JSimpleDB: Language-Driven Persistence for Java

Archie L. Cobbs
Techt, LLC
archie@techtllc.com

Abstract

Most software applications require durable persistence of data. From a programmer's point of view, persistence has its own set of inherent issues, e.g., how to manage schema changes, yet such issues are rarely addressed in the programming language itself. Instead, how we program for persistence has traditionally been driven from the storage technology side, resulting in incomplete and/or technology-specific support for managing those issues.

In Java, the mainstream solution for basic persistence is the Java Persistence API (JPA)[1]. While popular, it also measures poorly on how well it addresses many of these inherent issues. We identify several examples, and generalize them into criteria for evaluating how well any solution serves the programmer's persistence needs, in any language. We introduce JSimpleDB[3], a persistence layer for ordered key/value stores that, by integrating the data encoding, query, and indexing functions, provides a more complete, type-safe, and language-driven framework for managing persistence in Java, and addresses all of the issues we identify.

Keywords JSimpleDB, Java, Persistence, JPA

1. INTRODUCTION

Almost every non-trivial software application has the need to store data persistently, that is, on stable storage that lasts beyond the lifetime of a single execution, or even version, of the application. Since any piece of software, regardless of its complexity, is ultimately just a state machine, managing persistence is not a mere accessory to software development; it is truly fundamental, as nothing is more basic to a state machine than the management of its state.

Therefore it may seem surprising that in most programming languages, support for even basic durable persistence is an afterthought. By "basic" persistence, we mean the ability to store and retrieve structures and data types of the language across application restarts and upgrades. While virtually all programming languages provide the ability to read and write at the raw byte level, and to serialize and deserialize individual objects and values, few define a coherent framework for addressing the concerns particular to persistence programming: these include how queries are defined, indexing of data, schema version management, etc.

This is understandable. In general, programming languages provide support for data with "memory model" semantics, where all data lives in directly addressable memory and every access is atomic, instantaneous, and cannot fail. Persistence complicates this simple model with notions of non-determinism, transactions that can conflict or fail, communication delays and failures, and the requirement that all data be serializable into raw bytes from which it may be reconstituted later, possibly by a different version of the application having new and/or changed types and data structures. Moreover, there are so many different storage technologies with different features and trade-offs, it's unlikely there could ever be a single persistence framework at the language level that satisfied all needs.

As a result, persistence programming has been driven mainly from the storage technology side, resulting in a variety of special purpose API's and design patterns. Many persistence "solutions" are simply wrappers around whatever native API already exists for a particular storage technology. An example of this is Java Database Connectivity (JDBC)[4], which allows Java applications to interact with relational databases using Structured Query Language (SQL) statements and a straightforward API. Although easy to understand, JDBC leaves to the programmer the work of bridging the object-oriented and relational worlds, i.e., managing the "impedance mismatch"[2] caused by object-relational mapping (ORM). Hibernate[5], and its follow-on JPA, provide a more object-oriented API for relational database persistence, and are generally considered the current state of the art for basic Java persistence, at least when using relational databases.

As mature and widespread as JPA has become, bridging the object-relational divide still presents challenges in Java, a strongly-typed and relatively inflexible language. Simply put, JPA does not fit very naturally into it. This is not sur-

prising, because JPA's evolution was motivated primarily by the question: How can we most easily access the relational database functionality that already exists from Java? It was not attempting to address the reverse question: What is the best way to manage persistence in the Java language, and how might we implement that concretely using a relational database? JPA's design was constrained by the need for compatibility with the large variety of pre-existing relational database conventions, schemas and usage patterns. As a result, the simplicity and elegance of the Java programming language was compromised, at least with respect to persistence programming.

At the time JPA was developed, relational databases were ubiquitous, and so the assumption of their use represented a practical approach to persistence in Java. However, the past decade has seen an explosion of interest in non-relational ("NoSQL") database technology, and relational databases are now just one of many available options. Yet although "old" by modern standards, the Java programming language is still popular and widely used[6]. So the question of how to best manage persistence in the Java programming language is worth asking anew. More specifically: if we reject the requirement that we must expose functionality specific to relational databases and instead take a "language-first" approach, how could handling basic persistence in Java be improved?

In this paper we explore that question and provide some answers. We argue that from the programmer's point of view, JPA and similar solutions are not answering the right question: the real problem programmers have is how to manage persistence in a Java application, whereas the question they are answering is how to give programmatic access to a database. In this paper we show that the former question is broader and includes several additional issues that are non-trivial and inescapable, yet today remain largely unaddressed.

We start by taking a critical look at JPA. In the process we uncover several issues that are inherent to persistence programming in any language, and define criteria for evaluating how well a persistence solution addresses them. Finally we describe JSimpleDB[3], a new persistence layer for Java having the primary design goal of providing more complete, "Java-centric" answers to these inherent problems.

2. PROBLEMS WITH JPA

We examine problems with JPA as a way to motivate this exploration, but first, it's important to point out that JPA successfully achieves its goal, which is to provide a way for Java programmers to access the rich functionality of relational databases. Unfortunately, because exposing this functionality was of primary importance, preserving Java language elegance and simplicity, and providing a complete and coherent solution to all of the problems inherent in persistence, became unmet goals. Below we identify some of the resulting problems and consider how each problem might be better addressed in Java.

2.1 Configuration Complexity

Any persistence solution is going to require at least some configuration. JPA allows configuration via Java annotations, which gets high marks for natural language integration, but it requires 108 distinct Java annotations and `Enum` classes to do so. From this statistic alone, it is clear there is a huge amount of configuration complexity with JPA. Almost all of it derives from JPA's need to support the many different ways of storing data in relational tables.

How objects and fields are mapped onto the underlying database is technically not a programming language issue. However, JPA requires the programmer to understand and configure this mapping at the language level, because JPA must be flexible enough to support all the different ways of doing so. JPA adds additional configuration complexity when the `Criteria` API is used; it requires a compiler preprocessing step to generate the "meta-model" interfaces.

We define *configuration complexity* as the simple notion of how much work is involved in understanding and performing the configuration of persistence. In an ideal scenario, a persistence layer would manage how values and data structures in the language are mapped onto the underlying storage technology automatically, and not require the programmer to configure anything.

2.2 Query Language Concordance

Persistent data is not directly addressable in a programming language like normal, in-memory objects and data structures; some kind of query operation is required in order to retrieve it. A special language used to perform this operation is known as a "query language". Aside from plain SQL, JPA defines two additional query languages, the Java Persistence Query Language (JPQL)[12] and `Criteria` queries. JPQL parallels SQL at the Java class/field level, using class and field names instead of table and column names, and it allows stepping through reference fields and collections using joins. JPQL is transformed more or less directly into SQL. The `Criteria` API provides compile-time type safety, but requires additional preprocessing and is verbose and cumbersome to use; ultimately, it has roughly the same expressibility as JPQL. Even though written in Java, `Criteria` queries require the use of SQL-based specialized types and operations such as `From<Z, X>`, `MapJoin<Z, K, V>`, `Subquery<T>`, etc.

The first question here is: why do we need a "query language"? Java is a perfectly adequate language for expressing a wide variety of query-related concepts, including search, selection, projection, etc. Although SQL joins have no direct Java counterpart, joins are conceptually rooted in set theory, so it's reasonable to expect they too could be expressed as regular Java operations: an inner join as an intersection of two `Sets`; an outer join as a union of two `Sets`; etc.

Unfortunately JPA does not have the luxury to innovate here, as it must provide programmers complete query flexibility, with the ability to perform joins, sub-selects, and even query directly using SQL. However when those constraints are removed, many opportunities for simplification appear possible.

We define *query language concordance* as the degree with which the language used to perform persistence queries is consistent with the norms and conventions of the overall programming language. In this measure, more is better, and in the ideal case persistence queries would require no special knowledge from the programmer beyond a language's existing syntax, types, and data structures. Unfortunately, JPA is far from this ideal.

2.3 Query Performance Transparency

Java programmers are familiar with evaluating the performance of their own code. This often involves nothing more than a visual inspection backed by the knowledge of the performance of individual classes. For example, most Java programmers know that `Set.contains()` is "fast" but `List.contains()` is "slow".

Persistence query performance is a key part of overall application performance, but when persistence programming involves a special language for performing queries, that creates new challenges for understanding that performance. In fact, with JPA it's often completely impossible, when looking only at Java code that performs a query, to understand how well that query performs.

A simple example shows this. Consider the query `select Person as p where p.lastName = 'Smith'` that returns one row from a database containing one million rows. The database will either read one row (if the `lastName` column is indexed) or one million rows (if the `lastName` column is not indexed). Yet it's impossible to tell which will occur from looking at the query, which is identical in either case. This issue is inherited from SQL itself: query performance cannot be understood without understanding not only which columns are indexed, but also how the database builds query plans for each query. Programmers must in effect become relational database administrators (DBA's), well versed in interpreting the output of `EXPLAIN SELECT`, or else risk writing poorly performing software.

On top of the SQL and query plan layers, JPA adds an additional layer of obfuscation, namely the conversion from JPQL or `Criteria` queries into SQL. Compilation of JQPL queries often results in longer, more complex, and difficult to read machine-generated SQL queries, exacerbating the problem. By supporting JPQL and `Criteria` queries, JPA provides the programmer a more natural way to query than SQL than JDBC, but it also creates further distance between the programmer and the actual execution and therefore performance of the query.

We define *query performance transparency* as the level to which the performance implications of a query are obvious to a programmer inspecting it. Programmers already know how to evaluate the performance of normal code. Ideally, this knowledge would apply equally well when looking at persistence query code.

2.4 Schema Management

Software allows for unbounded experimentation and improvement because it is so easy to change. Continual updates and new releases are common. This has a downside when persistence is involved, because it creates a situation where data can be written by one version of an application but read by another. Since changes to software imply changes to the schema (i.e., structure and layout) of the associated persistent data, incompatibilities can occur. Unfortunately, if programming languages have treated persistence as an afterthought, persistence technologies have treated schema management as an afterthought's afterthought.

Relational databases are "schemaful" and therefore require explicit management of the database schema so that it agrees with what the application expects. "Schemaless" databases don't have this requirement; instead, they make the problem entirely the programmer's responsibility, which, if the goal is to simplify persistence programming, is going in exactly the wrong direction. Some JPA implementations provide basic tools to generate an initial schema (`CREATE TABLE`, etc), but not to automatically track, update, or verify it.

Ad hoc solutions are common in this space and over time schema tracking tools have evolved ([7], [8], [9]). This can be a tricky area, and mistakes can be disastrous, so any level of automation is welcome.

Several distinct issues arise with schema evolution. These are worth separate consideration.

2.4.1 Incremental Schema Evolution

A relational database can have only one schema at a time. Therefore, relational database schema migration tools typically verify and apply updates at application startup (or when the database is first accessed) in a "stop the world" operation during which no other data access is possible. Unfortunately, as storage has gotten cheaper and databases larger, the length of time the "world" must be stopped has also grown, prohibitively so for very large databases[10].

Clustered solutions are especially impacted. Not only must the entire cluster wait for the schema migration process to complete, but downtime is also required for the time it takes to upgrade every machine in the cluster. In other words, it's not possible to perform a rolling software upgrade in which one machine is upgraded at a time while the overall cluster is kept functional, because this would create conflicting schema expectations between the upgraded machines and the yet-to-be-upgraded machines.

Like most relational database persistence solutions, JPA provides no help here. Rolling upgrades are in fact possible, but they require manual development of a careful,

choreographed multi-phase upgrade and schema migration process, with intermediate schemas representing the union of the old and new[10]. This falls into the same unhelpful bucket as "schemaless" databases: the programmer is left to manually design and implement a solution to a delicate database management problem unrelated to the original purpose of the software.

Ideally, a persistence solution should support *incremental schema evolution*, where no "whole database" operations are ever required. It should allow schema information to be updated incrementally over time, it should make applying those incremental updates straightforward, and the process should be as automatic as possible.

Incremental schema evolution implies that the database can store some data under the old schema and some under the new, at least while an upgrade is in progress. Rolling upgrades with no downtime are then possible, for example by upgrading machines one at a time to a version of the software that can read both schema versions, and then "flipping the switch" to start writing data using the new schema version. Optionally, a background process can follow up by iterating through the database, migrating any remaining old schema data.

2.4.2 Structural vs. Semantic Schema Updates

JPA schema update instructions are typically written manually by the programmer, and are written at the SQL level, not the Java level. This creates obvious tedium and opportunities for errors. A fully automated schema migration solution would be ideal.

However, manual "fix-ups" will always be required in some situations. For example, consider a scenario where `lastName` and `firstName` fields are replaced by a consolidated `fullName` field. This change requires not only adding and removing columns (a *structural* update), but also copying the name information from the old columns to the new column (a *semantic* update). These are two distinct types of updates, and it's helpful to consider them separately.

Structural updates can easily be automated: it's not difficult to compare old and new schemas, and generate the corresponding changes in an automated way. Not so for semantic updates, which in general require information known only to the programmer. So, the best we can hope for is that it be easy and convenient for programmers to define those semantic updates–preferably using standard programming language conventions. This presents a challenge because, in general, semantic updates need access to data from both the old and new schemas, and data in the old schema may no longer be easily accessible in the normal way; in the previous example, the `getLastName()` and `getFirstName()` Java methods will no longer exist in the newer version of the code.

Summarizing the above discussion, an ideal persistence solution would provide automated *structural* schema updates, but also allow the programmer to define associated *semantic* schema updates in a convenient and natural way at the language level. The semantic updates would have access to all of the data in the old schema, even when that data is being removed by a structural update.

2.4.3 Schema Update Type Safety

Since JPA schema changes must be written manually, type safety violations can easily occur by accident. Part of the danger with such violations is that they are not always detected immediately. For example, many SQL databases perform implicit casting of integral columns to string values when evaluating expressions, meaning a forgotten schema update could go unnoticed in normal testing.

In the context of schema changes, one can consider *schema update type safety* as a property that guarantees, for example, that after an update an integer won't be misinterpreted as a string, or worse, a `Car` as a `Truck`, or cause instantiation of an `abstract` class. All of those examples can occur with JPA, the latter two when type discriminators are not carefully managed.

This problem can occur even when there are no structural schema changes. For example, suppose a Java `Enum` type is stored in an SQL column by ordinal, and then later a new value is added to the `Enum`'s identifier list. All subsequent values in the list will have their ordinal values incremented, and so every row containing any of those values has just been silently corrupted, even though no structural schema change has occurred and all data values remain valid. Since `Enum` type information is lost when converted into an SQL schema definition, JPA has no way to detect this scenario even if it wanted to.

If, after a schema change has occurred, a persistence solution makes it impossible to interpret a database value as the wrong type, or a different value for its type than before the update, then it guarantees *schema update type safety*. Obviously this is a highly desirable feature, especially in strongly typed languages like Java.

2.4.4 Schema Verification

Database schemas can change even when there are no schema changes in the software. When opening any transaction, it's theoretically possible that the database schema was changed incompatibly since the last access, perhaps by a different machine in the same cluster that was just upgraded. With JPA it is entirely up to the programmer to ensure that whatever data exists in the database is compatible with how the current revision of the software interprets it.

A closely related concept to schema update type safety is *schema verification*, which is the explicit verification that the schema in use by the database matches the schema in use by the software. Ideally, a persistence solution always performs schema verification. If so, then together with schema update type safety, this guarantees that type safety violations due to an incompatible schema can never occur in any situation.

2.5 Data Type Congruence

The Java language defines eight primitive types, each with precisely defined semantics. SQL supports a wide variety of different types, so obviously many SQL types cannot be cleanly mapped to Java types. The reverse is also true: there are Java values that are simply not representable in many relational databases.

This creates many odd and subtle corner cases. For example, SQL's `DECIMAL` type has bounds which are powers of ten, not two as in Java. MySQL's `DATETIME` type ranges from `1000-01-01 00:00:00` to `9999-12-31 23:59:59`, while Java's `java.util.Date` ranges from before the universe was created to over 200 billion years into the future, so attempting to persist the date of the fall of the Roman Empire (476 AD) will silently fail.

Many relational databases don't support the floating point values `NaN`, `-Infinity`, and `+Infinity`; these are often silently converted to zero, a very different value.

Java's `Enum` types have no straightforward representation in SQL. A Java `Enum` class is its own unique type, defined by an ordered list of distinct identifiers. With JPA, either the identifier or the list index (i.e., ordinal) is persisted, but all other type information is lost. The means the SQL column can store many other, invalid values; trying to read one results in a `RuntimeException` of course.

Regarding array types, JPA only supports one-dimensional `byte[]` and `char[]` arrays. JPA does not directly support the other primitive array types or multi-dimensional arrays.

In short, due to its SQL basis, JPA suffers from a lack of *data type congruence*. Even worse, values that don't translate often get silently coerced instead of triggering an error, requiring extra diligence on the part of the programmer to know and avoid all such "corner case" bugs.

A Java persistence solution should support Java types, and support them exactly, i.e., all of their possible values and nothing more. It certainly should not be possible to persist a value that is read back as a different value.

2.6 Transactional Constraint Validation

The ability to enforce data integrity constraints is fundamental with persistence programming. The persisted data is a data structure, and like any data structure it must be well-formed and satisfy certain invariants to make sense. Transactions are isolated (the "I" in ACID) so that these invariants remain satisfied from the outside world's point of view, while allowing them to be violated transiently within a transaction.

JPA supports enforcing integrity constraints via two types of validation: JSR 303[19] object/property constraints such as `@NotNull`, and SQL database constraints such as foreign key constraints and uniqueness constraints. Both are useful to ensure properly structured data and verify required invariants are preserved before a transaction can be committed.

However, these constraints are applied on JPA cache flush, or when the SQL statement corresponding to a change executes (such as `DELETE`), not necessarily at the end of the transaction. Therefore, the software may be prevented from moving the transaction through an invalid state, even if this is necessary and temporary (i.e., it was going to end up in a valid state at commit time). For JSR 303 constraints scoped to a single object this is typically not a problem, but for cascading JSR 303 constraints, foreign key constraints, and uniqueness constraints, all of which involve multiple objects, it is not uncommon. JPA's design, which is based on caching writes and sequentially flushing row-level updates to the database some time later, possibly in a different order, exacerbates this problem. Programmers are sometimes required to specially order their changes and intersperse strategic `flush()` statements to avoid transient foreign key constraint violations. This process requires either tedious trial and error, or a deep understanding of JPA's cache management behavior.

Validation constraints serve to guarantee that required invariants are preserved at the beginning and end of each transaction; what happens in the middle of the transaction is irrelevant to this goal. Therefore it makes sense for validation constraints to be verified only when a transaction commits, or when explicitly requested by the application. We call this desirable behavior *transactional constraint validation*.

There is another annoying issue with JPA uniqueness constraints in particular, which is their inflexibility with respect to special values. For example, different databases have conflicting–and unchangeable–rules regarding whether a unique column may contain multiple `null` values[11]. Similarly, a Java programmer may very reasonably want to exclude `NaN`, `-Infinity`, and/or `+Infinity` from the uniqueness constraint on a floating point column (assuming such values are supported). JPA provides no way to configure excluded values for uniqueness constraints.

2.7 First Class Offline Data Support

Open transactions are a relatively expensive resource: mutation state must be maintained, conflicts with other transactions must be evaluated and handled, a network connection may be allocated, etc. As a result software is typically optimized for relatively short transactions, and data is frequently copied out of the transaction for use later as needed. For example, a user interface may display results based on data that was copied out of a recent database query transaction, and then provide a "Refresh" button to trigger another query, another copy of data out of the new transaction, and another UI update showing that newly copied data. We might call such copied data *offline data*.

In JPA support for offline data is somewhat of an inadvertent side-effect of the fact that all transactional data access goes through the JPA cache: after closing a transaction, your available offline data is whatever objects you previously read into the cache and are still strongly referencing. JPA cache

granularity is (usually) per-object: all of an object's simple properties are brought into the cache when any object property is dereferenced because JPA reads the object's entire row. Collection properties may or may not be loaded, depending on how the object was queried and accessed.

Because it was designed for caching, not offline access, JPA's cache provides relatively crude control of what offline data is made available. In other words, the data you want to access (and therefore bring into the cache) *during* the transaction, and the data you want to keep a copy of offline *after* the transaction, are not necessarily the same, though obviously the latter is a subset of the former.

Perhaps more importantly, once the transaction closes, all of JPA's query capabilities are no longer available to query the offline data. You may have successfully copied all of the relevant `User` objects into the cache, but you can no longer query them by username–or in any other manner you haven't provisioned manually.

Because the use of offline data is so common, persistence solutions should provide *first class* support for it. This includes (a) precise control, separate from any caching, of which objects are copied out for offline access, and (b) the ability to query offline data using the same language and conventions as for querying "online" data, including index queries, joins, or whatever.

2.8 Data Maintainability

Databases are meant to be accessed by software applications, but sometimes it's necessary for humans to access them, either directly using a command line interface (CLI), or through simple scripting, etc. We can describe the ease of doing this as *data maintainability*.

JPA relies on the underlying SQL database to provide this functionality, and all relational databases provide comprehensive tools for doing so. However, these tools provide an SQL view of the data; all Java semantics have been stripped. A column storing `Enum` values by ordinal is just meaningless integers; it's impossible to query an object by its `hashCode()`; etc. Instead, the human must handle the reverse mapping of the data being manipulated back into the Java domain. An ideal Java persistence solution would provide tools for data maintenance that allow a human to access the data as Java data using regular Java types, methods, and expressions.

3. JSIMPLEDB

The issues described above motivated the design and implementation of JSimpleDB[3], a persistence solution for Java that takes a comprehensive, language-drive approach to managing persistence. JSimpleDB is a persistence *layer* that is compatible with any underlying database technology that can look like a simple sorted key/value store. Although started as a research project, JSimpleDB has since matured enough to become usable in a handful of small production commercial deployments. JSimpleDB is written in Java and available on GitHub[3] under the Apache license.

In this section we discuss how JSimpleDB addresses the specific issues described above to make persistence simpler and more natural for Java programmers.

3.1 Overview

JSimpleDB sits on top of a key/value store that is capable of storing arbitrary `byte[]` keys and values. The key/value store must be *sorted*, using an unsigned lexicographic ordering, so that it's possible to search for keys using (inclusive) lower or (exclusive) upper bounds, and to iterate over a range of keys in order. This API was chosen because it represents the lowest common denominator for many databases used for basic persistence today. This includes relational databases: any relational database can serve as a key/value store using a single table having `KEY` and `VALUE` columns of binary type with the `KEY` column indexed. JSimpleDB inherits whatever ACID semantics the key/value store provides.

JSimpleDB includes wrappers for several third party key/value databases, including relational databases (via JDBC), Oracle's BerkeleyDB[13], LevelDB[14], RocksDB[15], and FoundationDB[16]. JSimpleDB also includes several of its own key/value store implementations, including a novel distributed key/value database[17] based on the Raft[18] algorithm that inherits its linearizable semantics and fault tolerance. In theory, a balanced tree wrapper layer would allow JSimpleDB to function on top of unsorted key/value stores as well.

JSimpleDB performs no caching of keys and values; any caching needs are assumed to already be handled at the key/value store layer, or, a simple key/value cache wrapper layer can be added if desired. Contrast with JPA, where queried data is cached in memory twice: once by the relational database, and again by the JPA session.

3.1.1 Persistent Classes

Persistent model classes are annotated with `@JSimpleClass`. Persistent fields are defined by `abstract` Java bean property getter/setter methods; JSimpleDB generates concrete subclasses at runtime. For example:

```
@JSimpleClass
public abstract class User {

  public abstract String getEmail();
  public abstract void getEmail(String email);

  @JField(indexed = true, unique = true)
  public abstract String getUserame();
  public abstract void getUserame(String username);
}
```

The `@JField` annotation is used to override defaults. Since they are defined by `abstract` methods, fields can also be inherited into model classes from Java interfaces.

3.1.2 Storage and Object IDs

Each persistent Java model class and field has a unique internal integer ID called a storage ID, which (by default) is auto-generated by hashing the class or field name. JSimpleDB makes pervasive use of a compressed encoding for these and all other integral values, where values close to zero (-118 through 119) are encoded in a single byte, larger values are encoded in two bytes, etc. This encoding is both self-delimiting and order preserving. The auto-generated storage IDs encode into three bytes, but users can manually assign storage IDs values closer to zero to save a couple of bytes.

Each persistent object has a unique 64-bit object ID consisting of the object's model class' encoded storage ID as prefix, followed by a unique suffix. To avoid highly contended auto-increment counters that would limit scalability, new object IDs are created by choosing the suffix randomly. The likelihood of two transactions choosing the same suffix is very small, so conflicts are rare, until the number of instances of a type approaches the size of the suffix bit space. Therefore, when large numbers of objects of the same model class are expected (billions or more), storage IDs must be manually assigned values close to zero, leaving seven suffix bytes and allowing for up to 2^{56} instances.

3.1.3 Persistent Fields

JSimpleDB supports three categories of fields: *simple*, *complex* (i.e., collection), and a lock-free 64-bit *counter* type. Simple fields are those with simple types: simple types are "atomic" and have a self-delimiting `byte[]` encoding, by which they are totally ordered. This ordering is (usually) consistent with the type's `Comparable` implementation. All simple fields are indexable. Composite indexes on multiple simple fields are supported, and these support prefix views. All indexes are implemented within JSimpleDB itself.

The supported collection types are `List` (which performs like `ArrayList`), `NavigableSet`, and `NavigableMap`. Any simple type may serve as the element, key, or value type for a collection type. Collection fields may be indexed, in which case they may be queried by element, key, or value, and return all matching objects; for lists and map values, the corresponding list index and key (respectively) is also available.

The simple types include the Java primitive types, primitive wrapper types, `String`, `Enum` types, and other predefined types for `Date`, `UUID`, etc. *References* to persistent model classes are also simple types. User-defined simple types are supported, and are "first class" in the sense that they have the same capabilities as any other simple type: they can serve as a collection element type, be indexed, etc. Single and multidimensional arrays of any simple type (except reference) are also simple types; arrays are passed by value and sort lexicographically based on the element ordering.

3.1.4 Data Mapping

The mapping of object and field data into the key/value store is straightforward. Object meta-data, such as an object's schema version, is stored under the object ID key. Letting "+" represent concatenation, simple fields are stored as object ID + field ID $\Rightarrow$ value. Fields equal to their default values are not stored. Lists are stored as object ID + field ID + index $\Rightarrow$ value; sets are stored as object ID + field ID + element $\Rightarrow$ (empty); maps are stored as object ID + field ID + key $\Rightarrow$ value.

3.1.5 Index Mapping

Indexes are represented by inverted keys and empty values: for simple fields, field ID + value + object ID; for lists, field ID + value + object ID + index; for sets, field ID + value + object ID; for map keys, field ID + key + object ID; and for map values, field ID + value + object ID + key. Therefore indexing a field exactly doubles that field's storage cost. JSimpleDB keeps index information up-to-date automatically and atomically as fields are mutated.

3.1.6 Reference Fields

Reference fields are always indexed (SQL equivalent: foreign keys require indexes). Among other benefits, this allows JSimpleDB to support configurable behavior with respect to references and deletion. Reference fields can be configured so that when a referring object is deleted, the referred-to object is also deleted ("delete cascade" in JPA terminology). Conversely, the behavior when a referred-to object is deleted is configurable (analogous to SQL's `ON DELETE`), one of: delete the referring object (inverse delete cascade), disallow (throw exception), nullify the reference, or do nothing.

One subtle difference between JPA and JSimpleDB involves nullifying a reference from a collection: in JPA the reference is simply set to null, whereas in JSimpleDB the element is *removed* from the collection; in particular, this is impossible using `ON DELETE SET NULL` with a `List`.

JSimpleDB also provides lifecycle annotations `@OnCreate` and `@OnDelete`. The annotations, and forward and reverse delete cascades, may result in loops and reentrancy; JSimpleDB properly handles them.

3.1.7 API Layering

JSimpleDB is built in three well-defined, independent layers. At the bottom layer is the key/value store API. On top of that is the core layer, which handles most of the work relating to data encoding/decoding, query views, indexing, and schema management, but makes no assumptions about how the data is modeled, and is not explicitly object-oriented. At the top layer, concrete subclasses of the user-supplied Java model classes are generated and connected to the core API layer, including the automated derivation of core API schema information, and core API object IDs are wrapped by actual model class instances. All Java model class instances are

scoped to a specific transaction. JSimpleDB provides convenience methods for copying an arbitrary graph of objects between transactions if needed.

Having described the basic mechanics, in the following sections we describe how JSimpleDB addresses the issues and criteria described in Section 2 to provide a more complete and language-driven framework for managing persistence in Java.

3.2 Configuration Complexity

Whereas JPA defines 108 annotations and `Enums` for configuration, JSimpleDB defines 14, including those related to features not supported in JPA; only one is required (`@JSimpleClass`). The mapping from objects and fields to key/value entries is fixed and therefore requires no configuration. As a result, JSimpleDB has low *configuration complexity*. Even so, JSimpleDB provides some new flexibility. For example, an index can be defined on a field declared in a Java interface, and the resulting index query will return instances of any implementing Java class, regardless of its place in the class hierarchy.

JSimpleDB does support some new configuration related to its unique features (described below). For example, validation can be set to automatic, manual, or disabled; because it proactively tracks schema versions, the current schema version number must be specified; and transactions may be configured whether or not to allow recording new schema versions.

3.3 Query Language Concordance

JSimpleDB does not have a query language. Instead, it provides three operations to initiate a search: query for a specific object by object ID, query for all objects of a given Java type, or query an index. After that, the query proceeds using conventional logic on Java collection types and model objects. The net effect is a high degree of *query language concordance*.

A query for objects of type `T` returns a `NavigableSet<T>`; `T` can be any Java type. The objects in the set are ordered by object ID. An index query on a field of type `F` in `T` returns a `NavigableMap<F, NavigableSet<T>>`, i.e., a mapping from each field value to the set of objects containing that value in the field. The map is sorted by field value, and each set is sorted by object ID. Referring back to Section 3.1.5, one can see that these collections are direct views of the underlying key/value store: all returned collections are "live" and stay up-to-date with respect to changes.

Here are two examples of `User` methods that perform index queries. This assumes the `lastName` and `regDate` fields are indexed:

```
// Find users by last name
public static NavigableSet<User>
 getByLastName(String lastName) {

  // Get a view of the User.lastName index
  NavigableMap<Date, NavigableSet<User>> lastNameMap
   = JTransaction.getCurrent()
    .queryIndex(User.class, "lastName", String.class)
    .asMap();

  // Get users with the specified last name
  return lastNameMap.get(lastName);
}

// Get users registered in the past day, sorted
// in reverse order of their registration time
public static Stream<User> getRegisteredToday() {

  // Get a view of the User.regDate index
  NavigableMap<Date, NavigableSet<User>> byRegDate
   = JTransaction.getCurrent()
    .queryIndex(User.class, "regDate", Date.class)
    .asMap();

  // Get all users registered in the past 24 hrs
  Date cutoff = new Date(
    System.currentTimeMillis() - 86400000);
  return byRegDate
    .tailMap(cutoff)
    .descendingMap()
    .stream()
    .flatMap(NavigableSet::stream);
}
```

Set operations such as intersection and union are used to effect the equivalent of database joins. JSimpleDB provides efficient implementations of set union, intersection, and difference that preserve element ordering and the `NavigableSet` interface when the orderings are the same. Here's an example of a query by first and last name, using individual indexes on the `firstName` and `lastName` fields:

```
// Find users by first and last name
public static NavigableSet<User> getByNames(
  String firstName, String lastName) {

  // Query the User.firstName index
  NavigableSet<User> firsts
   = JTransaction.getCurrent()
    .queryIndex(User.class, "firstName", String.class)
    .asMap()
    .get(firstName);
  if (firsts == null)
     return Collections.<User>emptyNavigableSet();

  // Query the User.lastName index
  NavigableSet<User> lasts
   = JTransaction.getCurrent()
    .queryIndex(User.class, "lastName", String.class)
    .asMap()
    .get(lastName);
  if (lasts == null)
     return Collections.<User>emptyNavigableSet();
```

```
  // Return intersection of the two sets
  return NavigableSets.intersection(firsts, lasts);
}
```

`NavigableSets.intersection()` iterates efficiently: a complete iteration requires $O(N * M)$ key/value iteration steps, where N is the number of sets being intersected, and M is the size of the *smallest* set.

3.4 Query Performance Transparency

Because queries are written in normal Java, JSimpleDB provides high *query performance transparency*. All of the `NavigableSets` and `NavigableMaps` returned by JSimpleDB are based on the underlying sorted key/value store and provide the efficient performance expected of their type. Therefore, understanding the performance of a query is straightforward from looking at the code.

Considering the previous JPQL example `select Person as p where p.lastName = 'Smith'`, where the performance was not obvious from the query, because it depended on whether `lastName` was indexed. In JSimpleDB the code to perform this search will *necessarily* look very different depending on whether the `lastName` field is indexed: if so, it's linear code; if not, there will inevitably be some scanning of every `Person` in the database, making the inefficiency of the query obvious.

However, there is a cost for this transparency. JSimpleDB effectively shifts the responsibility for creating the "query plan", albeit one written in Java, onto the programmer. This is not a trivial change: all of SQL's conveniences, including aggregate functions like `AVG()`, `GROUP BY`, etc., must be implemented by the programmer in Java. Fortunately, this requirement dovetails nicely with Java's new lambda syntax and `Stream` API, with methods like `IntStream.average()`, `Collectors.groupingBy()`, etc., to make this job relatively straightforward.

In exchange for this new responsibility, query logic and performance implications are no longer hidden. Moreover, once the "query plan" is written in Java, it naturally leads the programmer to reason about it and optimize further if appropriate, for example, by deciding to index a field, or adding a new field to incrementally maintain derived data such as an average, in effect creating a custom "index" (see `@OnChange` below for how JSimpleDB facilitates this).

In some cases, it is the inability to write an efficient query in JSimpleDB that reveals mistaken assumptions underlying performance problems. For example, developers sometimes assume that if the `lastName` and `firstName` columns are both indexed, then an ordered query like `select Person as p order by lastName, firstName` will be efficient. In actuality, that's only true when there is a *composite* index on both the `lastName` and `firstName` fields (in that order); otherwise, the sorting on `firstName` has to be performed in memory or temporary tables. JSimpleDB forces this reality to the surface.

3.5 Schema Evolution Support

JSimpleDB provides explicit *schema evolution support* and guarantees Java type safety at all times.

Each object has a schema version number stored in its meta-data. In addition, for each schema version in use by any object in the database, the associated schema is itself stored in a special meta-data region of the key/value store. This recorded schema contains sufficient information for JSimpleDB to decode the fields and values of any object of that version, even if the original Java model classes are no longer available. Although schema versions are identified by numeric values, JSimpleDB imposes no constraints on how schema versions are numbered.

3.5.1 Incremental Schema Evolution

JSimpleDB provides *incremental schema evolution*: objects are migrated "on demand" when the schema version in use by the transaction does not match the object's schema version. This happens when the object is first accessed after an upgrade, or when explicitly requested. No whole database operations are ever required. The migration includes (de)indexing if necessary: a field may be indexed in one schema version but not another, so when an object transitions between the two schema versions, it gets automatically added to or removed from any indexes accordingly. Indexes never need be built from scratch.

JSimpleDB poses one primary restriction on how schemas may evolve: a field with the same storage ID must have the same type in every schema version in which it is *concurrently* used in a database. This requirement is necessary because an index may contain objects with different schema versions, but all entries in the index must have a consistent encoding.

As a consequence, since field storage IDs are (by default) derived from the field's name, in cases where a field's type, but not its name, has changed, the `@JField` annotation must be added to change the (internal) name or storage ID. This is an uncommon situation; the most likely cause in our experience is when an `Enum` field's identifier list changes (effectively changing the field's type); see below for an example.

3.5.2 Schema Verification

JSimpleDB performs *schema verification* at the start of every transaction. Each transaction has an associated schema derived from the Java model classes being used to access it. When a transaction is opened, JSimpleDB verifies that the transaction's schema is compatible with the schema previously recorded in the database under that same version number. If a transaction's schema version is not yet recorded in the database, and the transaction is so configured, it will be added automatically. Through this mechanism JSimpleDB tracks and verifies the schema in use.

Schema information is stored in a range of the key space reserved for meta-data. A recorded schema version occupies

a single key/value pair. Because this key/value pair will be read at the start of every transaction, scalability is a concern. In order for a JSimpleDB to be scalable, the underlying key/value store must not degrade when the same key is read by every transaction.

3.5.3 Structural vs. Semantic Schema Updates

Structural schema updates are entirely automated by JSimpleDB. This is possible thanks to the explicit recording of schema information in the database. As described above, this happens in an incremental fashion as objects are encountered after an upgrade.

Semantic schema updates are implemented using instance methods annotated with `@OnVersionChange`. Immediately after an object is structurally updated, `@OnVersionChange` method(s) are invoked. The values of all fields, including any removed fields, from the previous version of the object are provided to each method.

Considering the previous example where `lastName` and `firstName` fields are consolidated into a single `fullName` field, here's how the original model class might look:

```
// Schema version #1
@JSimpleClass
public abstract class User {

  public abstract String getLastName();
  public abstract void setLastName(String lastName);

  public abstract String getFirstName();
  public abstract void setFirstName(String firstName);
}
```

and here's how the new model class, including the semantic schema update, might look:

```
// Schema version #2
@JSimpleClass
public abstract class User {

  public abstract String getFullName();
  public abstract void setFullName(String fullName);

  // Semantic update for version 1 -> 2
  @OnVersionChange(oldVersion = 1, newVersion = 2)
  private void update(Map<String, Object> prev) {
    String lastName = (String)prev.get("lastName");
    String firstName = (String)prev.get("firstName");
    this.setFullName(lastName + ", " + firstName);
  }
}
```

A nice artifact of this design is that semantic update code remains private to the affected Java class. Other classes always see a fully upgraded object.

3.5.4 Schema Update Type Safety

JSimpleDB guarantees *schema update type safety*. This creates some interesting corner cases when old schema classes or field values have Java types that simply don't exist in the new schema version.

For example, any change to the identifier list of an `Enum` effectively changes its type, and therefore JSimpleDB requires a schema change. As mentioned above, this means a different name or storage ID must be used by fields having that `Enum` type. In addition, the old field's values, which are instances of the old `Enum` type, cannot be represented as such in the new code, which only defines the new, incompatible `Enum` type. Instead, JSimpleDB supplies old `Enum` field values as `EnumValue` objects, which are just an `int`, `String` pair. Here's an example showing an original model class:

```
// Schema version #1
@JSimpleClass
public abstract class Vehicle {
  public enum Color {
    RED,
    LIGHT_GREEN,
    DARK_GREEN,
    BLUE
  }

  public abstract Color getColor();
  public abstract void setColor(Color color);
}
```

and the new model class with the renamed field and semantic schema update:

```
// Schema version #2
@JSimpleClass
public abstract class Vehicle {
  public enum Color {
    RED,
    GREEN,   // was LIGHT_GREEN or DARK_GREEN
    BLUE
  }

  @JField(name = "color2")
  public abstract Color getColor();
  public abstract void setColor(Color color);

  // Semantic update for version 1 -> 2
  @OnVersionChange(oldVersion = 1, newVersion = 2)
  private void update(Map<String, Object> prev) {
    EnumValue colorName
      = ((EnumValue)prev.get("color")).getName();
    if (colorName.endsWith("_GREEN"))
      colorName = "GREEN";
    this.setColor(Color.valueOf(colorName));
  }
}
```

The other case of an old schema field value that cannot be represented happens with references to a Java model class that has been removed. These references appear to `@OnVersionChange` methods as instances of an opaque

type `UntypedJObject`. The old object's fields can be accessed using JSimpleDB introspection methods.

3.6 Data Type Congruence

JSimpleDB persists Java types, not SQL types, and therefore has high *data type congruence*. JSimpleDB supports values like `NaN` and `+Infinity`, and `Date` has its full range. Primitive types are different from primitive wrapper types: only the latter can be null. As a side-benefit, since primitive types are simple, they can be used for collection types; wrapper types are still used for generic type parameters, so the net effect is that null values are disallowed.

One interesting issue occurs with nullable `Comparable` types. `Comparable` types are not required to sort `null`, but all simple type values, including `null`, are ordered by JSimpleDB. By convention, `null` values sort last.

3.7 Transactional Constraint Validation

JSimpleDB supports JSR 303 validation, but it also gives the programmer control over when validation occurs. Internally JSimpleDB transactions maintain a set of IDs of objects with pending validation. Normally this set is processed only at the end of a transaction, thereby providing *transactional constraint validation*. Validation can also be manually triggered at any time, or disabled altogether on a per-transaction basis.

JSimpleDB automatically registers an object for validation whenever any JSR 303 annotated field is modified. Objects can be registered manually as well. This is useful in combination with the `@OnValidate` annotation, which denotes an instance method to be invoked during validation.

`@OnValidate` is useful for checking constraints that apply across multiple fields and/or objects. For these constraints, a common problem with JPA is determining where to put the code that detects any change that could invalidate the constraint (and therefore should trigger revalidation). JSimpleDB adds support for this and related use cases with the `@OnChange` annotation, which denotes a method that should be invoked whenever any of the fields named in the annotation change. A novel feature is that these fields may exist in the same object, or in any other objects reachable through an arbitrary path of references.

To demonstrate this use, consider a validation constraint that ensures a manager's salary exceeds the average of his direct reports. This constraint cannot be automatically checked in JPA using JSR 303 annotations, and a manual implementation would be awkward. This is also an example of a constraint that could easily be violated transiently during a transaction (e.g., bulk salary update), even though it is ultimately satisfied at commit time, and so transactional validation is important here as well.

In JSimpleDB such a constraint might look like this:

```
// My salary
public abstract int getSalary();
public abstract void setSalary(int salary);

// My direct reports
public abstract Set<Employee> getReports();

// Invoked on my or any report's salary change
@OnChange({ "salary", "reports.element.salary" })
private void onReportSalaryChange() {
  this.revalidate(); // enqueue for validation
}

// Validate my salary vs. my direct reports.
// Invoked at end of transaction if my salary,
// or any of my reports' salaries, has changed
@OnValidate
private void checkSalaryInvariant() {
  int avgReportSalary = this.getReports()
    .stream()
    .mapToInt(Employee::getSalary)
    .average()
    .orElse(0);
  if (avgReportSalary > this.getSalary())
    throw new ValidationException("need a raise!");
}
```

Even though this constraint involves non-local objects, the implementation is local, automatic and efficient: the manager object is registered for validation (a fast operation), but not actually validated, any time the manager's or any report's salary changes, and then the actual constraint check is performed only once, at commit time, after all of the salary adjustments have been completed. The code to handle (a) detecting the relevant changes and (b) actually checking the constraint are together in one logical place in the code.

The implementation of `@OnChange` involves inverting object references, i.e., following a "reference path" of object references in the reverse direction starting from the object being modified; this is possible because reference fields are always indexed. JSimpleDB also makes this reference path inversion capability, something not available in normal Java programming, available for general purpose use.

JSimpleDB also includes support for excluded values in uniqueness constraints, e.g.:

```
@JSimpleClass
public abstract class Item {

  @JField(unique = true,
    uniqueExclude = { "-Infinity", "+Infinity" })
  public abstract float getRank();
  public abstract void setRank(float rank);
}
```

3.8 First Class Offline Data Support

JSimpleDB provides *first-class offline data support*. The key mechanism for this is the *snapshot transaction*. A snapshot transaction is simply a transaction opened on an initially empty, in-memory key/value store, and configured to use the same schema as a regular transaction. Any Java query

that works on normal transactions works on snapshot transactions. This includes index queries.

JSimpleDB provides convenience methods for copying objects between regular and snapshot transactions (or between any two compatible transactions). The object copy operation is performed directly on the key/value store for efficiency. Programmers can precisely control which objects are copied. For key/values stores that support efficient snapshot views through multiversion concurrency control (e.g., LevelDB), JSimpleDB also allows creation of snapshot transactions based on read-only snapshots, eliminating the need for copying entirely.

Snapshot transactions are also useful in their own right. Persistence necessarily requires serializing and deserializing state, and snapshot transactions make this capability useful outside of the persistence context. For example, snapshot transactions are useful as containers for sending serialized messages over a network. They are just sets of key/value pairs, so encoding them is trivial.

Network communication and persistence are surprisingly similar problem domains. In both cases, (a) application objects and data types must be (de)serialized; (b) the application doing the writing and the application doing the reading might have different versions (e.g., during a rolling upgrade), so some form of schema management is needed; and (c) the recipient needs to query the received data in some way, so some kind of query support is needed. Snapshot transactions provide a solution to all three issues.

3.9 Data Maintainability

JSimpleDB includes a CLI for accessing a database via the command line in order to provide *data maintainability*. Three modes are supported: raw key/value, core API mode, and Java mode. The CLI supports integrating custom commands and functions, and can be embedded into an application.

The key/value mode provides direct access to the key/value store and is normally only used in special circumstances, for example when backing up or exporting key/value data for import elsewhere.

The core API mode gives a structural, rather than object-oriented, view of the data. It does not require the original Java model classes. All field data is accessible using its usual Java type, with two exceptions: reference field values are seen as object IDs, and `Enum` field values are seen as `EnumValue` identifier/ordinal pairs. Data can also be imported/exported as XML at either the key/value layer or the core API layer.

The Java mode requires the associated Java model classes and provides a fully object-oriented view of the data. In this mode, humans view and interact with persistent data on the same level as the application. The CLI includes a parser for Java expressions, allowing arbitrary Java-based logic to be applied on the fly. For example, it allows filtering objects by `x.hashCode() % 23 == 7` or any other Java expression.

4. RELATED WORK

The large majority of libraries for basic persistence in Java are targeted at relational databases. Several of these implement JPA, including Hibernate[5], DataNucleus[26], OpenJPA[23], and EclipseLink[22]. Other object-relational mapping libraries include Torque[32], Cayenne[34], Jaxor[30], pBeans[31], SimpleORM[28], and JULP[27]. Many projects were started before JPA evolved to become the mainstream ORM solution.

JPA implementations that are not restricted to relational databases include DataNucleus[26] and ObjectDB[20]. DataNucleus supports a wide variety of back-end databases. ObjectDB has its own non-relational database format and implements automatic structural schema migration; semantic update support is limited to convertible types and renaming of classes and fields based on an XML file.

The Java Data Objects[21] (JDO) specification pre-dates JPA and is more general, being independent of any specific storage technology. It is broadly similar to JPA and also includes a query language (JDOQL). JDO implementations include DataNucleus[26], ObjectDB[20], Castor[33], TriActive[29], Speedo[25], and XORM[24].

Although JDO is storage agnostic, like JPA it is designed with the assumption that fundamental tasks like the structuring and encoding of data, the execution of search queries, and the maintenance of index information are performed by the back-end database. In other words, the implied job of the persistence layer is to pass existing database functionality through to the Java programmer in the most painless way possible. Because databases focus on the storage of data rather than any issues inherent to managing persistence in a programming language, it's not surprising that JPA and JDO don't address many of these issues, e.g., query performance transparency, schema evolution management, and offline data.

5. CONCLUSION

Persistence is a fundamental concern in programming. Unfortunately, few programming languages provide a coherent framework for addressing the many issues inherent to persistence programming. Persistence programming has instead been driven instead from storage side. As a result, programming languages like Java suffer from solutions that compromise the simplicity, robustness, and type-safety of the language, are specific to one type of storage technology, and for the most part don't address those inherent issues. Java and its mainstream persistence solution JPA provide a vivid example.

JSimpleDB takes a language-driven approach, instead asking: What is the most natural way to address persistence in the Java language, and what is the minimum functionality required from the storage back-end to get there? JSimpleDB's answer, an ordered key/value store, is simple enough to be supported by many different back-end tech-

nologies, but also sufficiently powerful to allow JSimpleDB to provide data encoding, query functionality, and index maintenance itself. This high level of integration in turn makes it possible for JSimpleDB to better address the issues inherent to persistence programming.

By exploring some of the problems and complications with JPA, we have identified several of these issues, and defined corresponding measures of how well any persistence solution addresses them:

Configuration Complexity
How hard is it to configure? Are we forced to (ab)use the programming language to address what are really database configuration issues?

Query Language Concordance
Does the code that performs queries look like normal code in the language, or do we have to learn a new "query language"?

Query Performance Transparency
Is the performance of a query obvious from looking at the code that performs it?

Data Type Congruence
Are all field values supported? Do we always read back the same values we write?

Transactional Constraint Validation
Does validation only occur at the end of the transaction? Is it easy and convenient to define arbitrary custom validation constraints, even those that span multiple objects/records?

First Class Offline Data Support
Can it be precisely defined which data is actually copied out of a transaction? Does offline data have all the rights and privileges of "online" (i.e., transactional) data? Does this include index queries and a framework for handling schema differences?

Data Maintainability
Can data maintenance tasks be performed using the normal types and values of the original programming language? Are there convenient tools for manual and scripted use?

We define these issues specifically related to schema management:

Incremental Schema Evolution
Can multiple schemas exist at the same time in the database, to support rolling upgrades? Can data be migrated incrementally, i.e., without stopping the world? Are any whole database operations ever required?

Structural Schema Updates
Are structural schema updates performed automatically?

Semantic Schema Updates
Is there a convenient way to specify semantic schema updates, preferably at the language level, not the database level? Do semantic updates have access to both the old and the new values?

Schema Update Type Safety
Is type safety and data type congruence guaranteed across arbitrary schema migrations?

Schema Verification
Is the schema assumed by the code cross-checked against the schema actually present in the database?

We show that it's possible to address these issues in the Java language in a coherent, natural way, and give affirmative answers to all of the above questions. The database back-end is left to do what it does best–store and retrieve data–while the programmer is given an improved solution to the actual problem at hand: managing persistence.

Acknowledgments

Thanks to Marco Chiesa, Mark Thomas, David Templin, Malcolm Davis, and Seth Hammock for providing helpful review and comments.

References

[1] The Java Persistence API. https://en.wikipedia.org/wiki/Java_Persistence_API

[2] Object-relational impedance mismatch. https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch

[3] JSimpleDB. https://github.com/archiecobbs/jsimpledb/

[4] Java Database Connectivity. https://en.wikipedia.org/wiki/Java_Database_Connectivity

[5] Hibernate. http://hibernate.org/

[6] January Headline: Java is TIOBE's Programming Language of 2015! http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[7] Flyway. http://flywaydb.org/

[8] Liquibase. http://liquibase.org/

[9] MyBatis. https://github.com/mybatis/

[10] Online Schema Change for MySQL (MySQL At Facebook) http://goo.gl/uUulWX

[11] Does MySQL ignore null values on unique constraints? http://goo.gl/8W1q12

[12] The Java Persistence Query Language. https://en.wikipedia.org/wiki/Java_Persistence_Query_Language

[13] Oracle Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

[14] LevelDB. http://leveldb.org/

[15] RocksDB. http://rocksdb.org/

[16] FoundationDB. https://en.wikipedia.org/wiki/FoundationDB

[17] New key/value database based on Raft. https://groups.google.com/forum/#!topic/raft-dev/uYlZzVCIJTs

[18] The Raft Consensus Algorithm. https://raft.github.io/

[19] Bean Validation. https://en.wikipedia.org/wiki/Bean_Validation

[20] ObjectDB. http://www.objectdb.com/

[21] Java Data Objects. https://en.wikipedia.org/wiki/Java_Data_Objects

[22] EclipseLink. http://www.eclipse.org/eclipselink/

[23] OpenJPA http://openjpa.apache.org/

[24] XORM http://xorm.sourceforge.net/

[25] Speedo. http://speedo.ow2.org/

[26] DataNucleus. http://datanucleus.org/

[27] Java Ultra-Lite Persistence. http://julp.sourceforge.net/

[28] SimpleORM. http://www.simpleorm.org/

[29] TriActive JDO. http://tjdo.sourceforge.net/

[30] Jaxor. https://sourceforge.net/projects/jaxor/

[31] pBeans Persistence Layer. http://pbeans.sourceforge.net/

[32] Apache Torque. https://db.apache.org/torque/torque-4.0/index.html

[33] Castor. http://castor-data-binding.github.io/castor/

[34] Apache Cayenne. http://cayenne.apache.org/

