OODBMS ARCHITECTURES

An examination of implementations

Robert Greene - rgreene@versant.com
V.P. Product Strategy, Versant Corporation

Abstract:

The Object Oriented Database Management System (OODBMS) has been in existence now for nearly 2 decades. The major vendors conceived and began implementing them in the late 80's accommodating OO languages like Smalltalk, C++ and Java, with commercial products shipping in the mid 90's. In the beginning, there was a great expectation that the OODBMS would replace the RDBMS as the database of choice for future applications.

The "great expectation" has not come to fruition and it is an assertion in this paper that ONE of the major reasons for this comes down to OODBMS architecture. That is, the OODBMS architecture and it's impact on the expectations of early adopters. Many successfully deployed applications have demonstrated that choosing the right OODBMS architecture allows the building of high performance, highly concurrent and scalable solutions. This paper seeks to examine the issue of expectation along with examining the differences between the 3 most common commercial OODB architectures.

Introduction:

At the time the OODB started commercial deployments, relational technology was already well rooted and while still a hearty battle ground, the major relational vendors had all staked out a portion of the database market. At the time, businesses were actually transitioning away from making application database infrastructure technology decisions to making application business capability decisions. The database comparisons had all but been made and heavily publicized through benchmarks such as TPC. Application suites with the highly desired business capabilities either came from an incumbent relational database vendor or were designed to support any relational database by adhering to the SQL standards. The characteristics of the databases were close enough that there was no particular advantage to be held by supporting one vendor over others. The general perception was that relational databases were very similar in capabilities and these similarities were well documented. So, the decision of which database to use was often more political than technical, facilitating business alliances rather than necessitating a choice for "best" solution left up to the purchaser of the application. The general perception was indeed true, there are little differences from one relational vendor to the next, and performance and scalability numbers showed variations by percentage points rather than orders of magnitude.

This is where OODB architecture and user expectations get into the mix. While RDB architectures are very similar (server centric, index based, relational algebra execution engines) exhibiting performance and scalability characteristics differentiated by small percentages, OODB architectures vary considerably and exhibit wildly different characteristics. After so many years of conditioning that RDB's behave essentially the same, it was rather natural to apply the same reasoning to the ODB and proclaim, they are basically the same. In making that assumption, if by chance an early adopter choose an

ODB who's architecture was ill suited for their applications needs, the reasoning lead immediately to the conclusion that no ODB was suited to solve their needs. From this illogical thinking came the permeation of misconceptions regarding the OODB: they are too slow, they don't handle high concurrency, they don't scale with large data, etc, etc. These are all misconceptions and the reality is that you need to carefully consider your application characteristics and understand which OODB architecture is best suited to fulfill them. Choosing the right OODB architecture can mean orders of magnitude difference in performance and scalability characteristics rather than a few percentage points as found in relational implementations. Armed with this understanding, lets now explore the differences so that you can make an informed decision which will assist in leading you to the successful adoption of the technology.

The major factors:

There are several key distinguishing implementation differences that lead to the vastly different runtime characteristics. It is not the purpose of this paper to address all feature functionality, only major areas impacting performance and scalability under different application characteristics. The primary areas include: the core architecture, the concurrency model, the query implementation, the network model and identity management.

Core Architecture:

The following presents three well known OODB architectural implementations. Each OODB implementation provides degrees of distribution, parallel processing and remoting that are used to create elaborate system level designs, but these are the core architectural types underpinning those systems. In this paper we will call them: container based, page based and object based which reflects both their unit of transfer across network calls and lowest level granularity of locking. Each in turn provides caching, query processing, transactions and object lifecycle management (tracking of new, dirty, deleted). Lets take a look at each more closely.

Container based:

The “container-based” architecture is a client centric design. It uses standard or proprietary NFS to ship segments of disk around the network, called containers, to the clients which implement the majority of the database functionality. The user application code is linked with the database client libraries which provide caching of containers, query processing, transactions and object life cycle management. All objects must reside inside a container and a container can hold many objects, multiple containers can be used if the limit per container is reached.. In this architecture, application developers must ensure container models are layered over application domain models to facilitate access to objects within the database.

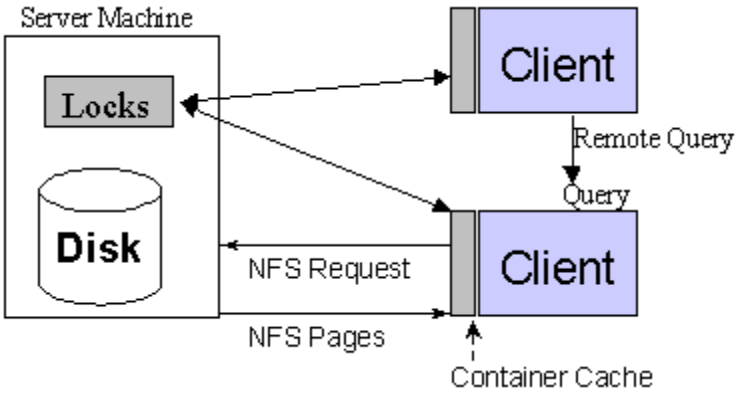


Figure 1 - Container Based Architecture

Page based:

The "page-based" architecture is a client centric design which runs a server process to fulfill page requests in a distributed virtual memory mapping model. It uses a server process to ship pages of disk around the network which get address translated into the virtual memory space of the application, where the majority of the database functionality is implemented. The user application code is linked with the database client libraries which provide caching of pages, query processing, transactions, object life cycle management. Special object placement strategies have to be implemented.

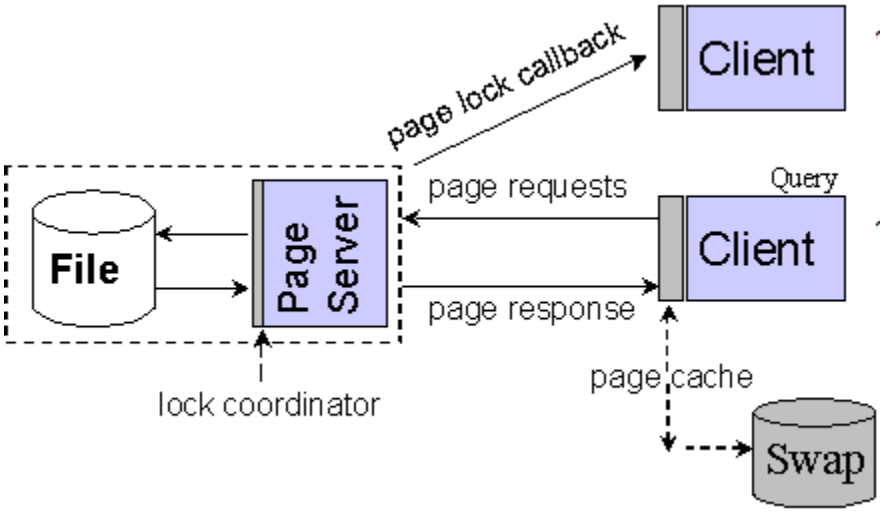


Figure 2 - Page Based Architecture

Object based:

The “object-based” architecture is a balanced design with caching and behavior in both the application and database server processes. The server process caches pages of disk and manages indexes, locks, queries and transactions. The user application code is linked with the database client libraries which provides object caching, local locking and object lifecycle management. No special object placement strategies need to be implemented.

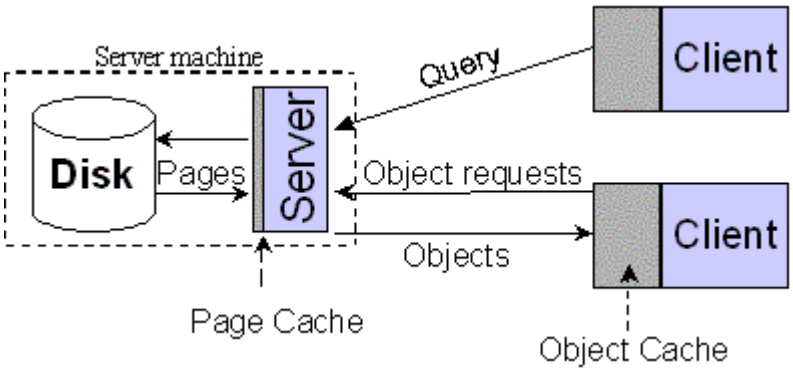


Figure 3 - Object Based Architecture

The Concurrency Model:

The three core architectures have differing concurrency control models as described below. The concurrency models solve the issue of transaction isolation and are closely tied with the network behavioral characteristics. Each architecture has it's ways of relaxing locking for certain types of applications where full ACID (Atomic, Consistent, Isolated, Durable) properties are not required. The following is focused on behavior required for the consistency and isolation of proper database transactions that are not relaxing the ACID properties. All implementations can cache locks locally at the client across transaction boundaries to minimize unnecessary lock requests. However, updates will always cause lock coordination and cache consistency operations. With container and page based systems, object placement and locking are tightly coupled while in the object based systems these issues are orthogonal. Lets take a look at how each architecture deals with these issues.

Container-Concurrency:

In this design, a separate lock server process coordinates concurrent access to information in the same container. Clients coordinate with a centralized lock process to request locks on containers in each database on a server machine before being allowed to cache a container at the client through the NFS page server. A request for update will establish a lock request queue on a container if there are existing readers of that container. The

requests will either time-out or go through when existing clients release their locks on the container. 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 queued read requests rush in and get their read lock, return the container pages, and if there are no other updates, the queue disappears. Clients who had cached an updated container must refresh the container on subsequent read. Containers can hold many objects, so it is possible to get false waits and false dead locks.

Page-Concurrency:

In this design, the page server coordinates concurrent access by tracking which applications are accessing each page, granting permissions to lock pages locally, and using lock callbacks. A request for update will cause the page server to use callbacks, requesting other clients to release their locks on that page and give up permissions to lock the page locally. All clients either release their lock and permissions to lock locally and the request goes through or it blocks on each objecting client for a specified duration. Each client that released it's permissions will refresh the page and acquire new permissions and locks on the next access of information on the page, going through the process just described. Locks and permissions are taken out at the page level and pages may contain many objects, so false waits and false deadlocks can occur.

Object-Concurrency:

In this design, 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. 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. Clients who had cached an updated object must refresh the object on subsequent read. In this architecture, locks are done at the object level so false waits and false deadlocks do not occur.

Concurrency Model Implications:

The different concurrency models can impact application performance in multi-user environments which have many updates. Applications which have conflicting updates are potentially hampered by all concurrency models, but that is why there are concurrency models. Applications which have heavy updates, but not necessarily conflicting updates, are potentially hampered further by both container and page based architectures due to the fact that locking occurs in a way that can block other objects not involved in the transaction introducing false waits and deadlocks.

In addition to potential pitfalls on false waits and deadlocks, the granularity of the objects involved in the application transactions can have greater impact in one architecture over the other. In the page based architecture, due to locking callbacks, a lot more network calls are required to coordinate lock release in a highly granular update system.

Applications which have relatively fixed models, well segmented into an independent cluster, with mostly non-conflicting updates, will generally do best in page and container based architectures. This is true because only high level locking is required and mostly non-conflicting updates will minimize lock coordination. One example which fits well with these characteristics is a CAD (computer aided design) type application. For applications of this type, using an OODB can provide orders of magnitude improvement in performance over traditional relational databases. Applications which have increasing levels of conflicting updates and increasing granularity of models, will generally do best in the object based architecture.

The implications of performance under concurrency are generally tied to the number of concurrent users or processes. The issues above are exaggerated in their affect as the user base or number of processes accessing the database system are increased. In general, the object based architecture is best suited for the largest numbers of concurrent users and or processes, especially in systems that are not well segmented for isolation.

The other issues surrounding the concurrency models involve long term expected use of the data. including evolution and maintenance of the applications. In many systems, fixed, well segmented data, can be achieved for a given application and its access patterns. An important question arises regarding the introduction of newer applications or increased capabilities of existing applications over the original data New applications and capabilities often require different access patterns. As the number of access patterns increase, designs that remain fixed and well segmented become much more difficult to achieve and changing the object placement strategy has a major effort and impact. In effect, an adopter of the technology should consider future use of the data carefully during the selection process.

As discussed below, note that lock coordination also has tremendous implications on the network model, because an update will require network activity to refresh client caches.

The Network Model:

The three core architectures have differing network models affecting the bandwidth utilization and performance in a distributed system design. Each network model is closely tied to the locking model where upon obtaining a lock, the subsequent transfer of information occurs unless already cached locally. The lowest level granularity of the transfer across the network is tied to the lowest granularity of the lock.

Container-Network Model:

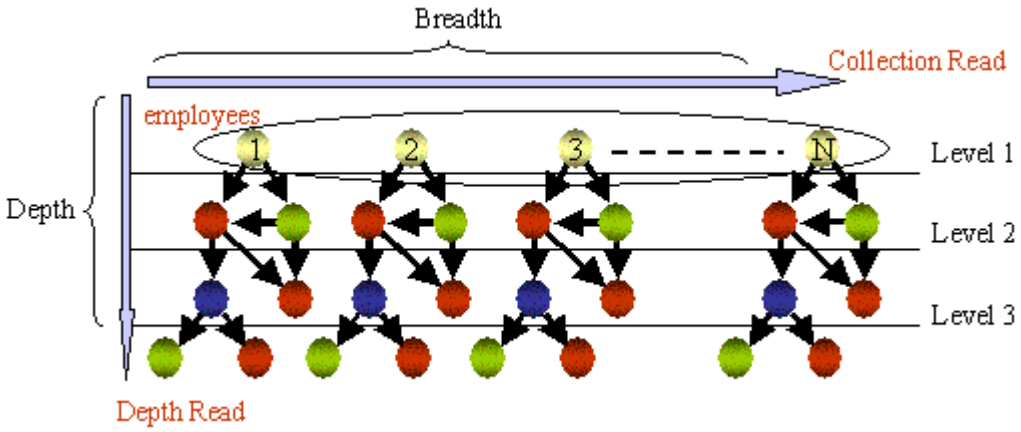
For this architecture, the unit of transfer on object request is the container. All objects must reside in a container and a client request for an object is resolved by the NFS server sending the container's pages across the network to the client where the container is cached and the object subsequently accessed. When transferring the container, all objects contained will also be transferred regardless if they are accessed or not. When transferring the container, a lock is placed on the container regardless which specific objects in the container are actually accessed by the clients requesting transaction.

Page-Network Model:

For this architecture, the unit of transfer on object request is the page. All objects must reside in a page and a client request for an object is resolved by the page server paging across the network to the client where it is address translated and cached and the object subsequently accessed. When transferring the page, all objects residing on the page will also be transferred regardless if they are accessed or not. When transferring the page, a lock is placed on the page regardless which specific objects on the page are actually accessed by the clients requesting transaction.

Object-Network Model:

For this architecture, the unit of transfer on object request is an object or group of objects depending on specified depth as illustrated below. An object may be a collection in which case many objects are transferred in a single network call. An object or collection request may include a depth request which will retrieve related objects at each depth in a single network call. When transferring an object, a lock is placed on the object. In the case of a collection and/or depth request, a lock is placed on each object sent to the client. For example, if all objects in the following illustration were to be loaded to the client, it would take 3 network requests. If only the employees themselves or one particular employee were requested, then only a single network call would be required. Customized depth requests are possible, allowing use case specific loading patterns which eliminate excess object loading while minimizing network calls.



Network Model Implications:

The different network models can impact application performance by impacting the number of network calls and simultaneously bandwidth utilization impacted by the amount of relevant data that is sent across the network. The behavior in a networked environment is closely tied to the application models and access characteristics.

An application who's characteristics are to access a mostly fixed model, well segmented into an independent cluster, in a non conflicting update and/or without lots of small transactional updates, will generally perform well with a container or page based architecture. This is because network calls are limited, grabbing a fixed container of related objects and moving them to the client cache in a single network call. Fixed models, tied to a specific use case where other logically related objects will be accessed at the same time in the same transaction, work well in these architectures because all logically related objects will already be located in the client cache after first object access and network calls will be minimized with efficient bandwidth utilization. The object based architecture will generally require more network calls, a call for each level of depth in the model. For example, this could lead to 1-to-3 ratio in network calls for a model like that shown in the figure above. However, this is assuming all objects in the figure are in the same page or container. If they were segmented (physically clustered) differently, then it is possible to have even more network calls in the page or container based architecture than in the object based architecture. For this reason, the networked performance and for that matter, the performance under concurrency, is critically tied to the physical layout of objects on disk for both the page and container based architectures. In an application with these characteristics, bandwidth utilization will be nearly as efficient in all architectures because groups of logically related objects will be transferred and excess objects will not be moved across the network.

As conflicting updates increase, performance with page and container based architectures will decrease and the object based architecture will provide an advantage. This is because conflicting updates will necessitate cache refresh in conflicted clients. For page and container based architectures, this can mean that pages or containers are needing to be shipped around the network for cache consistency. The biggest penalty here is network bandwidth, because if only a single object of a few bytes is updated, several thousand bytes, the page or container, will need to be refreshed across potentially many client processes. The full page or container refresh is required because this is the smallest unit of granularity in the network model.

To clarify what is meant here by conflicting updates, "conflicting" is not necessarily the update of the same object, but updates of objects that are in the same page or container. This is why for container and page based architectures, the model needs to be well segmented in a physical cluster, because if not well segmented, then even updates to non related objects can cause conflicts at the page or container level which leads to lock waiting and network utilization to ensure cache consistency. The reason small transactional updates need to be avoided with container and page based architectures, is because an update must write the smallest unit of data transfer, the page or container, across the network for long term storage. If updates can be well batched together in a

single transaction impacting a single page or container, then performance will not suffer as a result of the architecture and in fact application performance will excel tremendously as compared to the traditional relational system.

As conflicting updates increase, the object based network model shows advantage because the granularity of transfer is much smaller. If an object only a few bytes in size is updated and requires a refresh from clients for cache consistency, only that few bytes is transferred across the network. Further, the notion of conflicting updates changes a bit. In all architectures, any update requires cache consistency work. So, if models are fixed and well segmented, then the object, page and container based architectures behave nearly the same except the size of data transfer across the network is different. However, if the models are not well segmented and/or object relationships span pages or containers with only a few objects in the page or container getting impacted, then many pages or containers with lots of excess objects potentially need to be refreshed and that means a lot more network bandwidth will get utilized transferring the excess objects around the network in addition to the objects that truly need refreshing.

Another implication of the concurrency and network model for each architecture is in how they behave in multi-tiered architectures, embedded inside an application server middleware. All of the issues above apply, because caching is done locally in the application servers and so the issues and behavior described above are required to keep the cache consistency across the cluster of application servers which are in essence a cluster of OODB database clients.

Of course, all of the above is again impacted by increasing numbers of users or processes accessing the database. The issues above are exaggerated in their affect as the user base or number of processes accessing the database system are increased. In general, the object based architecture is best suited for the largest numbers of concurrent users and or processes, especially in systems that are not well segmented for isolation or require flexibility in future proofing more access patterns.

Hopefully from the above discussion it is clear that each architecture's network model has it's advantages and disadvantages. The page and container architectures require careful physical layout of application models to achieve optimal performance. In a fixed model that is well segmented, this results in efficient network utilization. This benefit comes at the cost of rigidity in access patterns which can impact long term data use and performance under high concurrency. The object based architecture requires a logical use case understanding for optimal performance. Unlike with page and container based architectures which need that understanding for physical modeling, the object based architecture needs the understanding so each application can use special coding or meta data to define use case based model loading. The effort is rewarded by facilitating high concurrency and scalability for extremely large numbers of transactions while improving flexibility in allowing overlapping access patterns for future data use.

The Query Implementation:

Object database implementations focus on their ability to navigate seamlessly between related objects using the language constructs. The native support of a navigational access pattern is a key advantage of OODBMS over the RDBMS complex join concept. Essentially, relationships are a static part of the system rather than runtime calculated, which makes them inherently faster to use. For each implementation, it is theoretically possible to define one master object which once retrieved would provide navigational access to everything else in the database. However, as a matter of practicality, application implementations are typically use case driven. It does not make sense to needlessly bring in objects that are not relevant to the use case. Further, the scalability requirements of large databases mandate efficient use of disk, network and memory. So, object databases provide a means of querying to retrieve the first level objects of a use case and then allow subsequent navigation from those starting objects. Again, it is not meant here to explain the detailed behavior of each query engine, but instead the fundamentals of the implementation. Specific details like the ability to return id's, for lazy loading, instead of actual objects, etc are left out.

Container-Query:

The container based implementation uses query processing at the client. The "client" may be another remoted instance of a client, but it is a process separate from the NFS page server process. You can contrast this with the typical relational database which is the opposite in architecture. In the RDB, everything is contained within the database server process: query, indexing, locking, and page management making it a server centric implementation versus a client centric implementation. In the container based implementation, all objects from the database server involved in the query must be identified by database/container which includes any potential indexes, and loaded into the client process for query execution. Once all potential objects or indexes are loaded and the query is executed, the results returned are containers holding the resulting objects that satisfied the query. The result as seen from the network and locking perspective may therefore contain many objects that did not actually satisfy the query predicate. If the client was remoted, then results are returned through several tiers to the original requesting client process. Through a combination of system wide identifiers, multi-threading, remoting and aggregation, parallel distributed queries are possible.

Page-Query:

The page based implementation uses query processing at the client. In the page based implementation, all objects from the database server involved in the query must be contained within collections that are loaded into the client process for query execution. Queries and indexing can only operate on these collections. Once the collection of objects is loaded and the query is executed, the results are references to the objects that satisfied the query predicate and implicitly the pages containing those objects have already been loaded across the network and address translated into the client memory

space. The result as seen from the network and locking perspective may therefore contain many objects that did not actually satisfy the query predicate.

Object-Query:

The object based implementation uses a query execution engine that runs within the database server process. Any object in the database is reachable via query even if it has no relationship to other objects. Object attributes can be indexed with maintenance done by the database server. The query is done as a statement sent to the server, executed using an optimizer and indexing on the server, and a result set of objects returned to the client. The only thing transferred across the network is the statement for execution to the server and the collection of objects satisfying the predicate back to the client. Through the use of system wide identifiers, multi-threading and aggregation, parallel distributed queries are possible.

Query Model Implications:

The different query models can impact application performance significantly. The primary factors in achieving optimal performance are: (1) where does the query execution take place, (2) flexibility on what can be queried, (3) indexing capabilities.

The object database implementations have always focused on navigational access as the primary means of retrieving information from the database. Again, it is not the intent here to dissect all query capabilities, like projection, aggregation, views, mathematical evaluation, cursors, composite indexing, etc, etc. The intent here is to provide an understanding of the core implementation. As a general rule, none of the implementations were as complete as people have come to expect in the relational world where all access to data is via query. Fortunately, this has been changing, as all the vendors gain a new understanding of the complementary role of query and navigational access. It is reasonably safe to say, that after nearly 2 decades, all of the implementations have matured capabilities that make them useful, the real question is now one of scalability and performance.

The single biggest killer of performance in query operations is when the potential objects that satisfy the query need to be moved to another process space for predicate evaluation. This can have tremendous performance implications, especially when there is large data involved. The page and container based query implementations suffer from this problem.

The page based architecture, in particular, is most vulnerable to this problem as only objects which exist in special collections can be the subject of a query and indexes apply only to those collections. All pages containing the objects in those collections or if indexed the indexes for those collections, must be loaded across the network for query execution. Those pages will undoubtedly contain many unnecessary objects wasting network bandwidth and performance as more time can be spent transferring the pages than actually evaluating the query. Think of the case where there are a million Foo objects, but you are only looking for one of them. Plus, any insertion then requires that potential indexes are maintained and since they are potentially distributed across many

client processes, there are potentially many page invalidation's and refreshes across the network for the collection and indexes adding to the network bandwidth consumption.

The container based architecture behaves in the same way as the page based architecture. However, queries are remotable, can be split and indexes are maintained in a centralized process. So, even though containers of objects and/or indexes must be loaded into another process for evaluation, execution can be done in parallel and only the containers satisfying the results of the predicate need to be moved across the network to the originating tier of the calling client. While not ideal, this is more scalable than the query design in the page based architectural implementation.

The object based design, which offers server side indexing and query execution similar to relational databases, offers the optimal query capabilities among these architectures. There are no excess objects that need to get moved across processes, indexes are managed locally and execution optimization is performed in the database server process returning only the object or set of objects that satisfy the predicate. Further, multi-threading in the client process allows the query execution to be performed in parallel across multiple physical databases.

With page and container based systems, the index management is typically integrated into the application code. That has an impact on the maintenance of the applications. With object based systems the index is managed by the server. New indices can be defined without changing the application.

Identity Management:

The core architectures all differ in their implementation approaches to identity management, yet they fundamentally differ in two ways. Identity is either logical or physical in nature. Object databases use identity in order to establish uniqueness and also to implement relationships. The difference in implementation has profound implications on long term operational behavior and flexibility. In addition, the particulars of implementation for physical identity have an impact on data scalability for systems requiring storage of multi-terabytes of information.

Physical Identity:

What is meant by physical identity is that all objects in the system of distributed databases have a unique identifier that is dependent on the physical location of the object within the system. The characteristics of object physical identity are: mutable, reusable, immobile, rigid.

One implication is that objects have a difficult time moving about the system without impacting other objects with which they are related. This is due to the fact that object relationships wrap the identity of the related object. If an object is moved in the system, it's physical location changes and therefore its identity changes (mutable). All existing objects in the system which had reference to that object must now be found and the relationship fields patched to wrap the new physical location (immobile). As the

database gets larger this becomes an increasingly expensive operation. While there are many advanced system designs that could use object movement to control active data sets for performant large scale systems, object movement by design could be avoided.

Unfortunately, object movement can be caused as a side effect of other non system design related operations. For example, schema evolution could change the layout of an object in such a way that it no longer fits in it's existing space. In such as case, the schema evolution would necessitate object movement and as a side effect cause tremendous stress on the system requiring extended downtime to make the schema change. Object growth due to variable length fields could cause the situation. Changing clustering schemes can cause this situation.

Other implications include things like performance degradation due to fragmentation. This is a classic problem in relational databases for applications that have large numbers of deletions where application performance degrades steadily over time. Most RDB vendors sell add on tools that do database reorganization to be used when the threshold of required response is breached. In the case of OODB's and physical identity, if an object is deleted, other objects cannot be compacted on the page and space reclaimed, because it would cause their identity to change and existing relationships to become invalid. Therefore, they suffer the same problem as relational databases where more and more empty space is created over time causing slower and slower response.

A key difference in implementation for physical identity systems leads to another significant scalability difference in the area of raw data management on disk. By taking an address translation approach, the page based architecture is unable to provide a true federated data distribution on disk. It is possible to segment data within a single database so that applications work on sub portions of that database in their process space, but it is not possible to have a data distributed (federated) database in the disk subsystem that is accessible to a client as a logical data source. This means that while the container based implementation of physical identity has data scalability capabilities into the terabyte or even petabyte range, the page based implementation is restricted to the megabyte or gigabyte range.

Logical Identity:

What is meant by logical identity is that all objects in the system of distributed databases have a unique identifier that is independent of the physical location in the system. The characteristics of logical identity are: immutable, never reused, mobile, flexible. The implications are that objects can freely move about the system without impacting other objects with which they are related. An object can be moved from one physical database to another and existing relationships from other objects remain intact and existing code continues to run because they are based on logical identity rather than physical location. Advanced designs like data archiving with active data sets are possible with no location aware coding in your application. Internal operations like schema evolution and page compaction can take place online in a future proof manner without risk of running into extended outages. Further, the use of logical identity enables the ability to provide a true application unaware federation of disk based databases as a logical entity, allowing

scalability into the terabyte or petabyte ranges without special coding to deal with the distribution.

Conclusion:

Understanding the core architectural implementation of an OODB and matching it with the characteristics of your application use can make the difference between success or failure in the adoption of OODB technology. Unlike relational technology, a given OODB's architecture can exhibit orders of magnitude difference in performance and scalability under a given set of application characteristics. Page and container based architectures provide a good solution for applications with relatively fixed models, low or medium concurrency and well segmented data. With object based architectures the concepts of: object locking, object placement and object shipment are independent and provide the best solution for high concurrency, growing and evolving applications and data.

An adopter should consider carefully such issues as data growth over time, number of concurrent users or processes, expected future requirements on changing domain models, granularity of the model, and the anticipated demand on the network. It has been proven time and again that the object database can be a superior solution to relational technology for applications with complex domain models that don't fit well in the relational storage paradigm. Choosing an OODBMS with the right architecture for your application is the key to harnessing the power of the technology and ensuring success.

References:

[1] Versant Object Database Fundamental Manual - release 7.0.1.0, July 2005
http://www.versant.com/developer/resources/objectdatabase/documentation/database_fund_man.pdf

[2] Objectivity Inc. Technical Overview, Release 9, January 2006.
http://www.objectivity.com/Misc/docs/oodb_techOverview.pdf

[3] Locking and Concurrency, Objectivity Inc, 1999
http://wwwasd.web.cern.ch/wwwasd/lhc++/Objectivity/V5.2/Java/guide/jgdLocking.fm.html

[4] ObjectStore Architecture Introductory, Dirk Butson
http://www.techsmiths.com/wapug/exch03/white_papers/objectstore/OStoreVMAA2_redux.doc