OODBMS Architectures Revisited

by Adrian Marriott, Progress Software

Abstract

The purpose of this paper is to explore the analysis offered by Greene in his paper *ODBMS Architectures: An examination of implementations* [1], and to correct what I believe are some incorrect statements made regarding page-based server architectures. I simply address the subjects Greene discusses in the order that they appear in his paper. To help orient the reader the original headings are included, and the offending passages are quoted, before I examine his claims.

Introduction

First Paragraph, Page 3

"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."

To achieve maximum performance and scalability the most important thing is choosing the right *application architecture*. OODBs give applications much more direct access to the persistent data, so application architecture has much more impact on performance than is the case with an RDB. Or to put it another way, when using an OODB the application architect has much more power to optimize performance than when using an RDB. Consequently the application architecture has more effect on performance and scalability than the choice of OODB product.

To effectively exploit an OODB a use-case driven approach is recommended, as this informs the entire design of the application. The process architecture design should consider which processes are responsible for which use-cases. Partitioning of the dataset should aim to identify which objects are accessed by which of these processes. Transactional analysis should analyse the transactional requirements of each use-case, and the objects that are accessed in each transaction. Object interaction diagrams are useful in this respect. The OO design phase should include the design of optimal access and index structures to support the navigation paths of the most important use-cases, and the concurrency characteristics of the system should be explored with techniques such as CRUD charts. In short, using standard OO analysis and design techniques to produce the correct application architecture is essential when implementing large, high-performance, scalable and reliable OODB based applications.

Without this effort, the choice of OODB product is unlikely to change the performance characteristics by orders of magnitude one way or the other.

Paged Based

Last Paragraph, Page 4.

"Special object placement strategies have to be implemented"

This statement is false. Page based architectures in themselves do not ***require*** the programmer to consider locality of reference. Being able to co-locate objects together that are used together is a technique which offers the programmer a very effective way of optimizing processing, based on the physical characteristics of the system. Application developers can optimize performance through object placement strategies that co-locate objects that are used together, and place objects that are not used together on to distinct pages. Using placement strategies is ***optional***, albeit recommended, in just the way optimizing transient algorithms is recommended; page based architectures still work without object placement strategies.

Object Based

First Paragraph, Page 5

"The "object-based" architecture is a balanced design with caching and behaviour in both the application and database server processes. The server process caches pages of disk and manages indexes, locks, queries and transactions."

The concept of 'balance' here is a subjective aesthetic assessment. 'Balance is in the eye of the beholder', as it were. More importantly, the server process in a page based server architecture, such as ObjectStore, could, and in ObjectStore's case, does, cache pages from disc. The ObjectStore page caching mechanism is designed to efficiently exploit the operating systems ability to cache pages from recently accessed files in memory. Furthermore, the page-based server also mediates all database access, tracking which clients have which permits on which pages, manages server-side transactions, and generally tracks page life-cycles. So the page-based server architecture is 'balanced' to the same extent, and to the same degree of meaninglessness, as the object-based server architecture, in that the server and client cooperate to achieve certain ends.

Whether to execute query and index processing in the client or in the server is a complex question. It isn't necessarily more 'balanced' always to do it in the server. Database clients don't have to run on the same machine as the server, and the clients don't have to run all on a single machine, thus there is potentially far more CPU power and cache memory capacity on the client side than on the server side. If indexes fit in the client caches and do not change too often, queries can take advantage of this larger computing power by executing on the client side. This is the ObjectStore strategy of moving the data close to the application. But if queries do not have indexes available to optimize them, or if the indexes cannot be cached effectively in the clients, it may be better to execute queries as close as possible to the disc, which means on the server side. Which architecture performs better in a given case depends on the detailed characteristics of the data, the application, and the deployment.

Concurrency Model

Paragraph 2, Page 5

“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.”

ObjectStore ***never*** relaxes the ACID properties of transactions. The one thing that is true of ObjectStore is that the cache is ***always*** transactionally consistent. This is one of the best features of ObjectStore. Programmers *really* don’t have to worry about cache synchronisation, because the way ObjectStore manages the cache and automatically fetches and locks pages on an as-needed basis, and the efficiency with which this process operates, means that the ACID properties never have to be compromised for reasons of performance.

“…updates will always cause lock coordination and cache consistency operations.”

Under certain conditions it is possible for updates in ObjectStore to result in ***no*** lock coordination and cache consistency operations precisely because the locking is done at the page level. Updates to objects on pages already locked for update proceed with zero overhead. This means that the cost of locking can be amortized across all the objects on that page.

In contrast, if each object is locked individually, then this locking overhead can represent significant processing, particularly when dealing with many thousands or millions of small objects.

“With … page based systems, object placement and locking are tightly coupled while in the object based systems these issues are orthogonal.”

Here Greene is high-lighting that with object level locking, when you lock an object, you are guaranteed to *only* lock that object. This means that the lock on any object does not accidentally lock another object. In this sense, object level locks are ‘de-coupled’, or independent, from one another.

However, what Greene omits to mention is that in real-world systems object-level locks can be expensive to manage and are often unnecessary, from the perspective of the actual use-case being implemented. Most algorithms require a number of objects to be accessed before a transaction can be usefully committed, so having the flexibility to efficiently lock them simultaneously when you need to, but also be able to lock each one independently where necessary, can be an advantage from a performance perspective.

In page based systems, object placement will result in simultaneous locks on objects located on the same page, but whether these are *spurious* locks, or pernicious in any way, depends on the details of the use-case. If there are two objects O_1 and O_2 located on the same page, and the program is executing a transaction which updates both these objects, then the simultaneous locking presents no problem. In fact it represents an advantage because the lock management overhead is reduced.

Greene also fails to observe that in page based systems where you really need object-level locking you can simply place an object on a page by itself. If the objects O_1 and O_2 are to be accessed independently by different programs simultaneously, with ObjectStore, the programmer can ensure they are allocated on different pages and will therefore be locked independently.

ObjectStore offers a very simple and flexible API that enables objects used together in the same use-case to be located on to the same set of pages, and to separate objects which need to be locked independently on to different pages. So during the design and implementation of an ObjectStore based system the programmer can organize the persistent data into different areas of the database – called clusters – such that data is held on the minimum number of pages possible. When the objects on these pages are accessed by the program, they are dispatched from the server on the minimum number of pages. This can offer huge performance advantages. Ultimately all persistent data are stored on magnetic discs, and physically disks are block-structured devices with very high performance differentials between random access and same-block or sequential-blocks access. By allowing this physical disc structure to show through to the application it gives the application programmer the opportunity to radically optimize performance.

However, this powerful feature can also cause problems if used incorrectly. For example, if a programmer distributes objects used together in the same transaction across many thousands of pages, then both read and write performance will be compromised. So with the increased exposure of the application to the physical organization of the database in an OODB, as compared to an RDB, comes an increased ability of the application designer to improve *or degrade* performance.

Page-Concurrency

Paragraph 2, Page 6

"Locks and permissions are taken out at the page level and pages may contain many objects, so false waits and false deadlocks can occur."

It is true that placing two objects on the same page and attempting to update them both simultaneously from different clients running separate transactions, or if one is updating an object while another is attempting to read the other, this will result in contention. Note that this does not result in a logical problem or a database corruption issue or a non-transactional view of data. Greene maintains that this presents a *potential* problem. How serious is this in reality?

There are basically three database operations we must consider here; reading, creating and updating objects. In real-world systems, most database access typically consists of 'mostly read' scenarios. If both objects are read simultaneously, and no updates to these pages occur, no contention occurs and both clients can legitimately access the objects.

What about object creation? ObjectStore clients control ‘locality of reference’ by using database entities called clusters. Clusters represent ‘physical’ collections of pages. A cluster can contain one or more pages, but each page is in only one cluster. From the API perspective ObjectStore programmers allocate objects into clusters, not on to particular pages. As objects are allocated into clusters new pages are added to the cluster and it grows in size. If two clients create objects in the same cluster simultaneously, by default ObjectStore allocates these objects on to different pages in the same cluster. In short, contention for object creates is rarely an issue.

So Greene’s point only concerns object updates on pages simultaneously being read or updated by other clients, accessing *different* objects that happen to be located on the same set of pages. In this situation there are two basic approaches. The most effective way to avoid this issue is by using object placement strategies where necessary. This can totally eliminate unnecessary waits and deadlocks.

However, there is a complexity which arises when attempting to cluster objects to account for multiple use-cases. Different use-cases can have competing or contradictory clustering requirements, where one use-case might best be optimized by clustering objects one way, and another use-case might prefer another. These complexities do arise in real deployments, and there are several approaches to solving them.

- One is to ignore the less important use-case, and only concentrate your efforts on the most important one. This approach works, for example, when the primary system use-case conflicts with a much less important secondary use-case that is executed infrequently.

- Another approach is to re-visit the object model, particularly the access structures, and re-design these with a view to optimizing the clustering for two (or more) important use-cases. Here the designer is explicitly aligning the data structures and the clustering so they ‘cooperate’ together at runtime. For example, this can involve storing extra state to fulfil the second use-case, so the objects that are less-than-optimally clustered are not touched during its execution. There are many variants of this approach.

- A third approach compromises the otherwise sacrosanct ‘store-it-once’ rule of database design, and optimizes two important read-only use-cases by transferring the cost on to update use-cases. Here the same data is clustered in two different ways to maximally optimize the read use-cases by introducing the requirement to write to two different data structures when updates occur.

ObjectStore also offers a special access mode called MVCC, which allows clients to read pages from the database with guarantees that these reads will not block updates, will never cause, or be the subject of, deadlocks, and guarantees a transactionally consistent, but possibly not up to date, view of the read data. Again this can totally eliminate problems of contention, and deadlock.

In real-world deployments, designing the data model so that it supports placement strategies that optimize multiple use-cases is one of the most creative parts of using

ObjectStore, but can pay the very highest dividends in terms of overall system performance. This approach, combined with MVCC, can and has enabled competent designers to eliminate deadlocks and reduce contention to zero, or near zero.

Another point is, it's not spurious lock conflicts that cause problems, because they are spurious – there is no real reason the objects are co-located on the same page, they just happen to be – so moving them into different clusters is usually a straight-forward affair. A more difficult problem are genuine lock conflicts, where two clients require exactly the same objects but with conflicting locks simultaneously, as is commonly the case with simultaneous updates to object extents, or index and access structures that are at the root of object graphs. In this area, both object-based lock architectures and page-based lock architectures probably have equal difficulty.

Concurrency Model Implications

First Paragraph, Page 7.

"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 call-backs, a lot more network calls are required to coordinate lock release in a highly granular update system."

I'm not quite sure what Greene is alluding to here. Why should having *fewer* locks increase the amount of network traffic required to manage locks? I think the point he intends to make is that spurious lock conflicts (he calls them false lock conflicts) can increase the amount of thrashing of cached locks and therefore increase the amount of network traffic. This is true but is not what is said. Of course the thrashing due to spurious lock conflicts only occurs if there *are* spurious lock conflicts, and as noted above, there are effective strategies to eliminate these and therefore any page thrashing that would result.

While it is true that in page based servers it is possible in some scenarios to get 'page thrashing', where two clients continually access the same pages, this is commonly symptomatic of an architectural design problem at the client process level. It can be avoided by implementing an N-tier process architecture where all updates to a particular set of objects are directed to a single update process which is responsible for those objects. This has the added advantage that those objects are more likely to be in the client cache and accessed at in-memory speeds.

Greene also fails to note here that the same thrashing can occur at the object level in systems which use object-level locking. Where a group of objects are required by two clients, and these clients require incompatible locks on these objects, for example they both wish to update the objects, these objects will thrash to and fro between clients, except this time, because all the locks are acquired on a per object basis, the lock overhead management can be higher. So both architectures can exhibit thrashing behaviour.

Paragraph 3, Page 7

"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."

What Greene fails to note here is that server centric systems, such as Versant, can quickly exhibit scalability issues as the number of users increase because much of the processing has to occur on the server. In a client-centric architecture, where the main algorithmic processing takes place on the client, the load is distributed to all the client machines, virtually eliminating the server as a bottleneck. My point is that which is best will depend heavily on the context, what use-cases are being implemented and the details of the dynamic behaviour of the particular system concerned. Greene has over-simplified the issues here.

Scalability may require awareness of the performance characteristics of magnetic disc hardware, regardless of whether the database lock granularity is objects or pages. Pretending that random access is as fast as sequential access is not going to work when databases get too large to be cached entirely in main memory. So no matter what OODB architecture is used, the application may need to be aware of object placement in order to perform satisfactorily.

Network Model Implication

Paragraph 3, Page 9

"The biggest penalty here is network bandwidth, because if only a single object of a few bytes is updated, several thousand bytes, the page …, will need to be refreshed across potentially many client processes."

Mathematically true, but largely irrelevant. For most of today's network technology, latency is more of a performance limiter than bandwidth. The network often has bandwidth comparable to or greater than the bandwidth of the disc. But the latency for a round trip through a local area network to a server process and back can easily exceed the time required to transmit a megabyte or more through the network

Sending a network packet containing a few bytes and sending a network packet containing 4096 bytes (1 page) is 'to all intents and purposes' equally efficient. It is the number of packets rather than their size that is the issue. Data shipping only becomes an issue when there are thousands of little updates on many different pages in the database, but again this can often be solved by the use of placement strategies, which will ensure that all these little objects are located on the same page, in which case all the updates will be efficiently sent in the same network packet.

There are many cases where ObjectStore has demonstrated that sustained updates can be written to disc at speeds only limited by the write speed of the disc; so in the case of small objects in the 100 byte size range, for example, this means many hundreds of thousands per second. This is accomplished by smart object placement and clever indexing structures which convert object updates to object creates, concentrating all the changes onto as few pages as possible. This technique is often combined with a

C++ pool allocator to further reduce the cost of individual object construction by creating genuine C++ arrays persistently in the database.

On reading Greene here, some people might think he implies that updated pages are ‘pushed’ out to client caches when they have been updated which, although potentially page-based server architectures could implement, is entirely untrue of ObjectStore. ObjectStore pulls pages on an as needed basis, so these updates would not be immediately sent to many clients, but only sent if the client accesses the page concerned, and so presumably needs it.

Paragraph 2, Page 10

“…if the models are not well segmented and/or object relationships span pages … with only a few objects in the page … getting impacted, then many pages … 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.”

This statement is only partially true. This problem only occurs *if* the placement strategies used are incorrect. Of course, the solution is to segment the model correctly – i.e. to align the object placement with the use-case access – such that these excess transfers do not occur. As mentioned before, by careful placement of object instances in the database, it is possible to exploit the page-based architecture so that all the objects necessary and sufficient to complete a particular use-case are located on the minimum number of memory pages. These pages will then be transferred in the minimum number of network packets. This arrangement is maximally efficient.

Paragraph 4, Page 10

“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.”

This generalisation is not borne out if you take into account the problems of server side bottlenecks and the small-object lock management overhead which he conveniently ignores. The issue of future proofing and schema evolution is addressed at length below.

Page Query

Last Paragraph, Page 11

“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.”

This statement is false. Greene appears to think that page-based OODBs do not use indexes during query execution, but this is incorrect. ObjectStore indexes are designed to ensure that only the pages of the index are fetched onto the client when executing the query. The indexes are very dense data structures and are always allocated on their own dedicated database pages, therefore they can be efficiently fetched from the server, and concurrent queries from different clients can run on different machines simultaneously.

If the network overhead of transferring pages is too onerous, then by running ObjectStore clients on the same machine as the server, it is possible to entirely eliminate this network overhead. The server by default memory-maps the caches of all local clients, so that page transfer is really fast, and these effectively work as a single process. So by decomposing your system into an N-tier architecture at the process level, it's possible to run ObjectStore clients on machines either remote from, or local to, the server process, and thus to eliminate network overhead, or distribute processing across multiple machines, as best suites each particular context.

However, putting the client process on the server machine may eliminate network communication overhead, but it does not eliminate page thrashing if, for example, there are a large number of concurrent updaters or, because there is no index to optimize queries. So ObjectStore has some flexibility, but does not cover all cases where server-side query processing is superior.

However, contrast this with a server-centric architecture where all the queries *must* be run on the server machine, there is little option to distribute these queries out and exploit the genuine parallelism of multiple machines when this make sense. A server-centric arrangement can result in server side bottlenecks as the number of concurrent users increases, and the only resolution is to purchase bigger, more powerful, server machines.

Object Query

Paragraph 2, Page 12

"The object based implementation uses a query execution engine that runs within the database server process."

From a theoretical stand-point, using a page-based architecture does not exclude the possibility of executing queries within a database server process. The implication here is that this is somehow a limit on page-based architectures, and only solvable by resorting to an object based design. Whether server-side query processing is offered is a contingent fact about a particular product, not a necessary limit imposed by either page or object architectures.

As mentioned above, using ObjectStore it is possible to decide where you want to execute your queries, on the client machine or on the server. This means some server bottlenecks, such as those arising from CPU contention, can be mitigated, should they occur, by relocating processes onto other machines and exploiting true parallelism.

"Any object in the database is reachable via query even if it has no relationship to other objects."

Firstly a minor logical point on this comment: using a page-based architecture does not exclude the possibility of querying for arbitrary objects within the database.

Theoretically and practically it is possible to find and process any object in any file or database system. This is provably the case when you consider that it could be implemented, albeit inefficiently, as a linear search through the entire database. It is simply a matter of how a particular product is designed, as to whether this service is offered, and if so, how it is implemented.

Although at first sight ad-hoc query might appear to have no drawbacks, in that flexibility is offered – you can query for anything you require whenever you want it and from any point in your code – there are issues here.

One of the most important features of the OO paradigm is encapsulation; the hiding of implementation details behind an interface. Objects are essentially indivisible computational 'atoms', each of which has a contract with other objects or clients. The state of an object's data members are, or should be, private to preserve this encapsulation, and upon an updater method being called, the object implementation enacts the contract, changing this private internal state such that the post conditions and invariants, and other features of the contact, are preserved by the time the call returns. The relations between objects form part of this private state. Because all the code that implements the contract is implemented within the class, and this is the only code that accesses private data members, then by enforcing encapsulation the compiler to a large degree enforces the contract, by automatically excluding the possibility of any external code corrupting the internal state of an object

This arrangement means that if something needs changing, or if something goes wrong, because only code local to an object can possibly access this internal state, it is relatively easy to find, and change or fix the required code. Contrast this to the case where there are public data members which can be accessed from everywhere in the code base. There is no encapsulation in this important sense, and it is plain to see that this compiler-enforced encapsulation is a fundamental feature of OO technology – a very powerful way to control complexity – not just a purist pursuit.

When objects are stored persistently it is this private internal state that is being written to disc. Allowing arbitrary code to access the internal state of any object compromises encapsulation, in this important sense, leaving the objects' state open to change by code which is external to the objects class. This means that any arbitrary code can come along and change the state of any object. In other words the methods on the class can no longer guarantee to be the only code that accesses this private state, so the object contract can no longer be enforced by the compiler. The practical implication here is the *overall coupling of the entire system increases*, it tends to get more complex, harder to maintain and more difficult to test. There is likely to be more runtime checking which may detrimentally impact performance.

So although having ad-hoc access to any persistent object might seem like a wholly good idea, from a theoretical and practical OO perspective, the situation is more complex – there are potential down sides. It should be noted here that this problem is identical to the case where objects are stored in relational tables – which is what many object-based implementations are essentially doing.

“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.”

Firstly, the placement of this statement in the section entitled ‘Object Query’ and an omission of mention of query optimisation under the section entitled ‘Page-Query’ implies that optimization cannot be performed during query processing on object database systems which use a page-based architecture. Of course, this is not the case.

ObjectStore processes queries on the client side, and if suitable indexes are present, these are used to optimize the query processing. The presence of indexes does not in itself alter the way in which database pages are served out by the server process. The index data structure itself, is treated like any other persistent data structure within ObjectStore. As the index is used to optimize the query by the client side processing, page requests for parts of the index structure itself will be sent to the server process. Thus, only those parts of the index that are required to process a query are fetched into the client process. Since indexes are always located on their own dedicated set of database pages, index use involves the minimum number of database pages being transferred to the client – spurious and unnecessary page contention is avoided. With the correct choice of indexes, and in the right context, this optimization process can hugely reduce the number of pages requested from the ObjectStore server process.

Upon returning, a call to the query method returns a collection of pointers to the result set, a set of persistent objects within the database. These objects have ***not*** been accessed by the query processing, so the pages containing them have not been fetched into the client process.

At this point, I need to point out another distinguishing feature of ObjectStore, which contributes to its flexibility and sheer performance. This is the idea of bespoke indexing. Architectures which execute server-side queries will naturally use indexes on the server side, but this usually means that the actual structure of these indexes is down to the database vendor rather than the user. Server-side query processing limits the possibility of the user determining the data-structure actually used by the index. Because ObjectStore permits the storage of real C++ pointers and genuine C++ arrays, the programmer is able to implement efficient bespoke indexing structures which optimally support any particular use-case. If there are multiple use-cases which demand maximum performance, then the target objects can be referenced by bespoke indexes peculiar and optimized to each. This approach sees two broad groups of objects in the database; the business objects, which store the actual data relevant to the domain being modelled and which comprise the whole point of the system, and the access structures, or indexes, which support navigation to the business objects pertinent to specific use-case invocations, or queries. Programmers can implement optimal data structures for processing objects in the database, even for example using STL, just as they would data structures on the heap. These access structures are fetched page-by-page just as are ObjectStore collection indexes when a client-side query is being executed, so these bespoke access structures are exactly equivalent to collection indexes in that they both optimize access to the business objects, and in similar ways. I do not think that this type of bespoke indexing can be implemented unless a client-side query architecture is utilized, or user C++ code is somehow linked into the server side process, which industry experience has shown raises server security and reliability issues.

"Through the use of system wide identifiers, multi-threading and aggregation, parallel distributed queries are possible"

Page based architectures can, and do, use system wide identifiers. Parallel distributed queries are just as possible in a page-based OODB such as ObjectStore, given the key ingredients that clients can be multi-threaded and clients can use persistent objects from more than one database at the same time. Because of address mapping, a client's pointer to a persistent object is the equivalent of a system-wide identifier; no database context is required to interpret the pointer.

ObjectStore also provides system wide identifiers called 'soft pointers' which uniquely identify any persistent object across all ObjectStore databases, globally, at any scope, and independently from the context of any executing process, and these are schema compatible with raw C++ pointers, and across 32-bit and 64-bit architectures. They are also very efficient, in that once used, they operate as quickly as direct C++ pointers.

Page-based architectures can, and do, use multi-threading in both their product implementation and from the programmer's perspective. ObjectStore enables the user to write multi-threaded code using multi-thread safe libraries designed to support multi-threaded coding.

Page-based architectures and ObjectStore in particular, can directly support parallel distributed queries. In fact, as described above, ObjectStore excels at this by allowing system designers to distribute query processing out to client machines, avoiding server bottlenecks, or execute queries local to the server. ObjectStore was designed from the ground up with distribution in mind.

Query Model Implications

Paragraph 4 Page 12

"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."

It is interesting that the subject of navigational access is not addressed adequately here, because this is probably one of the most important features of any object database system, and as Greene states, navigational access is the primary means of retrieving information from an object database.

The fact that navigational access is primary calls into question the implications of the query model as a whole. If the primary means of data access is navigational how can query performance, or the characteristics of the query sub-system, be critical to the overall performance of an object database product? The answer is only in the case where a user chooses to implement their system based on queries instead of navigational access, which, as quoted, is contrary to the focus of most object database vendors. So why is the subject of navigational access so quickly dismissed?

The answer probably has nothing to do with the theoretical differences between object based server architectures and page based server architectures, but more to do with product differences between ObjectStore and Versant.

From this perspective navigational access is one of ObjectStore strengths. ObjectStore can store and use real C++ pointers in the database. Why is this an advantage? Because *in principle* it allows any arbitrary C++ object model to be stored persistently. In practice, there may be other constraints that mean it is not possible, or less than ideal to do so, but the fact remains; you can store any C++ object model in ObjectStore.

C++ arrays underpin the design and implementation of many highly optimized and efficient data structures, such as pool allocators. C++ arrays allow the programmer to reduce the considerable cost of per-object construction by amortizing the cost of a single array construction across each object in the array. When used persistently, the ability to store real C++ arrays means that database writes can be incredibly fast.

If these features are combined, storing real C++ pointers and C++ arrays persistently in the database, and the ability to efficiently navigate through these pointers, then it becomes possible for programmers to design *absolutely optimal index structures* for any given use-case, as described earlier. As far as I know, this is very difficult from within C++ with any other technology.

Once everything is in the client cache, navigational access is probably faster, in a page-based system than in an object-based system because hardware-supported pointers are used and no pointer translation is needed. In theory an object-based system that swizzles pointers when putting objects in its cache is conceivable, although it's harder to do than in a page-based system. So in *theory* there may not be any real performance difference between the two approaches here; any real performance differential in this area between vendors will best be discovered by empirical measurement, and whether this difference is important overall will depend on the details of the system use-cases.

The point that needs to be emphasised here is that navigational access is central to almost all OODB applications, yet this subject is barely mentioned in Greene's paper.

"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."

This paragraph contains many inaccuracies and half-truths. Firstly, page-based server architectures are not ipso facto vulnerable to the requirement of moving pages into another process space while processing queries. If a particular product has been implemented as such then it *may* be, but page-based server architectures in themselves could implement server-side query processing if this was thought to be a desirable approach. Whether or not moving pages into another process space to execute querying is a problem will depend on many things, not least the use-case and the efficiency of the product implementation.

In the case of ObjectStore, as described previously, at length, query processing when suitable indexes are present, only requires, at most, the pages containing the index structure be fetched into the client. I shall not labour the point further here about mitigating server bottlenecks.

Secondly, page-based server architectures are not forced, in principle, to restrict queries to objects which exist in particular collections, which may or may not have indexes. Particular products, may implement a collections library in this way, but this feature is orthogonal to whether a page-based server architecture is used.

ObjectStore does implement queries by providing a query method on its collection classes and this returns a sub-set of the elements within it, based on a complex predicate, and these queries can be indexed and efficiently processed. ObjectStore also allows arbitrary objects to be accessed in two ways; with an object cursor and using dynamic extents. So any orphaned objects can be reached within an ObjectStore database. The reason why ad-hoc queries to global objects is not recommended as the primary mode of access is because of the encapsulation issues described elsewhere.

So Greene's comment that *"only objects which exist in special collections can be the subject of a query"* does not make any real point; it is like complaining that a relational database only allows tables to be queried. In addition it is false anyway, since arbitrary queries are supported by ObjectStore through dynamic extents.

An interesting side issue is the meaning of orphaned objects. Orphaned objects present within an ObjectStore database are those objects that have zero references from other objects. This means that no program can access these objects, unless they use an object cursor. Assuming that navigational access is being used, as is recommended, then these orphaned objects are symptomatic of a persistent memory leak; the client code has failed to delete an object which it no longer needs.

"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."

Finding a single Foo object in a collection of millions of objects is typically a sub-second call in well-designed ObjectStore based systems. If it's not then someone on the programming team has some work to do!

Greene here is painting a very inaccurate picture of ObjectStore index maintenance. Inserting an object into an indexed collection triggers automatic index maintenance. In the scenario presented, where a single Foo object is inserted into a collection of millions, only one, or maybe a few, index pages will be updated. When the

ObjectStore transaction commits (assuming it does) then the pages containing the index data structures are returned to the server and written to disc, no different to any other page. If other clients access the index, these few pages will be served out on an as-needed basis. The updated index pages are only transferred when they are used by other clients, and no other pages i.e. those containing the million Foo objects, are transferred as a result of this index update.

"Further, multi-threading in the client process allows the query execution to be performed in parallel across multiple physical databases."

Greene is not really differentiating object-based and page-based server architectures here. The context of this sentence implies that this feature is only enabled by the server-side query execution, and by implication, only object-based server architectures. The truth here is that page-based server architectures do not in themselves enforce a single threaded programming model, or stop querying being performed in parallel across multiple physical databases. The proof is there exists a page-based server architecture that offers just these features. ObjectStore, a page-based server architecture if ever there was one, enthusiastically promotes and supports multi-threaded programming on the client side and allows clients to access multiple physical databases distributed around a network, residing on different platforms, each under the jurisdiction of a different ObjectStore server process, and all from within the same transaction. What Greene says here does not differentiate page-based and object-based server architectures, and it does not even differentiate products to any real extent.

"With page … 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."

The fact that index management code is or is not integrated into application code is not related to a system being a page-based server. ObjectStore has several ways to offer index maintenance, some of which, as Greene observes, require minor changes to application code. But this code is not complex, or in any way a problem, it's just code that does a useful job. From a code-maintenance perspective ObjectStore indexes do not usually present an issue. In many situations explicit index maintenance is unnecessary, and is automatically handled by the ObjectStore runtime.

It is also untrue that in ObjectStore indexes cannot be defined without changing your application, because this functionality can be provided by writing another dedicated index management application tailored to manage your particular indexes. The API for adding indexes is very straight-forward and for C++ programmers, as easy as writing DDL statements to add indexes onto relational tables. It is therefore a relatively simple matter to write a stand-alone application to add or drop indexes onto any collection in the database, even while the system is in use by other applications, and those applications will exploit such indexes as are created as soon as the transaction that adds them has committed. So ObjectStore indexes can be added and removed, dynamically on-the-fly and without changing the main application code.

Identity Management

Paragraph 5, Page 13

"In addition, the particulars of implementation for physical identity have an impact on data scalability for systems requiring storage of multi-terabytes of information."

A 64-bit pointer can address 17,179,869,184 gigabytes or 16 Exabytes of RAM. How big ARE your datasets? Seriously, having 64-bits of persistent address space hardly constitutes a practical limit on data set size, so if Greene is implying here that this is an objection to page-based server architectures he is mistaken.

ObjectStore does have a limitation of 2Tbytes per physical database file, but one can use multiple physical databases to work around that limit. In most real-world deployments, large datasets are usually decomposed into multiple physical databases each significantly smaller than this size, and distributed across many server machines, so practically this restriction never presents a problem.

Physical Identity

Paragraph 6, Page 13

"The characteristics of object physical identity are: mutable, reusable, immobile, rigid."

Greene neglects the most positive aspect of physical identity here. They are characteristically very fast to de-reference – much more so than logical identity tokens, which require access via some form of indexing structure. It is precisely the 'immobile' and 'rigid' characteristics that make physical references so quick.

I am not denying that in principle both object-based and page-based server architectures could implement object access using physical identity, but in practice page-based servers can exploit this idea relatively easily. In the case of ObjectStore, raw C++ pointers provide for extremely fast access to data, and a very intuitive C++ programming style, which makes working with the database libraries very productive.

Last Paragraph, Page 13

"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 …"

What Greene says here is true. Object-ID based addressing has an advantage over 'physical', location-based addressing when it comes to schema evolution. Using object-IDs allows schema evolution to be done online and incrementally. Objects can be evolved as they are encountered, because they can be moved around without changing their address if they no longer fit where they used to be. This is definitely an advantageous feature of object-ID based reference.

However, this does not come for free, there is a downside. The client pays for this added flexibility by having to translate from an object ID to a physical address every time an object is accessed. Computers cannot use object IDs, they have to have a physical address at some point to access the object. No doubt Versant have implemented some degree of caching to prevent the cost of this from being prohibitive, but at some level there is an architectural trade-off here, between performance and flexibility. With object ID based addressing you pay for schema evolution continually during production and deployment, whereas with location based reference you pay for it all at once, but only when you use it.

So, the questions remain: How difficult is it to evolve an ObjectStore database? And how fast is schema evolution?

Schema Evolution Complexity

From an engineering perspective it is immensely complex, because unlike a relational database where the persistent schema is known and constant – all the data is in tables, columns and rows – the persistent schema is entirely arbitrary, defined by the user's actual C++ code. So any generic schema evolution utility must be able to handle literally any possible C++ object model that it might encounter, and be able to transform this into any other. To mitigate the degree of complexity and for performance reasons, the ObjectStore evolution utility mandates off-line evolution, and, to accommodate the sheer complexity of possible schema changes, it is common for programmers to write specific C++ evolution code and link it with the ObjectStore schema evolution library to evolve a database.

So from the user's perspective it also involves some complexity. Let's look at a couple of examples:

Initializing a Newly Added Data Member from an Existing One

In C++ it's prohibited to have two classes with the same name but with different memory layouts in the same address space. Yet here it appears necessary to do exactly this so we can access the old data value while creating the new object. There are several ways to achieve the desired result here, but the one used by the ObjectStore evolution tool involves three steps.

1. Do a schema evolution to add the new data member. It will be initialized by default to zero. Don't remove the old data member in this step.

2. Run a relatively simple user program that finds all these evolved instances in the database and initializes the new data member according to the contents of the old data member. There are several options here:
    - Using an object cursor
    - Navigating through application-specific data structures
    - Incrementally, the first time the application program updates the evolved objects.
    - Combined into step 1 of the evolution process by using a post-evolution transformer function. Schema evolution will automatically call this function once for every evolution candidate encountered.

3. The last step is to run a cleanup process that removes the old data member. This cleanup can be done at any convenient time since it makes no logical difference, it only saves storage.

Instance Re-Classification

Instance reclassification is where an existing leaf class, one with no subclasses, is refactored by introducing several subclasses. Each existing instance will be assigned to one of the subclasses according to an application-specific criterion. Again this is a three step process.

1. First the customer adds the new subclasses to their schema. Each new subclass simply includes the base class without adding anything to it. The definition of the base class is unchanged.

2. Next the customer writes and runs a program which examines each instance of the base class, chooses a subclass, and changes the instance's type using a call to an ObjectStore function to change its type. At this point there are no changes to the persistent layout of the object so all pointers referencing the re-typed object remain valid

3. Finally the base class and the subclasses are changed to their intended definitions. This may involve deleting the data members of the base class and adding some of them back as data members of the subclasses. After compiling the new schema, the customer runs schema evolution.

Many evolution scenarios can be accommodated directly by the direct use of the ObjectStore schema evolution tool. The complexity of finding all the objects, evolving them, relocating them onto other pages if necessary and fixing up reference pointer values and types are all provided for by the tool. The details of when it's not necessary to write C++ to evolve a database is best described in the *ObjectStore Advanced C++ API User Guide* [2], to which the interested reader is referred.

Where it is necessary for the user to write some C++ code, this code is relatively straight-forward and usually involves finding all the 'evolvees' in the database and either calling an ObjectStore function or initializing data members. Either way, the code is generally fairly pedestrian. The complexity of moving evolved objects to new locations if they change size, and fixing up the references, both their values and the pointer types, is provided for by the schema evolution library.

The point here is that ObjectStore schema evolution *is* complex from an engineering perspective because of location based addressing, but much of this complexity is hidden from the user by database evolution utilities supplied with the product.

Schema Evolution Speed

This question is a version of the classic 'how long is a piece of string?' It depends on the context, the details of the schema evolution, the speed of the disc system, the size of the database, the amount of physical RAM and the number of CPUs on the schema evolution machine.

The ObjectStore schema evolution tool has been designed and optimized to work most efficiently in high-end server environments, where ObjectStore databases are typically deployed.

There are three basic phases to schema evolution:

1. The first pass executes pre-evolution transforms, the details of which are unimportant here.

2. In the next pass the location of all the evolution candidates are ascertained and their new target locations are calculated, and this information is recorded in a highly compressed, but fast, map from old location to new location.

3. Lastly, the evolution phase fetches all the pages in the database, evolves the objects, moves them to their target locations if required, and fixes up the value of any referencing pointers. It is necessary to fetch all the pages because it would be far too expensive to pre-compute which pages contain *references* to evolution candidates. Furthermore, since the order of objects in persistent memory is not changed, evolved objects are not the only objects that move.

However, in terms of page fetches, there are not three passes through the database. The first two phases can be executed without fetching and mapping the pages containing the actual objects. Every page in an ObjectStore database has associated meta-data, which gives the location and type of every object and pointer on the page. This information is highly compressed and typically much smaller than the page itself. The first two phases only need to read this meta-information, and so run orders of magnitude faster than if the entire page was fetched and mapped. So in terms of page fetches, this is not three passes through the database, but more like 1.2 passes.

Greater performance is also achieved by judicious use of multi-threading. For most of the duration of all these phases, evolution can exploit multiple threads of execution, which efficiently utilizes multi-CPU boxes typical of high-end servers. The overhead of thread synchronization is reduced by assigning each cluster in the database to a thread, so the thread can process the entire cluster, and then when complete, move on to the next cluster.

The last phase, where the pages are actually fetched does not require the pages to be actually mapped, consuming address space. Evolution can take place on the 'relocated-out' form of the page. This means that the evolution can scale to quite substantial databases in the 20 gigabyte range. ObjectStore datasets larger than this are typically split into multiple separate physical databases, and distributed across several machines. Each of these databases can be evolved separately and in parallel, so this size more than adequately accommodates most situations.

In the last phase where there is significant disc I/O, (it is assumed that schema evolution processing is NOT run across a network for obvious reasons), I/O is done in large sequential blocks, not random access which would be much slower.

All this combines to give a throughput equal or close to the disc write speed on a four CPU machine; so something in the region of 7 gigabytes per hour. The performance has been measured to be approximately linear with the size of the database. The number of objects to be evolved does not appear to make much difference, except if it's zero, in which case the last phase is skipped.

There is one remaining question. If schema evolution is so efficient partly because it does NOT fetch and map pages, what makes the page-based server architecture good for other programs? Surely this is evidence in favour of using some other architecture?

The answer is that the page-based server architecture allows C++ clients to be written as if they were working on in-memory data structures instead of having to use a special API to navigate through the database. To the programmer using ObjectStore the best metaphor is of a ‘frozen transactional heap’ rather than a database. This reduces application complexity and also increases the speed for applications that do a lot of work on a subset of the objects which can be cached.

The schema evolution use-case is just the opposite. It touches every page in the database but does a very small amount of work on each one.

Paragraph 4, Page 14

“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.”

This is simply incorrect. Greene seems to be assuming that the entire database has to fit into a client's virtual address space. Of course that's not true. Only the pages (actually 64 KB ranges of pages) that are actually touched, and those that are referenced by ‘hard’ pointers on touched pages, consume virtual address space. Furthermore virtual address space is dynamically reassigned by ObjectStore to allow an application to access a larger portion of persistent storage. Even on 32-bit client machines there is no limitation on database size except for limitations on file size, and no restriction against federating multiple databases. With the advent of 64-bit virtual address space there are really no practical limits on address translation.

Conclusion

So to finish, I think it has been shown that Greene's paper contains many inaccuracies and misrepresentations. The aim of this paper is to re-establish a more honest discussion of the advantages and disadvantages of page-based and object-based server architectures. Hopefully, this paper goes some way to enabling readers to make a more informed judgement about which object database suits their needs.

References

[1] OODBMS Architectures: an examination of implementations, R. Greene, 2006
http://www.odbms.org/download/028.01%20Greene%20OODBMS%20Architectures%20September%202006.PDF

[2] Advanced C++ API User Guide, Chapter 7 Advanced Schema Evolution, P.201
http://media.progress.com/realtime/techsupport/documentation/crystal/crystal13/customcd/OStore/6.3/doc/pdf/user2/user2.pdf