

How to Evaluate an Object Database



Table of Contents

Table of Contents	2
Figures	3
Introduction	4
The Product Category	4
Pure ODBMSs	4
Persistent Storage Managers	5
Object Wrappers	5
Object-Relational Databases	6
Product Architecture and Functionality	6
Performance	7
Concurrency and Lock Granularity	7
Client-Server Implementation	9
Server-centric Approach	9
Client-centric Approach	9
Balanced Client-Server Approach	9
I/O Efficiency	10
Queries	12
CPU Utilization	13
Benchmarks	14
Scalability and Distribution	15
Transparent Data Distribution	15
Replication	17
Thread Utilization	17
Productivity	18
Programming Language Access	19
Schema Management and Evolution	20
Tools Support	21
Vendor-Supplied Tools	21
Third-Party Tools	22
The Right Vendor	22
Conclusion	23
Appendix: Object-Oriented Database Evaluation Checklist	24
About Versant	32
Versant Overview	32
Versant History, Innovating for Excellence	32

Figures

Figure 1. Roles in the database market. Many factors motivate the rise of objects, including the use of object programming languages and complexity of data. Supporting distributed, multi-user applications that manage objects requires the use of an object database. 6

Figure 2. Object vs. page locking. Object locking maximizes concurrency and minimizes contention by only locking objects that are actually in use. Page locking, in contrast, locks entire pages—needlessly blocking access to data, causing spurious deadlocks, and slowing overall performance. Some vendors try to masquerade page locking as object locking by costly object relocation to empty pages, or by wasteful object padding — making objects so large that only one object fits per page. 8

Figure 3. Three client-server architectures. The balanced client-server architecture is a hybrid approach that provides the best of both worlds: the network traffic reduction of server-centric architectures combined with the ability to exploit client-side processing power found in client-centric architectures. 10

Figure 4. Saw-toothed performance. On-line space reclamation, reorganization, and reuse is critical to avoiding saw-toothed performance. Systems that lack important space management capabilities require periodic, manual unloads and reloads to sustain performance. 11

Introduction

The move towards object database management systems (ODBMSs) is being driven by a number of important forces in the computing industry. These forces include the requirement to manage new types of complex data often associated with graphical, client-server applications, the desire to persistently store objects manipulated in object programming languages, and the need to manage this information in an environment that provides the traditional benefits of database management systems (DBMSs).

Application developers have historically turned to one of two choices to store information: flat files or relational database management systems (RDBMSs). While flat files let programmers store arbitrarily complex data, they lack the ability to coordinate concurrent access, manage data distribution, and provide transactional integrity. Relational databases offer these traditional database benefits, but their table-oriented data model is unable to adequately model complex data and relationships, and is unable to store programmatic objects in their native form.

Object databases offer the best of both worlds: the flexibility to store arbitrarily complex data objects combined with the power of true database management systems. Thus, for organizations managing complex data, using object-oriented programming languages, or both, the decision to move from conventional to object database technology is often surprisingly simple.

More complicated, however, is the selection of a particular ODBMS. Understanding the subtleties of the various products on the market can be time consuming and difficult. Since object databases are a relatively new technology, it can be hard for prospective users to determine which issues are relevant to consider during ODBMS evaluation. Together, these factors can make ODBMS evaluation a bewildering experience for the prospective user.

As a leader in the object database industry, Versant has assembled this guide to assist prospective users in identifying the criteria for evaluation of object database management systems. As with relational databases, the selection of an ODBMS represents a long-term investment, and is a decision that should be made with care. If there are further questions about this guide, or about Versant products, please feel free to contact us at 1-800-VERSANT, or at +1-510-789-1500. Versant can be reached via email at info@versant.com and found on the web at <http://www.versant.com>.

The Product Category

All ODBMSs, and products that resemble ODBMSs, fall into one of four categories. The first key question in evaluating ODBMS and ODBMS-like products is to determine which category of product best fits the application's needs.

Pure ODBMSs

Pure ODBMS products provide traditional database functionality (e.g., persistence, distribution, integrity, concurrency, and recovery), but are based on an object model. They typically provide permanent, immutable object identifiers to guarantee data integrity. Pure ODBMSs also generally provide transparent distributed database capabilities (e.g. transparent object migration and distributed transactions) and they often include advanced database

functionality as well. Like their relational peers, pure ODBMSs typically come with integrated database administration tools.

Pure ODBMSs are best suited to object-oriented data management applications that must support multiple users. These applications also tend to have significant requirements in the areas of scalability, concurrency, distribution, and/or performance and transaction throughput. Often, users requiring pure ODBMSs have tried a project using an RDBMS and found the relational model too restrictive. In other cases, users migrate to pure ODBMSs from hard-to-maintain internally developed database managers and file systems.

Persistent Storage Managers

Persistent storage managers (PSMs) developed from the requirements of the standalone computer-aided design (CAD) market, and are designed to manage the persistent storage of a relatively small number of objects. PSMs typically lack permanent object identifiers, and thus offer a lower level of data integrity than pure ODBMSs. These systems were designed to meet the needs of standalone CAD and other applications with relatively little data and low concurrency requirements. They typically offer limited or cumbersome data distribution, coarse-grained locking, poor on-line space management, and no schema evolution. Standard maintenance tasks in such systems typically require a database shutdown, reconfiguration, and restart - which may be an acceptable maintenance cycle for a single-user system, but is typically inappropriate for multi-user database applications.

In standalone applications, persistent storage managers are often a good fit. In multi-user environments, however, developers usually elect to use pure ODBMSs because of their multi-user and distributed data management capabilities. Persistent storage managers are generally not designed to meet the requirements of the broader market, and are best used for their intended purpose of local, standalone persistent storage for programming language objects, such as is found in standalone CAD applications.

Object Wrappers

Object wrappers are software layers placed on top of RDBMSs that provide an object-oriented interface to the underlying relational database manager. Object wrappers have typically not been built with the needs of any particular application in mind, but rather have been constructed to try to capitalize on the general marketplace interest in object technology. Object wrappers offer some of the generic benefits of object technology, such as improved programmer productivity by supporting re-use of code.

While they may look good on paper, the object wrapper's reliance on a relational database as the underlying storage manager presents serious performance problems for complex data management. While "papering over" an RDBMS with an object-oriented interface may improve programmer productivity in some areas, it does nothing to solve the fundamental problem that certain types of data are difficult to model in an RDBMS. In order to achieve any sort of reasonable performance, wrappers generally place severe restrictions on an application data model. Potential users have realized this fundamental flaw, and object wrappers have generally not fared well in applications that manage complex data. Object wrappers are perhaps best viewed as programmer productivity tools-like 4GLs-and not as a separate class of database management system.

Object-Relational Databases

Many relational database vendors are now attempting to respond to the market's enthusiasm for object technology. While some relational database vendors are pursuing the object wrapper strategy, others are attempting to patch object-like functionality into their relational databases. Like their network database predecessors who responded to relational databases with extended network databases (e.g., IDMS/R), several relational database vendors have responded to object databases with extended relational or object-relational databases (i.e., ORDBMS).

Relational databases offer a flexible and powerful environment for the management of conventional character and numeric data. However, none of the common extensions to RDBMSs (e.g., triggers, stored procedures) increase the RDBMSs ability to model and manage the more complex data that is the hallmark of ODBMSs. In addition, no extended RDBMS offers the standard benefits of the object model. For example, while stored procedures can offer a limited form of encapsulation, they fail to adequately support inheritance and polymorphism, and thus do not deliver the reusability, productivity, and application quality offered by object technology.

Since they do not support the O-O paradigm completely, and do not model complex relationships (i.e., 1:N, M:N) efficiently, object-relational databases are perhaps best viewed as "better relational databases" rather than as limited-function object databases. ORDBMS products are most suitable for use in the wide range of applications in which RDBMSs already prevail. To put it another way, beneath the object extensions provided by object-relational databases are the same old relational engines with the same old limitations that are associated with the handling of complex data.

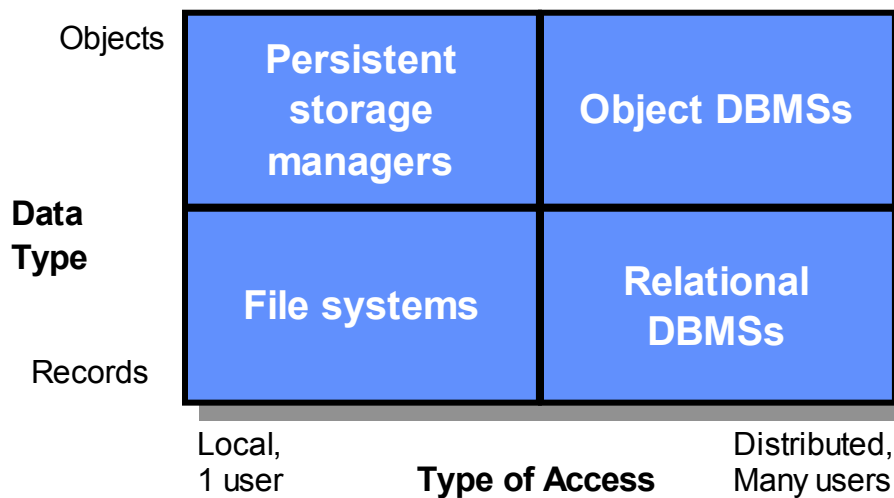


Figure 1. Roles in the database market. Many factors motivate the rise of objects, including the use of object programming languages and complexity of data. Supporting distributed, multi-user applications that manage objects requires the use of an object database.

Product Architecture and Functionality

Analyzing these four general categories of products should help give a sense for which type of product is required to meet the needs of a particular application.

Performing this analysis assists in paring down the large number of ODBMS and ODBMS-like products to a smaller set (typically two to four) in order to perform a more detailed analysis of each product's ability to meet the requirements of the target application.

Key areas to examine in any ODBMS evaluation include performance, scalability, distribution, and application development productivity. A discussion of each area follows.

Performance

High performance management of complex, non-record-oriented data is one of the key advantages of ODBMSs. Although somewhat dated, the industry-standard Cattell 001 benchmark [R.G.G. Cattell, Object Data Management: Object-Oriented and Extended Relational Database Systems, 1991] clearly demonstrated that all ODBMSs are faster than even the fastest RDBMSs in the management of complex data. Determining whether the performance of a particular ODBMS will be suitable for a target application can be accomplished in three ways:

- Analyzing product performance features;
- Reviewing industry-standard benchmark results; and
- Writing custom benchmarks of the application.

The easiest way to analyze performance is to investigate support for certain features in each of the candidate ODBMSs. While feature analysis can never guarantee that an ODBMS will meet application performance requirements, feature analysis can be successfully applied to determine which ODBMSs are likely to suffer from critical bottlenecks that will limit their performance. Thus, feature analysis is better used to eliminate potential products because of implementation weaknesses, rather than to ensure overall performance.

Following are some important areas to investigate when evaluating ODBMS performance features.

Concurrency and Lock Granularity

A key determinant of performance is the level of concurrent access supported by the database. An important goal of any database is to maximize concurrent usage while minimizing contention - and waiting - for resources. It is well established in RDBMSs that row-level locking is the preferred lock granularity to minimize contention. Leading relational databases such as DB2 and Oracle lock at the row level, which is the relational analog of locking at the object level. Furthermore, industry benchmarks demonstrate that row-locking systems are faster than page-locking systems. For proof of this point, one only needs to compare the results of the TPC benchmarks as performed by Oracle with the results of the same test run with Sybase—arguably the fastest page-based relational database engine.

The typical alternative to object-level locking is page-level locking. Most ODBMSs lock at the page level. Page-level locking can cause severe concurrency problems in multi-user applications. This relatively coarse level of locking granularity can result in "false waits," where two users working on different data collide with each other, causing one to wait because the data elements happen to reside on the same page. False waits not only cause undesirable lock waits, but can also cause false deadlocks, where transactions are needlessly rolled back and reapplied, wasting system resources and irritating end users. It

is not at all uncommon for even a handful of applications (or concurrent users) to literally be brought to their knees in terms of database throughput as a result of false wait and false deadlock situations.

There is yet another level of lock granularity that exists in the ODBMS marketplace. This level of locking granularity, termed “container level locking,” is of an even coarser level of granularity than page-level locking. In the container-based model, all objects in a database are required to reside in an arbitrary collection called a *container*. There can be multiple containers in a database, but the total number of containers is quite finite. As with page-level locking where all objects on a page are locked, all objects in a container are locked when a container is accessed. Since a container has the potential of holding a large number of objects, this model can also cause severe concurrency problems in multi-user applications, just as with the page-based locking model.

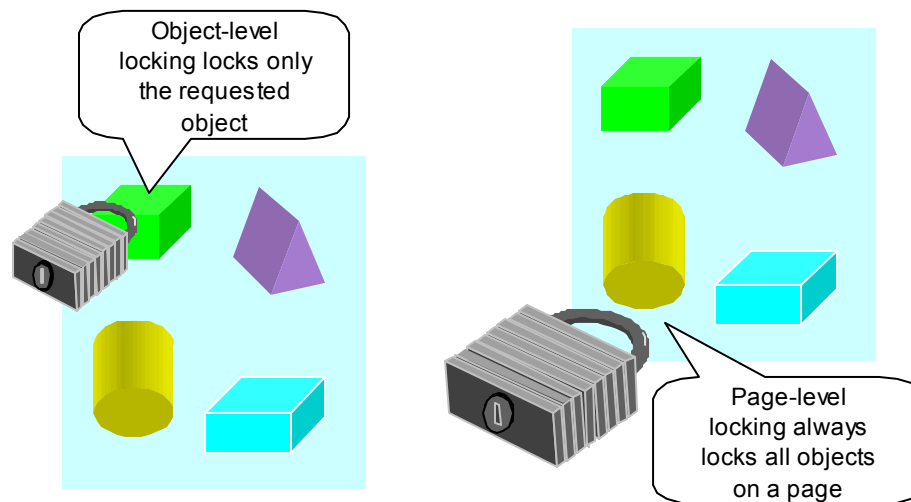


Figure 2. Object vs. page locking. Object locking maximizes concurrency and minimizes contention by only locking objects that are actually in use. Page locking, in contrast, locks entire pages—needlessly blocking access to data, causing spurious deadlocks, and slowing overall performance. Some vendors try to masquerade page locking as object locking by costly object relocation to empty pages, or by wasteful object padding — making objects so large that only one object fits per page.

Of equal importance to the level of lock granularity supported by the database is the flexibility of the locking model implemented by the database. Generally speaking, there are two models that have been implemented by the various ODBMS vendors. Virtually all ODBMS products support the very traditional pessimistic, or 2-phase, locking model. This is the same model that has been implemented by the various relational database products, and is represented in programmatic constructs by semaphores and mutexes. Pessimistic locking is a very rigid model suitable for concurrent transactions that are relatively short in duration.

In addition to the pessimistic model, some ODBMS vendors have implemented variations of another model called “optimistic locking.” The optimistic model is very useful in highly concurrent, “read mostly” transaction environments. Optimistic locking reduces or eliminates the need to wait for a lock to be granted before a process can access data. A true optimistic model allows any process to access and update any object at any time. The variations on the truly optimistic model implemented by some ODBMS vendors limit updates to a particular, pre-specified process. These variations go by names such as “multiple readers, one writer” and “multi-versioned concurrency control.” In most cases they prove to be not nearly as flexible or scalable as a truly optimistic model.

Thus, key concurrency control questions to consider include:

- Does the ODBMS lock at the page or object level?
- How are false waits minimized or eliminated?
- Can false deadlocks occur?
- Does the ODBMS support both 2-phase (pessimistic) and optimistic locking?

Client-Server Implementation

While virtually all database management systems are built with a client-server design, there are three major ways to implement the client-server model. Each has its own distinctive strengths and weaknesses.

Server-centric Approach

RDBMSs, for example, implement a server-centric model where the client application performs no database functions, and merely sends SQL to the server for database processing. While this model is appropriate for host-based computing, the server-centric model has been made less pragmatic by the current generation of high-performance client workstations equipped with many megabytes of low-cost memory. Performing all database processing on the server has two key disadvantages in the modern client-server environment. First, it requires network I/O for all database interactions. Second, it fails to exploit the power of today's client machines, which means that users underutilize their client hardware, and are forced to move to costlier, more powerful servers. In many ways, the server-centric RDBMS implementation of the client-server model is a holdover from the days of host-based computing.

Client-centric Approach

Most ODBMSs place more processing responsibility on the client. They typically implement a client-centric model, where the server side of the ODBMS is reduced to an unintelligent page server, incapable of performing any function other than sending requested pages to clients. This client-centric approach heavily exploits client-side processing power. The trouble with the client-centric approach is these page servers are unaware of object structure or content, and are thus unable to perform typical database activities such as queries and index maintenance on the server. This causes excessive network traffic because all data and indices needed to support a query must be moved from server to client memory before the query can be completed. For standalone, single user systems, this is not troublesome because the client and server reside on the same machine. For multi-user client-server systems, however, the ODBMS must take better steps to minimize network traffic.

Balanced Client-Server Approach

The third major approach is a balanced client-server implementation that partitions database functionality between client and server. Balanced client-server systems typically perform some database functions on the client (e.g., transaction management, session management), and some on the server (e.g., lock management, logging, queries, index maintenance). The balanced client-server design minimizes network transfers by performing dual caching—on both the client and server. When required, the server will retrieve and cache disk pages, permitting subsequent access to other objects collocated on cached pages, returning those objects to the client without any additional disk I/O being required. Repeated reads of objects cached by the database in client memory (also known as warm object traversals) can be performed without any network traffic at all. In addition, the ability to perform server-based query processing ensures

that only requested data will be transmitted from server to client, again minimizing network traffic and maximizing performance.

Key questions to consider related to client-server architecture include:

- Does the ODBMS implement a server-centric, client-centric, or balanced client-server design?
- How does the particular client-server implementation impact network traffic?
- Are objects or are pages the unit of client-server transfer?
- Is dual caching (i.e., caching at both the client and server level) supported?

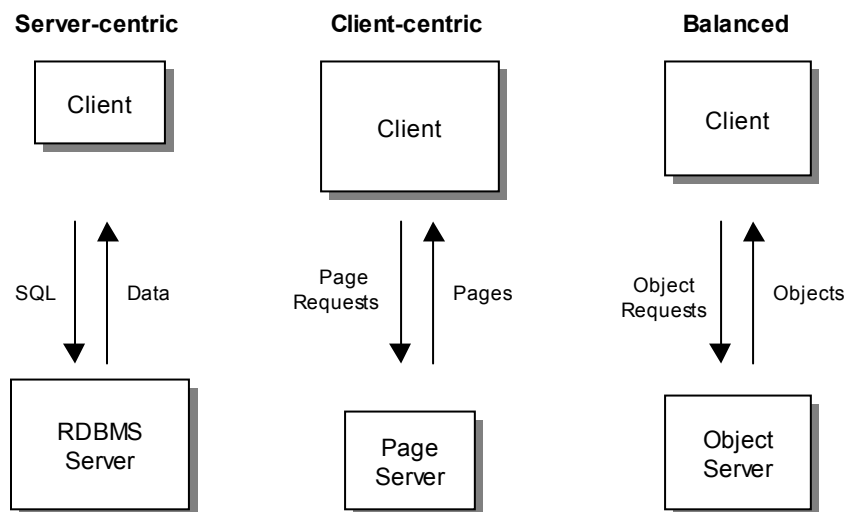


Figure 3. Three client-server architectures. The balanced client-server architecture is a hybrid approach that provides the best of both worlds: the network traffic reduction of server-centric architectures combined with the ability to exploit client-side processing power found in client-centric architectures.

I/O Efficiency

I/O efficiency is an unglamorous and often overlooked area of database technology. Nonetheless, in the vast majority of commercial applications, database performance (object, relational, or otherwise) is more dependent on I/O than on any other computing resource, because the performance "cost" of I/O (both disk and network) outweighs other costs by orders of magnitude.

Perhaps the most important item to consider is whether a given ODBMS's I/O subsystem will support sustained performance as time passes. One of the most common problems with databases is "saw-toothed" performance where, upon first install and after database reorganizations, performance is excellent. Performance will typically degrade over time due to database fragmentation, however. Systems that fragment the database do not support "sustained performance," because they are unable to perform two important tasks: on-line reclamation of deleted space (which also prevents databases from monotonically growing in size), and on-line reorganization of data. Benchmarks typically do not detect saw-toothed performance because they are not typically run for a sufficient amount of time to

test the anti-fragmentation techniques (or lack of them) implemented in a given system.

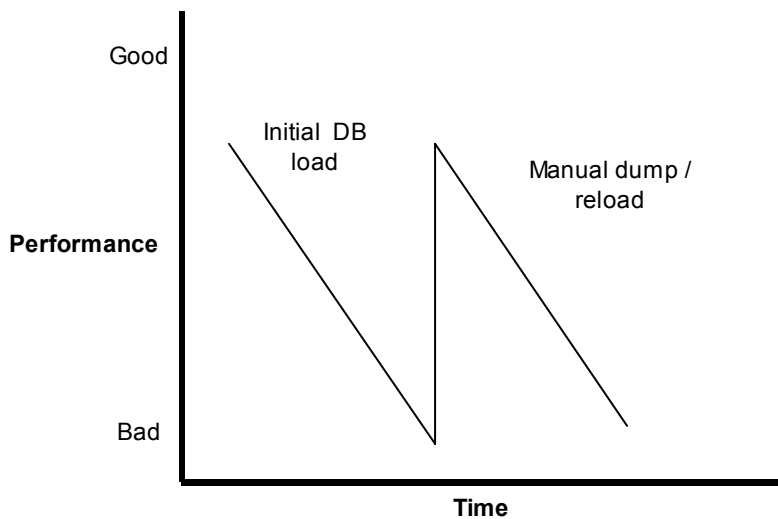


Figure 4. Saw-toothed performance. On-line space reclamation, reorganization, and reuse is critical to avoiding saw-toothed performance. Systems that lack important space management capabilities require periodic, manual unloads and reloads to sustain performance.

Another aspect of disk management is the product's ability to explicitly collocate - or cluster - data on disk with other data that would typically be accessed at the same time. Good physical clustering is key to reducing disk I/O, particularly in large databases. It is also useful if objects can be re-clustered as the objects they are associated with change over time.

Disk access is not the only form of I/O one should be concerned with, however. Network I/O efficiency is also important in a client/server environment. In particular, a database should be able to move groups of objects to and from the client when it is appropriate for the application. To optimally use network bandwidth, only objects that are explicitly requested by the client should be returned. Page-based ODBMS systems pass all objects that reside together on a physical disk page to the client, whether they are needed or not. A more optimal solution is for the client to be able to read and write logical groups of objects when required by a particular pattern of access. (For a more detailed explanation of clustering - both physical and logical - see the Versant white paper *Database Scalability and Clustering*, which can be found on the web at <http://www.versant.com>.)

Finally, it is important that a database can be expanded as the amount of data stored in it grows over time. It should therefore be possible to add additional database volumes as they are needed and possibly long after the database was initially created. For I/O efficiency, it should be possible to place those volumes on separate disk drives and controllers. In 24x7 applications, it is important that additional volumes can be added with the database on-line.

As a database grows larger, the potential for database "hot spots" increases. A hot spot is a volume within a database that is accessed with great regularity while other volumes are accessed much less frequently. Database hot spots can become a bottleneck to data access, and can thus impact performance. To avoid hot

spots and assure scalable performance, the database should be able to consistently take full advantage of all disk devices and I/O controllers on the system. The database should allow data to be physically partitioned - and clustered - across these devices based on logical patterns of access.

Key questions related to I/O thus include:

- How does the I/O subsystem support sustained performance?
- Does the system suffer from "saw-toothed" performance over time because of internal fragmentation?
- Does the product dynamically re-use deleted space?
- Does the ODBMS perform on-line data reorganization?
- How else does the product provide for sustained clustering of data?
- Can disk volumes be added without taking the database offline?

Queries

Queries consume disk I/O bandwidth as data is retrieved and network bandwidth as that data is passed from server to the client and back again. While many ODBMS products fail to recognize their importance, server-based evaluation of queries is extremely important for reasons of scalability and concurrency. This avoids wasting precious network and client memory resources to transport and instantiate data that the client does not actually require. Additionally, to minimize data pages that are accessed, the database server should support and automatically maintain various index mechanisms (i.e. b-tree and hash).

The server should also provide automatic query optimization to ensure that complex queries are processed in the most efficient manner possible. The query optimizer should be able to recognize navigational query structures (where object references are transparently traversed on the server to resolve the query). It should not only support traditional predicate operators such as "and / or" compounding, equality, less than, greater than, and string wildcards, but also advanced set-based query mechanisms such as intersection, subset, superset, etc. Additionally, it is useful if the database server supports collection-based queries to avoid having to fetch the contents of a (potentially large) collection across the network into client memory to be scanned for particular elements of that collection. To facilitate timely response and efficient client memory utilization, the database should support a movable cursor window for queries that return large result sets.

Some products require that objects reside in some sort of collection or container in order to be queried. As mentioned above, a collection-based query is a useful feature. Use of this mechanism should not, however, be a requirement. The mechanism requires that the application explicitly add all queryable instances to a collection in order to be queried. If there are varied patterns of access, the application might actually have to add a new instance to several collections - and remember to remove it from those collections when the instance is later deleted from the database. Additionally, the ODMG standard requires that the database automatically maintain class extents so that whenever new instances are committed to the database they are immediately available for query whether or not they reside in a collection.

Some points about queries that are important to consider are:

- Does the product perform queries on the server?
- Are indices maintained automatically on the server?
- Does the product provide automatic query optimization?
- Does the product support navigational queries?
- Does the product support advanced query operators such as set operators and string wildcards?
- Does the product automatically maintain class extents? Can it query all instances of a particular class in the database?

CPU Utilization

The previous sections discussed memory usage, caching, and disk and network utilization. Another major computing resource to consider is CPU. While I/O typically dominates database performance, certain applications that repeatedly reread cached data (i.e., perform "warm" object traversals) can be sensitive to CPU utilization of the ODBMS on warm object traversals.

Some ODBMS vendors have overzealously cut CPU utilization on warm traversals at great costs in product functionality, distribution, heterogeneity, and data integrity. While the temptation exists to minimize CPU utilization per traversal regardless of real-world application requirements, users should be sensitive to the fact that warm traversal performance is not a quest unto itself, but rather one of many requirements for a complete ODBMS application. Users are encouraged to set aggressive warm traversal performance thresholds that must be surpassed, but they should also focus on the many other factors contributing to ODBMS performance (e.g.. cold traversals, lock granularity, caching, query processing, disk space management). Be aware that some vendors and many benchmarks encourage a misleading focus based solely on warm traversal performance that can leave customers with an inaccurate view of real-world performance, which by the way is typically dominated by other factors – often by orders of magnitude.

The fact of the matter is that most ODBMSs can perform warm object traversals at rates far beyond what even the most demanding applications require. Too much focus on warm traversal performance is misleading because real-world performance is typically dominated by other factors such as disk and network I/O. To get excellent real-world performance, ODBMSs should optimize the most expensive computing resources (i.e. those with the greatest latency and overhead), not the least expensive.

In today's day and age, where SMP systems are relatively commonplace, of far more importance to the CPU utilization question is the database's ability to fully exploit all CPUs made available to it on both the server and the client. It is important to know whether the database product under consideration uses threads inherently, and whether or not those threads are true operating system threads that may be scheduled asynchronously on multiple CPUs.

Key questions in the area of CPU performance include:

- What factors (e.g. object manipulation, interprocess communication, method invocation, database I/O, network I/O, etc.) will really dominate performance of the target application?

- What tradeoffs (e.g. performance vs. functionality, distribution, heterogeneity, or data integrity) have been implemented in the ODBMS related to warm traversal performance?
- What are the warm traversal requirements of the target application?
- Can the product take full advantage of SMP resources by utilizing threads on both the client and server?

Benchmarks

There are two industry-standard benchmarks for measuring ODBMSs. Sadly, both are quite dated, and neither has been updated in a number of years. The first, called the Cattell 001 benchmark, was published by Rick Cattell of SunSoft, and measured single user performance on a mix of operations on complex CAD data. As mentioned previously, the most important observation to take from the Cattell benchmark is that ODBMSs are much faster than RDBMSs for the management of complex data. The other benchmark, which was performed at the University of Wisconsin and called the 007 benchmark [Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton: *The 007 Benchmark*, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. ACM Press 1993: 12-21], examined only a few ODBMS products and limited its testing to common feature sets in a largely single-user environment. The 007 benchmark thus did not fully exploit the capabilities of the products tested, and also did not adequately explore concurrency issues that would normally be a part of any detailed scalability testing.

The Cattell benchmark is not particularly useful for comparing ODBMSs, as each ODBMS won at least one of the various tests (thus all ODBMS vendors have claimed victory on this benchmark). While the 007 benchmark did in fact attempt to compare ODBMS products, as with the 001 benchmark, all vendors who participated won some number of tests and thus claim victory in this benchmark as well. Therefore, users should both be wary of these standard benchmark performance claims and carefully investigate the individual tests in question to determine their relevance to the target application.

Of course, the best code to benchmark is the target application. Next best is a benchmark that resembles the target application. Benchmarks that do not resemble the target application are worse than no benchmark at all. For example, if the target application will have many users and a database size of several GB, a simple benchmark with 1 or 2 users and a database that will easily fit in memory will not adequately put a database through its paces from the perspective of the target application.

If the evaluation includes running a custom benchmark, be sure to adequately model the following:

- The database design (to detect hot spots);
- The transaction mix (profiles, reads vs. writes);
- The number of concurrent users (to detect concurrency bottlenecks);
- The network topology (standalone vs. client-server);
- The database size and cache management / allocation; and

- The passage of time (all systems run fast immediately after the initial database load).

Scalability and Distribution

Scalability is one of the most desired attributes of information systems, and the term scalability has a host of varying meanings. Scalability means portability across platforms. Scalability means the ability to distribute data across a network for load balancing, performance optimization, and to ensure locality of reference. Scalability means the ability to sustain performance across the passage of time. Finally, scalability means the ability to maintain predictable performance as the number of users, database size, or both, increase.

General scalability questions include:

- Is the ODBMS portable across multiple platforms?
- Does the product take full advantage of both single processor systems and symmetric multi-processors?
- Can it maintain consistent, predictable performance as the application scales over time?

The following sections examine some of the deeper questions in the area of scalability.

Transparent Data Distribution

The notion of transparent distributed databases was first popularized by noted author Chris Date in the 1980s. During that time transparent distributed databases remained largely undelivered by the relational database vendors. While building transparent distributed databases has proven more difficult than initially expected, the demand for transparent distribution has increased substantially as both the demand for distributed network management applications and the demand for distributed, Internet-based data management applications increases.

Because ODBMSs were created during the client-server era, most ODBMSs were built with some (although varying) support for distributed databases. The key requirement of any distributed database is perhaps best summarized by Chris Date's "rule zero" of distributed databases: "a distributed database should appear to the user as a local database." That is, the distribution of data should be totally transparent to the user.

Transparency, in an ODBMS context, means that the way an application references an object is independent of the physical location of the object. That is, to be transparent, object reference must be independent of (i.e. orthogonal to) object location. More specifically, transparency means that applications must be able to navigate across objects without regard to their location. Transparency means that objects can be migrated across nodes to ensure locality of reference without any dependence on application code. Transparency also means that an object on one node must reference remote objects in exactly the same way that an object references local objects.

Furthermore, transparency means that object identifiers (OIDs) must be independent of object location, so that OIDs do not have to change when an object is migrated from one node to another. Preserving OIDs across migration is of

paramount importance for data integrity, because OIDs are the "glue" that holds an object database together. OIDs are in fact the mechanism used to manage the complex relationships among objects within the ODBMS. If an object's OID changes when it is relocated, the database will be corrupted until all objects network-wide that point to (i.e. are associated with) the relocated object are found and modified. This is clearly both expensive and difficult, if not impossible, in the distributed environment.

Many ODBMS products utilize OIDs that are based on physical object location. This solution is easier for the vendor to implement, but can limit an application's ability to relocate data - either within a database (e.g. for automatic space reclamation) or across databases. Few ODBMS products support logically based (i.e. independent of object location) OIDs. Logical OIDs are more difficult for the vendor to implement, but prove to be a far more scalable as a database grows either in terms of size or in terms of number of users.

Transparency refers not only to data objects, but also to transactions. To be transparent, an ODBMS must guarantee the atomicity (i.e. the all-or-nothingness) of transactions regardless of whether they update instances residing on one or many nodes. While the commit operation for local and distributed transactions differs (single vs. two-phase commit), the use of either technique should be masked from the user, and be chosen automatically by the ODBMS. Certain RDBMSs, for example, do not hide distributed transactions, and require the programmer to manually coordinate two-phase commits. A manual two-phase commit is known as a "programmatic" (as opposed to automatic) two-phase commit.

Another aspect of distribution in today's computing environment is the possibility that heterogeneous client and/or server platforms might be interacting with each other. At a minimum, it is highly likely that a UNIX server will need to interact with Windows NT or Windows 95 clients. It is also possible that heterogeneous UNIX platforms, potentially serving different departments and application needs, might need to share certain data. These applications might also require that this distributed sharing of data support heterogeneous languages (i.e. objects might need to be shared across C++ and Java applications). For long-term flexibility, it is therefore important to understand how transparently data can be shared across both platforms and languages. For C++ applications, the notion of compiler heterogeneity is important as well. Termed *Compiler Independent Heterogeneity*, or CIH, this feature allows C++ objects to be shared across heterogeneous platforms even though the compilers on those platforms might require that the data be laid out a bit differently from one another.

Transparent distribution provides many benefits. Maintenance is reduced because application source code need not be modified when objects are relocated. Performance is enhanced because objects may be migrated to ensure locality of reference and provide load balancing. Finally, data integrity is ensured because in transparent ODBMSs, OIDs are logical (i.e., not physical) and therefore are not dependent on physical location, and thus do not change when objects are migrated across systems.

Important questions related to distribution follow:

- Does the ODBMS support transparent distributed databases?
- Is the way a programmer references an object independent of (or orthogonal to) its physical location?
- Are object identifiers physical in nature (and thus change upon migration) or are OIDs logical and immutable?

- Will application source code require maintenance if objects are relocated to different nodes to improve performance?
- Does the ODBMS support transparent cross-node references?
- Does the product support two-phase commit?
- Is two-phase commit support automatic or programmatic?
- Does the product support required platform and language heterogeneity?

Replication

Another aspect of distribution is the notion of data replication. Replication is generally used for one of two purposes:

- Fault tolerance and recoverability; and
- Locality of reference and load balancing.

When all is functioning normally, replication for fault tolerance should provide transactionally consistent data across two servers. The data needs to be transactionally consistent to ensure that in the event of a failure a logically consistent view of the data is presented to the user by the non-failed server. Additionally, replication for fault tolerance should involve only two servers. Supporting more than this is a logistical nightmare, and becomes exponentially more difficult to resolve in the event of multiple failures and/or an unstable network environment. Finally, replication for fault tolerance should provide automatic resynchronization of data from the non-failed to the failed server once the failed server is available again, and the failed server should be brought automatically back into service once the resynchronization is complete.

Unlike replication for fault tolerance - which should be limited to two servers only - replication for locality of reference or load balancing should be capable of maintaining replicated data on many servers. These replication mechanisms should be not only able to replicate entire databases, but should also be able to replicate only a subset of the data stored in any particular database. This form of replication should support both event-driven, asynchronous replication and application-driven replication via programmatic APIs.

Key questions to ask include:

- Does the product support synchronous replication for purposes of fault tolerance?
- Does the fault tolerant solution provide automatic failover and resynchronization?
- Does the product support n-way asynchronous replication for purposes of load balancing or locality of reference?
- Is it possible to replicate only a subset of data in a database?

Thread Utilization

To scale up predictably in terms of number of users or size, an application must take full advantage of the resources available to it. This includes making

efficient use of real memory on the system and making full use of the CPUs that are present on the system. In many cases, the best way to do this is with threads.

From the perspective of the database server, it is important to know if the server is multi-threaded. Beyond this, however, it is important to know whether the latching mechanisms used to protect shared data resources are *fine-grained* or *course-grained*. Course-grained latches can allow a single process to pass through the database kernel a little more quickly than a fine-grained model, but they will typically only allow a single thread to be actively executing kernel code at any point in time. In contrast, a fine-grained model has a slight bit of additional overhead but will allow many kernel threads to be executing concurrently, thus taking advantage of multiple server CPUs if they are available. For example, a fine-grained model will allow queries and commits to occur simultaneously from different transactions; a course-grained model will not.

To build highly scalable database client applications (such as application servers), it is important that the client application be multi-threaded. To do this effectively, database libraries must be thread-safe. It is also useful if the database client can support multiple simultaneous transactions against the database. This allows application servers to utilize a transaction "pool" approach, which proves to be a highly scalable solution in application server and real-time message handling applications. For more information on threads and database sessions, see the white papers [Multi-Threaded Database Access](#) and [Multi-Session and Multi-Thread Applications](#) which can be found on the Versant website at <http://www.versant.com>.

Finally, be sure to understand the thread implementation model underlying both the client and server software. In the past, some vendors have supported their own version of threads rather than the native threads supplied by the operating system. These non-native thread implementations do not tend to scale well because the "threads" can't be scheduled by the operating system to run on multiple CPUs simultaneously. When native threads are used the application can take full advantage of all CPUs available on the system to accomplish its tasks.

- Is the database server inherently multi-threaded?
- Are client APIs fully thread-safe?
- Do client APIs support multiple, simultaneously active transactions within the scope of a single process?
- Does the product utilize native operating system threads, or its own thread implementation?

Productivity

The ultimate purpose of any database is to enable the rapid construction of powerful application programs. Too often, prospective users are encouraged to focus solely on database functionality, and assume that the process of actually building applications in each of the various ODBMSs is roughly the same. Some vendors encourage prospective users to focus solely on migration of existing code, rather than on development of new code as if no new code were being written to support new applications. For the most part, because users invest in databases to leverage application development resources, productivity of application development should be a key area of focus in ODBMS evaluations.

Programming Language Access

Generally speaking, most ODBMSs offer language interfaces that are superior to those offered by prior-generation RDBMSs. ODBMS language interfaces are superior for two reasons. First, ODBMSs generally eliminate the relational database "impedance mismatch" created by the use of two separate languages: one for database programming (SQL) and another for application programming (C++, Java). ODBMSs eliminate this mismatch by integrating tightly and transparently with the host programming language, so the host programming language operates as both an application and database programming language. The degree of transparency in the language integration, however, does vary among ODBMSs.

Second, ODBMSs do not force users to choose between openness and integration. To get tight integration between the database and application programming language in RDBMSs, programmers had to use highly proprietary, SQL-superset 4GLs. While these languages offered a much more integrated interface than the "EXEC SQL" interface between SQL and object-oriented programming languages, these 4GLs are proprietary to each vendor, and the use of these languages locks users into a single vendor for a long period of time. The alternative to these "proprietary, yet integrated" relational database 4GLs has been "open, but disintegrated" interfaces to standard programming languages (e.g., C or FORTRAN). In general, because ODBMSs are tightly integrated with open, standard languages, ODBMS users are not forced to make the difficult choice between productivity and openness; they can have both.

The major difference among ODBMSs in the area of programming language access relates to freedom of choice. In their efforts to tightly integrate the ODBMS with the programming language, many ODBMS vendors have either bound to only a single language (e.g., C++ only or Java only), or have a strong "language of choice" (e.g., a usable Java interface and an unusable C++ interface) in their products. Such forced choices are too restrictive given the state of today's market. While many feel that C++ will dominate large scientific, technical, engineering, and other advanced object applications, many also believe that Java will dominate the area of standardized MIS applications and of course the web. Choosing an ODBMS that limits freedom of choice and forces permanent selection of a language is strategically unacceptable and does not allow programmers to use the right programming language tool (be it C++, C, Java, or some future language) for the right job.

One last thing to consider is the nature of the implementation of the language interfaces. The basics of some products are easy to learn, but it proves hard to master the subtleties of the product needed to build large applications. Other products are a bit harder to get started with, but prove to be more intuitive and the interface is more consistent as the application scales. Also, the ODBMS language interfaces should not require separate hierarchies for transient and persistent instances. Transient and persistent instances should be able to use the same data structures and methods. Finally, to ensure that the language interfaces remain current with evolving standards, the product should not require use of proprietary compilers or virtual machines and should support the ODBMS-independent APIs as defined by the *Object Data Management Group* (ODMG). (See <http://www.odmg.org> for more information.)

The area of language interfaces is complex. Key initial questions to consider include:

- Does the product impose a programming language on application developers, or can application developers use the language that they view is best suited for the application at hand?

- Does the product provide transparent interfaces to multiple programming languages?
- Can transient and persistent instances of the same class be treated in exactly the same way?
- Does the product use proprietary compilers or virtual machines?
- Is the vendor committed to supporting the relevant standards?

Schema Management and Evolution

In any system, schema modifications inevitably occur as development moves forward on a project. To handle these schema changes, virtually all RDBMSs support the ALTER TABLE command, which allows administrators to add new columns to tables, or to change the types of existing columns. In object database terms, the notion of ALTER TABLE is called schema evolution, which means that the schema for a database can be evolved from one state to another as work moves forward on a project.

Schema evolution is typically implemented in one of three ways in various ODBMSs. The first is manual (i.e. not implemented) schema evolution the user must take the database offline, unload it, destroy the class, recreate the class, and then reload the old data. Second is "pseudo on-line" schema evolution—the user sends a single ODBMS command to perform schema evolution which, in the background, performs the exact steps outlined under manual schema evolution: an implicit dump/drop/recreate/reload of the class. Third is "lazy-update" schema evolution, where two versions of the schema object for a class are automatically maintained, and each instance is migrated from the old to the new schema on demand (i.e. only when referenced).

Lazy-update schema evolution has two key advantages. First, lazy updates do not require system downtime for schema evolution; they may be performed on-line. Second, lazy updates amortize the cost of migrating all the instances of a class across a broad time period.

It is not enough to simply support on-line schema evolution, however. Since the applications' class definitions are the database schema, the database must also provide mechanisms to facilitate migration of deployed applications to the new schema. The database should therefore be able to detect an application / database schema mismatch and throw an exception, or it should be able to transparently ignore attributes that are unrecognized by older versions of the application. (This is termed *loose schema mapping*.) Ideally, the database should support both mechanisms and allow the application developer to choose which is best for his or her environment.

In addition to schema evolution, programmers often require the ability to access the schema for a database at runtime. They may also require the power to create classes at runtime for added flexibility.

Key questions in the area of schema management and evolution include:

- Can schema information be dynamically modified with the database on-line?
- Does the product require all instances to be evolved to a new schema representation at once?

- Does it use a lazy-update algorithm to amortize the cost of schema modifications?
- Does the product support loose schema mapping?
- Does the system support run time or compile time-only schema access?
- Can the product support creation of classes at run time?

Tools Support

Vendor-Supplied Tools

One thing to consider when evaluating high-end database products is the availability of tools supplied by the vendor for use in database administration, performance tuning, etc. While there are a wide variety of tools that can be (and are) provided by the various vendors, of particular importance are backup and recovery utilities, and utilities related to performance analysis and tuning.

In the area of backup and recovery, a robust database product should obviously support automated backup capabilities. Perhaps less obvious is the fact that for large databases the product should support both full and incremental backups. Incremental backups are important simply because it takes far less time to archive the changes to a database each day than it does to archive the entire database. For example, it takes less time to write 100 MB of changed data to tape than it does to write 1 GB of data (900 MB of which is unchanged from day-to-day). Additionally, for a product to claim 24x7 availability, it must support the ability to perform backups with the database on-line. Since some ODBMS products support highly distributed applications, it is also important that the ODBMS be able to perform distributed backups across multiple databases in a way that ensures that data can be restored in a logically (i.e. transactionally) consistent manner. One final useful feature is support of a roll-forward recovery log. A roll-forward recovery log keeps track of all transactions since the last backup was performed. This allows a failed database to be recovered to the point of the last committed transaction by the restoration of the last backup followed by the application of the roll-forward log.

Tools for performing database administration should support not only a GUI-based environment, but also command line versions so that they may be incorporated into scripts and such. These tools will support database browsing, data export / import, schema management, instance management, index creation / deletion, lock management, granting database access privileges, etc.

Some relevant questions to consider are:

- Does the ODBMS vendor provide tools that support on-line backup for true 24x7 operation?
- Does the database support roll-forward recovery?
- Does the product provide parameters for tuning database behavior, cache and buffer sizes, etc.?
- Does the product provide utilities for analyzing database client and server performance?

Third-Party Tools

While it is important that a vendor provide tools to support the database, another important point to consider is the vendor's support of other "off-the-shelf" software tools. This support might be provided through standard interfaces or custom integration.

Of key importance is the ability to access the database and generate reports using off-the-shelf ODBC-based tools. This support should not simply permit access to the ODBMS, but should actually facilitate it. This means that the "relational" representation of the object schema (required by ODBC) should be automatically generated. Just as in the language APIs, queries should be able to query against inheritance hierarchies and access all objects relevant to a query. For reasons of scalability, the product should perform automatic cache management and utilize movable cursor windows when appropriate for large queries. Finally, limitations of a particular ODBMS architecture (i.e. the requirement that all objects reside in some sort of container to be queried) should not be exposed to the ODBC interface.

Integration with other tools, such as modeling tools, object request brokers, application servers, etc. is also useful to some applications and should be explored.

For example, some questions – and products – to consider might be:

- Can standard report writers (i.e. *Crystal Reports™*) access the ODBMS using industry standard ODBC mechanisms? How is the "relational view" of the object schema generated?
- Have integrations with object request brokers (e.g., Iona's *Orbix™*) been accomplished?
- Can design tools such as *Rational Rose™* be used to generate ODBMS-aware object models?
- Are integrations with high-end (and highly scalable) products such as BMC's *Patrol™*, BEA's *Weblogic™*, IBM's *WebSphere™*, and Tibco's *TIB/Rendezvous™* available?

The Right Vendor

In practice, the most important part of any software decision often has little to do with technology. Since database software requires training, support, ongoing maintenance, and enhancement, the most important part of any product evaluation is often an evaluation of the vendor itself. Choosing the right vendor is particularly important in the ODBMS marketplace, because a number of companies have been attracted to both the current ODBMS market, and to the anticipated growth in ODBMS usage.

Many years ago, in the early days of the relational database era, the situation was similar. There were many different vendors with different types of products: pure RDBMSs, SQL front-ends for network databases, extended network databases (e.g., IDMS/R), and others. Just as the extended network database approach did not meet the requirements of the relational market, the object-relational database will not meet the needs of the object database market. To meet relational needs, the apparently safe choice of IDMS/R surprisingly became more troublesome at the time than the seemingly risky choice of Oracle or one of the

other relational vendors. Similarly, for many applications today the safer choice for new, complex applications is often a pure object database rather than a relational or object-relational one.

Therefore, a critical issue that stands before any ODBMS evaluator is the strength of the ODBMS supplier. The following questions will help in identifying companies that are more stable, and thus that have a much greater likelihood of being the winners in the ODBMS market race.

- Does the firm have a clear vision and a realistic plan for fulfilling it?
- Is the vendor a major player in the industry, with substantial real world deployments of its ODBMS solution?
- Is the vendor a publicly traded company?

Conclusion

This guide has outlined the major areas that should be investigated when evaluating object database management systems. Specifically, it has focused on performance, scalability, distribution, application development productivity, and vendor strength. Each of these areas will prove critical to the success of the target applications in the long term, and thus each should be an important part of the ODBMS evaluation.

We hope that this guide has been valuable to you in formulating an ODBMS evaluation plan. If there are further questions about Versant as the evaluation proceeds, please feel free to contact Versant by phone or via the Internet. (Please refer to the *Introduction* for contact information.)

We believe that as you move forward with your evaluation that you will find out what hundreds of others have already discovered—that Versant is the leading ODBMS for building and deploying production object database applications.

Appendix: Object-Oriented Database Evaluation Checklist

The following checklist is intended to assist in the ODBMS evaluation process. Key features, functionality, and architectural aspects of various products are outlined. It should be noted that the “check boxes” associated with Versant capabilities are not marked. This was a deliberate decision on Versant’s part. While Versant’s technical representatives would be happy to assist in the completion of the matrix, it is our opinion that performing the research necessary to complete it is key to understanding the differences between the different products. In other words, “the journey is as important as the destination.”

Product Category

(select only one per column)

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Pure ODBMS (provides persistence, distribution, integrity, concurrency, and recovery)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Persistent Storage Manager	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Extended- or Object-Relational	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Object Wrapper (software layers placed on top of RDBMS)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Performance

(select all that apply)

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Balanced client-server model	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Server-centric model	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Client-centric model	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Queries occur on the server	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Server side data cache	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports group reads (i.e. reading a specified list or all objects in a container into the client at once)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports group writes	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Minimizes network traffic	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Client side object cache	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Automatic space reclamation and reuse	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Locking

(select all that apply)

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Object level locking	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page level locking	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Container level locking	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Prevents false waits	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Prevents false deadlocks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Automatic deadlock detection	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports optimistic locking for highly concurrent applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Scalability

(select all that apply)

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
On-line addition of volumes	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
On-line data reorganization	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
On-line data migration	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports very large databases (VLDBs)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports a highly distributed data model	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports large numbers of concurrent users without performance degradation	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Truly multi-threaded server engine with fine-grained latching of critical data structures	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Thread-safe client interface	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports MTMS (multi-thread, multi-session) with multiple concurrent database transactions within the same application process	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
On-line, dynamic schema evolution	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports loose schema mapping	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Performs automatic index maintenance on the server	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports server-based queries against collections	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports navigational queries	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports movable "cursor" windows to manage the	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
result sets of large queries				
Supports asynchronous data replication for locality of reference or load balancing	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Integrity

(select all that apply)

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
DBA tool for index creation / deletion	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Immutable object identifiers (i.e. OID's are never reused)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports n-way asynchronous database replication for load balancing and locality of reference	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports 2-way, synchronous fault tolerant database replication	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Supports event notification (i.e. database triggers)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Maintains class extents automatically to support ad hoc queries (i.e. find all persons with age > 40) - as required by the ODMG standard	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Full backups occur on-line	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Incremental backups occur on-line	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Roll-forward recovery	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Automatic two-phase commit	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Data Distribution

(select all that apply)

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Uses a single pointer type regardless of location (i.e. no near/far pointers)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Data distribution model is transparent to the application	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Object identifiers based on physical location of object (i.e. on a page, in a database, etc.)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Logical (i.e. location independent) object identifiers	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Object identifiers are never reused	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Object identifier never changes, even when an object is migrated to another database	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
True compiler independent heterogeneity	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
True platform independent heterogeneity (i.e. data written on any supported platform may be accessed by another supported platform without any explicit padding or extra code)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Standards

(select all that apply)

Feature / Functionality	Versant	ODBMS A	ODBMS B	ODBMS C
Provides command line and GUI-based DBA utilities	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Adheres to ODMG standard	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ODBC support	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Automatic generation of relational schema for ODBC access	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports ANSI standard C++	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports Java	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports C	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Integrates with leading application servers using J2EE standard interfaces	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports object sharing across languages	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Supports storage, import, export of XML	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Import / export of object graphs in ASCII format	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

About Versant

Versant Overview

As a leading provider of object-oriented middleware infrastructure, Versant offers the **Versant Developer Suite** and **Versant enJin**. Versant has demonstrated leadership in offering object-oriented solutions for some of the most demanding and complex applications in the world. Today, Versant solutions support more than 650 customers including AT&T, Alcatel, BNP/Paribas, British Airways, Chicago Stock Exchange, Department of Defense, Ericsson, ING Barings, MCI/Worldcom, Neustar, SIAC, Siemens, TRW, Telstra, among others. The Versant Developer Suite, an object database, helps large-scale enterprise-level customers build and manage highly distributed and complex C++ or Java applications. Its newest e-business product suite, Versant enJin, works with the leading application servers to accelerate Internet transactions.

Versant History, Innovating for Excellence

In 1988, Versant's visionaries began building solutions based on a highly scalable and distributed object-oriented architecture and a patented caching algorithm that proved to be prescient. Versant's initial flagship product, the Versant Object Database Management System (ODBMS), was viewed by the industry as the one true enterprise-scalable object database. Leading telecommunications, financial services, defense and transportation companies have all depended on Versant to solve some of the most complex data management applications in the world. Applications such as fraud detection, risk analysis, yield management and real-time data collection and analysis have benefited from Versant's unique object-oriented architecture.

VERSANT Corporation

Worldwide Headquarters

6539 Dumbarton Circle
Fremont, CA 94555 USA
Main: +1 510 789 1500
Fax: +1 510 789 1515

VERSANT Ltd.

European Headquarters

Unit 4.2 Intec
Business Park
Wade Road, Basingstoke
Hampshire, RG24 8NE
UK
Main: +44 (0) 1256 366500
Fax: +44 (0) 1256 366555

VERSANT GmbH

Germany

Arabellastrasse 4
D-81925 München,
Germany
Main: +49-89-920078-0
Fax: +49-89-920078-44

VERSANT S.A.R.L.

France

10 rue Troyon
F-92316 Sèvres Cedex,
France
Main: +33 1 450 767 00
Fax: +33 1 450 767 01

VERSANT Italia S.r.l.

Italy

Via C. Colombo, 163
00147 Roma, Italy
Main: +39 06 5185 0800
Fax: +39 06 5185 0855

VERSANT Israel Ltd.

Israel

Haouman St., 9
POB 52210
Jerusalem

1-800-VERSANT
www.versant.com

versant_template.doc

©2001 Versant Corporation.

All products are trademarks or registered trademarks of their respective companies in the United States and other countries.

The information contained in this document is a summary only.

For more information about Versant Corporation and its products and services, please contact Versant Worldwide or European Headquarters.