

An In-Depth Look at the Architecture of an Object / Relational Mapper

Patrick Linskey
BEA Systems, Inc.
235 Montgomery Street, 15th Floor
San Francisco, CA 94104
+1 415 402 7328
plinskey@bea.com

Marc Prud'hommeaux
BEA Systems, Inc.
235 Montgomery Street, 15th Floor
San Francisco, CA 94104
+1 503 922 0749
mprudhom@bea.com

ABSTRACT

Object / relational mapping (ORM) tools can deliver a number of benefits to object-oriented, data-centric applications. These benefits are usually artifacts of the implementation of the tool used for the mapping, and not of the ORM paradigm itself. We will examine some of the significant features that the Kodo[1] ORM implementation delivers, by looking at the architecture of Kodo to see the source of these features.

Categories and Subject Descriptors

D.2.3 [Coding Tools and Techniques]: *Standards*.

General Terms

Performance, Design, Reliability, Standardization, Languages.

Keywords

Object / relational mapping, ORM, Kodo, OpenJPA, JPA, JDO, Java Persistence API, BEA WebLogic Server

1. INTRODUCTION

Object / relational mapping is the technique of converting records in a relational database into object instances in an object-oriented programming environment. Kodo is an implementation of the EJB3 Java Persistence API (JPA) specification[2], which defines a standard for object / relational mapping in Java environments. The “object / relational mapping” moniker brings a certain set of features to mind: translation of object allocation and modification operations on a graph of object instances into a sequence of SQL INSERT and UPDATE statements; and translation of object-level queries, load, and delete operations into SQL SELECT and DELETE statements. For object-oriented development teams, there is significant value in decoupling the relational schema and the domain object model using a mapping layer. However, these initial high-level capabilities gloss over much of the interesting and truly valuable work that an ORM tool can deliver.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 12–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006...\$5.00.

Aside from the obvious domain translation problem, object / relational mapping can accelerate data access through caching and optimize data writes to the database through its knowledge of transactional boundaries. Additionally, an ORM tool can simplify the process of tuning data access patterns on a per-use-case basis. Lastly, there is nothing about ORM in particular that requires any significant transaction-level overhead.

The following sections will discuss these particular feature groups in the light of how they are implemented in the Kodo architecture. Since Kodo uses OpenJPA[3] internally to provide much of its functionality, much of this architectural discussion also applies to the OpenJPA product.

2. DATA CACHING

Kodo employs a look-aside data caching pattern, which is particularly well-suited to the optimistic transaction pattern. The cache stores a decomposed view of the developer's domain object model, breaking down relationships into object ID references. The cache can be clustered across multiple JVMs via a variety of protocols.

2.1 Optimistic Locking

The JPA specification requires support for *optimistic* transactions, and Kodo's data cache is designed for that model. Optimistic transactions differ from pessimistic or *datastore* transactions in that they minimize database resource utilization and, in read-mostly situations, reduce blocking on shared resources. When using a datastore transaction, locks are pre-emptively obtained on records in the database, and held for the duration of the transaction. Optimistic locking algorithms assume that there will probably not be any real resource contention, and so no lock is obtained in the database. Instead, additional per-record data is read from the database (typically, a numeric or timestamp column). Then, when that data is later updated in the database, the implementation adds the original per-record lock data to the WHERE clauses of any UPDATE or DELETE statements to guarantee that the optimistic lock column data does not differ from the values read earlier in the transaction. If the WHERE condition fails, the JDBC driver will indicate that the statements failed to update any records, and Kodo will raise an optimistic lock exception to the application code for some sort of failure handling. (Optimistic read locks are implemented by issuing a SELECT statement including the original lock values at transaction commit time.) This model is well-suited for environments that do not usually concurrently edit data, and can provide the same transactional consistency guarantees as datastore

locking. The main difference is in the behavior in the face of concurrent modifications: optimistically locked transactions will assume that all will end well, and will fail completely if a concurrent write is detected. Datastore transactions, on the other hand, will block until there is no more concurrent access. Optimistic locking is therefore better-suited for applications in which it is unlikely that there will be concurrent writes, either because the application is read-mostly in nature, or because data mutation is naturally partitioned at the instance level.

Kodo's look-aside data caching implementation provides significant performance benefits for any read-mostly application. It is the percentage of reads, not the time between writes, that determines the effectiveness of the cache. Such an implementation is well-suited for optimistic locking, as any application using optimistic locking must already take into account the possibility of an optimistic lock exception. The existence of the data cache – especially in a clustered environment – slightly increases the likelihood of an optimistic lock exception, but does not otherwise change the semantics of the application. In concert with the JPA specification's rules for optimistic read and write lock semantics, this provides a rigorous locking model that demonstrates similar characteristics to the more familiar database-side locking employed in many database applications.

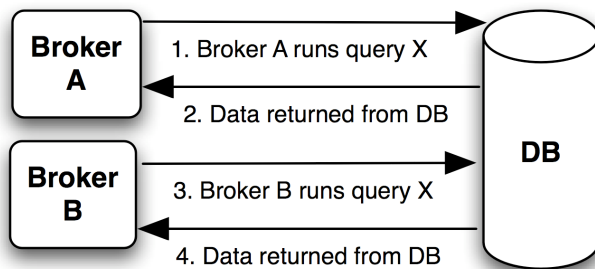


Figure 1: ORM behavior without caching

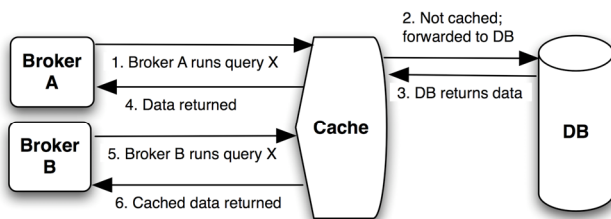


Figure 2: ORM behavior with caching

2.2 Data Cache Overflow Concerns

To avoid consuming all the memory in a system, Kodo's data cache is typically limited to a fixed size, measured in number of entries. Additionally, the data cache can be partitioned on a per-class basis, and each partition can have separate size and implementation strategies.

In an appropriately-tuned system, a given data cache partition may be big enough to hold substantially all of the data of a particular type in the database, if the data set size for the type is small enough. However, this is certainly not always the case, especially with large data sets. So, the data cache implementation must provide some means for removing instances from cache

when the cache is full. Caches of this nature commonly use a least-recently-used algorithm, and Kodo provides such an implementation. However, the default behavior in Kodo is a strictly random overflow policy. In the former case (in which the data cache contains substantially all of the data set), overflow will never happen, and in the latter case, the size of the data cache is typically so large that a random eviction is highly unlikely to remove an instance that will be used soon.

Intuitively, there is some greater likelihood that a random strategy will not perform as well in the latter case as would an LRU policy. However, there is a not-insignificant cost to maintaining LRU bookkeeping information, and this cost can turn into a scalability bottleneck in the face of a large number of concurrent transactions. So, while a random strategy may result in a slightly lower hit ratio, it will often result in considerably more favorable scalability characteristics.

Of course, the only way to know what the best strategy is for overflow in a particular data set is to know about the runtime access patterns for the data set. Since Kodo allows different cache partitions to use different implementation strategies, including the choice of LRU versus random eviction, the application developers can choose the policies best suited for their needs.

2.3 Stale Data and Data Caching

Any real-world deployment must assume that the database is the ultimate home and arbiter of any data stored in the system. So, any caches must accommodate the database as the primary source of information. To that end, Kodo employs a number of techniques to ensure that stale data makes its way out of cache in a timely manner.

Kodo's primary means of cache clean-up is via its clustering protocols, which enable multiple Kodo instances on different machines to notify each other of cache modifications. Kodo also exposes APIs that allow third-party code to notify Kodo of external changes that it may not have any means to know about.

2.3.1 Clustered Cache Communication

Many database applications eventually encompass more than a single physical database client machine. As multiple database clients are introduced to the system, it is important that the separate client machine caches are kept up-to-date with any remote changes that impact their locally cached data. Kodo's default cache implementation does this in a relatively non-coherent manner[4], meaning that caches are not eagerly kept in sync, and that there is a window of potential cache inconsistency among cluster members just after a transaction commit. Kodo asynchronously broadcasts invalidation messages around the network¹ containing object IDs identifying which instances were mutated or deleted in a successful transaction. Upon receipt, the remote machines in the cluster evict any instances matching those in the message from their data caches. The next time that one of

¹ Kodo includes implementations that communicate over TCP, JMS, or, in a BEA WebLogic Server environment, over the WebLogic cluster communication protocol. It is straightforward to plug in different cluster communication protocols, or entirely different cache implementations that handle cache coherence via other means.

these changed records is accessed, it will be re-loaded into the cache from the database.

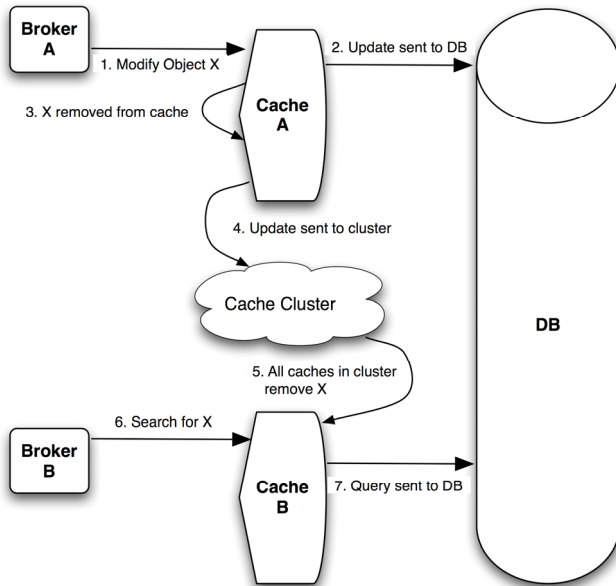


Figure 3: Database Updates in a Clustered Environment

2.3.2 Eviction Policies

However, in the face of legacy systems accessing the database, hooking into Kodo's clustering protocols may not be possible. As a second line of defense against stale data, Kodo provides two means of periodically evicting data from the cache.

The first is a simple per-type time-to-live setting, which allows the developer or the deployer to designate how long instances of a given type should be kept in cache. A stock trading application, for example, might designate that stock prices are valid for 30 seconds. Kodo stores this time-to-live information along with the instance in the cache, and checks the information lazily when accessing data in the cache. This means that the cache may contain a number of instances that have exceeded their time-to-live setting, which will be evicted the next time that they are accessed.

The second is a cron-style[5] eviction schedule. With this scheduling mechanism, a developer or deployer can configure Kodo to drop all records in a given cache partition at a given time, on a per-week scheduling basis. This can be particularly useful when there is a known batch process that updates the database on a particular schedule (every Friday night, for example).

2.3.3 Cache Healing

Given the algorithms outlined above, it is possible for stale data to get stuck in cache. If no timeouts or schedules are specified, this stale data may get stuck in cache indefinitely. In an environment that uses optimistic locking, this can effectively prevent any data modifications (or any transactions employing optimistic read locks) from completing successfully. To avoid this situation, Kodo's cache performs an analysis of optimistic lock exceptions to attempt to determine if the cache may be out-of-date.

First, Kodo always evaluates all optimistic locks against the database. If any of these optimistic lock checks fail, the data cache intercepts the optimistic lock exception generated by the

low-level JDBC[6] interactions. It extracts the failed instances from the thrown exception, and analyzes the data in the instances to see whether or not the data should have successfully committed based on the current data cache values.

If the optimistic lock values in the data cache differ from the values in the failed domain objects, then the data cache is probably correct, since both the data cache and the database agree that the failed transactional records are out-of-date. However, if the optimistic lock data in cache are the same as the values in the failed records, then the data cache must be out-of-date. This is because the only way that the optimistic lock failure could have occurred is if the optimistic lock checks against the database failed, and the database is always the final arbiter of data validity. In this situation, Kodo will automatically evict the questionable data from the cache, and the exception will propagate on up to the application code. When the application code repeats the transaction, the out-of-date entries will no longer be in cache, so Kodo will fetch fresh values from the database. In this way, Kodo ensures that stale data can never get stuck in the cache.

3. QUERY CACHING

Kodo's query cache layers on top of the data cache, and stores the object IDs of instances returned in the execution of a given query, or the values of projections and aggregates returned from an execution. These object IDs are then dereferenced against the data cache when the same query is executed at a later time. The query cache is populated with these lists of primitive data and object IDs once a given non-cached query execution is fully-traversed, and only if the query execution does not involve any dirty changes made in the current transaction.

The query cache is a pure slave to the data cache. Query cache entries are removed from cache when any instance in the cached results is evicted from the data cache for any reason. Additionally, when a query is executed, Kodo analyzes the fields in the access pattern of the query. When changes are made to fields in a query's access pattern, the potentially-affected queries are dropped from cache.

Further, if a query is executed in a transaction after changes were made involving the query's access pattern, the query cache is disregarded, and the query is evaluated as if the cache were not present.² So, queries are never evaluated against the values in the data cache itself, but instead the query cache is treated as a simple performance-boosting lookup mechanism.

4. STATEMENT ORDERING AND BATCHING

When performing a sequence of SQL operations via JDBC, it is possible to realize significant performance gains by grouping together identical statements and employing the JDBC statement batching facilities. *Statement batching* allows a single prepared statement to be sent to the database with many sets of parameter values, rather than performing individual database round-trips for

² Ignoring the query cache, if there are changes in the access pattern of the query being executed, Kodo will either flush the changes to the database and execute the query purely against the database, or will load all instances of the candidate type into memory and evaluate the query in memory.

each set of parameter values. This can result in significant performance improvements due to reduced network round-trips and internal database optimizations during query execution.

However, often, the optimal or intuitive way to write a given sequence of operations in a transaction from the point of view of the application developer does not coincide with the optimal statement batching ordering for that transaction. Additionally, for a transaction to commit to the database successfully, the statements issued to the database must meet the referential integrity constraints outlined in the database.

Since data mutations in Kodo always happen within the bounds of a transaction, Kodo can take advantage of a larger scope than the typical per-statement scope that is familiar in JDBC applications. Within this transaction-level scope, Kodo assembles a directional graph representing the foreign key constraints among the statements to be written to the database. It then walks this graph with a breadth-first algorithm to build an ordered list of SQL statements to issue to the database. The algorithm ensures that all non-cyclical referential integrity constraints³ will be met, and then optimizes statement batching by collocating identical (and therefore batchable) statements within levels of the graph traversal.

This analysis means that applications using Kodo will automatically benefit from optimal statement batching, regardless of whether or not the application logic is written to take batching or referential constraints into account.

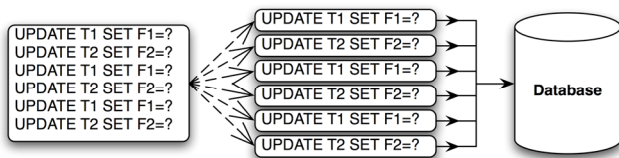


Figure 4: SQL statement execution without statement batching

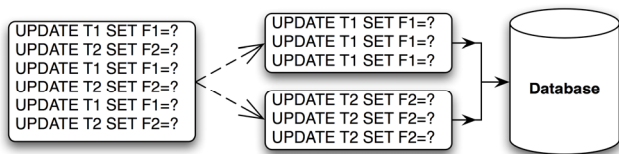


Figure 5: SQL statement execution with statement batching

³ When faced with cyclical referential integrity constraints, Kodo will issue extra database operations to attempt to avoid triggering a constraint violation. That is, in the cyclical case Kodo will attempt to insert null values into columns that involve cyclical constraints, and then update the columns after the other side of the cyclical relationship has been inserted. When there are no cyclical constraints, Kodo can guarantee that it will perform the minimal number of operations for a given transaction.

5. FETCH GROUPS FOR DATA ACCESS TUNING

In applications that interact with remote resources, a significant amount of time is often consumed in the network stack, sending the request across the network and waiting for the response. One of the best ways to improve the performance of these types of applications is to reduce the number of network requests performed. So, when querying a database, it is usually optimal to load all the data that will be necessary for a given unit of work at the same time. For example, if a unit of work is about to process a purchase order and its line items, it might make sense to perform a single SELECT statement that loads the purchase order and joins the line items into the query at the same time.

However, often, the transitive closure of the graph of objects needed in memory differs from use case to use case, even when executing the same or very similar code paths through application code. The optimal graph to load frequently depends on what will be done with the data being accessed, rather than on exactly which records are being accessed. To continue with our purchase order example, one use case might only require the purchase order IDs and descriptions, possibly for creating a summary report. A detailed report might query for the same purchase order records, but access all the line items and associated products.

Kodo's fetch groups decouple application logic and query definitions from the specification of the transitive closure of the graph of objects to load. The programmer can statically define named fetch groups, which can then be overridden or fine-tuned at deploy time without making changes to the source code of the application itself. Application code can reference these statically-defined fetch groups, and Kodo will alter the SQL it creates to reflect the needs of all the fetch groups specified when accessing the database. This decoupling of fetching heuristics from application logic means that an application can separate per-use-case tuning from the application logic itself. For example, in our purchase order example, a web application might use a servlet filter to configure fetch groups differently for summary requests versus detail requests, but then let the application logic for finding and processing purchase order requests remain exactly the same. This allows for greater code reuse, and simplifies the application delivery process by separating tuning concerns from the application logic development lifecycle.

Additionally, the application code can add or remove individual fields programmatically. This dynamic fetch group configuration is particularly useful when exposing some degree of graph control to the end user of the application. For example, applications that provide custom query and reporting capabilities might not know which fields will be needed until the end user of the application chooses which fields to display in the report. When executing a query or loading an object by ID, Kodo loads the union of all the active static fetch groups and dynamically-specified fields.

6. FIELD-LEVEL TRACKING OF ACCESS AND MUTATION

When using Kodo, an application interacts directly with an object model provided by the application developers. This object model uses standard Java programming techniques, and Kodo is responsible for mapping fields in the model into columns in the database.

In order to perform its duties of loading data and writing changes back out to the database, Kodo must know what is happening to the object model as the application manipulates it. To do this, Kodo instruments the Java bytecodes[7] in the domain model to eagerly intercept any field accesses, including both field reads and writes. Another common technique in the ORM field is to create a copy of the domain objects as they are loaded, and then compare values when flushing to the database. Kodo's field interception architecture is advantageous both from a performance standpoint and from a memory utilization standpoint.

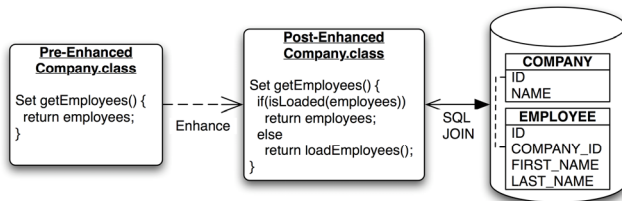


Figure 6: Bytecode enhancement modifications

6.1 Field Tracking and Performance

Kodo's field interception means that whenever a field is mutated, Kodo knows about that mutation immediately. Kodo then adds the instance to a collection of unflushed dirty records, and marks the changed field as such. When the data is flushed to the database⁴, Kodo has direct references to all the unflushed data in the current transaction. This means that Kodo can immediately walk through these instances and write SQL just for those instances. Additionally, since the exact fields that have been mutated are known, the generated SQL will only update the minimal set of columns. As a result, flushes execute in time proportional to the number of modified records since the last flush. This is significantly advantageous when compared to implementation architectures that obtain copies of data and then diff these copies at write-back time; in such architectures, flushes run in time proportional to the number of instances accessed during the entire transaction. This number can be considerably higher than the number of unflushed dirty instances for long-running transactions, or for transactions that access more data than they write.

This field-tracking architecture is further beneficial because data comparison – especially of string data – can result in considerable CPU load, for no real benefit. Since Kodo knows exactly which fields in each instance were changed, Kodo does not need to do any comparisons when flushing a particular dirty instance.

6.2 Field Tracking and Memory Overhead

Tracking dirty fields means that Kodo has only marginal impact on memory overhead. Generally-speaking, Kodo holds onto weak references⁵ to the application domain objects. When the

⁴ Flushes can happen because of a commit operation, a flush operation, or a query whose access pattern involves types and fields for which instances have been modified in the current transaction.

⁵ *Weak references* in Java differ from normal references (known as *hard references*) in that they do not count as a reference to the garbage collector. So, when the garbage collector runs, any instances that are accessible only through weak references are eligible for garbage collection.

application code mutates a record, Kodo hardens its reference to that record by putting it into a collection of dirty records. (This is the same data structure discussed in 6.1.) After a flush operation, Kodo weakens these references. As a result, Kodo only holds on to hard references to unflushed dirty records, and so the lower bound on the memory consumption of a transaction is proportional to the number of unflushed dirty records. (Of course, if the application code holds hard references to data, memory consumption could be considerably higher.)

ORM implementations that employ a comparison-based flush algorithm must hold onto the before image of all data loaded from the database during a transaction, meaning that they must create a copy of all loaded data as it is read. Further, since such architectures do not actively track which instances have changed, those implementations cannot weaken any references to the application domain objects or the copies of the loaded data until after commit. So, with such implementations, memory overhead in a transaction is proportional to the number of instances read in the transaction as a whole.

Clearly, the set of unflushed dirty records is smaller than (or, at worst, equal to) the total number of instances read in a transaction. With appropriate incremental flushing in the application code, this number can be held below a fixed size, even in the face of arbitrarily large transactions.

7. LARGE TRANSACTIONS AND DATA CACHING

As discussed in the last section, Kodo is designed to handle large transactions in a performant and memory-conservative manner. However, Kodo's data caching features discussed earlier can conflict with large transactions.

7.1 Large Transactions and Memory

Large transactions can erase much of the memory gains outlined in the last section. In order to maintain a *read committed* isolation semantic, Kodo does not update the data cache until a transaction has successfully committed. As a result, Kodo must maintain information about what to update until the transaction is committed. Looking at the caching architecture as it has been presented so far, this means that Kodo must maintain hard references to all mutated data until the end of the transaction, so that the changes can be copied into the cache. This is worse than Kodo's normal behavior, in which it holds onto only those instances that are unflushed, but is still better than maintaining a copy of all data accessed in a transaction.

However, sometimes the number of modified records in a transaction is prohibitively large. In these situations, the application code can instruct Kodo to only track the classes that are mutated for the purposes of data caching, rather than all changed instances. When a transaction is in this mode, all instances of the types that were modified will be removed from cache at commit time, and the cluster communication message will only contain this type information. This is clearly more disruptive to the contents of the cache, but has much better memory characteristics in the face of transactions that mutate large amounts of data, such as a batch upload of new data or a monthly data cleanup process.

7.2 Large Transactions and Cache Churn

When caching is enabled, as records are loaded during a transaction, Kodo copies the loaded data into the data cache for use by subsequent transactions. Since the cache is designed to minimize database round-trips, this is typically the right behavior. However, when loading large amounts of the database into memory during a given transaction, this can have the unintended side-effect of pushing out much of the data currently in cache. In some situations, this is exactly the worst possible behavior. For example, consider an application use case that performs a weekly analysis of data in the database for export to some reporting process. This process might load many records that will not be accessed again until the next weekly report. In the process, it might push out operational data that may be needed by some other part of the application just a few minutes later. For algorithms that access data once and then do not revisit the data for a long period of time, loading the data into cache has no real benefit, and can have significant costs in the form of effectively clearing the cache.

For such use cases, the application code can selectively configure Kodo to not populate the data cache with data that is loaded in a particular transaction. Of course, any data that is in cache already is used, and the cache is kept up-to-date with any modifications that occur. But, Kodo prevents the cache from being churned by the transaction in question.

8. CONCLUSIONS

In this paper, we introduced several key features of the Kodo object / relational mapping framework, and discussed the impact of those features on applications using Kodo. We focused on features and architectural choices that have an impact on runtime performance and scalability. (Note that we did not discuss some of the more immediate topics of ORM, such as the actual mapping of classes in Java to tables in an underlying relational database; of course, these features exist in Kodo.) We examined the behavior of the data cache in clustered and heterogeneous environments, and looked at how to tune data access with fetch groups when the database must be consulted. We discussed the benefits of Kodo's transaction-level scope and eager change-tracking strategy when it comes to writing changes back to the database, and how all these features interact with each other.

All of the features discussed in this paper are available as part of the BEA Kodo product, and thereby available in the BEA WebLogic Server product, which bundles Kodo as its JPA persistence provider as of version 10.0.

The Kodo roadmap includes optimizations to some of the limitations discussed in this paper. In particular, query caching could be optimized in a number of areas without implementing a complete in-memory query evaluation engine, and a bucket-based less-recently-used data cache overflow policy would probably be more ideal than either of the current data cache overflow strategies. Additionally, a number of the data cache and fetch plan features discussed in this paper require a certain level of manual tuning; we plan to develop adaptive algorithms to eliminate or reduce the need for manual intervention.

9. ACKNOWLEDGEMENTS

The Kodo product is a large codebase encompassing years of development and design. We thank the entire team that built this product. In particular, we would like to thank Abe White for his contributions to the codebase and his architectural intuition. Also, we are indebted to Mike Carey, not only for offering an invitation to author this paper, but also for his help in reviewing it as it came together.

10. REFERENCES

- [1] BEA Systems, *Developers Guide for Kodo*, BEA Systems Inc., 2006.
- [2] EJB 3.0 Expert Group, *JSR 220: Enterprise JavaBeans Version 3.0 Java Persistence API*, Sun Microsystems, Santa Clara, CA, 2006.
- [3] The Apache OpenJPA project, *Developers Guide for OpenJPA*, the Apache Software Foundation, 2007.
- [4] Wikipedia, *Cache Coherency*, http://en.wikipedia.org/wiki/Cache_coherency
- [5] Vixie, Paul, *Cron manual page*, Free Software Foundation, Boston, MA, 1993.
- [6] Jon Ellis, Linda Ho and Maydene Fisher, *JDBC 3.0 Specification*, Sun Microsystems, Santa Clara, CA, 2006.
- [7] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Sun Microsystems, Santa Clara, CA, 1999