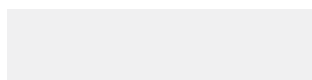


MMOG CONFIG AND CODING TIPS

Programming and Configuration Tips for Achieving Optimal Performance
Using the Versant Object Database in MMOG Applications



INTRODUCTION

The Versant Versant Object Database (ODBMS) solves the challenges faced by MMOG companies when creating a scale-out solution for read/write transactions under zero latency conditions. The following are some coding and server configuration tips that are essential for MMOG applications. This paper is not intended as a substitute for the documentation, but rather as a supplement to quickly move you in the direction of an advanced user. These tips are meant to give you a quick start in achieving the optimal evaluation of Versant technology.

The Versant Object Database

Using the Versant Object Database for data storage brings powerful advantages to applications that use complex C++ and Java object models, have high concurrency requirements, and large data sets. For more general information about the Versant Object Database please refer to

<http://www.versant.com/developer/resources/objectdatabase>

If you are new to Versant's object database technology, the screencasts showing how to create a Versant enabled application might be a good starting point to kick off your project:

<http://www.versant.com/developer/resources/objectdatabase/screencast>

PROGRAMMING NOTES FOR ACHIEVING OPTIMAL PERFORMANCE

1. Basics of using Versant

Versant provides a `VSession` class to connect to one or more databases, manage your client cache, and manage transactions. Versant's transparent persistence implementation means that almost all of your Versant related code will be utilized for:

- a) getting a connection (cache context a.k.a. the "session")
- b) doing a very limited amount of queries (discussed later)
- c) committing at appropriate points

The current thread is attached to a Versant session implicitly when the session is created. When using multiple threads per session, a `VSession::set_session()` method is used to attach the current thread to the session (and implicitly detach the previously attached thread).

Here is a quick example application.

```
#include <iostream>
using namespace std;
#include <cxxcls/pobject.h>
#include "World.h"

int main(int argc, char* argv[])
{
    VSession *aSession;
    World *one;

    cout << "Where your proper arg check code ☺" << endl;

    char* db_name = argv[1];
    cout << "Beginning session on database \" " <<
        db_name << "\"..." << endl;

    /*
    Create a new connection to the specified database.
    This is also your "session" or cache context.
    The db_name is of the form: dbName@hostName
    Or dbName@hostName:port or dbName@i.p.:port
    Current thread of execution is implicitly attached
    to the session.
    */
}
```

```

aSession = new VSession(db_name, "GameNav");

/*
   Now you can create/update/query, etc.
   Note- "new" is replaced by O_NEW_PERSISTENT macro
*/
one = O_NEW_PERSISTENT(World)("Shard1");

/*
   Here our "one" just went into the database.
*/
aSession->xact(O_COMMIT);
cout << "Ending session" << endl;
delete aSession; // cleans up resources
}

```

2. Session Cache Control

Versant provides multiple caches, one for the database server instance (one per db) and one for each session instance. Here we discuss taking control of the session (client) cache using Versant's advanced API capabilities to achieve zero latency response.

2.1 Cache Retention

Cache retention across transaction boundaries is controlled by use of options to your commit calls, `VSession::xact()` and `VSession::xactwithvstr()`. In particular, you will want to use the option `COMMIT_AND_RETAIN` which will keep your objects in your session cache across transaction boundaries. Note there are various flavors that change how locks are also affected. Generally, you will use Optimistic or No lock policies so the above will be used. See reference doc for all possibilities.

2.2 Targeted Flush

Generally, you will want to target particular sets of objects to be flushed to the database (committed). Therefore, you will want to use `VSession::xactwithvstr()` API. This allows you to tell which particular set of objects in the cache to flush. You should also use the options `COMMIT_AND_RETAIN` | `O_FLUSH_VSTR`.

2.3 Pinning Objects in Cache

The application cache size is controllable using configuration parameters in the profile.fe. The default behavior is to flush back to the database server or release objects to make room if the cache is getting full. To make sure certain sets of objects are never released from the cache you can use "pin regions" defined by the `beginpinregion()` API. Note - you can also pin individual objects. Of course, if your cache is sized large enough, committing with `COMMIT_AND_RETAIN` will always ensure all objects are retained in cache.

3. Logical Databases

A logical database encapsulates the idea of using a virtual database instance in your application that is really composed of many separate physical databases. Once you've created a logical database, you can use it in query operations to execute in parallel across all physical nodes.

3.1 Dependent Query Implementation

Logical Database concepts are supported by the VQL 6.0 Query implementation. VQL 7.0 introduced OrderBy which is not implemented in the logical database concept. Depending on when you are reading this FAQ, it is possible that VQL 7.0 now accepts a logical database. Please check your release notes or use VQL 6.0 for your testing.

3.2 Logical DB Example:

```
const char* defaultDBName = "FirstDB";
const char* connectedDBName = "Nth_DB";
const char* virtualName = "VirtualDB";
LinkVstr<Player> found;

...code to begin session and connect to databases
```

```
session->newlogicaldb(virtualName);
session->addtologicaldb( virtualName, defaultDBName );
session->addtologicaldb( virtualName, connectedDBName );
```

```
// THIS QUERY LOOKS IN ALL DATABASES IN LOGICAL DB: virtualName
found = PClassObject<Player>::Object().select
    (virtualName,TRUE,Pattribute
    ("Player::name") == "SPECIAL");
```

4. Server Configuration

These are parameters set in a file called profile.be. This file sits in the database directory and is read at startup of the database. Simply stop and start the database (CMD>stopdb -f dbName : CMD>startdb dbName) to allow the database to read your changes.

4.1 Creating and Sizing of Versant Databases

The default capacity of a newly created Versant database is 128MB; no doubt you will need to create larger databases. Versant utilizes two log files used for recovery purposes, these log files default to 2MB in size. If you update a large amount of data in a single transaction, you will want to make these log files larger than 2 MB (rule of thumb is to make them large as the largest transaction times the number of concurrent update sessions).

Assume you wish to create a 2 GB database named "mydb" with 50 MB logs. Simply perform the following steps:

- 1) From command line run: "makedb mydb"
- 2) Update profile.be in database directory with the following changes (changes in *italics*):

| | | |
|---------|--------------|--------------|
| sysvol | 2048M | system |
| plogvol | 50M | physical.log |
| llogvol | 50M | logical.log |
- 3) Then run: "createdb mydb"

In addition, you can add more physical files to keep data in called "additional volumes". There are a number of ways to accomplish this, but the easiest way is to use the command line utility, addvol. Here is an example usage to add a 2 GB volume:

```
addvol -n volume2 -p volume2 -s 1024M mydb
```

This volume would be created in the database directory; if you wish to create the volume somewhere else, you have the option of using a full path name for the -p parameter (i.e. -p /disk1/mydb/volumes/volume2).

4.2 Enabling Asynchronous Commit

More than likely, you do not want the client application to block on I/O when issuing a commit call. In order to enable asynchronous I/O you need to add the following parameters to the profile.be.

```
//Enable asynchronous I/O
commit_llog_sync -1
//You can vary the integer - number of batched commits
commit_llog_flush 100    ...note - ignore existing "off"
```

4.3 Increase the total memory space of database cache

The default memory cache allocation for the database is only 16M. So, you will likely need to increase this (max_page_buffs) for your testing. The number represents a number of 16K pages. So, 1024 is really 16M of memory. For example, if you want 1G of cache memory allocated, set max_page_buffs in the profile.be as follows:

```
max_page_buffs 65536. // Make sure your OS is configured appropriately
// Set "heap_size" to this value times 16K, it will fail on start if OS cannot provide this amount of memory.
```

4.4 Increasing transaction memory space

Especially if you are using asynchronous commit, you will need to increase the size of the memory buffers used to hold transaction records. Versant uses piggy backed double buffers for transaction logging, which means many transactions can get put into 1 buffer while another is doing I/O. The result is a "batching" of I/O operations when many concurrent threads are performing transactions. When using asynchronous commit, this operation is certain. So, increase the following parameters in the profile.be. The actual value should be: 2*commit_llog_flush*size_of_objs*number_of objects_in_tx

```
llog_buf_size 100M
plog_buf-size 100M
```

5. General Application Performance Programming Note

5.1 Changing the RDB mindset

The following provides a discussion of the key conceptual differences in developing applications with an object -vs- relational database management system.

OBJECTS -VS- RESULT SETS

Transactional application design over that past decade has traditionally been done using relational databases. Relational databases have at their core the notion of returning “data” to the application as a result set. This result set is the product of queries performed against the database in the form of SQL, JDBC, Stored Procedures (database specific SQL), PSQL, etc. Result sets of “data” are then mapped into objects that represent the state of the application at runtime.

Queries and their ensuing result sets still play an active role in application design when using object oriented databases. However, when using an object oriented database, the notion of a result set changes. Result sets are no longer represented by the retrieval of “data” that must now be mapped into objects, but instead by the actual objects found as the result of your query execution. The mapping effort is no longer required.

Further, transactional applications designed with a relational database mandates the use of a query facility and their affiliated result sets as the sole means of access to information in the database. When using an object oriented database, the query facility is not the only means of retrieving information from the database and in fact plays a substantially diminished although still important role in application design.

5.2 Navigation -vs- Query

Using an RDB, the dependency on a query facility means a careful consideration must be given regarding the access of any application object. This is true because underlying that object access must be a database query incurring network overhead and server side computation of the result set. The efficiency with which you perform the queries becomes critical to application performance. Considerations such as pre-loading result sets of data to reduce the number of physical queries to the database and the physical layout of tables in the relational database including duplication of data, indexing, partitioning, etc are critical.

Using an object oriented database, queries should be generally limited to the retrieval of only a small number of key objects. There are perfectly valid application designs that only query to initialize some set of “root” objects at application startup and then never query again. However, more often than not, retrieval is associated with a particular use case and query is used to retrieve some set of relevant root objects and then *navigation* is used to access the graph of related objects.

So, if query is limited to some small number of objects, then the question arises, “how do we retrieve the rest of the ‘data’ in the application?” The answer is that ‘data’ comes in the form of an object’s state and objects are retrieved via *navigation* of your object model. Navigation is the retrieval of related objects through the normal processing of business logic.

Navigation is physically manifested by using language specific constructs, the object message send, to retrieve related information. In your application design, getter methods are executed and related information is returned in the form of objects. This is possible, because object oriented databases implicitly assign and manage object relationships on behalf of the developer. Further, an OODB understands how to interact with the message send construct and using those relationships, knows how to interact with the database and cache for object retrieval on behalf of the developer.

As relates to performance, when relationships become more complex, navigation becomes vastly superior to query. The reason is that navigational access is a relatively constant time operation. While query operations degrade substantially as the number of JOIN operations increase.

6. Navigational Concepts

6.1 Closure defined

Given a starting point object or set of starting point objects, closure is defined as the identification and retrieval of related objects to your application process relative to that starting point.

Closure is important to the performance of applications using an object oriented database. By default, closure is performed at runtime while performing navigation during the processing of normal business logic in your application. Each time you use a message send within your application business logic, the Versant object manager determines if closure is needed (object not currently in cache) and if necessary will take the required action to locate the related object from the database into cache. This process of closure on a single object is also known as a dereference of the object.

For example, lets consider an HR application that knows about departments each of which has a collection of employees. Lets consider a use case where the application is processing a bonus for all employees within the engineering department. First, the use case could query to retrieve the department named engineering. Once you have the department, you would then execute normal business logic navigating your object model and probably send a message `getEmployees()` to the department which would return the collection holding the employees. You might then get an iterator on the returned collection object and begin to iterate across each employee in the collection and send the message `giveBonus()` to each employee. As your application performed this business logic, each time a message send is made, the Versant object manager determines if the related object is currently cached and if not, it does the dereference. Therefore, when you send `getEmployees()` a dereference occurs and the collection object is located in the Versant object manager. Then as you iterate the collection and send the message `giveBonus()` a dereference is occurring and each employee is being located into the application memory.

This transparency of object access eases the developer's efforts by hiding all of the work necessary to find the related object in the database and return it to the application process. However, one can imagine how this could be terribly inefficient for certain use cases. What if the engineering department has 3,000 employees? An individual dereference is occurring for each of the 3,000 employees as closure is obtained while processing your use case. Each dereference is comprised of a network RPC and constant time lookup in the database server, perhaps even some disk I/O to bring the related object into memory.

6.2 Breadth -vs- Depth loading

Helping the closure process to be more efficient can greatly improve the performance of any application using Versant. This is in many ways similar to pre-loading result sets when using an RDB though as you will see improving closure efficiency is far easier.

Closure efficiency can be improved for a use case by the application of two simple API calls. One API is typically used to cover "breadth" loading and the other covers "depth" loading and they are known as `getobjs()` and `getclosure()` respectively.

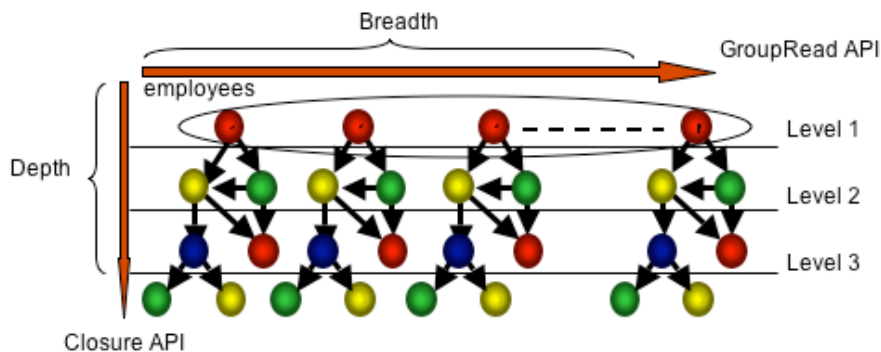


Fig. 1 – Breadth –vs- depth loading

Breadth loading is done when you have a target set of objects that will be touched during a specific use case. Versant does not by default dereference any object until you send a message to it in your application logic. Referring to the HR application example used above, when the message `getEmployees()` is sent to the department, a single collection object is returned to the application process. The other objects in the collection, the employees, are not loaded. In this case, you could use the closure call `greadobjs()` with the collection of employees to more efficiently load the employee objects into the application process. The `greadobjs()` call will result in a single RPC that will cause all objects in the collection to be streamed to the client side application process. Now when you create your iterator and iterate through the employees calling `giveBonus()` all activities will be on objects in your application process memory and no interaction with the database server will be required.

Depth loading is done when you have an object or target set of objects and you will be touching not only those objects, but related objects down to some specific level relevant to your use case. Using the same HR example, imagine that you are updating records for employees health care policies to a new insurance plan and recalculating partial payment rates based on the number of dependents for the individual employee. In this case, you will have to access related insurance Policy objects and perhaps related Dependent objects. In this case, the `getclosure()` call can be made with the collection of employees and a depth level specified such that the employees and related Policy and Dependent objects are streamed back to the application process very efficiently. Here again, when you process your business logic all objects will be in memory and no interaction with the database server will be required.

6.3 Fan out of standard depth loading API

There is one more level of efficiency that is being left out of the above discussion on `greadobjs()` and `getclosure()`. The `getclosure()` API call has no intelligence regarding the path below the starting point objects. If you have a large fan out of objects below the starting point then you need to realize that all referenced objects down to your specified depth level will be read into the application process. In the situation where your use case is only going to touch a limited number of objects down a particular path, you may be needlessly transferring large numbers of objects. The picture below illustrates this case.

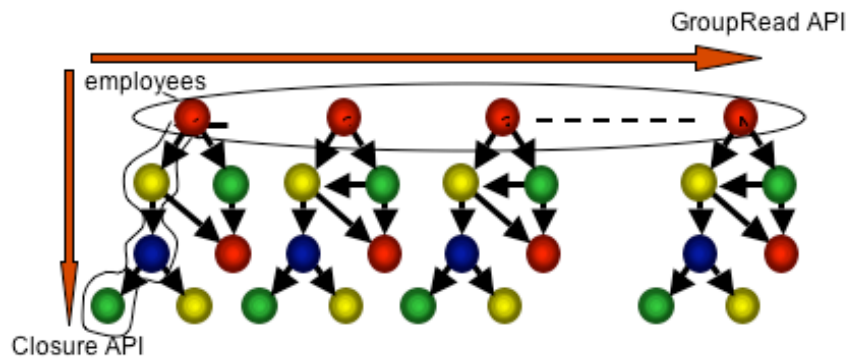


Fig. 2 – Touching a limited number of objects down a particular path

Referring to the above illustration, if you were to specify a `getclosure()` call with a depth level of 4 then all objects shown in the illustration will get loaded into client memory. In this case, there are 3 extra objects from each root employee that gets loaded across the network and yet will never be used by application logic. If you can eliminate this needless transfer, you might improve the overall bandwidth utilization of your network and in addition may speed up the response time of your application.

As it turns out, the internal implementation of the `getclosure()` call is using a `greadobjs()` call for each level of closure. In the above example this means that a total of 4 RPC's are actually getting called to retrieve all objects to client side memory. If you were to take control of this closure activity and accumulate object references at each level on your own, you could emulate an intelligent version of the `getClosure()` call that is sensitive to your path of interest.

CONCLUSION

This paper discussed some essential information that will allow you to successfully start using the Versant object database for basic evaluation testing on MMOG types of applications. In particular, being aware of the very basic configuration points for data storage, memory buffers and asynchronous I/O will enable you to scale your initial testing to more realistic data and concurrency sizes. Further, an understanding of the key concepts of caching and navigation will get you thinking about how you can truly use the power of object oriented design in your efforts. Finally, using the simple closure API's in your implementation, you will have extraordinary performance and network optimization at your fingertips.

NON-SQUARE DATA MANAGEMENT.

Get rid of rigid row and column structures when it comes to storing and retrieving complex data. Release the full power of a consistently object-oriented software application design. Non-square data management with Versant's object database technology – rapid development, high performance and massive scalability.

Versant's Object Database Management Systems (ODBMS, OODBMS) are used in a wide variety of industries to store and access hierarchical, and graph-oriented data in Java, C++ and .NET applications. It helps companies to handle complex information in environments that have high performance and high availability requirements. Using the Versant Object Database – instead of traditional relational database systems – customers cut hardware costs, speed and simplify development, significantly reduce administration costs, and deliver products with a strong competitive edge.

Versant USA

Versant Corporation
225 Shoreline Dr, Suite 450
Redwood City, 94065 CA, USA
+1 800 VERSANT
info@versant.com

Versant Europe

Versant GmbH
Wiesenkamp 22 b
22359 Hamburg, Germany
+49 40 60990-0
info@versant.com

www.versant.com

© 2001-2007 Versant Corporation. All products are trademarks or registered trademarks of their respective companies in the United States and other countries.

The information contained in this document is a summary only. For more information about Versant Corporation and its products and services, please contact Versant.