018', 'John Doe', 'Inheritance',
        'Best practices', 2)

INSERT INTO announcement (board_id, createdOn, owner, title, validUntil, id)
VALUES (1, '2016-01-17 09:31:12.018', 'John Doe', 'Release x.y.z.Final',
        '2016-02-17 09:31:12.023', 3)
```

```
INSERT INTO topic_statistics (views, id) VALUES (1, 2)

INSERT INTO topic_statistics (views, id) VALUES (1, 3)
```

Unlike the joined table inheritance, each persisted subclass entity requires a single insert statement.

When fetching all `Topic` entities associated with a specific `Board`:

```
List<Topic> topics = entityManager.createQuery(
    "select t from Topic t where t.board.id = :boardId", Topic.class)
    .setParameter("boardId", 1L)
    .getResultList();
```

Hibernate uses `UNION ALL` to fetch rows from the base class and every subclass table in this particular inheritance tree.

```
SELECT
    t.id AS id1_3_,
    t.board_id AS board_id5_3_,
    t.createdOn AS createdO2_3_,
    t.owner AS owner3_3_,
    t.title AS title4_3_,
    t.content AS content1_2_,
    t.validUntil AS validUnt1_0_,
    t.clazz_ AS clazz_
FROM (
    SELECT id, createdOn, owner, title, board_id,
        CAST(NULL AS VARCHAR(100)) AS content,
        CAST(NULL AS TIMESTAMP) AS validUntil, 0 AS clazz_
    FROM topic
    UNION ALL
    SELECT id, createdOn, owner, title, board_id, content,
        CAST(NULL AS TIMESTAMP) AS validUntil, 1 AS clazz_
    FROM post
    UNION ALL
    SELECT id, createdOn, owner, title, board_id,
        CAST(NULL AS VARCHAR(100)) AS content, validUntil, 2 AS clazz_
    FROM announcement ) t
WHERE  t.board_id = 1
```

When loading a `TopicStatistics` while also fetching its associated `Topic` relation, Hibernate must use `UNION ALL` for the inheritance tables to construct the desired result set:

```
TopicStatistics statistics = entityManager.createQuery(
    "select s from TopicStatistics s join fetch s.topic t where t.id = :topicId"
    , TopicStatistics.class)
    .setParameter("topicId", topicId)
    .getSingleResult();
```

```
SELECT
    ts.id AS id1_4_0_, t.id AS id1_3_1_, ts.views AS views2_4_0_,
    t.board_id AS board_id5_3_1_, t.createdOn AS createdO2_3_1_,
    t.owner AS owner3_3_1_, t.title AS title4_3_1_, t.content AS content1_2_1_,
    t.validUntil AS validUnt1_0_1_, t.clazz_ AS clazz_1_
FROM topic_statistics ts
INNER JOIN (
    SELECT id, createdOn, owner, title, board_id,
        CAST(NULL AS VARCHAR(100)) AS content,
        CAST(NULL AS TIMESTAMP) AS validUntil, 0 AS clazz_
    FROM topic
    UNION ALL
    SELECT id, createdOn, owner, title, board_id,
        content,
        CAST(NULL AS TIMESTAMP) AS validUntil, 1 AS clazz_
    FROM post
    UNION ALL
    SELECT id, createdOn, owner, title, board_id,
        CAST(NULL AS VARCHAR(100)) AS content,
        validUntil, 2 AS clazz_
    FROM announcement
) t ON ts.id = t.id
WHERE t.id = 2
```

The identity generator is not allowed with this strategy because rows belonging to different subclasses would share the same identifier, therefore conflicting in polymorphic `@ManyToOne` or `@OneToOne` associations.

Considering that the `Board` entity defines a `@OneToMany` `Topic` association:

```
@OneToMany(mappedBy = "board")
private List<Topic> topics = new ArrayList<>();
```

Fetching the collection lazily generates a separate select statement, identical to the previous `Topic` entity query.

When fetching the `topics` collection eagerly:

```
Board board = entityManager.createQuery(
    "select b from Board b join fetch b.topics where b.id = :id", Board.class)
    .setParameter("id", id)
    .getSingleResult();
```

Hibernate requires a join with the result of unifying all three topic-related tables.

```
SELECT
    b.id AS id1_1_0_, t1.id AS id1_3_1_, b.name AS name2_1_0_,
    t1.board_id AS board_id5_3_1_, t1.createdOn AS createdO2_3_1_,
    t1.owner AS owner3_3_1_, t1.title AS title4_3_1_,
    t1.content AS content1_2_1_, t1.validUntil AS validUnt1_0_1_,
    t1.clazz_ AS clazz_1_, t1.board_id AS board_id5_3_0__, t1.id AS id1_3_0__
FROM board b
INNER JOIN (
    SELECT id, createdOn, owner, title, board_id,
        CAST(NULL AS VARCHAR(100)) AS content,
        CAST(NULL AS TIMESTAMP) AS validUntil, 0 AS clazz_
    FROM topic
    UNION ALL
    SELECT id, createdOn, owner, title, board_id,
        content,
        CAST(NULL AS TIMESTAMP) AS validUntil, 1 AS clazz_
    FROM post
    UNION ALL
    SELECT id, createdOn, owner, title, board_id,
        CAST(NULL AS VARCHAR(100)) AS content,
        validUntil, 2 AS clazz_
    FROM announcement
) t1 ON b.id = t1.board_id
WHERE b.id = 1
```

Performance considerations

While write operations are faster than in the joined table strategy, the read operations are only efficient when querying against the actual subclass entities. Polymorphic queries can have a considerable performance impact because Hibernate must select all subclass tables and use `UNION ALL` to build the whole inheritance tree result set. As a rule of thumb, the more subclass tables, the least efficient the polymorphic queries will get.

12.4 Mapped superclass

If the `Topic` class is not required to be a stand-alone entity, it's more practical to leave inheritance out of the database. This way, the `Topic` can be made `abstract` and marked with the `@MappedSuperclass` annotation so that JPA can acknowledge the inheritance model on the entity-side only.

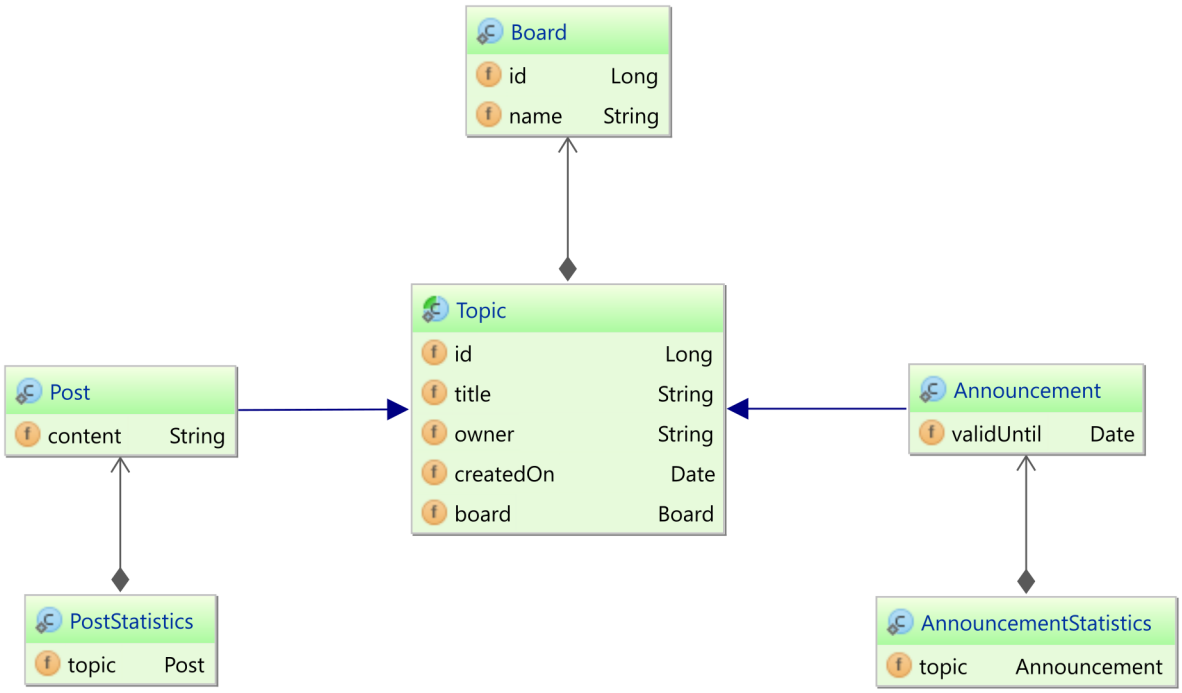


Figure 12.5: Mapped superclass class diagram

Having a single `TopicStatistics` entity with a `@OneToOne Topic` association is no longer possible because the `Topic` class is not an entity. This way, each `Topic` subclass must define its own statistics associations (e.g. `PostStatistics`, `AnnouncementStatistics`).

The `PostStatistics` and the `AnnouncementStatistics` entities looks as follows:

```
@Entity @Table(name = "post_statistics")
public static class PostStatistics
    extends TopicStatistics<Post> {

    @OneToOne
    @JoinColumn(name = "id")
    @MapsId
    private Post topic;

    private PostStatistics() {}

    public PostStatistics(Post topic) {
        this.topic = topic;
    }

    @Override
    public Post getTopic() {
        return topic;
    }
}

@Entity @Table(name = "announcement_statistics")
public static class AnnouncementStatistics
    extends TopicStatistics<Announcement> {

    @OneToOne
    @JoinColumn(name = "id")
    @MapsId
    private Announcement topic;

    private AnnouncementStatistics() {}

    public AnnouncementStatistics(Announcement topic) {
        this.topic = topic;
    }

    @Override
    public Announcement getTopic() {
        return topic;
    }
}
```

To retain the inheritance semantics, the base class properties are going to be merged with the subclass ones, so the associated subclass entity table will contain both. This is similar to the table-per-class inheritance strategy, with the distinction that the base class is not mapped to a database table (hence, it cannot be used in polymorphic queries or associations).

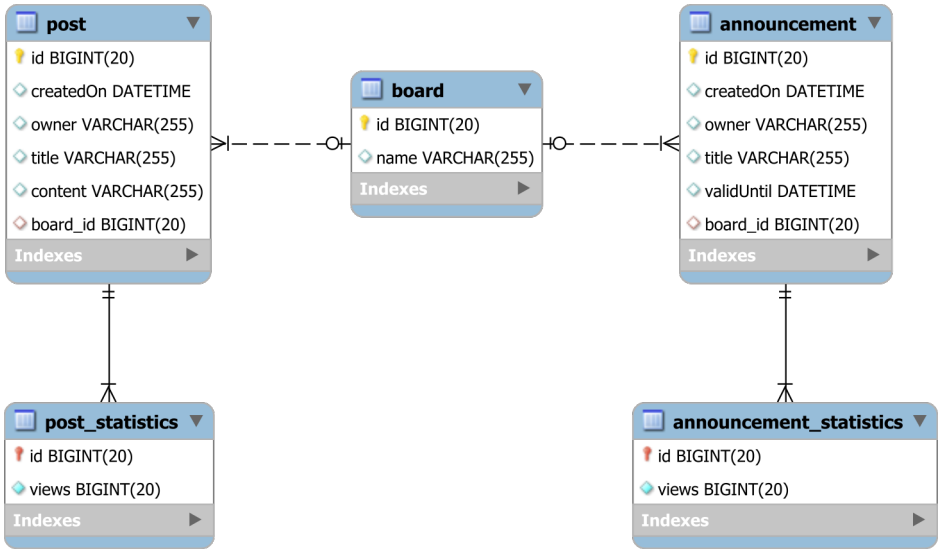


Figure 12.6: Mapped superclass

The `post` and the `announcement` tables feature columns that were inherited from the `Topic` base class. When persisting the same `Post` and `Announcement` entities and saving statistics using the newly defined entity classes:

```
TopicStatistics postStatistics = new PostStatistics(post);
postStatistics.incrementViews();
entityManager.persist(postStatistics);

TopicStatistics announcementStatistics =
    new AnnouncementStatistics(announcement);
announcementStatistics.incrementViews();
entityManager.persist(announcementStatistics);
```

Hibernate will generate the following SQL statements:

```
INSERT INTO post (board_id, createdOn, owner, title, content, id )
VALUES (1, '2016-01-17 09:11:07.525', 'John Doe', 'Inheritance',
    'Best practices', 1)
```

```
INSERT INTO announcement (board_id, createdOn, owner, title, validUntil, id)
VALUES (1, '2016-01-17 09:11:07.525', 'John Doe', 'Release x.y.z.Final',
    '2016-02-17 09:11:07.529', 2)

INSERT INTO post_statistics (views, id) VALUES (1, 1)

INSERT INTO announcement_statistics (views, id) VALUES (1, 2)
```

Querying for statistics require specifying the actual `Topic` subclass statistics:

```
PostStatistics statistics = entityManager.createQuery(
    "select s from PostStatistics s join fetch s.topic t where t.id = :postId",
        PostStatistics.class)
    .setParameter("postId", postId)
    .getSingleResult();
```

This entity query generates a single SQL join statement:

```
SELECT
    s.id AS id1_4_0_ ,
    p.id AS id1_3_1_ ,
    s.views AS views2_4_0_ ,
    p.board_id AS board_id6_3_1_ ,
    p.createdOn AS created02_3_1_ ,
    p.owner AS owner3_3_1_ ,
    p.title AS title4_3_1_ ,
    p.content AS content5_3_1_
FROM post_statistics s
INNER JOIN post p ON s.id = p.id
WHERE p.id = 1
```

Polymorphic queries against the `Topic` class are not permitted and the `Board` entity cannot define a `@OneToMany Topic` collection either.

Performance considerations

Although polymorphic queries and associations are no longer permitted, the `@MappedSuperclass` yields very efficient read and write operations. Like single and table-per-class inheritance, write operations require a single insert statement and reading only needs to select from one table only.

Inheritance best practices

All the aforementioned inheritance mapping models require trading something in order to accommodate the impedance mismatch between the relational database system and the object-oriented Domain Model.

The default single table inheritance performs the best in terms of reading and writing data, but it forces the application developer to overcome the column nullability limitation. This strategy is useful when the database can provide support for trigger procedures and the number of subclasses is relatively small.

The join table is worth considering when the number of subclasses is higher and the data access layer doesn't require polymorphic queries. When the number of subclass tables is large, polymorphic queries will require many joins, and fetching such a result set will have an impact on application performance. This issue can be mitigated by restricting the result set (e.g. pagination), but that only applies to queries and not to `@OneToMany` or `@ManyToMany` associations. On the other hand, polymorphic `@ManyToOne` and `@OneToOne` associations are fine since, in spite of joining multiple tables, the result set can have at most one record only.

Table-per-class is the least effective when it comes to polymorphic queries or associations. If each subclass is stored in a separate database table, the `@MappedSuperclass` Domain Model inheritance is often a better alternative anyway.

Although a powerful concept, Domain Model inheritance should be used sparingly and only when the benefits supersede trade-offs.