ODBMS.ORG www.odbms.org

Editor: Roberto V. Zicari

TechView Product Report: Versant Object Database.

Nov 1, 2008

Product Name:	**Versant Object Database**
Company:	**Versant Corporation**
Respondent Name:	**Robert Greene, V.P. Product Strategy, Versant Corp.**

1. Support of Programming Languages

How is your product enabling Java. .NET/C#. C++, developers to store and retrieve application objects? What other programming languages do you support?

The Versant Object Database enables developers using OO languages such as Java, C# and C++ to transactionally store their information by allowing the respective language to act as the DDL (Data Definition Language) for the database. In other words, your in memory model is the database schema model.

In general, the core of what is necessary to achieve persistence, is to declare the list of classes in your model which you want to be stored in the database, then provide a relatively thin transaction demarcation API to your use cases. Respective language integrations adhere to the constructs of that language, including syntactical and directive sugars.

Addition API's exist, beyond simple transaction demarcation, providing for the more advanced capabilities necessary to address practical issues found when dealing with performance optimization and scalability for systems with large amounts of data, lots of concurrent users, network latency, disk bottlenecks, etc.

Versant also has language support for Smalltalk and Python users.

2. Queries

Does your product support querying? If yes, can you describe the querying mechanism, giving examples of queries?

Yes, Versant supports queries via a server side indexing and query execution engine. Query support includes both a Versant specific and standards based query language syntax. Versant provides this query capability in a number of forms depending on the developers chosen language binding. For example, in Java we provide VQL (Versant Query Language), JDOQL, EJBQL and OQL. In C++ Versant

provides VQL and OQL, with C# support for VQL, OQL and LINQ. Versant will do optimization of query execution based on available attribute indexes.
Versant also has support for standard SQL queries against the Versant database using ODBC/JDBC drivers through the Versant SQL product known as Wistle.

Here are some simple language examples:

Example Versant native C++ query (standard based examples adhere to the standards, so look at those specs if you want to see their syntax):

The basic form of a C++ query looks as follows:

```
VQuery query(vql, "dbName@hostName" )
query.compile();

VQueryResult<Singer> result;
query.execute(result);
query.release();
```

The above parameter vql can be substituted with any valid query string. For example, the following which returns a collection of Person objects who's best friend is another person as opposed to a Pet:

```
"select selfoid from Singer where Person::bestFriend isa Person"
```

Or something more complex like:

```
"select selfoid from Person order by Person::ssn * 3 + Person::address-
>Address::zip DESC"
```

Or something with runtime bound parameters like:

```
"select selfoid from Person where Person::ssn + $1 > $2 and Person::name
like $3"
```

Where the parameters are bound with code in the following way:

```
int a=1000;
int b=3000;
const char * name="Tom*";

VQueryParameter param1(a);
VQueryParameter param2(b);
VQueryParameter param3(name);
query.set_param("1",param1);
query.set_param("2",param2);
query.set_param("3",param3);
```

Example Versant Native Java Query (standard based examples adhere to the standards, so look at those specs if you want to see their syntax):

The basic form of a Java query looks as follows:

```
Query query = new Query(session, vql);  //session is a db connection
QueryResult result = query.execute();
result.close();
```

The above parameter vql can be substituted with any valid query string. For example:

```
"select selfoid from model.Singer where bestFriend isa model.Person"
```

Or something more complex like:

```
"select * from model.Person orderby (ssn + 10) DESC ,(dateOfBirth * 2) ASC"
```

Or something with runtime bound parameters like:

```
"select selfoid from model.Person where ssn + $ssn1Param > $ssn2Param and
name like $nameParam"

Where parameters are bound with code in the following way:

      query.bind("ssn1Param", new Integer(1));
      query.bind("ssn2Param", new Integer(2));
      query.bind("nameParam", "Tom");
```

How is your query language different from SQL?

The native Versant Query Language (VQL) is very similar to SQL 92. It is a string based implementation which allows parameterized runtime binding. The difference is that instead of targeting tables and columns, you target classes and attributes.

Other elements of the OO paradigm apply to the query processing. For example, a query targeting a super class will return all instances of concrete subclasses that satisfy the query predicate. Versant is a distributed database, meaning you can define a logical database composed of many physical database nodes, so queries are performed in parallel when using logical databases.

Versant query support includes most of the core concepts found in relational query languages including: pattern matching, join, set operators, orderby, existence, distinct, projections, numerical expressions, indexing, cursors, etc.

Does your query processor support index on large collections?

Yes, though it is not necessary to have a collection in order to have a queryable object with a usable index. Unlike other OODB implementations, any object in a Versant database is indexable and accessible via query. Indexes can be placed on attributes of your classes and those classes can then be the target of a query operation. Indexes can be hash, b-tree, unique, compound, virtual and can be created online either using a utility, via a graphical user interface or via an API call.

Example of creating an index using a utility:

```
cmd>dbtool -icb bioinformatics.NonTransRegion attrName  dbName@hostName
```

Example of creating an index using Java API:

```
ClassHandle cls = session.locateClass ("mypackage.Person");
cls.createIndex ("name", Constants.UNIQUE_BTREE_INDEX);
```

Example of creating and index using C++ API:

```
PClassObject<Person>::Object().createindex("Person::name",O_IT_BTREE,
"dbName@hostname");
```

3. Data Management

Do you provide special support for large collections?

Yes, we have pagination support for large collections using a special node based implementation. These "Large" collections are designed in such a way that access is done so that only nodes needed by the client are brought resident into memory, instead of having to load the entire collection. These large collections are created and operated on just as the other persistent collection classes. The API is also consistent with the appropriate language constructs i.e. C++ STL, Java iterators, C# enumerables, etc.

A further point is that normal collections of objects by default are only a collection of object identifiers. So, these can be very large, yet have a small resident memory footprint. As you iterate the collection, objects are dereferenced into client memory space in either a configurable batch mode or one at a time. So, even in the case where you are dealing with collections of billions of objects, you can easily handle the load/release memory cycles as you iterate the contents.

If you are trying to query the contents of the collection, that can be done on the server using the "in" operator (or other set based operators like subset_of, superset_of, etc) without ever having to load the collection to the client memory space. As an example in C++, lets assume a person has a collection of phone numbers at a given address, then a query for people who's phone numbers contain the area code 925 would looks like the following:

```
"select * from Person where 925* in Person::address->Address::phoneNumbers"
```

4. How do handle data replication?

There are several mechanisms that really depend on the motivation behind the replication. It is for high availability or for distribution or integration.

High Availability: *Versant does synchronous pair replication. This requires zero code changes, just install one configuration file specifying the buddy node names and presto, you are now in full replication for fault tolerance.*

The way it works is as follows: New connections notice the existence of the replica file and on connect, check the file for a buddy pair and if it exists, connect to both buddies. Note, this could be a distributed database so that there are lots of buddy pairs. Then all transactional changes are committed synchronously to the buddy database server processes.

If any one of the databases in the buddy pair should become unreachable, the in-flight transactions are handled so that there is no commit failure, instead in-flight transactions on node failure will continue to the node that is still alive in the buddy pair. On the machine where the node is still alive and processing transactions, a

new process will start that monitors for the crashed database to become accessible again. Once the previously failed node is alive, the monitoring process starts replicating all changes that have occurred since the time of failure to bring the two buddies back into full synchronization. Once they are in full sync, a flag is set and on the next transaction clients will move back to full synchronous operation. All of this is handled without any user involvement.

In the case of extreme failure, like a broken disk drive, etc, the replicated node can be recreated from an online backup of the live node. Simply install a new disk drive, take an online backup of the live node, restore on the failed machine, start the monitor to sync the last few transactions and restore full replication at clients.

Distribution: *This is handed using Versant Asynchronous Replication (VAR). This is a channel driven, master-slave or peer-to-peer replication framework with rule based conflict detection and resolution.*

The way it works is that an administrator uses a utility to define replication channels. Channels are named entities that define a scope of replication within a physical node. The "scope" can be anything from full database replication to something as fine grained as anything definable by a Versant query. Once the channels are defined, applications can register as listeners on these channels, at which point changes from those channel begin to flow to the respective clients.

These channels provide both persistence and reliable messaging. So, in the event that a connection is lost between a registered listener and a channel, ongoing changes will be guaranteed delivery once the connection is re-established. There are multiple transport protocols that can be configured for optimization in highly reliable LAN networks or high reliability in unreliable WAN type of environments.

In the event that you have bi-directional channel replication, a set of conflict detection rules are put in place so that conflicting changes can be resolved at runtime without disrupting channel activity.

Note: There are other forms of data distribution for various operational purposes.

For example, there are utilities like vcopydb that will make a heterogeneous copy of a database for use in QA or reporting or development purposes.

```
cmd>vcopydb soucedb@hostname targetdb@hostname.
```

There are binary dump or step format or XML export utilities that can be used copy all or a portion of a database for import into other operational data stores.

There are incremental warm standby back facilities where incremental backup snapshots can be taken and applied to a standby database that can be used both for reporting and as a warm standby in the event of a prolonged product outage.

Integration: *It is not clear what this means given the context of data replication. You may want to refer to the section below on Integration with relational tables. In general, integration requires some kind of custom code. If you are using Java or C#, you can connect to both relational and Versant databases using ORM products. You can then load objects either from a relational database or Versant and then*

with some minor code implementation, disconnect those objects from the source and write them to a target. This can be used for import/export purposes in a batch processing mode for integration with other database systems. If you are using C++, we have a special consulting solution that will allow your application to talk with a relational database.

How do handle data distribution?

This question is a bit unclear. You may want to refer to the above. If what is meant here is, "how do you handle distributed data processing", then we can talk about Versant's data distribution architecture.

In general, we handle distributed data processing using a distributed 2 phase commit protocol across multiply connected databases. In this process, we have our own internal resource manager that is handling the distributed transactions. Versant also supports the XA protocol allowing external transaction monitors to control our transactional context, so for example you can plug into a CORBA or J2EE application server.

Versant allows object relationships to span physical resource (database) nodes. So, you can have shared information referenced from object graphs that reside in other databases and resolution of that information is transparent at runtime. For example, you may have several physical databases holding user information models that are partitioned by acct number holding aggregations on account activities such as trades and then have some more databases holding actual trade models and these users and trades can be related. So, you might do a query across all of the user databases and return a user (or set of users), then as you perform message sends to the user objects involving trades, the trade models will automatically be resolved across the distribution. If you perform updates of any of those objects, at commit time Versant will ensure that all changes commit back to their respective physical nodes in a completely ACID 2phase commit process.

Note - Object id's are guaranteed to be unique across all physical nodes. Objects could be "moved" (migrateobjs(objs[ ], fromDB, targetDB) from one physical node to another without any application code changes required.

An example in C++ would look like the following:

```
const char* defaultDBName = "P1_DB@hostName";
const char* morePlayerDBName = "P2_DB@hostName";
const char* tradesDBName = "T1_DB@hostName";
const char* virtualName = "VirtualDB";
LinkVstr<Player> found;
Link<Player> player;

….code to begin session and connect to databases

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

```
// THIS QUERY IS PARALLEL AND IN ALL DATABASES IN: virtualName
// In this example, would also include T1_DB, so players could also reside there too.
// Note you can create any number of logical databases with varying combinations.
found = PClassObject<Player>::Object().select
                  (virtualName,TRUE,Pattribute
                  ("Player::acct_num")  == 02321121 );

…..now access "found" contents (Players) and anything related
to a player will be resolved across all physical databases.

Player = found[0];
LinkVstr<Trades> trades = Player->getTrades();
Link<Trade> trade = trades[0];

…….now update anything you want in the model.

trade->incrementTotal(1000);
player->setLastAccessDate(today);

// HERE ALL CHANGES ARE 2Phase COMMITED BACK TO ORIGIN
session->commit( )
```

How do you handle schema evolution?

Schema evolution is handled via the normal update of your application class models and then applying those changes to the operational database. Those schema changes can be applied to an existing database either via utility or API. The result is a versioning of the database schema which occurs in sub second time regardless of the actual size of the database.

Existing objects in the database are now lazily evolved to the latest schema version. The approach is that no object is actually evolved unless it is made dirty (marked for update) and committed back to the database. So, in general this means an application with the new schema will not cause evolution, expect for new and updated objects.

There are utilities that can "crawl" a database slowly evolving all instanced to the latest version by grabbing sets of them, marking them dirty, committing. This is sometimes desired for embedded or real-time systems where every ounce of performance and space needs to be optimized.

In most cases, older clients get patch updates with the new schema in conjunction with updates to the server. So, the clients schema version is in sync with the database server. However, you can also use Versant's loose schema mapping facility. This is enabled by a flag in the client so that it does not complain about a mismatch in schema version and instead filters the incoming objects to match the old schema. Clearly, using this facility requires some forethought to avoid any unintended side effects.

The process goes as follows:

1) *You update your class definitions, i.e. add new subclasses, add attributes, rename attributes, remove attributes, etc and recompile. Now when the application connects to a Versant database, we will detect a schema version mismatch and you would normally get an error unless you take some action to avoid the mismatch.*
2) *The schema mismatch can be avoided using a number of techniques.*

   One, you can use a utility to describe the new schema to the database. The utility will show a list of incompatibilities and ask how you want them to be resolved. Your action will depend on whether you are in development, QA, production, etc. Regardless, you can do things like drop the existing class, evolve the schema version and keep all existing objects, rename and retype, etc.

   Two, you can automate the evolution process via connection options. This is normally used in development mode and allows the schema to automatically evolve any mismatches on connect and continue preserving the existing objects.

   Three, you can use specific API's to dynamically evolve the database schema. This is an advanced topic, involving what we call Versant runtime classes. Basically, you can create completely dynamic schema structure for the database so that you can create new classes and attributes on the fly as part of your normal application processing.

3) *If you are going to have clients with the older schema continue to operate on the database, you should set loose_schema_mapping in the application profile file to true.*
4) *Optionally, start a utility to crawl the database and force version migration of all existing instances.*

The general guidelines for schema evolution are that any schema changes can be made and existing instances preserved, without having to write custom evolution code, with the exception of two things. 1) Changes to the middle of an inheritance hierarchy. So, you cannot do something like insert a new class into the middle of a hierarchy, without loosing your existing objects, unless you write custom code to do this operation in a series of steps. 2) Incompatible type changes like Array to a String.

All other forms of evolution like renaming attributes, deleting leaf classes, adding leaf classes, adding new classes, adding or removing attributes, etc can be done online and without custom code.

If you have a need to do things like set non standard default values for newly added attributes, you can do this in callback functions within your objects. There are a set of standard object lifecycle callbacks that get invoked in activities like cache load. You can use those callbacks to check for default values and take action if necessary.

Please describe an object lifecycle when an object is loaded from a database: When are members of an object loaded into memory? When are they discarded?

The lifecycle of an object load can be controlled on a use case basis.

By default, objects are loaded only when they are sent a message. This includes the default behavior for queries which only return a collection of references to objects that satisfied the query predicate, not the actual objects. When an object is loaded, all it's non-reference attributes (primitives) are also loaded and remaining reference types follow the same pattern as the referencing object.

When a message is sent to an object we look into internal structures to see if the object is already in client memory. If not, we do an RPC to load the object. At the time we load the object, we will also look at the connections locking strategy to decide how to deal with locking the object on load. We have both global locking strategies that can be applied to your connection and extremely fine grained control to override behavior for a particular use case.

Once an object is loaded and locked it stays in the client cache, with an equivalent lock in the server, until one of a number of events occurs.

The most common event, the current transaction ends with commit. In the default case, this will release the lock and object from memory. However, note that there are forms of commit that will do combinations of things like, keep the cache and the locks and start a new transaction, keep the cache, but release the locks and start a new transaction. These forms and others are used to optimize cache effectiveness when using non-default locking strategies like optimistic locking or when you have a series of transactions that form a task and operate on the same set of objects.

Another possibility is that your client cache starts to get full. In this case, we may decide to swap objects back to the server process to make space and do some work that will have to be done at commit anyway. We do this in a fully transactional way, so that even if modified objects get swapped to the server, they will still be undone if the transaction is rolled back. Also, you have the ability to "pin" objects into the client cache to prevent swapping of important sets of objects, enabling the use of direct memory pointers without concern for memory faults.

Another possible event is a query call which has the option set to flush the cache of objects in the target class, so that changed objects currently in your cache become part of the current query execution evaluation.

Other possibilities include API calls that result in explicit release of the object, like a call to refresh or a call to release.

There are many ways to override the default behavior. Those are in fact commonly used to performance tune on a use case basis. For example, if you are going to iterate over a collection of 1000 objects, you don't want to do 1000 RPC's. So, you can give the collection of references to a call to groupRead which will use a single RPC and load all objects. Similarly, you can make a call to getClosure which will use groupRead behavior to load all referenced objects in a graph from the starting point, down to your specified level of reachability. Further, queries have options to

set a lock and load result sets rather than just references or to use cursors. There are API's to explicitly load objects into cache and set higher lock levels than the connection defaults, etc.

5. Data Modeling

Which Data Modeling notation do you recommend for the design of your data?

Versant does not have any particular recommendation in this area. You can use whatever makes sense for your effort. In the end, the implementation model in the language of choice becomes our DDL, so ultimately that becomes our data model. We have seen users take varied approaches including using Generic modeling to Entity-Relationship modeling, Object Role modeling, Semantic modeling, etc.

6. Integration with relational data

Could you integrate flat relational tables into your object database? If yes, how?

A large percentage of Versant's customers do some form of Integration with relational tables. This can be accomplished in a couple of ways depending on the requirements such as: on-line/off-line, batch based, transactional, etc.

XA:
Versant supports the XA protocol for distributed transactions. This allows Versant to participate in online distributed transactions with relational databases. The form of interaction with the relational tables can take many forms from custom code to ORM solutions to J2EE application servers (Entity Relationship Modeling) to message passing to ORB's, etc. The XA API allows the Versant database to act as a resource controlled by an external transaction monitor coordinating changes to both Versant and relational databases in the same transactional context.

ORM:
Versant can interact with relational databases using Java ORM technology such as JDO (Java Data Objects) and Hibernate JPA. These standards based implementations have the ability to detach objects from their transactional context and then attach them to another connection. There are restrictions in that Versant requires the application to use a concept known as database identity in order for replication to work with relations intact. Versant does not support the ORM form of application identity in anything other than a disconnected data form.

XML:
Versant has tools that enable the import and export of XML data. So, for example, batch based replication of data can be accomplished by exporting objects from the Versant database in the form of XML, if necessary applying an XSLT transform and then importing into relational tables. Of course, the opposite is also possible.

In addition, with Java, the most common approach using XML is to dynamically replicate information using JAXB which runtime converts objects into and out of an XML form. Using JAXB, the Versant database only needs to work with objects

rather than importing an XML form. In essence, XML coming from relational databases are converted to objects at runtime using JAXB and those objects are then persisted into the Versant database.

Custom Code:

Users of C++ are especially challenged in integrating with relational databases. Versant provides consulting productized frameworks to help these customers with their integration challenges, but does not make those solutions, which require customization for each application, available in a productized form.

7. Transactions

How do you define a transaction? Pls. use a simple example.

Versant by default is always implicitly in a transaction when connected to the database. So, for example:

In JVI (Java Versant Interface):
//establish a connection to a database and implicitly start a txn
TransSession session = new TransSession("dbName@hostName");
.....do some work

//commit all work done and implicitly begin a new transction
session.commit();
....do some more work in the next transaction and commit again.
session.commit();

In C++:

*VSession *aSession;*
```
char* db_name = "dbName@hostName";
```

//establish a connection to a database and implicitly start a txn
VSession aSession = new VSession(db_name, "session_name");
.....do some work

//commit all work done and implicitly begin a new transction
aSession.commit();
....do some more work in the next transaction and commit again
aSession.commit();

In addition, we support the XA protocol and apply that to certain standards based API' such as JDO and JPA which require explicit transaction demarcation. So, there is a non-implicit form of transaction where transaction begin/end must be declared.

Is there a way to discard objects that have been modified in the current transaction from memory?

There are several ways to do this in Versant. You can do it globally for the current transaction by issuing a rollback which also implicitly starts another txn or you can

do it in isolation or globally using specific calls within the same transaction. For example:

```
//end current txn, restore modified objects in cache to original state, start a new txn
session.rollback( );

//explicitly release modified object from memory
session.releaseobj( objRef );

//explicitly release modified objects from memory
session.releaseobjs( Object[ ] );

//explicitly undo modification to an object in memory
session.refreshobj( objRef );

//explicitly undo modification to objects in memory
session.refreshobjs( Object[ ] );

//explicitly send/release modified objects to the server, but don't commit
session.flush( Object[ ] );
```

Do you provide a mechanism for objects on clients to stay in synch with the committed state on the server? If yes, please describe how it works.

Versant by default uses a pessimistic locking strategy to ensure that objects in the database server are in sync with client access in an ACID way. This is done by using a combination of locks against both schema and instance objects. In brief*, the database server process maintains lock request queues at the object level to control concurrency of access to the same object. A request for update will establish a queue if there are any existing readers of an object. The request either goes through when all current readers release their locks or times-out (an exception which can be handled by client is thrown). Locks are generally released at transaction boundaries. When a queue is established by an update request, all other subsequent requests fall in queue behind the update request. Once the update request has been filled, all read requests in the queue rush in and get their read lock, return the object, and if there are no other updates, the queue disappears. In this architecture, locks are done at the object level so false waits and false deadlocks do not occur.*

Note that Versant supports other ways of keeping client caches in sync. For example, we can use an optimistic locking strategy, using a classic timestamp mechanism. We also have forms of client cache synchronization using multi-cast. We also have an event mechanism where clients can register for triggering events within the database server to be used for synchronization or for business logic work flow.

How are transaction demarcations for your product expressed in code?

See the above examples. Except when using our XA API, transactions are implicit with commit.

Describe the strategies possible with your product for concurrent modifications by multiple clients.

See the above locking and caching strategies.

8. Persistence

Are there requirements for classes to be made persistent? If yes, please describe them.

For users of C++, Versant requires that the upper most class in an inheritance hierarchy inherit from a base class "PObject", which handles database activities. So, the typical header file will have something like this:

```
class Player :
      public PObject
{
```

Then there is a file setup schema.imp that declares which classes in your model are to be made persistent and that file is used in a pre-compilation phase where Versants necessary magic is added to your persistent classes. Finally, the resulting schema.cxx file is compiled and linked with your application.

Schema.imp file will have entries that look like the following:

```
#include "Player.h"
O_CAPTURE_SCHEMA(Player);
```

The pre-compilation phase is done with a utility as follows, though note this is typically automatically setup in your visual development environment so the process is automatic whenever you do a build:

*schcomp.exe -DVERSANT_ANSI -I$(VERSANT_ROOT)\h - **compiler flags** schema. imp*

Does your product require enhancement of application classes for Transparent Persistence? If yes, briefly describe the additional steps required for a build process of an application.

When using Java or .NET this same thing described above with C++ is accomplished using post processing byte code enhancement. You setup a file that declares which classes are to be persistent and then use a utility, or API, or IDE integration to enhance the classes before running or debugging.

The file has entries like the following for each class:

```
a biodriver.Navigation
p bioinformatics.Genome
c bioinformatics.Centromere
n *
```

Where:

```
a – aware ( for people who don't use setters and getters in
non-persistent classes that use persistent classes )
p – persistent ( persistent at new )
c – capable ( persistent at makePersistent( obj ) )
n – not persistent
```

Note – Versant provides other Java APIs based on standards JDO and JPA. In those versions of the API we adhere to the standards defined for declaring persistence whether it be some kind of XML or annotation.

Enhancement is then done using a utility as follows (similarly with .NET) or more commonly you have an Eclipse plug-in or Microsoft Visual Studio integration which automatically does the right stuff during the build process:

```
cmd>java com.versant.Enhance *
```

9. Storage

Does your product operate against a single database file or are multiple files required?

Versant supports, multiple file and multiple process configurations. Data storage is done in a single or multiple files, but there are supporting files for the logging subsystem (logical and physical log files). These logging files are used for high performance and scalability under concurrent user loads and for online database backup processes.

10. Architecture

Does your product support multiple clients connecting to a single database?

Yes, as described above, Versant is a multi-user client server database and has production applications with 1000's of concurrently connected users. That being said, Versant can also run linked and embedded in the same address space as your application process.

For which Operating Systems is your product running?

Versant's platform support includes: Solaris, Linux, Windows (NT thru Vista), AIX, HP-UX. We support these platforms in both 32 and 64bit.

11. Application

What kind of applications are best supported by your product*?*

We often describe the "best kind of application" to use a Versant database as those applications requiring an application specific database of an OLTP nature. In other words, a non-traditional I.T. type of transactional application. That being said,

there are certain characteristics, which when exhibited in an application, indicate a stronger value add by Versant.

Those characteristics are: complex models, large amount of data, large number of concurrent users. Any one of those three characteristics starts down a path of Versant value and at the extreme end, where you have all of those characteristics, Versant provides clear distinguishing value. The whole reason Versant still exists is that we provide better performance and scalability for applications with the above characteristics over traditional relational technology.

To that end, Versant is found in applications within many different Vertical industries where those characteristics come into play. So, Versant runs global trading platforms for the worlds largest stock exchanges, network management for the worlds largest telecommunications providers, intelligence analytics for defense agencies, reservation systems for the largest airline/hotel companies, risk management analytics for banking and transportation organizations, massive multi-player gaming systems, network security and fraud detection, local number portability, advanced simulations, social networking, etc, etc as a representative set of industries and applications exhibiting those characteristics.

12. Performance

What benchmark do you use to measure the performance of your system?

Versant uses internal performance and scalability benchmarks to monitor and measure our behavior over time across releases, patches and generations of new hardware. It's an ongoing effort to improve in performance and scalability.

Versant has done other non-standard benchmarking activities in a public forum. For example, you can find a study done with IBM in 2002 when integrated with J2EE application servers using the following link to the IBM Redbook site:

http://www.redbooks.ibm.com/abstracts/SG246561.html

At one point in time, Versant ran the 007 benchmarks, but that is long outdated originating back in the early 90's. There are currently no industry benchmarks that make sense for object databases, so we don't run any of the benchmarks recognized as standards in the database industry.

Versant took a serious look at the latest TPC-E, which was supposed to be the new OLTP standard database benchmark with new complex models aimed at being representative of today's computing environment. The TPC-E is based on a financial trading system model.

Unfortunately, it was impossible to get real comparative results from TPC. The reason is that the TPC specifies requirements regarding what part of the code resides in the "driver" of the benchmark and what part resides in "database" functionality. However, the driver to application logic interface is completely defined at the data level. So, what this means is that when measuring relational access you would not incur any overhead for mapping into a C++ object. The

mapping of the raw data into what ever form was necessary in the driver to implement the business logic was completely outside of the benchmark measurements. Of course, when it comes to the object database, you need to now un-map the C++ objects into the driver data structures and in doing so, measure the cost of that activity as part of the benchmark timings.

This is the exactly the opposite of a real world application. In the real world, people write object oriented applications resulting in object oriented models. Now if you choose a relational database, you need to map/un-map from your objects to the relational data structures. The TPC-E was written in a way as to exclude the "mapping effect" from the measurements, which by the very nature of how an object database works means the TPC-E was written in a way that forces measurement of an "un-mapping effect", an activity which does not occur in a real world application.

Thus with TPC-E, the true cost of computing is removed for relational and even worse added to object databases. It was very disappointing because Versant had high hopes of finally delivering a public result representative of our technologies true value proposition, the reason we have been selected for so many of the worlds most demanding applications.
##