

Editor: Roberto V. Zicari

## **TechView Product Report: ObjectStore.**

Nov. 19, 2008, Revised August 2010.

Product Name: **ObjectStore**  
Company: **Progress Software Incorporated**  
Respondent Names: **Don White, ObjectStore Engineering  
Manager.  
Jonathan Leivent, ObjectStore Architect.**

### **Support of Programming Languages**

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

ObjectStore provides language bindings for Java, C#, and C++ that are equally comprehensive in their support for:

- Transparent persistence: all fetch, load/store, lock/unlock operations are fully automated and invisible to the programmer
- In-memory caching of persistent objects
- Fully serializable ACID transactional guarantees
- Page-level lock concurrency
- Distributed cache coherency management, including object and lock caching between transactions, and write-back LRU eviction for modified and unmodified objects
- Multi-Version Concurrency Control
- Collections, querying, and indexing
- Online Backup/Restore with archive logging
- Asynchronous replication
- Server failover
- 2PC, including XA and JTA interfaces

ObjectStore has two language binding mechanisms, one for C++ and C#, and the other for Java.

The C++/C# virtual-memory mapping-based mechanism enables the application to safely navigate through ordinary C++ pointers (no need for smart pointers) and perform pointer arithmetic freely (including the important C++ case of pointers to subobjects), yet with automatic fetching and appropriate read/write locking of persistent objects on access. Access and navigation between in-memory persistent

objects is machine-instruction equivalent to (and just as fast as) native transient object access and navigation. Address space recycling at programmer-specified points allows the application to access more persistent data within a transaction than would fit into memory at once. ObjectStore can persist almost any C++ type, including "plain-old-data" structs, individual fundamental-typed objects like "ints" and "char\*" vectors, and 3rd-party types not designed for persistence (such as STL), in addition to all customer-authored C++ classes. C# ObjectStore applications persist C++ objects, but can access them using native or CLR code. Whether or not an object is persistent is determined when it is allocated.

The Java mechanism is based instead on copy-on-access persistence by reachability. It uses class file code injection to provide transparent locking on object access. This mechanism is tailored to the needs of the Java relocating garbage collection memory model, and doesn't require any JVM additions. As with the C++/C# mechanism, the Java mechanism also allows the application to access more persistent data within a transaction than would fit into memory at once, but does so by leveraging the Java garbage collector combined with an internal automatic address-space recycling capability. A JDO interface is also supported.

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

Yes. ObjectStore has two query formats: our original one based on C++/Java expressions, and a new one based on XPath. ObjectStore can also accept ODBC queries using the external Progress OpenAccess product.

Here's a C++ example of a query using the original query format:

```
os_database *people_database;
os_Set<person*> *people = ...;
...
os_Set<person*> &teenagers = people->query(
    "person*",
    "this->age >= 13 && this->age <= 19",
    people_database
);
```

The above query is analyzed and evaluated in one step. It is also possible to separate query analysis, parameterization, and evaluation so that similar queries do not have to be analyzed individually, as in:

```
const os_coll_query &age_range_query = os_coll_query::create(
    "person*",
    "age >= *(int*)min_age_ptr && age <= *(int*)max_age_ptr",
    people_database
);
```

which creates and pre-analyzes a query with two parameters. This query can then be (repeatedly) bound to (different) parameter values:

```
int teenage_min_age = 13, teenage_max_age = 19;
os_bound_query teenage_range_query(
    age_range_query, // the pre-analyzed but unbound query from above
    (
        os_keyword_arg("min_age_ptr", &teenage_min_age),
        os_keyword_arg("max_age_ptr", &teenage_max_age)
    )
);
```

);

and finally evaluated (repeatedly, over different collections):

```
people->query(teenage_range_query);
```

Queries can also contain calls to string functions like `strcmp` and `strcoll`, comparison operators for which the user has defined a corresponding rank function, and member functions (that satisfy certain restrictions). Queries can also nest.

Query analysis performs query optimization and automatically takes advantage of any indexes that may exist for the collections involved.

### *3. How is your query language different from SQL?*

Our query language allows the use of non-member and member functions within the query. Queries return individual objects or collections of objects (not SQL tables), depending on the query. Queries can also be pre-analyzed and then repeatedly re-bound and re-evaluated.

### *4. Does your query processor support index on large collections?*

Yes. Indexes can be placed on arbitrarily large collections in ObjectStore.

## **Data Management**

### *5. Do you provide special support for large collections?*

In addition to supporting indexes on large collections, ObjectStore allows for iteration within a single transaction through collections that are larger than the available address space. For example, a 32-bit application could iterate through a 100 gigabyte collection, reading and/or writing every element, in a single consistent transaction, and commit all of those modifications at once.

### *6. How do handle data replication?*

ObjectStore includes an asynchronous replication feature, allowing multiple servers to maintain copies of a database. Replication is one way, so one master copy is updatable. The other copies are read-only. The read-only copies might be slightly out of date, but are always transaction consistent.

ObjectStore also includes an online backup feature to maintain multiple consistent backup versions of databases, with a continuous archive logging capability to keep the most recent backed up version nearly up-to-date.

### *7. How do handle data distribution?*

ObjectStore uses a client-server model. ObjectStore automatically maintains a cache of persistent data local to each client.

A unit of persistent storage could be present simultaneously in multiple client caches, as well as in the database hosted by the server. Clients access data from their private caches if it is available there, and request data from the server otherwise. Data is fetched from the server in page unit multiples, with a configurable pre-fetch option. Distributed data consistency is maintained through distributed cache coherency.

ObjectStore clients cache permissions to read lock or write lock pages across multiple transactions. A lock can be "owned" for read or write even while the corresponding page is not locked. Owning a lock for read or write gives a client permission to lock the corresponding page for read or write without contacting the server. This improves performance of transactions, removing the need for many messages between client and server, while remaining invisible to the application running on the client.

ObjectStore transactions can each access and modify multiple databases hosted on multiple servers, with full serializable ACID guarantees maintained across all servers. Any transaction that modifies data hosted by multiple servers will automatically use a two-phase commit (2PC) protocol.

### *8. How do you handle schema evolution?*

Two kinds of schema exist in ObjectStore: program schema and database schema. A program schema contains information about the definition of types within a program, including any ObjectStore-aware DLLs the program may use, and is constructed when the program is built. A database schema (normally part of the database itself) contains information about the definition of types present in the database, and is automatically populated from the program schema of the program(s) that populate the database. To prevent accidental corruption of a database, ObjectStore checks the types in the schema of a program attempting to access a database to see if they match the like-named types in the database's schema. If there is a mismatch, an exception will be raised with text detailing the mismatch.

As programs evolve, developers alter their decisions about the semantics of types within the program as they develop, maintain, and adapt the program over time to changing conditions. Evolution of database schema (and corresponding migration of data within the database to the target schema) is necessary to allow such modified programs to continue accessing the same databases.

In ObjectStore, the *ossevol* utility migrates an existing database and its schema to a new target program schema. The utility automatically constructs a mapping from the source to the destination schema, then executes this mapping on every affected object in the database. Due to the ambiguity of certain C++ constructs, *ossevol* provides an API that allows the programmer to design tailored evolution assist functions that are then used during the evolution process.

The *ossevol* utility builds a separate work database of intermediate state information so that if stopped it can be continued from where it left off.

Note that adding new types to a database does not require schema evolution. Such cases are handled transparently by adding the new types to the database schema at either the next database access, or when instances of the new types are created within the database.

*9. 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?*

In C++/C#:

Given a pointer to an object (acquired from another object, or from a database root), the first access of the object through that pointer in ObjectStore triggers a page fault, which will automatically request an appropriate lock on the database page containing the object, fetch the page, "swizzle" pointers within the page to correctly reflect the position of the pointers' targets within the application's address space, and map the page into the application's virtual memory to make the object accessible. If the page was initially accessed for write (if the very first machine instruction that attempted to access the page was a write instruction), it will be write locked, marked modified and mapped for write access. If the page was initially accessed for read, it will be read locked and mapped for read-only access. A read-only page that is subsequently accessed for write will be upgraded to write locked and mapped for write access.

One parameter for each ObjectStore client is its cache size - which determines the number of pages it can hold. If the cache size is N pages and the application attempts to access the N+1 page, it will undergo cache eviction, which will cause several of the least-recently used pages already in the cache to be evicted to make room for the newly accessed page (and several others). If those evicted pages were accessible at the time, their page frames are cleared and then protected against all access so that page faults will be triggered on subsequent access attempts. If the pages were modified, they will be "un-swizzled" and sent back to the server for safe keeping (the modified state can be fetched automatically again later in the same transaction).

Other than this cache eviction due to cache overflow, the other ways for a page to leave the cache are: explicit cache eviction by function call, and (when unlocked) callback due to another client attempting to modify the page. Otherwise, pages can remain in the client's cache as long as it remains connected to the hosting server with the owning database open - which can be for an unlimited number of successive transactions.

In Java:

Given a reference to an object (acquired from another object, or from a database root), ObjectStore makes sure that there is a "hollow" object of the proper type at the destination of the reference. By "hollow" we mean that its fields are currently uninitialized, and the object is marked as being hollow. An attempt to access the fields of a hollow object will cause it to "materialize" - which (by code injection into class files) involves a request for the appropriate lock (read or write), fetch, and copy of the object's contents into the "hollow" object in the heap. As with C++/C#, an object that is initially read will have a write lock requested for it on the first subsequent write attempt (through either field or method).

ObjectStore keeps weak references to unmodified objects, so that any unmodified object that becomes inaccessible by strong references from the application can be garbage collected by Java. For modified objects, ObjectStore keeps strong references, until the ObjectStore.evictAll() method is called. Calling ObjectStore.evictAll() causes

modified objects to be copied back to the server, marked as unmodified in the heap, and the strong references are exchanged for weak ones - allowing these objects to be garbage collected as well. Subsequent access to the fields of modified objects in the same transaction will re-materialize them by re-fetching the saved state from the server (in other words, evicting them doesn't lose data, just transfers it to the server to free up application memory).

## Data Modeling

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

We don't recommend any over any other. ObjectStore can persist any C++ or Java types created by any data model design tool.

## Integration with relational data

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

Since ObjectStore can support arbitrary data types and structures, one could in theory implement a whole relational database engine (tables, indexes, schema, etc.) within ObjectStore. In practice, that isn't done.

Integration is normally done between ObjectStore and external relational databases using the standard XA protocol (for sharing transactions but keeping data separate), or using Progress Software's DataXtend Semantic Integrator (DXSI), which automates distributed transactional data translation and sharing between multiple heterogeneous data sources.

## Transactions

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

ObjectStore has two transaction APIs - referred to as "static" and "dynamic" - names which indicate the type of code region they can encapsulate. In either case, a transaction consists of a starting call that specifies the transaction type, an arbitrary execution of application code in the body, and an end call that commits or aborts the transaction. Dynamic transactions can span multiple threads. Each client process consists of one or more sessions where each session is an independent sequence of transactions - allowing a single client process to execute multiple concurrent transactions itself. Transactions of either type can also be nested.

Here is a simple example of a static transaction in C++:

```
OS_BEGIN_TXN(txn1, o, os_transaction::update) {  
    // arbitrary application code to execute within the transaction goes here  
} OS_END_TXN(txn1);
```

Static transactions, because they encapsulate a static region of execution (everything within their block body), can automatically retry on aborts due to deadlock or timeout (or other restartable exceptions raised within them).

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

ObjectStore uses distributed write-back caching to allow a single transaction to modify an unlimited amount of data, even exceeding available RAM and virtual memory.

In C++/C#: An attempt by an application to modify more data than can fit into its cache during a single transaction will cause ObjectStore's least-recently-used (LRU) eviction algorithm to write back data modifications to the server. Address space can be recycled within transactions at programmer-specified points to avoid running out of virtual memory.

In Java: a method call (ObjectStore.evictAll()) can be used to cause modified objects to be copied from the Java heap back to the server, such that ObjectStore will no longer keep strong references to those objects. If the application also has no more strong references to those objects, the Java garbage collector can delete them.

In both cases: evicted modified data sent back to the server is held in the server's memory or in its log (or directly in the database itself, in the case of newly allocated database regions) fully isolated from access by other transactions. Subsequent re-access of that data by the application will fetch the accessed portion of that modified data from the server. The effect presented to the application is of an unlimited persistent heap without restrictions on the amount of data accessed or modified per transaction.

*14. 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.*

ObjectStore has two concurrency synchronization models: the standard pessimistic 2-phase locking model, and a multi-version concurrency control (MVCC) model. The choice is made individually for each database when opened, and can be changed by closing and re-opening the database in between transactions.

The standard model allows a single writer OR unlimited multiple readers to simultaneously access a single individually locked unit of persistent data (page). The MVCC model allows a single writer AND unlimited multiple readers to simultaneously access a single page.

The standard model maintains fully synchronous data consistency across client caches using a lock ownership caching and callback algorithm - any page that is in danger of becoming stale is evicted from a client's cache. An attempt by the application to modify data on a page for which the client doesn't already "own" a cached write lock will cause synchronous callbacks to request eviction of that page from all other client caches. Any client cache currently locking that page will deny the callback until it is finished with its current transaction, and so block the writer. Similarly, an attempt by the application to read data on a page not already present in its cache will cause a synchronous callback to be sent to any client cache that may

currently have "ownership" of a write lock for that page, with a similar blocking result if the write lock owner denies the callback.

The MVCC model allows read-only access to slightly out-of-date but still fully consistent persistent state. In this model, callbacks from writers to MVCC readers are used to inform MVCC readers of the availability of more recent state.

*15. How are transaction demarcations for your product expressed in code?*

Static transactions in C++ are demarcated as above in the answer to question 12. Dynamic transaction demarcations are function calls:

```
os_transaction *txn = os_transaction::begin(os_transaction::update);  
  
txn->commit(); or txn->abort();
```

Within Java, these are:

```
Transaction txn = Transaction.begin(ObjectStore::update);  
  
txn.commit(); or txn.abort();
```

The extent of dynamic transactions runs from the begin call to the commit or abort call within a single application session, where each thread is in one session at any point in time, and can jump between them.

ObjectStore also has a transaction checkpoint feature that allows a transaction to retain locks on pages across the checkpoint based on whether or not there is contention for those locks. A checkpoint is thus roughly equivalent to a commit, immediately followed by starting a new transaction and re-acquiring all uncontended-for locks held prior to the checkpoint. The checkpoint API is:

```
txn->checkpoint(); // in C++  
  
txn.checkpoint(); // in Java
```

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

The standard and MVCC models both allow at most one writer per persistent data unit (page) per point in time. ObjectStore attains modification concurrency by allowing data to be partitioned across multiple pages, and allowing an unlimited number of writers to simultaneously modify distinct pages. Application programmers can partition data according to use into different database files or data containers within databases (called segments and clusters) to guarantee concurrent write access.

## **Persistence**

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

ObjectStore places no requirements on persistent types at all, not even a requirement that all persistent types be represented as classes. ObjectStore can persist virtually any data type that can be allocated by an application, right down to individual "chars". Persistence in the C++/C# model is determined at object allocation by using a persistent overloading of the "new" operator, and in Java is based on a method call after allocation, or by reachability from other persistent objects. Any type can have both transient and persistent instances simultaneously within the same application.

For C++/C#, ObjectStore acquires knowledge of user and 3rd-party type definitions using a schema generator utility based on the Edison Design Group's popular front-end parser. For Java, ObjectStore uses data reflection and a post-compilation utility (also available as an Eclipse plug-in) to instrument class files.

*18. 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.*

No visible code enhancement is needed by ObjectStore to persist any type. In the Java model, code is injected into class files to enable object persistence. In the C++/C# model, type definitions are scanned by a schema generator (typically run as a build step) to give ObjectStore knowledge of the layout of those types. In either case, a single additional build step per application is all that is required.

## **Storage**

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

In ObjectStore, a single database is typically represented as a single file in the operating system. However, ObjectStore allows database schema to be stored in separate databases and shared (one database schema for many databases). ObjectStore also allows an application to use an arbitrary number of databases simultaneously, and allows relationships (pointers) to span across databases.

## **Architecture**

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

Yes. ObjectStore has support for many different configurations. A transaction can access multiple databases on one or more servers. Multiple clients can access the same database simultaneously. A single client process can even have multiple "sessions" which allow it to run multiple simultaneous transactions.

ObjectStore can also be configured to run as a single stand-alone process combining a single multi-session client with a single server.

*21. For which Operating Systems is your product running?*

- Both 32- and 64-bit versions of all Windows versions from XP forward.

- Both 32- and 64-bit versions of Linux (certified on Red Hat, compatible with other similar Linux) for Intel/AMD processors.
- Both 32- and 64-bit versions of Sun/Oracle Solaris for Sparc Processors.
- 64-bit HP-UX for PA-RISC Processors.

## **Application**

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

Applications that need transactional access to very large and complex persistent data graphs, often including persistence-unaware types (and associated procedures) authored by 3rd parties. Applications that require in-memory access and navigation speed for persistent objects. Typical customers are in finance, utilities, defense, telecom, travel, information services, geographical information management, and web services.

## **Performance**

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

For internal testing purposes, we use internal benchmarks. In the remote past, we were involved with standard object database benchmarks (such as oo7), but these are too generic to provide insight about how we are meeting our performance goals.

## **"Useful Resources"**

ObjectStore Product page: <http://www.progress.com/objectstore/index.ssp>

Cache-Forward Architecture Introduction:

[http://www.progress.com/objectstore/publications/cfa\\_intro/index.ssp](http://www.progress.com/objectstore/publications/cfa_intro/index.ssp)