Experiences using an ODBMS for a High-Volume Internet Banking System.

Vincent Coetzee
eBucks.com
PO Box 781171
Sandton, 2146, South Africa
(Voice) +27 11 282 8952
(Fax)+27 11 282 4529
vincent.coetzee@ebucks.com

Robert Walker
GemStone Systems
1260 NW Waterhouse Ave
Beaverton, OR, 97006, USA
(Voice) +1 503-533-3623
(Fax) +1 503-629-8556
bob.walker@gemstone.com

ABSTRACT

Few large corporate organizations make the decision to use an Object Database Management System (ODBMS) when developing high volume transactional eCommerce web sites. This report examines an application used to run a website that encompasses banking, online shopping, and the management of a Customer Loyalty Currency called eBucks.

This system demonstrates that an ODBMS can be used in a high volume web based transactional system. While the choice of this technology has many merits, there are drawbacks. These drawbacks are examined along with the solutions that have been used at eBucks to either solve or ameliorate them.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA. ACM 1-58113-751-6/03/0010.

Categories and Subject Descriptors

H.2.4 [Information Systems]:Database Management - Object-oriented databases
H.2.1 [Information Systems]:Database Management - Logical Design – Data Models
J.1 [Computer Applications]:Administrative Data Processing – Financial

General Terms

Performance and Design

Keywords

ODBMS, Banking, Internet, Transactional, High-Volume, Object-Oriented Databases

1. BACKGROUND

Three years ago FirstRand Bank decided to create eBucks.com, a new division whose goal is to service the bank's online customers, and to provide a portal for the newly created eBucks Loyalty Currency. FirstRand customers earn eBucks when they transact against any account managed by any of the FirstRand subsidiaries, with services ranging from banking to vehicle financing

to insurance. There was no system extant, either in South Africa or in the world, which would accommodate the precise needs of the new division. The decision was taken to develop a new system to provide these services to FirstRand customers, and to do so using a Java Application Server.

The specifications given to the designers of the system were spare, "build a banking and loyalty currency portal, and do it in 100 days". A large portion of the design work had been completed prior to this time, and a working system was delivered on schedule. It is our belief that this was possible due to the flexibility and power provided by using an ODBMS to provide persistent storage for the Application Server.

Currently the eBucks application so developed services a customer base of almost 700,000 users presenting over 36 million transactions a month. A significant number of these customers are small to medium sized businesses and hence present a large number of transactions to the system. In the next 12 months eBucks anticipates supporting over 1 million customers presenting 45 million transactions per month to the system. As of August 2003 the ODBMS contains almost 460 million objects. The system was delivered by a small project team consisting of 1 back-end Developer/Architect who built the CORBA servers accessing the ODBMS, 4 Servlet Developers who built the user interface portion of the system, and 1 front-end Architect.

2. THE ARCHITECTURE

We had worked with an ODBMS before, which one of us (Coetzee) had used in the Investment Banking division of FirstRand Bank to develop trading systems. He was confident in the abilities of an ODBMS and therefore elected to use a Java based ODBMS from the same vendor.

A significant amount of resistance was encountered as a result of this choice. Many large RDBMS vendors insisted that an ODBMS could not cope with the transactional volumes required of such a system, despite the fact that their own Application Server offerings were unproven at the time. One of the more well known Application Server vendors demanded a meeting with our CEO during which he insisted that an ODBMS could not support the transactional volumes required by our environment and went so far as to demand "how can eBucks do eCommerce without us?" Fortunately for us our CEO had a great deal of faith in our ability to make our chosen environment work correctly and backed our choice of technologies.

A basic requirement of the system was the ability to interact with the FirstRand mainframe, where a large COBOL based system managed the accounts of over 6 million customers. MQ Series from IBM was selected to provide communication with the mainframe. All of the customer data with the exception of most of the banking accounts was to be stored locally in the Java ODBMS.

2.1 Object Trees

Each Customer has a tree of associated objects containing all relevant information, such as addresses, accounts, contracts, insurance policies, and marketing information. The root of the tree is the Customer object itself. Since customers interact with the system via Interactive Voice Response (IVR), GSM SMS (locally called WIG), GSM WAP, USSD and the Internet, the

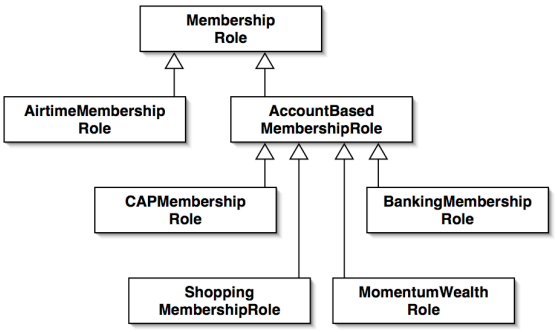

Figure 1 – Extract of Role Hierarchy

complexity of these object trees is high.

Deciding on how to structure our objects trees was difficult. Business required a system that was flexible enough to add functionality to customer objects on a regular basis, we did not want the customer class to become a "God" class. We therefore created an inheritance hierarchy of "role" classes. Each type of behavior and state required by a customer is encapsulated in a particular role class. These roles are then added to and removed from the customer object as needed. A small section of this class hierarchy can be seen in Figure 1.

Using this mechanism allows us to add functionality to a customer in an incremental way. For example to capture the state and behavior associated with a customer that has just joined the eBucks Loyalty Programme, but is already registered for some other functionality, we merely instanciate and populate a CAPMembershipRole (the CAP prefix is an abbreviation for Customer Appreciation Programme, the politically correct name for a Loyalty Scheme). This CAPMembershipRole stores the eBucks account proxy object as well as various other details associated with the customer's role when interacting with the Loyalty Scheme. This role is then added to the customer object's collection of roles, and accessed when needed by the various portions of the system. This approach sits well with the various caching mechanisms used by our ODBMS, since it does not require the entire state of a customer to be loaded when only a portion is required by our system. A typical banking customer would therefore end up with a tree of objects similar to the one shown in Figure 2.

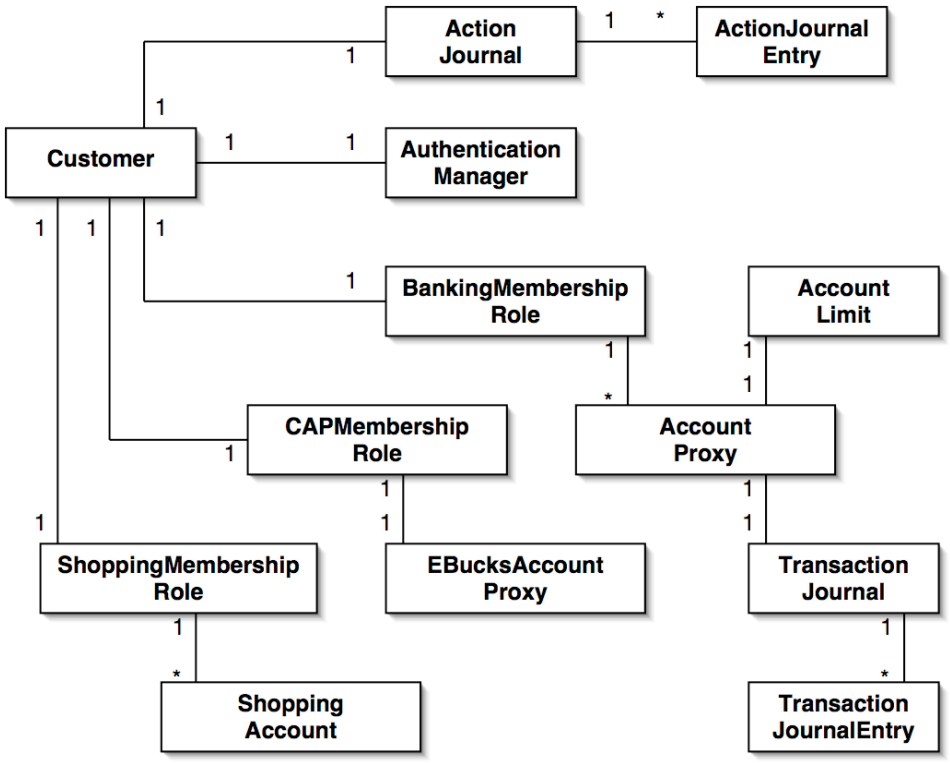

Figure 2 – Typical Customer Object Tree

In order to ensure that programmers could use simple Java methods to perform complex operations against the mainframe (without any in-depth knowledge of the mainframe) we decided that proxy account objects representing accounts not stored locally in the ODBMS (stored either on the FirstRand Bank mainframe, or on AS/400 systems in FirstRand Insurance) would also be created and inserted in the customer's object tree. These proxy accounts wrap the interaction with other systems and hence ensure a large degree of architectural insulation. This encapsulation has proved invaluable in ensuring the maintainability of the system, it has also reduced the complexity of the code in the system since all accounts, whether held locally or elsewhere, can be treated in the same manner. Managing an object tree of such complexity is less efficient in an RDBMS due the large number of Object Relational (O/R) mapping actions that have to be performed to translate between the RDBMS representation and the Java objects.

These requirements resulted in a three-tier architecture (see Figure 3). The first tier is a front-end Servlet tier that converts user interface requests into requests of application objects. The Servlets access the functionality of the second tier (the application objects) using CORBA. The second-tier application objects convert incoming eBucks transactions into transactions that can be performed locally or dispatched to the FirstRand mainframe (or other external systems) for execution. All of these transactions both successful and otherwise are logged locally in the ODBMS (in Journal objects) so that customers may query their transactions and potentially the reasons for their failure at a later stage. This implies that a single transaction against the FirstRand mainframe may in fact result in several transactions against the local ODBMS. The third and final tier is the mainframe (or other external system).

3. ADVANTAGES OF AN ODBMS

As a result of our experiences at eBucks we can list some of the advantages that we have found in using an ODBMS environment for our website.

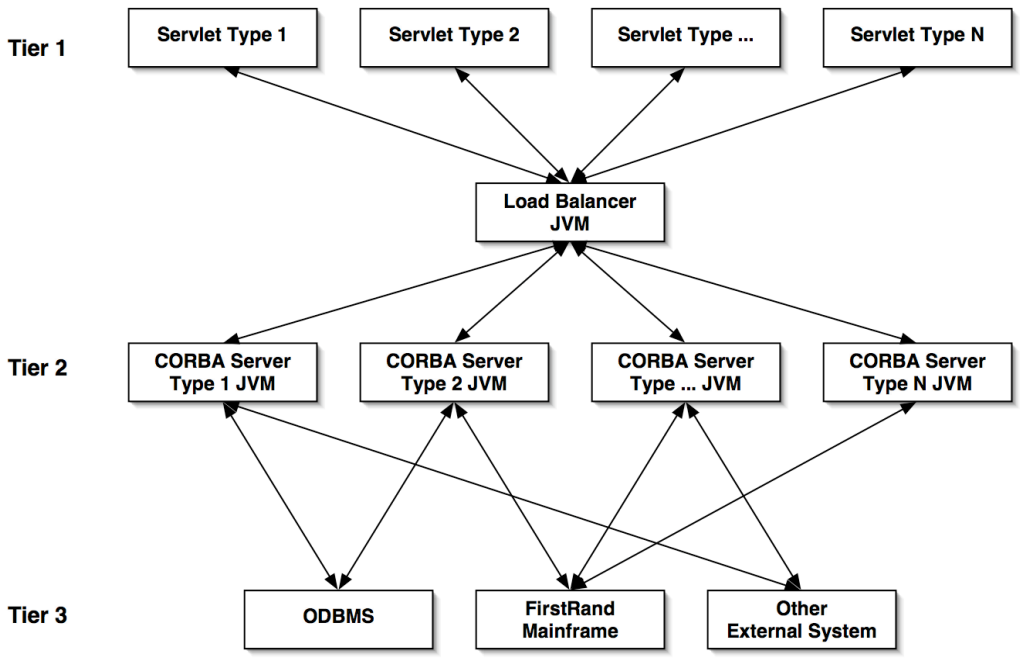

Figure 3 – Macro Architecture of eBucks Environment

3.1 Simplicity

We have no doubt that the simplicity of using an ODBMS contributes significantly to our ability to rapidly develop and deploy new functionality and bug fixes. It is extremely straightforward to create and persist Abstract Data Types, since the persistence mechanism directly supports classes and objects, requiring no extra work to map classes to tables (as required by an RDBMS implementation). An ODBMS also directly supports inheritance, which means that no difficult design decisions have to be made regarding the splitting of instance variable storage across tables.

Development cycles at eBucks are measured in weeks and days - not the months that have typically been the case in many projects that we have worked on. In our opinion much of our efficiency can be attributed to our use of a Java ODBMS. The transactional model of the ODBMS is easy to understand, there is no database language syntax to learn, over and above that of the implementation language. Developers do not need to use another language to express the database schema, because the Object Model as defined by the Java application classes *is* the schema.

3.2 Time to Market

Due to the ease of developing and deploying modules with complex Object Models, we gain significantly over our competitors in our ability to add functionality to our environment quickly. We believe that this advantage is one factor that has contributed to eBucks becoming profitable one year ahead of schedule.

3.3 Schema Management

The persistent Object Model is identical to that of the application class hierarchy. Managing a persistent Object Model is far simpler than managing an environment where the in memory structures are different than those on disk. However, care must be taken in designing the entry points to the large collections of root objects in the system. Less experienced developers often err in creating multiple root objects for the same underlying tree of objects, leading to confusion and incoherency in the model.

3.4 Code Independence

We have found that the completely non-invasive transactional model makes it possible to insulate most business objects in the system from having any knowledge of the underlying database. This is important in the event that we might have to migrate to some other environment at a later stage.

Knowledge of the persistence mechanism in the eBucks system is encapsulated in 27 of the 1000 (or so) application classes. While some O/R mapping environments will allow for a clean separation of this knowledge, many OO systems making use of an RDBMS must directly embed knowledge of SQL and tables deeply in the application classes themselves.

It is important to clarify the above point, no-where in our system does any object have to actively participate in the mechanism used to persist it. This is often the case in either purchased or home-grown O/R mapping frameworks, since classes need to have some knowledge of what tables they fit in. It is only due to the fact that objects are persisted as objects and not fragmented into various atomic database types that this is possible. The knowledge of the persistence mechanism that is embedded in the 27 or so “aware” classes is purely related to obtaining connections to the database and management of database specific transactional mechanisms.

3.5 Natural Object Model

In an environment that uses an ODBMS, objects that refer to other objects always “contain” the object that they reference. What we mean by this is that there is no distinction between an “in-memory” object and an “on disk” object. In systems that make use of an RDBMS, objects are often found to contain a key or an index to

the referenced object. Once again, some O/R mapping environments remove this complexity but more often than not, the fact that the referenced object is not actually stored as an object leaks through into the Object Model. Objects are always objects in the ODBMS model, and are always available. This naturalness of expression for the developer as well as the designer is very difficult to achieve in any RDBMS environment.

3.6 Collections

Many ODBMS vendors (GemStone included) provide highly optimized Collection classes that can be used to provide efficient management of vast numbers of objects. Iteration over (and management of) the collections is straightforward and can be used without any complicated setup routines. We believe this is a distinct advantage to developers.

3.7 Performance

Performance is vital to the online experience, online users expect snappy responses. In our experience some architects believe that doing things in a pure OO manner leads to poor performance. In many situations involving the storage and retrieval of objects, an ODBMS does deliver significant performance gains over other persistence mechanisms. We hasten to add that there are some instances when an RDBMS will outperform an ODBMS. These instances often occur when it is necessary to perform arbitrary queries over large collections of objects. We touch on this issue in section 4.2 below.

3.8 Design Recovery

By this odd term we mean the ability to quickly and easily correct design mistakes in the class hierarchy / schema. Since the schema of the database *is* the class hierarchy as defined by the Java classes in the system, changing a class definition to correct a mistake in design or implementation is relatively simple.

In systems where there is a distinction between the class hierarchy and the database schema correcting flaws can become a complicated matter. Because the schema of an RDBMS is available to other modules (possibly developed outside of the scope of the larger system) errors are often introduced when making changes to the database structure. Changes to the RDBMSs schema may differ from the expectations of external programs. Such conflicts cannot occur in a true ODBMS since there is no difference in structure between in-memory in-use instances and those persistently stored on disk. There are however other problems that can occur when changing schemas, we deal with these issues in section 4.3.

3.9 Retention of Objects

Developers make mistakes. In an RDBMS these mistakes may occasionally lead to situations where inadvertently deleting a row in a table can lead to dangling references, since triggers are often imperfectly implemented. An ODBMS with Garbage Collection based object removal completely eliminates this problem. Persistent objects are only “deleted” once they are no longer referenced by any other instances reachable from a well-defined root object. The ability to recover from an accidental deletion has often allowed eBucks to recover gracefully from potentially embarrassing situations. This has occurred when flawed code eliminates the primary reference to an object. Because other objects still referenced the “deleted” object, it was possible to restore the reference and thus recover the object. Obviously this only works if a persistent Garbage Collection has not been performed.

4.DISADVANTAGES

As with every technology some problems are eliminated but others arise. We have found that using an ODBMS does have some different problems associated with it. We discuss some of these and the eBucks solutions to them below.

4.1 Arbitrary Queries and MIS

One of the great advantages of an RDBMS is the ability to formulate complex queries and apply them to the database well after primary implementation. Intelligently designing the primary entry points to the large collections in an ODBMS system is vital, and often involves some degree of prescience due to the fact that business all too often cannot articulate the entirety of their requirements in advance. The mark of a good system is how well it tolerates these demands.

We have struggled in this area since navigating to a particular object or collection of objects means knowing how to get to a root object that has the desired target in its object tree. If business cannot define this key up front, which is often the case in MIS situations, then providing that sort of information can be difficult and time consuming. Some ODBMSs do not support these sorts of arbitrary queries well. Carefully building primary collections of objects that are indexed according to the keys that business might require is important. RDBMSs do not suffer from this problem, although in some cases it might be time-consuming to have to build an index on a large table after implementation.

The eBucks system resolves this issue by utilizing a series of HashMaps pointing to the customer objects. These keyed collections contain the references to the customer objects based on about every key we think is likely. This works, but every couple of months, business requests a mechanism for accessing customers on some basis we have not designed for. This would not be a problem in an RDBMS, since a new query with an appropriate series of joins would solve the problem. We continue our research in this area.

4.2 Warehousing

Data Warehousing has proved extremely problematic for us. We have progressed towards a solution by pairing our live ODBMS environment with an RDBMS environment for MIS and Warehousing. The production environment runs against the GemStone/J PCA, whilst the MIS and Warehouse systems run against a DB/2 database that contains relational data replicated out of the production objects on a monthly basis.

This provides the required MIS and Warehouse functionality, however we would like to keep the RDBMS more “in-synch” with the production system. We find it difficult to populate the RDBMS with data from the production ODBMS due to the ruinously long time it takes to traverse the full object tree in the ODBMS environment. Many of the performance boosts in the GemStone/J PCA are due to its sophisticated object caching and paging, these benefits almost completely vanish when the requirement is to touch every single object in the system, cache pressures become too large and inefficiencies arise.

We have recently attempted a new mechanism whereby we track changes made to any object in a Customer object graph, and then dump only the data from changed Customers. We are still struggling to perform this activity in a near real-time manner due to the poor performance we have experienced with many JDBC libraries. We do believe however, that this will ultimately be the best solution.

4.3 Schema changes

While changing a class hierarchy is simple and quick on the surface, it is important to remember that changing the “footprint” of class that has instances in the repository requires that every instance of that class be located and mutated. At this point in time, GemStone/J does not perform lazy mutation and hence this process can be very time-consuming. It is important to note that the system having its schema changed cannot be active while this operation is taking place. In a banking system, downtime means lost revenue, and this problem continues to hamper us.

A newer version of the GemStone/J PCA, namely Facets, does go some way toward alleviating this issue, but we believe that it will remain a limitation going forward. A simple trick however does allow one to accommodate this in some circumstances. The technique involves declaring the references to objects as references to Java Interfaces, rather than as references to an instance of a class. This means that when one wishes to change a schema, one can do it by adding a new class that implements the Interface. One then lazily migrates instances of the old class as they are accessed by the application. Once again, this approach demands a degree of foresight and thought, which is difficult to achieve in a fast paced growing business.

4.4 Garbage Collections

The GemStone/J PCA determines which objects are eligible for garbage collection by performing a full walk of the persistent object tree from a set of known roots; this is called a Mark For Collection (MFC). It is necessary to perform an MFC on the repository on a regular basis. This is so that garbage pages and objects are reclaimed, and the Object Table does not get too large. As mentioned earlier, walking the complete tree of objects is a time consuming exercise. Although the MFC itself does not significantly impact performance, certain operations invoked at the end of the MFC can have a brief adverse effect on the operation of the system. We therefore find it best to start our MFC on a Friday evening (our least busy period) and thus ensure that the portion that does affect performance usually takes place early on Sunday morning, another non-critical period. Obviously each organization would have to ascertain an appropriate timeframe that works for them. Having said this, it should be noted that a newer version of our ODBMS has made significant progress towards reducing the overall time required by the MFC routines.

4.5 Lack of Tools

Possibly one of the most frustrating aspects of using an ODBMS is the complete lack of third party tools for these environments. When using an RDBMS, tools such a data aggregators, report writers, data summary tools and so forth are readily available. Using an ODBMS means that many of these tools, most of which are required more for end-users than for developers, have to be custom written. Admittedly due to the almost total integration between “memory object space” and “persistent object space”, developing these tools is quick and easy, but it consumes developer resources that could be used far more productively elsewhere. We feel that the lack of third party tools is due in large part to the ODBMS marketplace having not yet reached a critical mass equivalent to that of the RDBMS world.

5.CONCLUSION

This summary of what we have learned is neither complete nor in depth. We have very successfully developed systems using the GemStone/J PCA in our banking environment, and we believe that some of the success of our business can be directly attributed to the benefits that the use of an ODBMS has given us. We plan to continue using these technologies going forward, as the advantages far outweigh the disadvantages. We’re confident that the ODBMS technology in use at eBucks will not only continue to scale as our business grows, but will also continue to provide us with a significant competitive advantage.