CREATING HIGH PERFORMANCE BIG DATA APPLICATIONS WITH THE JAVA PERSISTENCE API

By Dirk Bartels & Matthew Barker, Versant Corporation

Sponsored by Versant Corporation

EXECUTIVE SUMMARY

THE CASE FOR BIG DATA APPLICATIONS

Data management has reached another inflection point. A new breed of applications is pushing traditional, mostly relational database management solutions (RDBMS) beyond their limits, driven by an ever-growing mountain of data, ubiquitous access to information, the consumerism of IT, cloud computing, mobile computing, and, last but not least, the hunger for smarter applications.

These volumes of data are outgrowing advances in hardware design, while simultaneously analytics applications are pushing for faster, near-real-time capabilities to deliver on the promise of leveraging Big Data's benefits. It becomes clear that data management software must evolve rapidly to address these changing requirements. But instead of following the "one size fits all" model of the past, a more prudent approach to this new era of data management is to consider more closely the specific requirements of each application implementation and use the information management system that best meets those demands. For example, it is critical to estimate the scale of the data problem and the level of complexity (or richness) of the domain (data) model, among other important aspects, in order to pick the right data management technology(ies) to get the job done.

As we design these new types of applications, we must consider many issues that are closely related to data management: velocity, volume, variety, scale and concurrency, the richness of the domain models, and, critically, the value we want to derive from the data. But perhaps most importantly, these new requirements result in immensely dynamic demands, and designing applications to meet them also means planning for the unexpected. Applications built today must be able to easily add data sources, and integrate with other IT systems.

Programming standards play a vital role in enabling this flexibility. Standards establish a stable and reliable baseline on which the market can build compliant tools and solutions. They allow developers to pick and choose the right vendor or tool for the project at hand without requiring the extra costs to learn new skills or techniques.

VERSANT

THE NEED FOR PROGRAMMING STANDARDS

A programming standard helps to increase developer productivity and software quality. An established and widely adopted standard allows companies and developers to invest in that standard through training, education, and standard-compliant tools. For example, SQL is a notable standard, and after its breakthrough nearly thirty years ago a huge industry grew around it, creating a de facto database standard for today's traditional enterprise applications.

Software standards are not typically established by committees, but rather by the market itself. There are two dominant ways the market has created standards:

1. By sheer volume, companies like Microsoft (with Windows) and Apple (with iOS) have such dominance in their particular segment that their technologies have become de facto standards. The sheer number of supporters of the technology - application developers and consumers buying into the platform - creates the critical mass for a flourishing ecosystem that ensures the technology's longevity.

2. Other standards, such as SQL, and more recently Hadoop and MapReduce, are created by having support from a number of vendors, therefore, reaching critical mass as an open system rather than a closed one.

The Java Persistence API (JPA), much like SQL, is an open data management standard. Unlike the proprietary APIs being created by emerging NoSQL solution vendors, JPA is already part of a large ecosystem, and training, education and tool support are widely available. NoSQL products like Couch DB, Mongo DB, and Cassandra are facing an uphill battle to reach mainstream adoption and be attractive enough for third parties and developers to create an ecosystem for these systems. Their market is limited to early adopters, which tend to ignore the need for standards.

And while NoSQL is a new, complementary-to-SQL method for data management, it is desirable to extend JPA to also become the NoSQL standard for Java. Why rely on a proprietary and less mature API if similar functionality can be had with an established standard?

VERSANT

INTRODUCTION TO JPA

The Java programming language and its platforms, such as J2SE and J2EE, remain the undisputed leading platforms for enterprise application development. A large eco-system of tools, broad operating system support, and legions of skilled developers makes Java an excellent choice for enterprise application development.

With the growing size and richness of enterprise applications, it is literally unmanageable to consider a large scale development project in Java without using a persistence programming framework to help manage the complexity of the underlying data management system. Today's applications often require thousands of persistent models (or classes). Programming and maintaining a class separately from a database schema (for RDBMSs, tables, columns, indexes, and possibly further optimizations, such as stored procedures) without a higher level abstraction is no longer commercially feasible. Simply put, application models have become too rich to be managed at the relational level, and require a persistence framework to simplify development and implementation of the database layer.

Part of the broader Java platform is the so-called Java Persistence API (JPA), an established de facto standard for the persistence layer between application and database. JPA has been widely adopted as the programming interface for RDBMSs, using implementations such as EclipseLink, Hibernate JPA, Open JPA, and others. Compared to the more basic JDBC[1] programming API, JPA offers a higher level of abstraction, and hides and encapsulates most of the tedious, error-prone code that would be necessary using JDBC to "map" Java classes to a relational database schema. The "assembly" and "disassembly" of objects from and to the database is mostly automated. This allows the developer to stay focused on its application domain and to significantly improve productivity.

The JPA specification is maintained in the Java Community Process[2] (JCP). It originated from the open source tool Hibernate[3], one of the first and most successful object-to-relational mapping (ORM) technologies for Java, the Java Enterprise Beans (EJB) 3.0 persistence specification, and, most notably, the Java Data

---

1 JDBC is a Java-based data access technology (Java Standard Edition platform) from Sun Microsystems, Inc.. It is not an acronym as it is unofficially referred to as Java Database Connectivity
2 The JCP is the mechanism for developing standard technical specifications for Java technology. See jcp.org
3 Hibernate is an object-relational mapping (ORM) library for the Java language

VERSANT

Objects[4] (JDO) standard. The JDO specification was created by several object database experts who originally coined the notion of object persistence as a tight integration between the application and the database via a single, unified type system. With roots so close to the fundamentals of database creation and standardization, JPA represents an amalgam of the following philosophies to integrate database access into Java:

The "pragmatic" Hibernate, born out of the necessity to overcome the impedance mismatch between an object oriented application and a relational database by mapping objects into a relational database, which quickly gained large developer support.

The "dogmatic" EJB, born out of a specification to construct a true object system, which in the end proved to be too "heavy" and complicated to implement. Developers abandoned EJB persistence because they wanted to use so-called Plain Old Java Objects (POJOs).

The "pure objects" JDO, the most elegant and efficient specification, creating a single-type system to work with objects all the way from the application to the database tier.

WHY NOSQL MATTERS

At the same time, the changing requirements for data velocity, volume, concurrency, availability, and partition tolerance, to name just a few key issues, have inspired a new breed of data management systems, often referred to as NoSQL, or "**N**ot **o**nly **SQL**", databases. NoSQL databases represent a collection of database management tools that both implement a functional subset of what is known as a SQL database, and also introduce different functionality to address critical database needs for situations like web-scale applications, which are not readily available in SQL technologies.

Some of the critical differentiators, requirements and capabilities of NoSQL are:

KEY VALUE:
NoSQL databases use a simple, one dimensional vector, also called a key, to retrieve data elements from the database. First generation NoSQL databases use this key often as the only means to retrieve data to avoid any overhead, such as those

---

4 Java Data Objects (JDO) is a standard way to access persistent data in databases, using plain old Java objects (POJO), see also http://db.apache.org/jdo/

VERSANT

imposed by maintaining index structures. This is tolerable as long as the key is sufficient for data retrieval. However, for any other type of more complex query, or for looking into a sequential read of the entire data store, or maintaining additional structures outside of the key value store, initial NoSQL solutions become exponentially less efficient.

The "reduction" to the simple key:value pair query is somewhat justified, though, as it does eliminate overhead, and therefore provides higher performance. But it also plays into other important NoSQL capabilities, such as Partition Tolerance (see below), which would be much harder to achieve with the more complex meta data and query capabilities typically found in SQL and other enterprise databases.

PARTITION TOLERANCE:
Another foundational ingredient to NoSQL is that it gives the developer simple ways to partition the data, and to add or split partitions without shutting down the entire database system, which is a key to NoSQL's ability to provide High Availability (see below). Partitions are very important for designing and operating a seamless, horizontal scale-out architecture where new partitions can be added when the work load or the volume of data exceeds the capacity of the current cluster architecture.

This is contrary to the design of traditional SQL databases, which often require an upfront calculation for storage, concurrency, and other important considerations that must be "wired" into its setup. Adding partitions can be a tedious process, often requiring that the database services be shut down to re-configure the setup and recreate indexes entirely, among other costly complications.

AVAILABILITY:
Like the other NoSQL capabilities above, Availability plays into the design, as well. Many Web-based applications and services, like eBay (e-Commerce) or Facebook (social media), simply cannot afford to ever go offline. So even if one partition or server goes down, the show must go on! Therefore, designing a horizontal, scale-out database cluster, where individual partitions can easily be replicated, stopped, or restarted provides a "softer and more elastic" architecture so that even if a partition fails, that failure does not affect the rest of the cluster. NoSQL's simple key:value vector and Partition Tolerance enable this kind of architecture.

EVENTUALLY CONSISTENT:
All of the above benefits of simplification for the sake of scale and performance, however, come at a cost. SQL databases

VERSANT

offer ACID (Atomic, Consistent, Isolation, Durable) transactions, allowing the developer to put brackets around a series of database operations to enforce a consistent state of the data store, irrespective of programming failures, hardware failures, or simple latency in updating the state. ACID transactions, however, are typically not part of simple NoSQL databases. This lack of "assurance" in NoSQL's core design disqualifies it from being suitable for certain applications that cannot live without consistency, such as financial transaction applications.

Such "dumbing down of the database" was born out of pure necessity. However, as these new applications grow in complexity and size, they require more and more application code to work around NoSQL's limitations.

Lastly, most NoSQL products are proprietary by nature. It is up to the application developer to fully understand their application's requirements and learn how to map those into the proprietary technology when evaluating NoSQL products. This is another key reason why introducing JPA as an industry standard API for NoSQL is so important. Without a standard, every implementation will be vendor specific, choosing from best of breed solutions for specific application requirements, and switching from one NoSQL solution to another will become expensive if not impossible.

BIG DATA APPLICATIONS, JPA, AND OBJECT-TO-RELATIONAL MAPPING

JPA is a proven specification to develop traditional enterprise applications in Java. However, there are some well-known and well-documented short comings when using JPA as an object-to-relational mapping tool with rich data models:

HANDLING DEEP CLASS HIERARCHIES:

A deep class hierarchy presents problems for ORM tools even with the ease of annotations. Normalization dictates that each subclass be its own table (i.e. "one table per class"), but performance when using such mapping techniques can be prohibitively slow. Most developers collapse several classes into one "big table" that wastes space and makes it very difficult to find the optimal mapping within a deep hierarchy.

SCHEMA VERSIONING:

As it is already so difficult to execute optimal ORM with deep class hierarchies it becomes practically impossible to manage ORM as these data models inevitably evolve. Refactoring

VERSANT

ORM, testing for performance bottlenecks, and evolving table schema can become extremely cumbersome and time consuming.

EFFICIENCY WITH MANY-TO-MANY RELATIONSHIPS:

Many-to-many relationships present efficiency problems for JPA with ORM. With an ORM, the underlying RDBMS must create "intersection tables" to handle many-to-many relationships. This added burden requires more space, memory, and added load (i.e. additional JOINs) to create, maintain, and traverse these intersection tables.

SCALING:

JPA ORM scalability is relatively poor, as the necessary JOINs to obtain related data performs poorly both when data sets grow and when concurrent usage increases. Often times, additional indices are required just to allow efficient retrieval of related data, which further hampers update performance and increases the size of the database.

These types of engineering and operational issues arise in development when code size, model complexity, and data volumes grow. And with Big Data applications, even small issues quickly become bigger challenges that carry an exponential impact on the overall performance, the scalability of the application, and operational costs. Expensive re-engineering is often required to replace automatic mapping code with manually-written code to optimize access patterns and de-normalize the mapping to reduce the number of JOIN operations, to name just a couple resolutions to common problems.

Ultimately, these problems may eventually render the benefits of a standard API and the automatic ORM mapping useless, and the resulting code and relational schema becomes hard to manage and maintain again.

INTRODUCING VERSANT JPA AND THE VERSANT NOSQL DATABASE: HOW VERSANT SUPPORTS THE NOSQL PARADIGM

How can the NoSQL benefits and the apparent issues of ORM mapping in Big Rich Data applications be addressed?

VERSANT

KEY VALUE:

As noted, a simple, direct vector into a large database can be very valuable, especially when the database gets partitioned and distributed. Versant supports the key:value paradigm with Logical Object Identifiers (LOID), a unique global identifier that programmers can use much like a pointer in the application program. Furthermore, Versant translates pointers gracefully and transparently, making it is easy to map even complex object hierarchies into the database, including all of the graph semantics, without incurring any overhead.

PARTITION TOLERANCE:

The Versant Object Database (VOD) offers a fully distributed model, allowing data that is managed on one or many servers to be partitioned in multiple databases, but still be connected when needed, for example, for a traversal from a parent object to a child object across partition boundaries.

AVAILABILITY:

With the right design, these database partitions can be managed individually without any impact on the overall availability of the database cluster a partition belongs to. Furthermore, partitions can be replicated via built-in, asynchronous replication features.

CONSISTENCY:

Last, but not least, VOD allows the programmer to selectively read from the database outside of transaction boundaries, allowing "dirty" reads of data that might not be entirely consistent. At the same time, Versant also provides a complete two-phase transaction protocol for distributed databases that can be switched on selectively and used to enforce data integrity and consistency when needed.

USING AN OBJECT DATABASE WITH A NATIVE JPA LANGUAGE BINDING

In addition to supporting critical NoSQL attributes that enables programmers to use VOD for Big Data applications and design a horizontal scale-out architecture, Versant provides a native JPA

VERSANT

implementation. Versant JPA has been proven in many scenarios to be up to 10 times more efficient, requiring much smaller database clusters and significantly lowering the total cost of ownership.
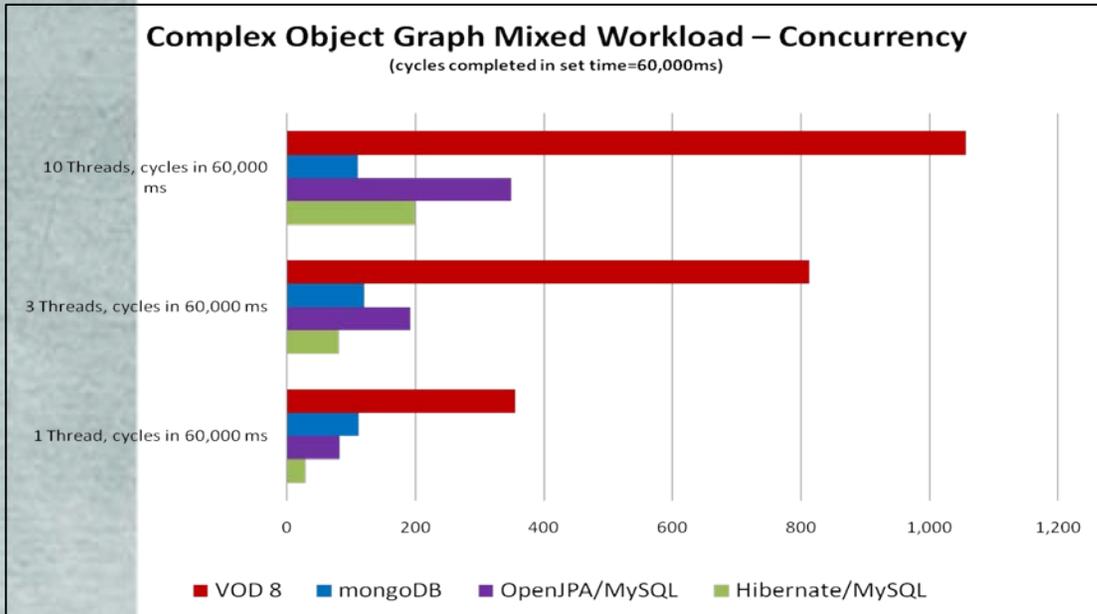

TABLE 1

TABLE 1 AND TABLE 2

VOD outperforms the competition with mixed workloads – and scales the best with multiple threads
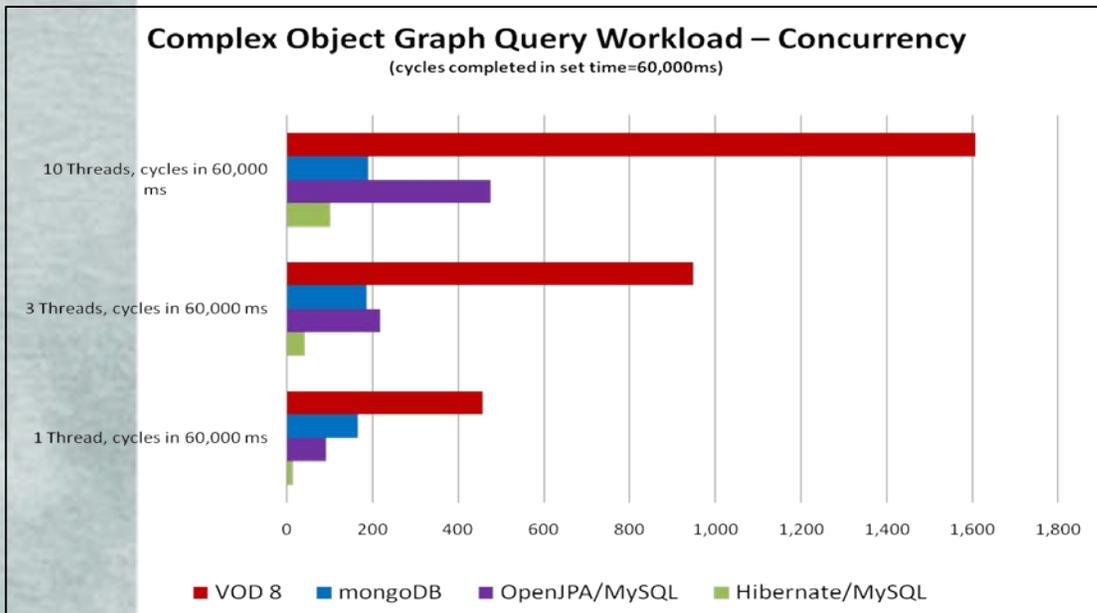

TABLE 2

VERSANT

Such high efficiency is accomplished through two key design characteristics:

1. **Native binding means no JOINs:**

   The most expensive operation in an RDBMS is a JOIN, which recreates a semantic relationship between two tables, for example, between an order table and an order item table. Each order item refers back to the order via the order number. In VOD, the reference is stored as a LOID, eliminating a large scan of an index structure to find all the associated data points to recreate the object (in this case, the order).
   The larger the database and the more complex the object structure, the more overhead is caused by performing JOINS.

2. **No mapping:**

   Similarly, a native object storage requires no mapping needed from the in-memory representation of data objects to the database representation. The database and the application "share" a single type system, resulting in a tremendous reduction in design and coding work, and, furthermore, requires less CPU cycles since nothing needs to be disassembled and re-assembled.

THE SYNERGY OF JPA AND NOSQL

The synergies of combining the JPA standard with NoSQL characteristics are profound. Combining them provides today's enterprise developer with the tools needed to properly support their organization by building applications that speed time to market, raise productivity and flexibility, and reduce the total cost of ownership in operations. The formula is:

JPA + NOSQL + VERSANT = BIG RICH DATA APPLICATIONS

For more information on Versant JPA and the Versant Object Database, visit
http://www.versant.com/products/Versant_Database_APIs.aspx
.

VERSANT