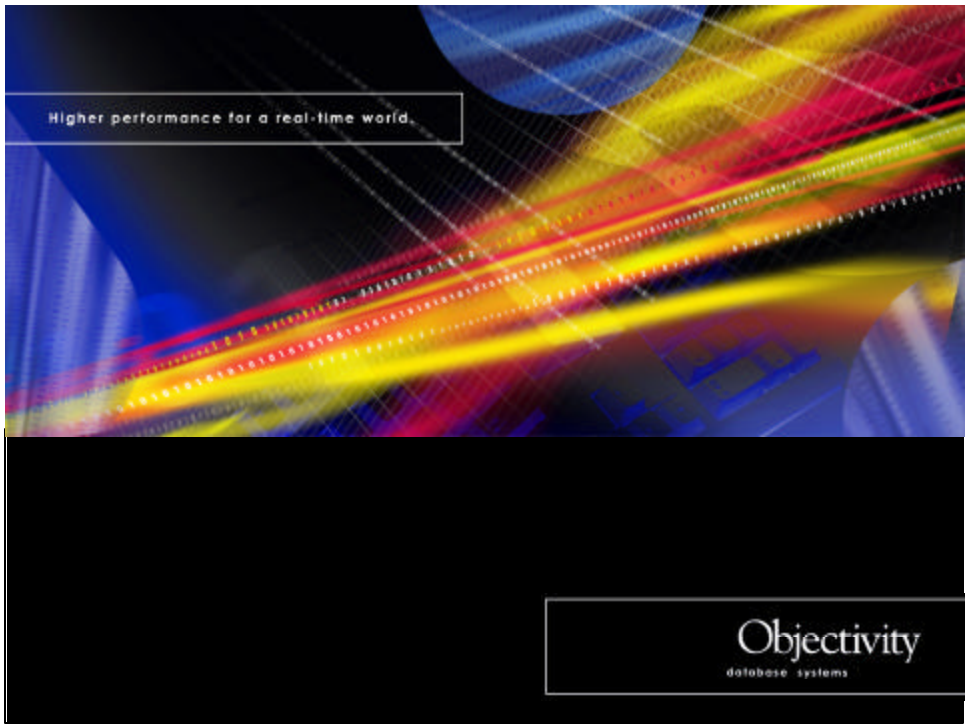


Hitting The Relational Wall

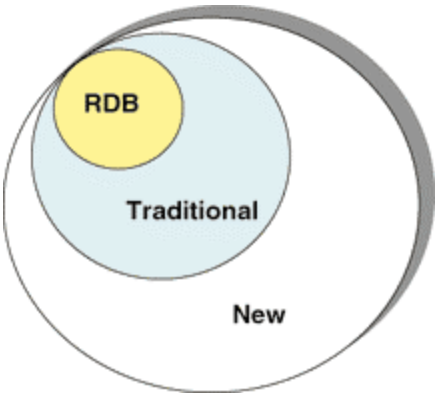
An Objectivity, Inc. White Paper

Introduction

Relational Database Management Systems (RDBMSs) have been very successful, but their success is limited to certain types of applications. As business users expand to newer types of applications, and grow older ones, their attempts to use RDBMS encounter the "Relational Wall," where RDBMS technology no longer provides the performance and functionality needed. This wall is encountered when extending information models to support relationships, new data types, extensible data types, and direct support of objects. Similarly, the wall appears when deploying in distributed environments with complex operations. Attempts to scale the wall with relational technology lead to an explosion of tables, many joins, poor performance, poor scalability, and loss of integrity. ODBMSs offer a path beyond the wall. This paper measures the wall, explains what model and architectural differences cause it, how to foresee it, and how to avoid it.

The Wall: Where an RDBMS breaks down

The generation of Relational Database Management Systems (RDBMSs) has brought many benefits to users, including *ad hoc* query, independence of data from logical application, and a large variety of front-end graphical user interface (GUI) tools. It has served a large number of business applications, and the industry has grown to over $4B annually, including tools. Why, then, should we look beyond RDBMSs? We should do so only when necessary to support new kinds of applications, new media data types, relationships, distribution, and scalability. In fact, most applications have never used RDBMSs, and many that have are now looking for a better solution, such as ODBMSs.



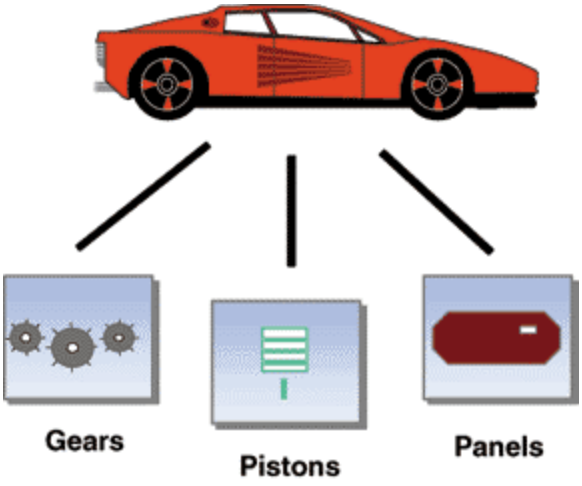
Most Applications are not served by RDBMS

Now three decades old, relational technology is showing signs of age. Although it works quite well when supporting simple, tabular data models, with short, simple operations, and centralized data storage and access, as in online transaction processing, or OLTP, anything beyond that encounters the "Relational Wall." This phenomenon is the result of applying the wrong tool (in this case, RDBMS). One might, for example, try to use a screwdriver to pound nails into a wall, and some nails might actually go in, but in the process many nails would likely break, along with screwdrivers and walls. Instead, using the right tool, a hammer makes the job much easier.

The relational wall is exactly the same; i.e., it is the result of using a table-based DBMS for non-tabular data, or a central server DBMS for distributed applications, or a centralized (and hence non-scalable) DBMS for large, enterprise scalable applications. The term "wall," though common in the literature, is a bit misleading, in that there is not a specific point where good turns to bad. Rather, as the complexity of the application increases, the mismatch between the RDBMS and the application grows. The same happens as data model complexity increases, as more relationships are managed, and as more distributed, scalable access is needed. The result is an explosion of tables, with slow joins between them, complexity difficult to manage, integrity loss, performance loss by orders of magnitude, all increasing. The user is left trying to squeeze his applications into tables, much like squeezing round pegs into square holes.

This wall is specifically a rapid rise execution time (slower performance), measured as complexity of the application increases. However, as we'll see below, it is also a rapid rise in the difficulty to change (loss of flexibility) the application and database. Also, a similar effect occurs in terms of extensibility; i.e., as complexity rises, the ability to extend applications, data models, and existing databases rapidly decreases. All these are aspects of "the relational wall."

Detailed measurement of the wall, and visual illustration is in the benchmark section, below. First, however, we'll review the concepts at a higher level, starting with a simple, but quite illustrative, example. Suppose you wish to store your car in the garage at the end of the day. In an Object DBMS (ODBMS), this is modeled with an object for the car, another for the garage, one operation "store," and you're finished. In a relational system, on the other hand, all data needs to be normalized; i.e., it must be flattened to primitives and sorted by type. In this example, it means that the car must be disassembled, down to its primitive elements, which must each be stored in separate tables. So, the screws go in one table, and the nuts in another, and the wheels, and side panels, and pistons, etc. In the morning, when you wish to drive to work, you must first reassemble the automobile, and then you can drive off.



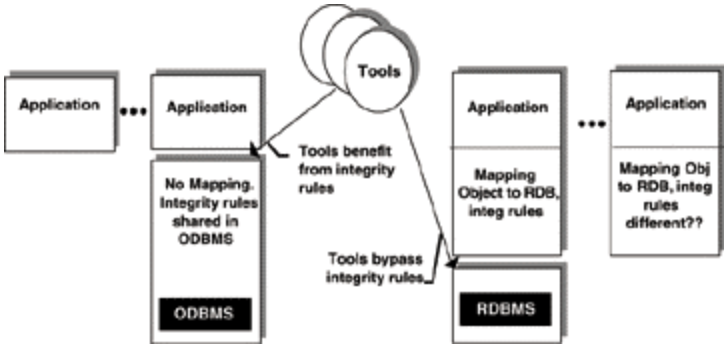
Storing a Car in an RDBMS Requires Disassembling

This may, perhaps, sound exaggerated. However, it is not very different from many applications we've seen. Whenever the application has complex (non-tabular, varying sized, interrelated) data structures, the application programmer must write code to translate from those data structures down to flat tables. This translation code has three problems:

Programming Time and Cost

Depending on the application, this translation code can easily grow to more than half the application. In one case (see American Meter Systems, below), a user moved from an RDBMS to an ODBMS (Objectivity/DB) and literally discarded one third of his application (and also achieved far better performance, taking the product from near failure to commercial success). Instead of programming, and maintaining, this translation layer, your programmers could instead be programming your applications, and saving a third to a half of their time and cost.

This effect is amplified during maintenance. Studies show that up to 80% of the lifetime cost of applications is in maintenance, during which time features are added, bugs are fixed, capabilities changed, all of which require changes to the data model underneath. When the application changes its data model, this tends to amplify, through the mapping layer, into many changes, both in that mapping layer and in the flat tabular data underneath. In the words of one user (see Iridium, below), this makes the mapping "brittle," and dramatically increases cost and potential errors.

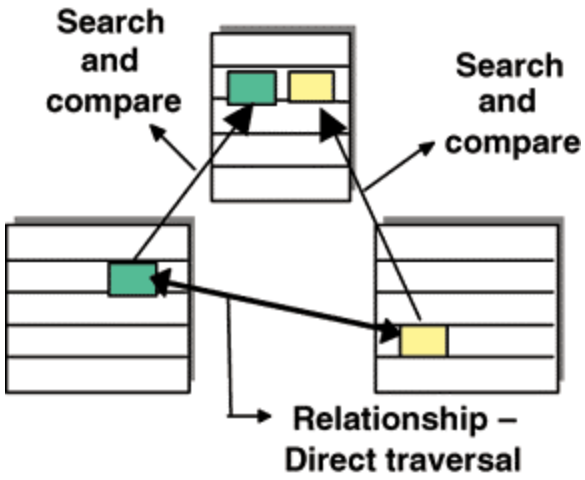


RDBMS Requires Mapping Layer

Integrity Loss

Since the mapping from flat RDBMS data to the application data is done within the application, by application programmers, there is the danger (even assuming all programmers program perfectly) that one programmer will do this mapping differently from other another programmer, resulting in a mismatch in their work and integrity violations. Such integrity issues can be subtle and difficult to locate, requiring detailed examination and comparison of many application programs. The problem is not only that the RDBMS works only at a primitive (tabular) data level, forcing the application to do the mapping, but also that this mapping remains part of the application. Instead, in an ODBMS, there is no mapping required. The DBMS understands and directly operates on high-level structures (such as the automobile, the engine, etc.) as well as low-level structures (such as piston, bolt). Queries may be performed at any level. The DBMS understands the structures (as defined in the object model, directly from the object model) and supports direct access at any level. The application programmers are then freed from the responsibility of doing the mapping. Instead, that complexity of structure and relationship is captured in the DBMS itself, and hence is shared among all users. Therefore, they all share the some structures and relationships and are guaranteed to be compatible at all levels, avoiding this integrity risk.

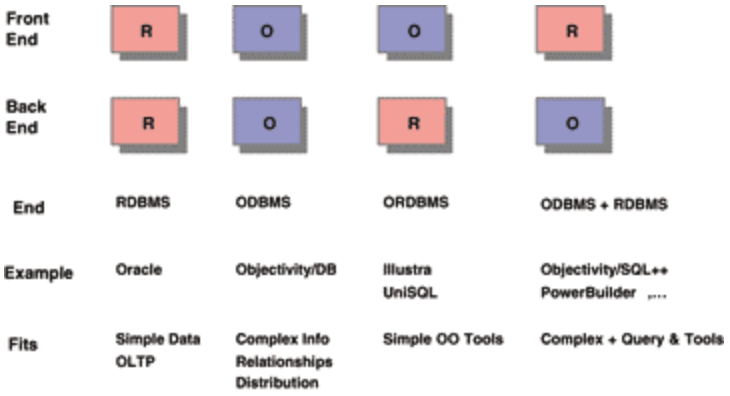
This risk of integrity loss extends all the way to end users via front-end GUI tools (PowerBuilder, Microsoft Access, Visual Basic, etc.). Such tools communicate to the RDBMS at the only level the RDBMS understands; i.e., at the primitive, flat level. Any application structures above that are lost. It is up to the end user to know those higher level structures and to reassemble them out of the primitives. Any error in such reassembly, of course, leads to integrity violation. An ODBMS avoids that risk while supporting the same tools. Since the ODBMS understands the higher level structures, as well as their operations and relationships, and makes these directly accessible to the tools, the result is that the end user can work directly at this higher level. In fact, by using grant and revoke (access permissions), certain users and groups may be limited to access only certain structures, and only certain attributes or operations on those structures. This guarantees that integrity will be maintained.



Slow Join vs. Fast Relationship

Performance

At runtime, disassembling and reassembling the automobile from its primitive components takes time. The more complex the structure, the more time. Any nested structures or varying-sized structures require multiple RDBMS tables, and joins between them. Similarly, any relationships require joins. The join, although it's quite flexible, is the weak point of relational technology because it's slow. It requires a search-and-compare operation, which takes time, and gets slower as the database tables get larger. (For more on this, see the next section.) Instead, all such complex structures, even dynamically varying-sized ones, are directly modeled as objects in the ODBMS, whose storage manager understands and is optimized to efficiently manage such structures. In addition, wherever the object model includes relationships, these are direct. There is no need to do the slow search-and-compare of the joins. Direct relationship traversal can literally be orders of magnitude faster (see examples below and benchmark). The more relationships you have, the more you'll benefit from an ODBMS.



ODB vs. RDB vs. ORDB

A high-level comparison of DBMS architectures is useful to clarify the differences between relational, object, and object-relational technology. Simplify DBMS architecture into a front end and a back end. Either of these may be based on either relational or object technology, yielding four possible combinations, as illustrated. If both front and back ends are relational, this gives a typical RDBMS. Similarly, if both are object, this gives a typical ODBMS.

An object front end layered over a relational back end is what's often called object-relational (ORDBMS). There are some new products in this area (UniSQL, Informix Illustra), and it's also the future direction of the major relational products (Oracle, Sybase, etc.). It's easy to understand why: it's very difficult to change the core engine technology of a DBMS, but very easy to add layers over it. These object layers can add functionality; e.g., they can offer additional data types and can ease integration with other object systems. However, the back end is still based on tables, so the front-end objects must still be flattened, disassembled into those tables, as in the car example above. If the application's data is simple and tabular, this is not a problem, but if it's more complex, the same performance problems remain.

For completeness, a relational front end may be layered over an object backend, as in the Objectivity/SQL++ product (see right-most column of figure). This is not intended to displace the all-relational (left-most column) approach. Rather, for those users who need the object backend, because of complex information, relationships, or distribution and scalability, this adds a familiar interface to support ad-hoc query and front-end GUI tools (based on ODBC). Besides extra functionality, this allows users who are familiar with SQL or such ODBC tools (PowerBuilder, Visual Basic, etc.) to immediately use ODBMS systems with no extra training and to share the same objects with C++, Java, and Smalltalk programmers.

This display of different architectural approaches leads some to wonder what is lost. RDBMSs offer many capabilities, including data-independence (allowing the automobile in the above example to be rebuilt into a different vehicle), views, SQL query, and a strong mathematical foundation that allows proving theorems on query optimization. The answer is that nothing is lost. These capabilities, and other RDBMS capabilities all continue to be available to the ODBMS user. The automobile parts may be searched by range of attribute values (e.g., all pistons with tolerances between x and y, made in Korea, etc.) exactly as in RDBMSs, using the same indexing techniques (b+ trees, hash tables, keys, etc.). Queries may be performed using the same standard SQL query language. All aspects of SQL may be used, including views, DDL (create table creates an object class, usable in C++ and Java), DML (insert, update, delete), triggers and stored procedures (actually for more general, via object methods), and security (grant, revoke) not only at the level of every attribute but also every method, allowing certain users and groups to be restricted to accessing only certain objects and accessing them only through certain methods, enforcing encapsulation and rules, guaranteeing integrity. The automobile's parts may be reassembled in arbitrary fashion, just as in the RDBMS case. The difference is that, once assembled, the ODBMS can capture the knowledge of that assembly, so later use of it will not require reassembling (searching, joining, etc.).

The mathematical foundation of relational theory has been quite useful academically for proving theorems, but in practice it is only useful for applications with simple, tabular data structures and simple operations. For these applications, the SQL query mechanism applies directly, and the optimizers directly apply to the user's problem. However, in most cases the application's data structures are not simple tables and its operations are not limited to simple, disconnected SELECT, PROJECT, and JOIN. In these cases, the application programmer must first translate (flatten) his problem into these primitives. The complexity of the problem is, in fact, managed completely by the application programmer, while the RDBMS, confident in its theorems, works only on the resulting primitive decomposition. The mathematical world of the simple RDBMS models do not apply to the original application problems. The application programmer is left to whatever devices he can manage on his own. In short, the complexity, whatever it is, is inherent to the application. With an RDBMS, all that complexity is left to the application programmer. With an ODBMS, the DBMS itself helps manage the complexity.

In the following two sections, we'll examine the underlying differences between RDBMSs and ODBMSs in order to understand the source of their different behaviors. First, we'll examine the information model differences, including:

- Relationships
- Varying-sized data structures
- User-extensible structures
- Flexibility and efficiency of structures
- Encapsulation of operations for integrity, quality, and cost reduction
- Complex queries

Here we'll see how the relational wall can appear not only in performance, but also as a barrier to modeling complexity, extensibility, and flexibility. Then, we'll examine architectural differences typical of RDBMSs and ODBMSs, including:

- Caching
- Clustering • Distribution
- Scaling, Bottlenecks
- Dynamically tunable locking granularity
- Legacy integration
- Safety and integrity

These show how the relational wall limits the user's ability to adjust the architecture of his application to fit his computing environment, to take advantage of the new, powerful microcomputers, to tune his system to fit his needs, and to maintain necessary integrity.

We'll follow with a description of a benchmark comparing a leading RDBMS with a leading ODBMS (Objectivity/DB), showing the relational wall quite explicitly. Then, we'll review a few successful production users of ODBMSs, showing in each case why they benefit from the ODBMS. This includes applications in telecommunications, financial services, internet/intranet/web information systems, multimedia systems, document management, enterprise-wide integration, manufacturing, and process control.

Information

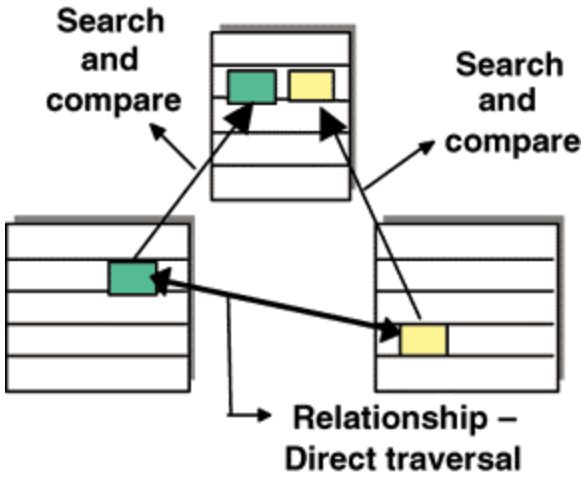
Model Issues
As the names imply, ODBMSs and RDBMSs vary in the models they allow the application programmer to use to represent his information and processing of it. RDBMS support:

- Tables (as data structures)
- Select, Project, Join (as operations)

All application information and processing must be translated to these. ODBMSs support:

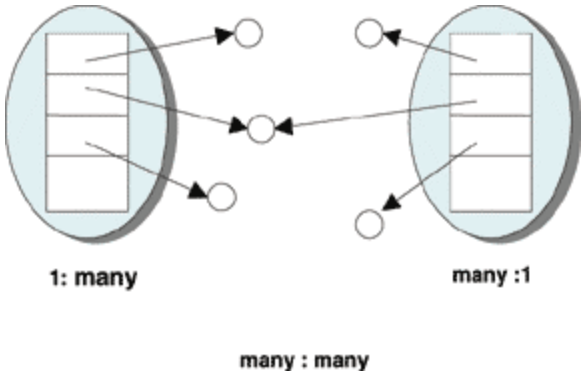
- Any user-defined data structures
- Any user-defined operations
- Any user-defined relationships

These result in several differences.



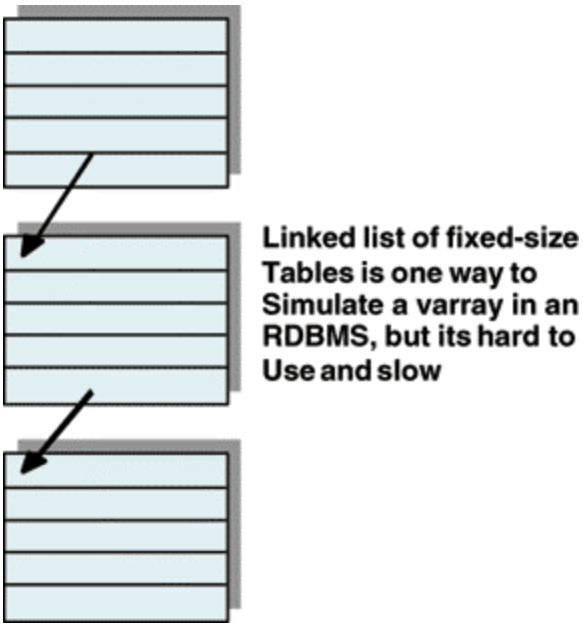
Slow Joi n vs. Fast Relationship

First, consider relationships. Many applications must model relationships of information; e.g., among customers / products / account-representatives, or students / faculty / courses, or clients and portfolios of securities, or telecommunications networks / switches /components / users / data / voice, or manufacturing processes and products, or documents/cross-references/figures, or web sites / html objects / hyperlinks. In a relational system, to represent a relationship between two pieces information (tuples, rows), the user must first create a secondary data structure (foreign key), and store the same value of that foreign key in each structure. Then, at runtime, to determine which item is connected to which, the RDBMS must search through all items in the table, comparing foreign keys, until it discovers two that match. This search-and-compare, called join, is slow, and gets slower as tables grow in size. Although the join is quite flexible, it is slow and is the weak point of relational systems.



Many:Many ODBMS Relationship

Instead, in an ODBMS, to create a relationship, the user simply declares it. The ODBMS then automatically generates the relationship and all that it requires, including operations to dynamically add and remove instances of many-to-many relationships. Referential integrity, such a difficult proposition in RDBMSs, usually requiring users to write their own stored procedures, falls out transparently and automatically. Importantly, the traversal of the relationship from one object to the other is direct, without any need for join or search-and-compare. This can literally be orders of magnitude faster, and, unlike the RDBMS approach, scales with size. The more relationships, the more benefit is gained from an ODBMS.

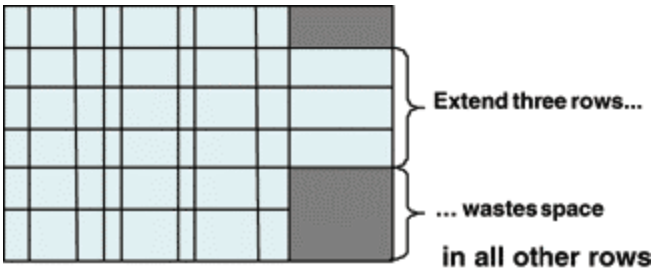


RDBMS Can't Do Varying Size

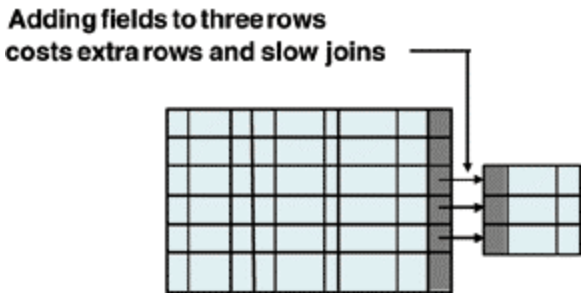
Another typical problem area in RDBMS is varying-sized structures, since as time-series data or history data. Since the RDBMS supports only fixed-size tables, extra structures need to be added, resulting in extra complexity and lower performance. In order to represent such varying sized structures the user must break them into some combination of fixed size structures, and manually manage the links between them (see figure for an example of this). This requires extra work, creates extra opportunity for error or consistency loss, and makes access to these structures much slower.

ODBMS Varying-Sized Structure

Instead, in an ODBMS, there is a primitive to support varying-sized structures. This provides an easy, direct interface for the user, who no longer needs to break such structures down. Also, it is understood all the way to the storage manager level, for efficient access, allocation, recovery, concurrency, etc.

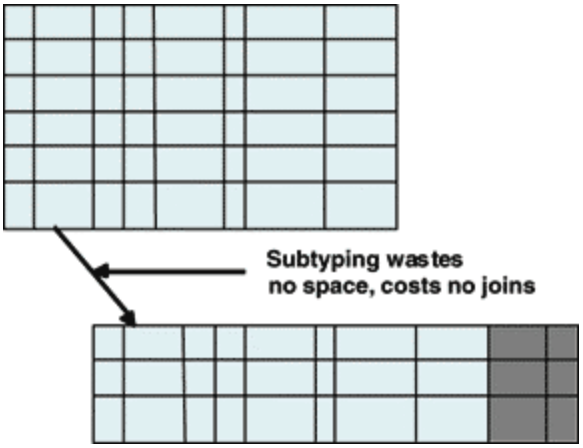


Adding Fields to Some Rows Wastes Space in All O ther Rows



Another RDBMS Approach Also Wastes Space Plus Costs Slow Joins

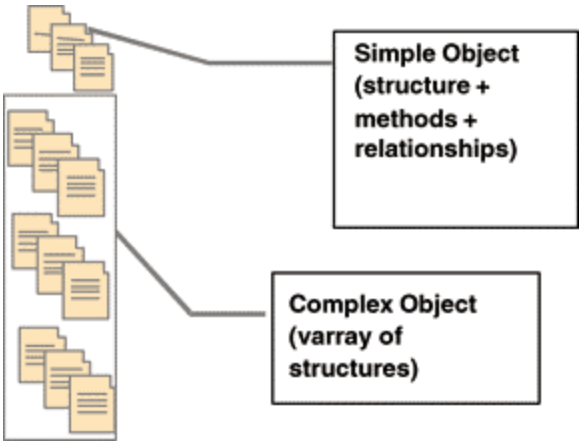
A similar situation arises when a user wishes to alter a few rows in a table. Suppose, for example, that two new fields are to be added to 3 rows (see figure). The RDBMS user is left with two choices. He can enlarge all rows, wasting space (and hence time, also, because of increased disk I/O). Or, he can create a separate structure for those new columns, add foreign keys to all rows to indicate which have these extra columns. This still adds some (though less) overhead to all rows of the table, but also now adds a new table, and slow joins between the two.



ODBMS Subtyping Adds 2 Fields to 3 Rows with No Waste, No Joins

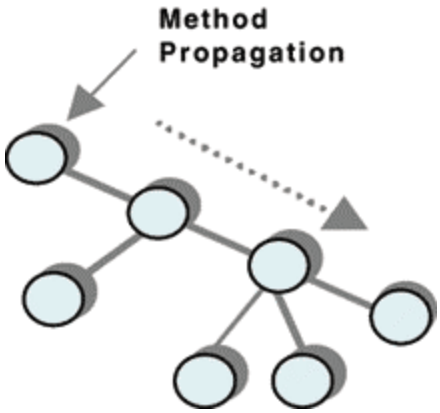
The ODBMS user has no such problem. Instead, the ODBMS manages the changes directly and efficiently. The user simply declares the changes as a subtype (certain instances of the original type are different), and the ODBMS manages the storage directly from that, allocating extra space just for those instances that need it, with no extra foreign key or join overhead.

Flexibility can be critical to many applications, in order to vary structures for one use or another, or vary them over time as the system is extended. The RDBMS structures are static, fixed, with hard rectangular boundaries, providing little flexibility. Changes to structures or additions typically are quite difficult and require major changes. With an ODBMS, the user may freely define any data structure, any shape. Moreover, at any time, such structures may be modified into any other shape, including automatic migration of preexisting instances of the old shape. Any new structures can always be added.



Simple and Complex Objects

Such structures in the ODBMS may be very simple, with just a few fields, or may be very complex. Using the varray (varying-sized array) primitive, an object can have dynamically changing shape. In fact, by combining multiple such varrays into a single object, very complex objects can be created.



Composite Objects

ODBMS structures can also include composite objects, or objects that are composed of multiple other (component) objects. Any network of objects, connected together by relationships, can be treated as a single (composite) object. Not only may it be addressed structurally as a unit, but operations may propagate along the relationships to all components of the composite. In this way, object may be created out of objects, and so on, to any number of levels of depth. Since the relationships are typed, a single c omponent object may also participate in a different composite, allowing any number of dimensions of breadth. Thus, arbitrary complexity may be modeled. More importantly, additional capability may always be added. Users may always add new composites, that perhaps thread through some of the old composites' objects as well as adding some new ones, without limit. There is no complexity barrier or relational wall in complexity.

In an attempt to allow storage of complex structures, RDBMSs have begun to add BLOBs , or Binary Large Objects. Unfortunately, to the DBMS, the BLOB is a black box; i.e., the DBMS does not know or understand what is inside the BLOB, as the ODBMSs do for objects. In fact, storing information in a BLOB is really no different than storing them in a flat file, and linking that to the DBMS. Flat files can be useful, but with a BLOB the DBMS cannot support any of its functionality internally to the BLOB, including concurrencies, recovering, versioning, etc. It's all left to the application.

Similarly, in an attempt to support user-defined operations, some RDBMSs now support stored procedures. On certain DBMS events (e.g., update, delete), such procedures will be invoked and executed on the server. This is much like executing methods in an ODBMS, except that ODBMS methods may apply to any events (not just certain ones, as in RDBMS); may execute anywhere (not just on server); and may be associated with any structures or abstractions. Also, many RDBMS features may not work with them, because they're not tables. Certainly, any robustness, limited scalability, etc., would not apply since they're newly added outside the RDBMS table-based engine.

Very similar to stored procedures are Data Blades or Data Cartridges. These are pre-built procedures to go wth BLOBs. They add the ability to do something useful with the BLOBs. In this way, they're similar to class libraries in an ODBMS, except that the ODBMS class libraries may be written by users (data blades typically require writing code that inserts into the RDBMS engine); and may have any associated structures and operations.

All these efforts (BLOBs, stored procedures, and data blades) stem from a recognition by RDBMS vendors that they lack what users need; viz., arbitrary structure, arbitrary operations, and arbitrary abstractions. Unfortunately, they take only a small step in this direction, providing only limited structures, predefined operations and classes. If the RDBMS were modified enough to generalize BLOBs to any object structure, and stored procedures to any object method, and data blades to any object class, it would require rebuilding the core DBMS engine to support all these, instead of just tables, and the result would be an ODBMS.

Data often changes over time, and tracking those changes can be an important role of the DBMS. The RDBMS user must create secondary structures and manually copy data, label it, and track it. The ODBMS user may simply turn on versioning, and the system will automatically keep history. This is possible for two reasons. First, the ODBMS understands the application-level objects, so it can version at the level of those objects, as desired. In the RDBMS, the data for an application entity is scattered through different tables and rows so there is no natural unit for versioning. Second, the ODBMS includes the concept of identity, which allows it to track the state of an object even as its values change. The history of the state of an object, of the value of all its attributes, is linear versioning. Branching versioning, in addition, allows users to create alternative states of objects, with the ODBMS automatically tracking the genealogy.

In addition to supporting arbitrary structures and relationships, objects also support operations. While the RDBMS is limited to predefined operations (select, project, join), the ODBMS allows the user to define any operations he wishes at any and all levels of objects. This allows users to work at various levels. Some can work at the primitive level, as in an RDBMS, building the foundation objects, which others can then use without having to worry about how they're implemented internally. Progressively higher levels address different users, all the way to end users, who might deal with objects such as manufacturing processes, satellites, customers, products, and operations such as measure reject rate, adjust satellite orbit, profile customer's buying history, generate bill of materials, etc. As new primitives are added, the high level users immediately can access them without changing their interface.

Encapsulating objects with such operations provides several benefits, including integrity, ability to manage complexity, dramatically lower maintenance cost, and higher quality. Integrity results from embedding desired rules into these operations at each level, so that all users automatically benefit from them. As we saw in the section above, RDBMSs allow (indeed, force) users to work at the primitive level, so they might violate or break higher level structures, or change primitive values without making corresponding changes to other, related primitives. The encapsulated operations prevent such integrity violations. The same remains true with GUI tools, because they, too, get the benefit of the higher-level objects with built-in operations.

Operations allow managing greater complexity by burying it inside lower structures, and presenting unified, more abstract, user-level interfaces. Encapsulation can dramatically reduce maintenance cost by hiding the internals of an object from the rest of the system. Typically, large systems become complexity-limited because any changes to one part of the system affects all others. This spaghetti-code tangle grows until it becomes impossible, in practice, to make more changes without breaking something else somewhere. Instead, with an ODBMS, changing the internals of the object is guaranteed not to affect any users of the object, because they can only access the object through its well-defined external interface, its operations. In addition, new sub-types of objects can be added, extending functionality, without affecting higher level objects, applications, or users. For example, a graphical drawing program might include a high-level function (and user interface menu or button) to redraw all objects. It would implement this by invoking the draw operation of all objects. Different objects might have very different implementations (e.g., circles vs. rectangles), but the high-level routine doesn't know or care, it just invokes draw. If a new sub-type of drawing object is added (e.g., trapezoid), the high-level redraw function simply continues to work, and the user gets the benefit of the new objects. Similarly, if a new, improved algorithm is implemented for one of the drawing object subtypes (e.g., a better circle drawing routine for certain display types), the high level redraw code continues to work as-is, unchanged.

Finally, objects encapsulated with their operations can improve quality. The old approach to structuring software, as embodied in an RDBMS, creates a set of shared data structures (the RDBMS) which are operated upon by multiple algorithms (code, programs). If one algorithm desires a change to some data structure, all other algorithms must be examined and changed for the new structure. Similarly, if an algorithm from last year's project turns out to be the one you wish to use this year, you must still copy it over and go through and edit it for the new project's data structures. Code reuse requires changing it, creating a new piece of software with new bugs that must be eradicated, and creating a new

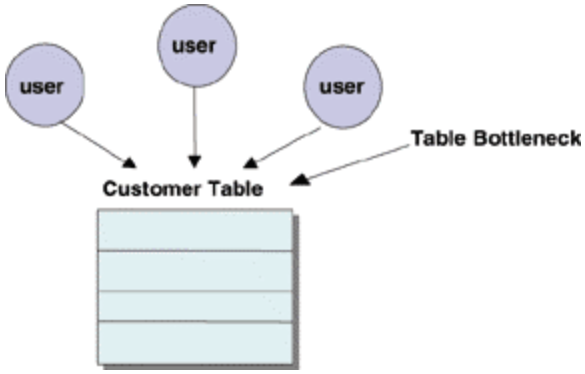
maintenance problem. Instead of building on past work, software developers are continually restarting from scratch. With ODBMS objects, however, the code and the data it uses are combined, so changes can be made to them together, without breaking or affecting the other objects. If an object from last year's project does what we need for this year's project, we can simply use it as-is, without copying and changing. It's already tested and debugged and working, so we gain higher quality by building on the work of the past. Even if the old object isn't exactly what we want, we can define a new subtype for that object (inheritance), specifying only the difference (delta) between the new and old. This gives the flexibility to allow reuse in practice, reduces the required new programming (and debugging and potential quality problems) to the much smaller delta, and reduces the maintenance by keeping the same, main object for both the new and the old system.

Last, we'll address complexity. Some might look at all this information modeling capability, including versioned objects, composites, subtypes, etc., and ask what price is paid for this increased complexity. In general, no price is paid. In fact, there is a reduction in complexity as seen by the user. The question is really backwards. The complexity lies not in the DBMS, but in the application, in the problem it's trying to solve. If it is simple, then the data structures and operations in the ODBMS will be simple. If the application's problem is inherently complex, the more sophisticated modeling capabilities of the ODBMS allow that complexity to be captured by the ODBMS itself, so it can help. Instead of forcing the application programmer himself to break that complexity down to primitives, the ODBMS allows him to use higher level modeling structures to directly capture the complexity, and then manages it for the user based on the structure and operations the user has defined. In short, the complexity is in the application, not the DBMS.

The same is true for queries. The RDBMS query system might look simpler, closed, easier to predict, and it itself is. However, the application's desired query is not. It must be translated from the natural, high-level application structures and operations down to the primitive RDBMS structures and operations, and the complexity is all in that translation. Instead, in an ODBMS, the high level structures and operations can be used directly in the query, and the ODBMS query engine executes the query itself, with no need for the user to translate to lower-level primitives. True, the resulting ODBMS query and query system is more complex, but that is because of the nature of the query. It was still true for the RDBMS-based complex application query. The only difference is who handles the complexity, the user or the DBMS.

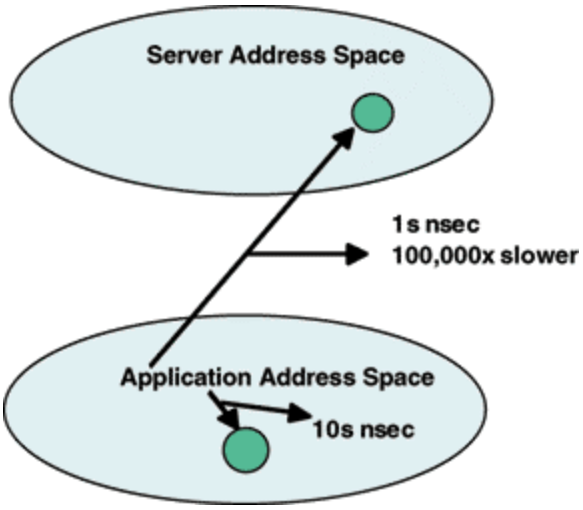
Architecture Issues

Next we'll turn to architectural differences between ODBMS and RDBMS. Technically, the terms "ODBMS" and "RDBMS" say nothing about architecture, but refer only to the information model (based on tables only, or supporting arbitrary structures and operations). However, in practice, the ODBMS and RDBMS products differ significantly in architecture, and these differences have a major impact on users. We'll see differences in clustering, caching, distribution, scalability, safety, integrity, and dynamic flexibility and tuning.



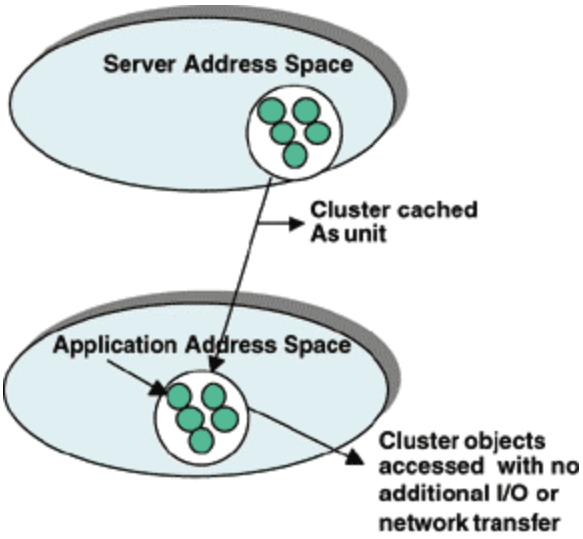
Customer Table Bottleneck

Since the RDBMS model is based entirely on tables, virtually all implementations are based on tables, too. For example, if there's a customer table, then all applications that need to access customers will collide, or bottleneck, on that one table. This bottleneck forces all such applications to wait for each other. The more users, the more applications, the longer the wait, defeating scalability. Instead, in an ODBMS, the customer objects may be separated and stored as desired; e.g., the customers in the USA may be placed in one database, on one server, while those in Asia may be placed in another database, on another server. Though access remains unchanged (see the discussion of distribution architecture, below), now those applications that access only USA customers (or only Asian customers), will not conflict at all, allowing them to run in parallel.



Caching Is Up to 10^5x Faster

In addition to the above clustering flexibility, ODBMSs add the ability to cache. In an RDBMS, all operations are executed on the server, requiring interprocess (IPC, and usually network) messages to invoke such operations. The time for such IPC is measured in milliseconds. With an ODBMS, the objects can be brought from the server to wherever the application is executing, or cached directly in the application's address space. Operations within address spaces occur at native hardware speeds; with 100 MIPS, that means they're measured in hundredths of microseconds, fully 100,000x faster than IPCs, a gain in five orders of magnitude. Such overwhelming performance advantages are a big part of why ODBMSs can be much faster. Although the first operation, to cache the object, requires IPC and runs in milliseconds, any later operations on that object occur directly, 10^5x faster. This is typical of other operations, too, in the ODBMS, where the overhead is incurred in the first use, but then no extra overhead at all is suffered in later operations.



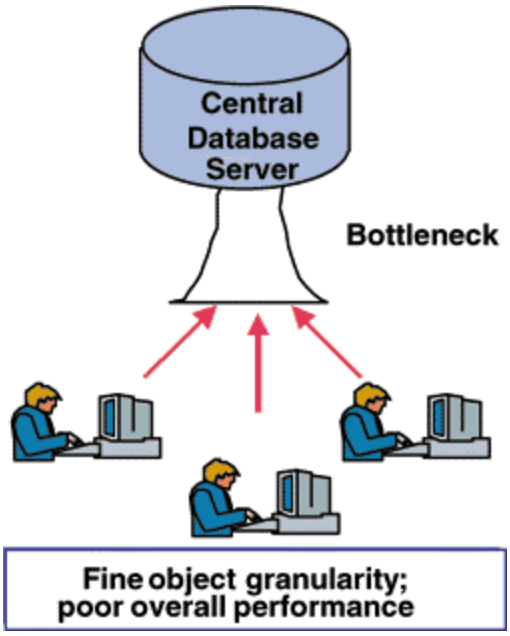
Combined Clustering and Caching

Combining caching and clus tering produces even greater benefits. The clustering discussed above can be applied across different types. While the RDBMS stores all the tuples of a given type together (in tables), the ODBMS adds the flexibility to cluster different types together. For example, together with a customer object, we might cluster the objects representing the products, services, or securities that customer has purchased. Then, when the customer object is accessed, the related objects come along into the cache for free. Later access to them occurs immediately at native hardware speeds.

In general, performance in a DBMS is limited by network transfers and disk I/Os, both occurring in the range of milliseconds. All other operations, measured in machine cycles, 10^5x faster, are a distant second-order effect. Caching and clustering, together, allow avoiding such slow network and disk operations, and can thereby dramatically, by orders of magnitude, increase performance.

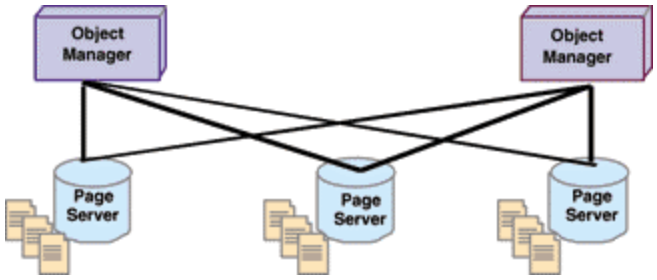
The basic architecture of ODBMSs and RDBMSs differ, too, which brings us to distribution. RDBMSs are built around central servers, with all data, buffering, indexing, joining, projecting, selecting occurring on the server. The user simply sends in a request and gets back an answer. This architecture is, in fact, the same architecture as the mainframes of the 1960s, where the user, on a 3270 IBM terminal, sends requests to the mainframe, which processes all requests, and then returns the answers. DBMS technology grew up on the mainframes and almost all DBMS architects have continued to follow that same, basic architecture.

Meanwhile, the world of computing has changed. Today, with powerful workstations, PCs, and high-speed networks, a typical corporate environment has far more computing power (MIPs) spread around on the desktops than it has in the central glass house. The mainframe architecture is unable to take advantage of all these computing resources. Some ODBMSs, however, have been built with a new generation of distributed architecture that can take advantage of all computing resources, by moving objects to whatever machines desired and executing there.



RDBMS Central Server Bottleneck

To examine this in more detail, consider the central server architecture of virtually all RDBMSs. All users send requests into the server queue, and all requests must first be serialized through this queue. Once through the queue, the server may well be able to spawn off multiple threads, which can be quite useful, but first, to achieve serialization and avoid conflicts, all requests must go through the server queue. This means that, as new users are added, they wait longer and longer at that central queue. It becomes a bottleneck (different from the table bottleneck described above) that limit multiuser scalability. When the RDBMS starts to get too slow all the user can do is buy a bigger server and hope that it can move requests through that queue faster. This is why RDBMS companies often encourage high-end users to buy very high-performance (and very expensive) parallel computers. Their mainframe architecture pushes them back towards mainframe machines.



Distributed Servers Architecture

Contrast this to a distributed ODBMS architecture (see figure). There are two main differences here to understand. First, the DBMS functionality is split between the client and server, allowing computing resources to be used, and, as we'll see, allowing scalability. Second, the DBMS automatically establishes direct, independent, parallel communication paths between each client and each server, allowing clients to be added without slowing down others, and servers to be added to incrementally increase performance without limit.

To see the client/server DBMS functionality split, consider first the server. Here, a page server moves a page at a time from the server to the client. This is advantageous because of the nature of network transport overhead; viz., the cost to send one byte differs little from the cost to send 1000 bytes. So, if there is any significant chance that any of the other 999 bytes will be used, it is a net gain to send the larger amount. A detailed measurement shows that any clustering overlap of greater than 10% yields better performance for page transport. Similarly, a measurement based on simulation of different architec tures, by David DeWitt, Univ. of Wisconsin, Madison, shows the same advantage to page transport. In this picture, the page size may be set (tuned) by the user, different for each database, to achieve the optimum performance. Such pages often contain 100s of objects. Therefore, compared to traditional tuple-servers or object-servers that send one tuple or object at a time, this page server can be up to 100s of times faster.

On the client side is the Object Manager. This allows application access (and method invocation) to objects within their own address space. As described above, such operations are in hundredths of microseconds, fully 10^5x faster than the millisecond range of cross-process operations, such as those that must communicate to the server. This Object Manager component of the ODBMS also supports object-level granularity, supporting a means to add functionality, guarantee integrity, and control access at the level of every object. (A detailed description of this mechanism appears below, under the Integrity discussion.) For example, if a page with 1000 objects is moved to the client, but the client uses only 10 of those objects, then overhead is incurred for only those 10, a savings of up to 100x in this example. Such overhead includes swizzling and heterogeneity translations.

Also, because access is controlled at the object level, various operations may apply to one object and not to another; e.g., one may version one object, but not another.

The client-to-server communication is also significantly different in this architecture. Instead of all users communicating via a single, shared central-server bottleneck, independent parallel communication paths are provided between each client and each server. (These are set up transparently, on-demand, and dynamically managed, cached, released, and reused, just as any other dynamic resource such as memory or open files.) The effect is that a server may simultaneously move pages to multiple clients at exactly the same time. If a new client is added, he will get his own, independent link, and the server will serve him and the previous clients entirely in parallel. With the central server bottleneck, adding clients necessarily forces other clients to wait. Here, however, adding new clients causes no extra waiting in other clients. The only time clients wait for each other is when they're attempting to access the same objects, in which case the usual locking protocols serialize them (see more, below, on concurrencies). Otherwise, the result is scalability in clients; i.e., clients may be added freely without slowing down others.

This scalability continues until (one of) the server(s) begins to approach its performance limits (in cycles per second, or I/O, etc.). Even, in this case, however, scalability can be maintained. The distributed ODBMS approach provides a single logical view of objects, independent of their location, database, server, etc. This logical view is transparently mapped to physical views. In particular, for full support of distribution, six criteria must be supported:

1. Single Logical View of all Objects

2. All operations work transparently across that view (e.g., atomic transactions, propagating methods, many-to-many relationships, etc.)

3. Dynamic support for the single logical view (e.g., as objects are moved, applications continue to run)

4. Heterogeneous support for the single logical view (e.g., objects and databases may be located on different hardware, networks, operating systems, and access via different compilers, interpreters, languages, and tools)

5. Fault Tolerance (e.g., when faults occur, such as network or node failures, the remaining parts of the system continue to function and provide services normally, within the physical limitations of unavailable resources; this requires servers to automatically take over responsibility for missing servers, including recovery, locking, catalogues, and schema)

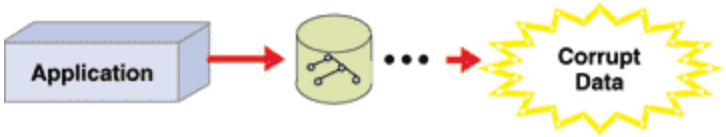
6. Replication and Continuous Availability (which adds to fault tolerance by supporting replication of user objects, thereby allow ing continued availability of that information even during fault conditions)

Note that this separation of logical view from physical requires that the DBMS make no assumptions about what resides or executes on the client or the server or any intermediate tier, but rather support object access and execution anywhere. For example, if the DBMS asserts that certain operations always take place on "the server," it is certainly not distributed, because the addition of a second server makes such a statement meaningless.

There are two benefits of this distribution that immediately apply. First, consider scalability. In traditional server-centered DBMSs (including RDBMSs and even some ODBMSs), when the server capacity limit is reached, all the user can do is replace the server with a more powerful one. This holdover from the mainframe architecture tends to push users back into mainframe-like, massive, expensive, and proprietary servers. Instead, in a distributed architecture, when a server capacity is reached, the user may simply add another server, and move some of the objects from the first, overloaded server to the second. It will dynamically set up its own independent, parallel communication paths. Due to the transparent single-logical view, clients and users see no difference at all. The entire system continues to run, and continues to scale. Further, this can be achieved with commodity servers (e.g., inexpensive NTs, rather than massively parallel super-computers). The result is scalability in servers as well as clients.

A second benefit of this distribution is flexibility. In the traditional server approach, each user (or application) connects to the DBMS by first attaching to a specific server, then request objects or tuples or operations. If, however, it is later desired to move some objects from one server to another, all such applications (or users) break, because the objects are no longer on the server they expected. Instead, the distributed single logical view approach insulates the users and applications from any such changes or redistribution of objects. All applications continue to run normally while online changes are made. This provides a new level of flexibility.

The above-described caching in the client's address space raises an issue of integrity maintenance. In the traditional server architecture, all DBMS operations take place in a different process (and often a different physical machine). The operating system inter-process protection mechanism acts as a firewall, preventing any application bugs from inadvertently accessing and damaging the DBMS structures and buffers. With caching, along with the 10^5x performance advantage comes the potential that an application pointer bug could now directly access such DBMS structures, damage objects, and corrupt the database. In fact, with stored procedures, found in most RDBMSs, the same danger exists because those user-written procedures run directly in the DBMS process, and thus can freely destroy the DBMS.



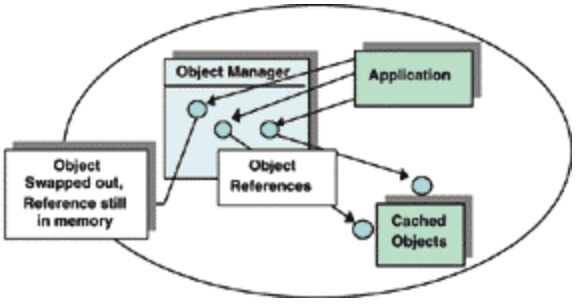
Cache References (or Stored Procedures) can Corrupt Database

Most other systems (whether stored procedures on the server, or ODBMS methods on the server or client) provide the user code to directly access object internals via raw pointers. Eventually, any such pointer will become invalid; e.g., if the object moves, is swapped out, and certainly after commit, because commit semantics require that the object be available for other users, to move to other address spaces and caches. Any programmer attempt to use such pointers when invalid will result in either a crash (segment violation), or, worse, will de-reference into the middle of another object, reading wrong values, writing into the middle of the wrong object, and corrupting the database. Any programmers familiar with pointer programming will recognize such pointer bugs; they're very common and very difficult to avoid. With a single-user program, they're not so serious. The programmer simply brings up the debugger, finds the bad pointer, and fixes the code. However, in a database shared among 100s or 1000s of users, the result can be a disastrous loss of critical, shared information. The loss may not even be noticed for days or weeks, after backups have been overwritten, and information is irretrievably lost.

Reference Indirection Guarantees Integrity

The Object Manager, however, avoids this problem at least for the most common cases of pointer errors, using the mechanism mentioned above, for object-level granularity. This is done by providing, transparently, one level of indirection. The application pointer de-reference looks exactly the same (e.g., objectRef-> operation()), but underneath, automatically, the Object Manager traps the reference, does a test, and a second indirection. Instead of a raw pointer, this is what's often called a "smart pointer." The pointer given to the application belongs to the application, may be stored in local variables, passed to functions, etc. In other words, once such a (raw) pointer is given out, it cannot be retrieved, and the DBMS has given up all control (and functionality). However, once this becomes a smart pointer, no matter what the application does with it, it always remains valid, because it always points to the intermediate (second) pointer (often called a handle). Since the application never sees that second pointer, it can automatically be maintained by the Object Manager, who automatically updates it whenever the object moves. Even after commit, the Object Manager can trap the reference at the handle, and as necessary re-fetch the object, lock it, cache it, perform heterogeneity transformations, then set up the second pointer, allowing operation to continue correctly and transparently.

The result of this Object Manager-supported handle indirection is to ensure integrity of every object reference, avoiding corrupt databases. The cost of this indirection is a test and pointer indirection, typically a couple cycles on a modern RISC machine, a couple hundredths of microseconds. Measurements of actual programs have shown this to be insignificant except in cases artificially constructed to measure it, because object de-reference is usually followed by other operations (comparing values, calculation, method invocation) which are much larger than a cycle or two. In any case, the benefit is always there, guaranteed integrity of object references. Returning to scalability, we see a second advantage to this handle indirection. In other approaches, with application-owned raw pointers directly into the object buffers, the DBMS is unable to swap objects out of virtual memory (VM). Once brought into the cache, since the DBMS knows nothing of where the application has retained such pointers, it simply cannot swap them. As VM fills, thrashing occurs. Worse, every operating system has a hard limit to VM. In some it's 1 GB, in others as small as .25 MB, and in practice it's the size specified for the swap file. In swizzling approaches, the swap file actually fills up, not just from objects actually accessed, but also from all the objects referenced by accessed objects; i.e., it fills according to the fanout of the actual objects accessed. Once that swap file fills up, no more objects may be accessed, at least within that transaction.



Swapping: 2-way Cache Management Enables Scalability

Instead, with the handle indirection, the Object Manager may dynamically support cache management or swapping in both directions. It swaps out the objects no longer being referenced, making room in the cache for new objects the application wishes to access. This effective reuse of the cache is the key enabler for scalability in objects, and can determine the difference between linear (with the Object Manager) versus exponentially slow scaling.

Scalability is also supported by other capabilities that fit nicely with this distributed architecture, including dynamic tuning of lock granularity and a variety of concurrency modes. To examine the first, note that most DBMSs pre-define and fix the locking granularity. In some it's an object, in others a page, but the choice forced upon all users. In some RDBMSs, the user may choose row or table locking, but row is almost always too slow, and table is almost always too coarse, creating too much blocking. This distributed architecture provides a means to dynamically tune the locking granularity. In the single logical view, locking is always (logically) at the object level. However, at runtime the ODBMS translates this into a physical lock at the level of a container, or a user-defined cluster of objects. When there are high-contention objects, they may be isolated into their own containers, to minimize blocking. On the other hand, when there are many objects often used together, then all of them (100s, 1000s, more...) may be placed into the same container, where they may all be accessed with a single lock. Since locks involve inter-process communication (milliseconds, 10^5x slower than normal memory operations), this can save 100s to 1000s of such slow messages and dramatically improve performance. In a production multiuser system, the optimum performance is achieved by tuning the granularity of such locks, some between finer granularity (to minimize blocking) and coarser granularity (for higher performance). Such tuning is not restricted, as in RDBMSs, to table (or type). Instead, objects of different types may be freely clustered together into locking containers to maximize throughput and performance. Also, the distributed architecture, because of its single logical view, allows such tuning to be done dynamically, without changing applications, while the system is online, simply by measuring statistics and moving objects among containers.

Concurrency modes may also dramatically affect performance. While some RDBMSs allow these, not all do, and even those that do are limited to applying them to tables. With the availability of all locking primitives, including locks with and without wait, with timed waits, etc., sophisticated users may customize their concurrency to meet their needs. Prepackaged concurrency modes help users in common situations, the most popular of these being MROW, or Multiple-Readers-One-Writer. In most systems, whenever one user is writing an object (or tuple or record), all other users are waiting, taking turns, which slows down everyone. With MROW, up to one writer and any number readers may run concurrently, simultaneously accessing the same object, with no blocking, no waiting, and hence much higher multiuser performance. Dirty reads, that allow readers to read what the writer is in the process of changing, give similar concurrency, but at the expense of lost integrity, because the readers may see partially changed, inconsistent information, dangling references, etc. Instead, MROW automatically keeps of pre-image of the object that the writer is writing, and all readers see this pre-image. Therefore, readers see a completely consistent image as of the previous commit. If the new writer commits, the readers have the choice of updating or not. For applications that find this semantics acceptable, and do more reads than writes, which describes by far most applications, MROW can be a huge performance boost.

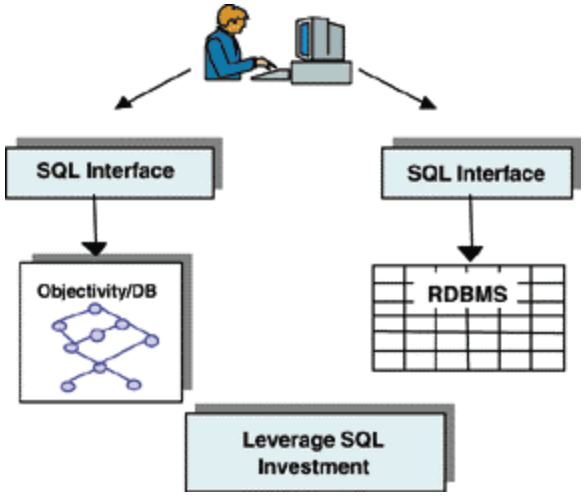
Another capability, cross-transaction caching, can affect performance significantly. In most systems, at commit time the cache is flushed to server disk. Then, if some of the same objects are accessed in a later transaction, the DBMS must again fetch them from the server, costing (again) milliseconds. Instead, the Object Manager preserves the client-based cache, even after commit. If the application attempts to access any of those objects, the Object Manager does a timestamp check with the server(s), and if no one else has modified the objects in the meantime, the application is allowed to continue access to those objects directly in the cache, taking advantage of the 10^5x performance boost.

Finally, replication can be a major boost in performance, not to mention reliability and availability, of large, multiuser systems. While RDBMSs can, in principle, provide some replication, they are hampered in two ways. First, their old central-server architecture makes implementation of replication difficult and slow (similar to the way it makes implementation of referential integrity difficult and slow). All user access is directly addressed to the server. Therefore, any replication must first go to that server, who then might be able to pass the request on to another server managing a replica, etc. Such multiple server interaction, required by the lack of a single logical view, turns a single access into multiple, slow (millisecond) inter-process communications. Second, the RDBMS lacks any knowledge of the appropriate units for replication. All that's available to the RDBMS server is tables of flat data, so that's all it can replicate, either full tables or full databases. The distributed ODBMS architecture, instead, solves both those problems. The distributed single logical view makes it transparent where the objects are located, so the system can directly access whatever replicas are available. Also, the definition of objects at multiple levels allows the system to replicate those objects the user wishes to replicate, rather than all of the database or all of a given type. Finally, unlike most RDBMS approaches, all replicas are automatically kept in synch dynamically, at each transaction commit, with guaranteed integrity even under faults (node and network failure), and with improved read performance, typically no slower write performance. Many of the issues we've discussed have related to performance (avoiding the server bottleneck, 2-way cache management, dynamic tuning of locking granularity, etc.) We'll turn now to a brief discussion on integrity, flexibility, and extensibility. Integrity, aside from basic mechanisms (such as the handle indirection, above), arises from users defining rules or constraints that must be observed. In an RDBMS, the ability to do this is limited to the level of flat, primitive data. Further, invocation of such user-defined rules is limited to the small set of predefined system-provided operations (delete, update, etc.), and requires writing stored procedures which can be tricky for typical programmers. In the ODBMS, such integrity rules are directly implemented as methods (or operations) on any objects, at any levels. Such methods may implement any level of operation, from simple update to application-level (e.g., route cell-phone message among satellites, manufacturing process, etc.), with all the necessary interdependencies. Users may write (and change) such methods in whatever languages they like, from c++, to Java, Smalltalk, or even visual basic. The result is that a much higher level of integrity is supported. Rather than limited to primitive data level integrity, the ODBMS supports integrity all the way to the application level.

We've seen such examples of flexibility and extensibility in terms of physically moving objects, adding servers, etc., all dynamically and transparently. It's significant, also, to note that the ODBMS also supports, with a large amount of transparency, logical changes, or changes to the schema. No matter how proficient the designer, systems almost always undergo change as designers find ways to improve the system and to add functionality. Such changes often change the application's data structures. In the RDBMS, the concept of a schema (user-defined data types) is limited to only one data structure, the table, with no operations or relationships. Applications must build their own structures and operations out of these primitives. When these change, the changes at the low level multiply, requiring manually re-doing the mapping. Often these changes will make existing databases in the field invalid, and require painful changes, such as dump and reload, not to mention rewriting of applications.

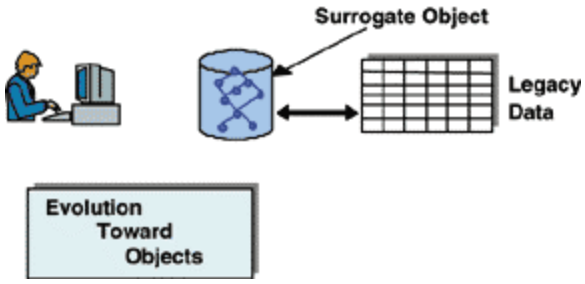
ODBMSs support the concept of schema evolution with instance migration. The user may define any structures (with associated operations and relationships). Each such type definition is captured in the ODBMS as a type-defining object. When the user changes a type, the ODBMS automatically recognizes this, analyzes the change by comparing to the in-database schema, and automatically, in most cases, generates code to migrate old instances of those types to the new type. The user may then decide whether to convert all such instances, only some (e.g., one database), none, or only convert objects on demand, as they're accessed via the new type. Application and system designer making such changes may customize such automatic migration by plugging in callback functions (e.g., to initialize new fields to some value calculated based on values of other fields). This may all be done dynamically, online, providing the ability to extend systems, add functionality, without breaking existing operations.

Finally, we'll close with a discussion of legacy integration. Few users have the luxury of throwing away all preexisting (legacy) systems and rebuilding from scratch. Instead, even as they wish to add new capabilities, new databases, and objects, they must continue to support and interoperate with these legacy systems. There are two main approaches to achieving this, the first based on SQL/ODBC, and the second on surrogate objects.



Legacy Integration with SQL and ODBC

Since the ODBMS now supports SQL and ODBC, user applications, tools, and even direct, interactive access may simultaneously access old, legacy RDBMSs along side newer, distributed ODBMSs. SQL supports a common language already familiar to many, in use by many applications and many tools. The ODBMS support for SQL is automatic. A class or type appears in SQL as a table; an instance as a row in the table; an attribute, operation, and relationship all appear as columns. Support of existing applications and tools requires ODBMS support for full SQL, not just predicate query, including DDL (create-table creates a new object type), DML (insert, update, delete, via methods), and security (grant and revoke user or group access at the level of every attribute and every method). This last, in fact, allows the ODBMS to selectively enforce encapsulation by requiring certain users to access objects only via certain operations. ODBC support allows off-the-shelf use of any of the popular GUI tools, including Visual Basic, PowerSoft, Microsoft Access, Crystal Reports, SQL Windows, Impromptu, Forest and Trees, etc. All such access, via ODBC, can interoperate simultaneously against both the legacy RDBMSs and the new ODBMSs. This also allows leverage of existing user training. Users familiar with any of these tools or legacy systems may immediately access the new ODBMSs, and, over time, they can learn more about and gain more benefit from the objects.



Surrogate Objects Integrate Legacy Systems

While this last, SQL/ODBC, approach leverages existing systems and user training, the other common approach, surrogate objects, provides a simple, consistent object view for users who prefer objects. Both approaches may be used simultaneously. First, the designer of the new, object system creates a surrogate object that stands for legacy data. It's up to the user to decide how those objects appear. Although it's very easy to map a row to an object, etc., it may be desirable to design the desired, future, ideal object view of all the systems, and then let the surrogate hide any complexity of mapping these to the legacy systems. This is achieved by implementing methods for the surrogate object type that can read and write the legacy database, sometimes doing so via legacy systems in order to continue the application-level integrity mechanism. Third party products can help in doing this mapping from objects to RDBMSs. Even proprietary flat files or mainframe files (ISAM, etc.) can be supported in this way. Once someone has written and installed these surrogate object methods, the ODBMS makes these objects appear exactly as any other ODBMS objects. They become part of the single logical view, so users and applications can access them just as they do any other objects . As they're accessed, they go off and read or write the legacy systems, but that's all transparent to the object users. The result is that users in the new, object world see the simple, object view only, yet they have full access to the legacy systems. The legacy systems remain functioning unchanged. Over time, if and when it makes business sense, some of that legacy information can be migrated into native objects. Of course, doing so requires changing (rewriting) the legacy systems, but it may be done incrementally, as necessary, and all object users so no difference at all.

Benchmark

We turn now to some benchmarks comparing RDBMS performance to ODBMS performance. The goal here is to search for the relational wall, and, if found, quantify it. This will show when it occurs, how quickly it rises, how large it gets, etc. The benchmark will also show some typical RDBMS operations and investigate how these behave under an ODBMS, in order to determine if anything is lost. In short, the results clearly show the existence of the relational wall, rising exponentially as complexity increases, to orders of magnitude slower than the ODBMS performance. It also shows the ODBMSs are comparable on simple data and simple RDBMS operations, but, again, become far superior, even in joins, due to internal relationship optimization, as complexity increases.

Any report on benchmarks must provide two caveats. First, the user is ultimately interested in his own application performance, not in that of the benchmark. Since ODBMSs address a far wider variety of applications than RDBMSs, this is even a larger caveat here. The best is for the user to benchmark his own application (or find a similar user that has benchmarked a very similar application). Second, benchmarks can become a battle among implementers, rapidly diminishing their relevance to users. Given any benchmark definition, any vendor can go to great lengths to optimize the implementation of that benchmark, even to change the kernel of his DBMS appropriately. Some of this has happened with the well-known TPC benchmarks which, in any case, lack measurements relevant to many ODBMS applications. In short, the benchmark game becomes a competition among benchmarkers, and a competition in which user relevance is lost.

To avoid these problems as much as we are able, we do two things. First, we chose a benchmark content that is generic. It is a model of objects connected to objects, measuring times to create such objects, traverse them, select them, query them, etc. Such a model might be viewed as a very common manufacturing problem, calculating a Bill of Materials on a parts explosion. At the same time, it can be viewed as a very common network management problem, modeling dependencies, fault reports, etc. Other applications include the World-Wide Web of HTML objects and hyperlinks; Inter and Intranets; document management systems, with documents containing possibly shared sections, chapters, paragraphs, and figures; financial services models of users, instruments purchased, and dependencies on other instruments and markets; multimedia systems with video clips and subclips; healthcare systems with patient records, graphs, x-rays, and relationships to doctors, hospitals, procedures; insurance systems with customers, policies, brokers, agents, dependencies, offices; etc. While no claim is made that this is by any means universal, it certainly is relevant to a wide variety of applications, and is relevant at a higher level of complexity than just the simple, flat operations of RDBMSs. Further, it is easy to instrument such a model and show its behavior as the complexity increases, simply by increasing number of parts, fanout, levels of depth, etc.

Second, we have contracted an independent third party to implement the benchmark. Rather than relying on vendors to execute the benchmark, this provides a level playing field in terms of hardware, network, environment, and even level of depth of work in optimizing and customizing the benchmarks. The implementers chose, Objex, Inc., which was lead by Jeff Kennedy. They have over 20 person-years of experience implementing and optimizing implementations using the same, leading RDBMS. Often, they have followed the vendor's own consulting team to further improve the customer's environment. This was chosen specifically so that there would be no question of ample RDBMS experience, so that the results would be fair.

In fact, the implementation for the RDBMS used the highest performance interface of the vendor (via the C language), and most of the well-known optimizations. For example, the database was split across three concurrently accessible disks, one for the data, one for the index, and one for rollback segment. Also, the buffer cache was increased, and the table storage parameters adjusted to accommodate the expected size in the benchmark. Within reason, typical of most users, best efforts were made for the RDBMS implementation. The ODBMS implementation was a straightforward C++ implementation of the model, using a single disk, and no special or unusual optimizations.

The consultants who performed the benchmark report:

"The installation and execution of the benchmark was on a standard Intel Pentium system running Microsoft Windows NT 4.0. The benchmark application was not run in a client/server mode, but instead was run with the benchmark applications running on the same machine as the databases. Parallel versions of the application were developed to access Oracle and Objectivity. Care was taken to make sure that the benchmark implementation did not favor either database, and in particular the timing component was written to eliminate any prejudice that the function might cause to the results.

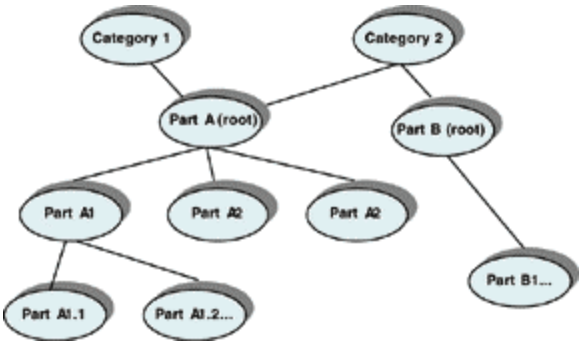
The results were very surprising. Objectivity outperformed Oracle in virtually all of the tasks at all depth levels except for small numbers of updates and deletes. As the depth increased in all of the tasks, Objectivity exponentially beat Oracle. In fact, the results seemed almost too hard to believe and the tests were rerun and corroborated using a separate program to make sure that we were in fact seeing what we thought we were seeing and that the data was indeed changing correctly. At low depth levels the results were mixed. Oracle was faster at updates and deletes while Objectivity was faster at inserts and connect/selects. At moderate or high depth levels the results showed Objectivity as being up to 30 times faster than Oracle. Basically, as the number of records processed increased Objectivity showed superiority in every task.

As veteran RDBMS application developers, we were not expecting Oracle to be so thoroughly beat. We truly expected the results to be more mixed with Oracle having superiority at such tasks as updates. With very little volume of activity, Objectivity shows its superiority in handling the activities that comprise the parts explosion benchmark."

The hardware configuration used was:

- Pentium 150 processor
- 32MB of memory
- 3 disk drives (2 SCSI with PCI controller and 1 FAST-ATA 2GB E-IDE)
- Windows NT 4.0
- Visual C++ 4.0

Both versions of the benchmark program create a set of Categories that has Part trees associated. The benchmark was designed to simulate a bill of materials parts explosion application in which several parts may be combined to form a single part as in the following diagram:



Benchmark Model

Benchmark Parameters

The benchmark is designed around several basic components:

- Root: The number of tree roots.
- Depth: The number of child levels below the root level.
- Fanout: The number of children at each level of the tree.

The programs accept parameters to determine the depth and fanout. In the previous illustration, Part A tree has a depth of 2 and fanout of 3 for Part A since it has Part A1, A2, and A3. In each run of the benchmark programs, six part roots were created (Part A- Part F).

Benchmark Execution

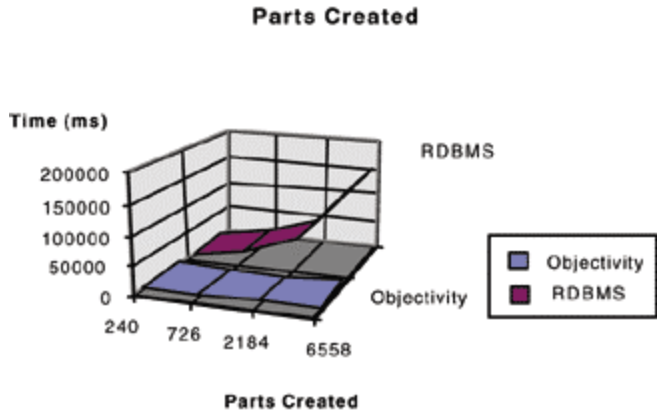
The benchmark program was executed with parameters creating six part trees with a fanout of 3. The depth was adjusted in each of the four runs to yield varying numbers of parts for each run. The four benchmark runs were performed with depth levels of 3, 4, 5, and 6. Each of the results below corresponds to the different depth levels used in the programs. Note that the SELECT phase was run after the parts were created and after the DELETE phase was run, so the number of parts selected is always less than the number created in the create phase.

Parts Creation

The first phase of the benchmark program was the create phase. The programs would first create the part trees within a single database transaction. The total number of parts created along with the total time was recorded for each benchmark run. Here are the results of those runs (times in milliseconds, the first row is for fanout=3 roots=6 depth=3, the second row has depth=4, etc.) The RDBMS consultants who ran the benchmark state:

"The results of the part creation phase show that even with a small number of parts, Objectivity is much faster than [RDBMS] in creating records. When the number of records rises, the difference in time between Objectivity and [RDBMS] is staggering. For the benchmark that created 6558 parts, Objectivity ran over 30 times faster. A separate program was written to validate that the records were actually created, because we couldn't believe Objectivity could create records so quickly. The other program printed all the parts, validating the test."

#	Objectivity/DB	RDBMS
240	931	2123
726	1342	15883
2184	2514	46617
6558	5047	154522



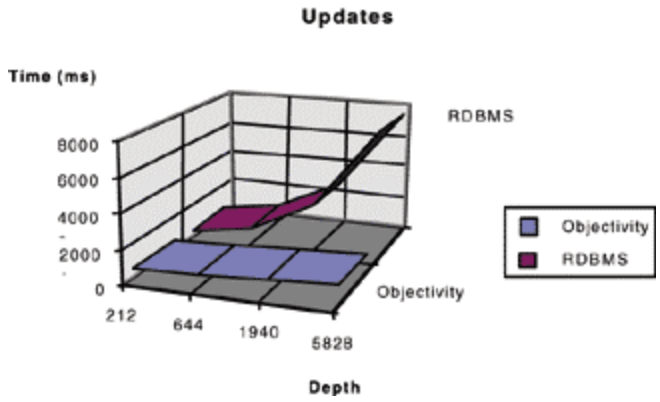
Object Creation Benchmark

Updates

The next phase of the benchmark program was the update phase. The program traversed the constructed Part tree and updated the description of every third part encountered. This update phase was repeated 5 times in each benchmark run and the time for each run, plus the total time in milliseconds were recorded. The results of the test are shown here (times are in ms).

For a very small number of parts, the RDBMS surpassed the performance of Objectivity. The RDBMS performance seems fairly linear based on the number of parts being updated. Objectivity's update performance increases dramatically as the number of records updated increases, with the final test showing Objectivity outperforming the RDBMS by over 7 times.

#	Objectivity/DB	RDBMS
212	5508	2614
644	6219	8071
1940	7691	26348
5828	10135	75950

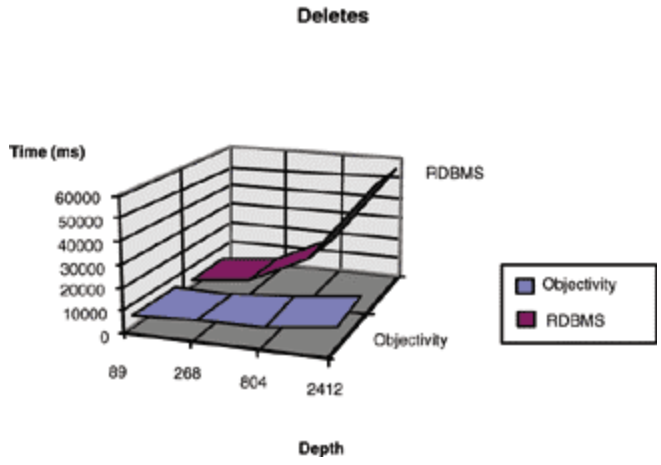


Object Update Benchmark

Deletes

The third phase of the program was the delete phase. Every third part in the tree that was a leaf node is deleted. This phase was executed 4 times in each run. With each successive run, fewer parts get deleted since the tree is smaller. The total time for all delete runs was then recorded. The results are shown here (again, times are in ms). Again the RDBMS outperforms Objectivity when the number of records being deleted is very small. Objectivity overtakes the RDBMS in the second test with 268 records, and dramatically outperforms the RDBMS when a large number of records are deleted. In the last test Objec tivity was over 5 times faster than the RDBMS in deleting records.

#	Objectivity/DB	RDBMS
89	4837	2985
268	5357	6109
804	6489	18737
2412	9964	56000



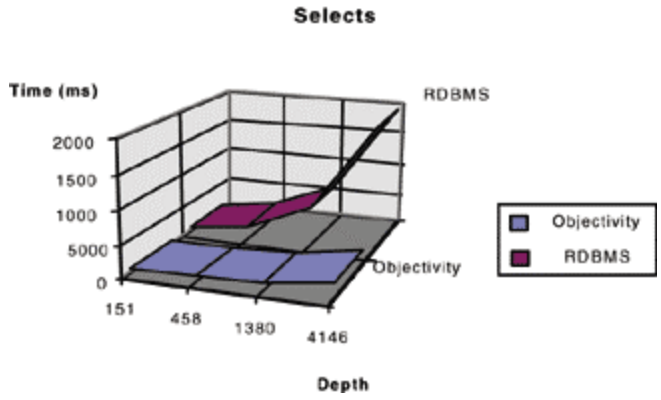
Object Delete Benchmark

Selects

The last benchmark run was the select phase. In this phase a separate program was run that scanned all Part trees and printed out each record. The total time to connect to the database and print each record was then recorded. Note that the records were printed to a file, so screen I/O would not play a part in the results. Here are the results of those runs (times in ms).

Objectivity is twice as fast as the RDBMS for a very small amount of records and "blows the RDBMS" off the scale for a large amount of records. Objectivity was able to print 1800 records per second in the last test, while the RDBMS averaged about 200 per second in all runs.

#	Objectivity/DB	RDBMS
151	380	721
458	531	1992
1380	972	5839
4146	2304	19007



Object Select Benchmark

This clearly shows the exponential superiority of Objectivity/DB over the leading RDBMS in typical network or hierarchy applications. One might wonder what is sacrificed for this advantage — perhaps SQL is slower or less functional, or ODBC tools are slower or less functional. To investigate this, we continue the benchmark to compare SQL and ODBC behavior. The results show that not only is there no loss in performance or functionality, but, as the complexity increases, the Objectivity/SQL++ query optimizer performs increasingly better than the RDBMS one, because it is able to optimize based on the objects and relationships in the database.

Summary

For simple data structures and operations, these benchmarks show that Objectivity/DB, the leading ODBMS, is comparable to the leading RDBMS. As complexity increases, it shows that the RDBMS performance slows exponentially while Objectivity/DB maintains linear performance. This dramatic exponential difference quickly scales to two orders of magnitude (100x), and beyond as size and complexity continue to increase. For SQL ad hoc queries, and for ODBC tools, the benchmark shows, similarly, that Objectivity/SQL++ and the RDBMS are comparable for simple data, but Objectivity/DB becomes much faster as complexity increases. Actual users have measured similar, and even more dramatic performance advantages of ODBMSs, up to 1000x faster (see Cimplex, next section). It is only a matter of how complex the application is.

Examples

Benchmarks can be a useful mechanism for comparing functionality and performance, but even more useful is to examine actual users, especially those in production. Towards this end, we'll briefly review several users, in a variety of industries and application domains. For each we explain what the application is, what benefits it receives from the ODBMS, and what experience it has had in production. Example domains include: Manufacturing (Cimskil, Exxon), Scientific and Research (CERN, Sloan Digital Sky Survey), Telecom Real-Time and Network Management (Iridium, QUALCOMM, Marconi, COM21, Nortel, Sprint), Financial Services (Citibank, Decision•ism), Document and Process Management (Payback Systems, Daikin, Intraspect), Process Control (Fisher-Rosemount, SEH, NEC), Data Warehousing (CUNA Mutual), Military/Transportation/Environmental Logistics and Control (BBN, Kintetsu, Siemens), and Energy (American Meter, Exxon). These also cover Asia, North America, and Europe, as well as a variety of different end user industries, and different technology applications, different and mixed platforms (Unix, Linux and Windows), a variety of different and mixed languages (Java, C++, Smalltalk), chosen from more than 150,000 deployed user licenses.

Cimskil
(Manufacturing)

Cimplex Corporation provides a mechanical manufacturing software package that embeds manufacturing data and experience in an object-oriented "knowledge base" to automate manufacturing engineering tasks. Cimplex, based on Objectivity, is in production in major manufacturing corporations, such as Boeing, Sikorsky Aircraft, Ford Motor Company, Ingersoll Milling Machines.

Cimplex accepts 3-dimensional solids-models of parts and assemblies from front-end CAD systems such as IBM/Catia, stores those in the ODBMS, and then automatically generates the manufacturing process to actually produce such parts and assemblies, including milling, drilling, etc. Users may graphically preview the process on screen, manually adjust it, and then allow Cimplex to download instructions to numerically controlled machines (robots, etc.) to actually manufacture the part.

In Cimplex, every point is an object. These are combined together into edges, faces, etc., along with higher level structures. Even simple solids models can produce large numbers of objects. As objects, each such structure is directly an object, with cubes containing varrays of faces and edges and 1:m relationships to points. Attempting to break this into tables produces an explosion of many tables and many more slow joins. When extended to complex systems as automobiles and aircraft, it can rapidly expand to billions of objects. In a benchmark, Cimplex reported that Objectivity/DB performed 1000x faster than the leading RDBMS, and 10-20x faster than other ODBMSs.

Exxon
(Configuration and FEA)

Exxon, a leading energy company, has deployed a system for managing configurations of drilling rigs and other assets, as well as analyzing such systems using the technique of Finite Element Analysis (FEA). This requires very large numbers of fine grained objects, and sophisticated calculations, modeling stress, heat flow, etc., that traverse and re-traverse these objects as they perform computations. Only an ODBMS can give the support necessary for such analysis. Before ODBMSs, most users had to build proprietary, flat-file systems. Now that ODBMSs can provide the performance needed and the integration with the host language that allows the programmer to simply use the language he knows (such as C++ or Java), these developers can concentrate on their application, while they transparently gain the benefit of scalability, integrity, etc.

Exxon deployed their first generation system some years ago, and is now developing their second generation. Their reasons for choosing the ODBMS over (O)RDBMS include:

Performance: The analysis operation is very navigational in nature, i.e. navigating from a particular node of a model to its element pieces, often in an iterative fashion. This makes database performance key to building a practical system that can perform its function in a timely manner.

Flexibility: The numerical intensive analysis operations being performed required varying-sized arrays of numerical information to be stored with each object. This fact precluded the use of a relational database, and made the flexibility of how the object database could model such information easily an important factor.

Robustness: Objectivity's architecture proved to provide the most robust storage capacity that could meet their application requirements. The caching approach provides high performance without sacrificing data integrity.

Heterogeneity: Transparent support for multiple platforms was another distinguishing factor of Objectivity. Exxon's deployment environment includes SGI and NT machines, and transparent interoperable support for these platforms reduced Exxon's development effort and risk on the project. Others were unable to support mixed use of heterogeneous platforms.

CERN
(Data Gathering and Analysis)

CERN is the leading worldwide consortium, headquartered in Geneva, Switzerland, for performing scientific experiments at the sub-particle level. For their new generation accelerator, they need to build the largest database ever built, which is expected to grow to 100 PB = 100,000 TB = 100,000,000 GB = 10^17 Bytes. After evaluating all other available DBMS technology, both RDBMS and ODBMS, they concluded that the only system scalable to this level is Objectivity/DB.

Their new accelerator for High Energy Physics (HEP) is approximately 28 km in circumference. When live, data will be pumped into the database at the rate of 100 MB/sec. Such data w ill store events, which are particle collisions and their results. Various users, around the world, will perform analyses of this data. One primary goal is to seek the Higgs boson, postulated to be responsible for creating mass (otherwise the universe would be all energy). Probing deeper and deeper into nature requires more energy, larger accelerators, and, because the desired events are more rare, larger amounts of data with more sophisticated analysis.

Although the new accelerator will not be online until approximately 2003, CERN is already using the ODBMS for several past experiments, and have measured up to 80MB/sec data input, with sizes up to a TB.

Size and distribution were the key driving needs that lead to ODBMS. The natural modeling of objects for the complexity of events was also quite useful. In size, no central-server architecture could possibly handle such large sizes. These older architectures scale only as far as the largest computer available, and there are no computers that could handle such huge volumes of data and concurrent access, even if they were affordable. Instead, the CERN team required a system which could use distribution and a single logical view to transparently allow adding multiple servers, multiple disks, to support many users and many databases simultaneously. Distribution is also important for access, because the users of the database are scattered worldwide, with access via the Internet. Here, the key was the distributed ODBMS ability to selectively replicate certain subsets of all the databases and automatically re-synchronize them.

For more information, see:

http://wwwcn1.cern.ch/asd/cernlib/rd45/workshop/objy_nov96/slides /index.html

http://asdwww.cern.ch/pl/cernlib/rd45/slides/objyrs_nov95.ps

http://wwwcn.cern.ch/pl/cernlib/rd45/index.html

http://asdwww.cern.ch/pl/cernlib/p59/p59html/drdc_1.html

http://asdwww.cern.ch/pl/cernlib/p59/index.html

Decision•ism
(Decision Supportware)

Decision•ism, Inc., of Boulder, CO develops and markets data mart management software that gathers and transforms data from various sources such as operational systems, source applications and data warehouses, into focused OLAP data marts that support decision-making by business professionals. Decision•ism selected Objectivity/DB for its flagship product, Aclue Decision Supportware, because of its speed, flexibility, and transparency.

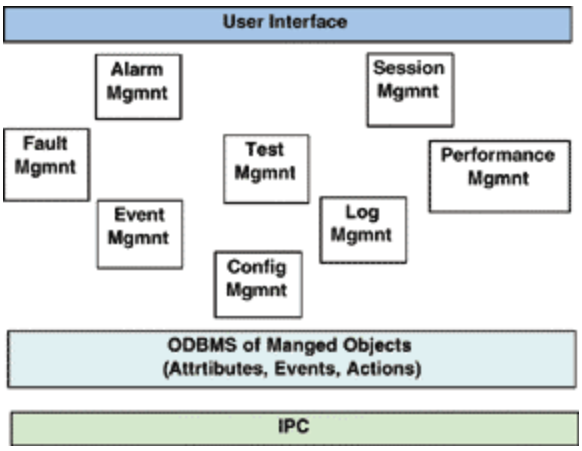
Aclue delivers on-line analytical processing (OLAP) data mart management. Aclue helps organizations gather, cleanse, transform and aggregate data from one or more production applications and/or data warehouses. Aclue then exports this data into smaller, focused data marts that suit the specific needs of individuals or groups of users. Aclue exports directly into Arbor Software's Essbase or Oracle's Express-multidimensional OLAP cubes.

Based on a central Business Information Model, Aclue serves as an information broker and ensures consistent and synchronized data delivery to multiple data marts via a "read-once, write-many" architecture. Objectivity/DB is used as the staging area to hold both the business information model (metadata) as well as the imported data. Cubes are then exported from the data collected in Objectivity/DB. Objectivity/DB's inherent support for the efficient storage and rapid, flexible manipulation of multi-dimensional data makes it a critical underlying technology within Aclue.

QUALCOMM
(Telecommunications)

QUALCOMM, Inc., San Diego, CA, is a leader in digital wireless communications technologies. The company manufactures, markets, licenses and operates advanced communications systems and products based on digital wireless technology. QUALCOMM had revenues of $107.5 million in the fiscal year ending September, 1992. Publicly traded since December 1991, its stock trades on the NASDAQ National Market System, symbol: QCOM. Founded 1985. Over 900 personnel are located at the company's San Diego headquarters and throughout the United States. Products and Technologies include OmniTRACS System, the most technologically advanced two-way mobile satellite communications and tracking system of its kind in the world. Introduced in late 1988, the OmniTRACS * Osystem provides data transmission and position reporting services for over 225 transportation companies and other mobile customers using more than 43,000 OmniTRACS terminals in the United States and Canada. The system also operates in Europe and service is planned to begin in Japan, Mexico and Brazil in 1993. CDMA, QUALCOMM's Code Division Multiple Access (CDMA) is a next-generation technology for digital wireless telephone communications. CDMA improves voice quality and increases capacity by as much as 10 to 20 times over today's FM analog cellular system. CDMA uses spread spectrum technology to break up speech into small, digitized segments and encode them to identify each call. A large number of users can thus share the same band of spectrum and greatly increase system capacity. Commercial deployment of CDMA cellular systems will began in several U.S. cities in early 1994. A North American standard based on QUALCOMM's CDMA technology was adopted in 1993 for the rapidly growing cellular telephone industry. Under development through a joint venture between QUALCOMM and Loral Aerospace Corporation, Globalstar is a low-earth orbit (LEO) satellite-based, cellular-like voice and data service for mobile or fixed applications. Globalstar will be designed to provide affordable and reliable voice, data, fax and position location services to customers around the globe. QUALCOMM's government products include Data Link Systems for government test and training ranges, and the QM6100 Universal Compatible Modem for Defense Satellite Communications System.

QUALCOMM's industry leading CDMA technology supports digital, wireless communications systems. The base stations that support all communications, routing of messages, voice, data, images, dynamic rerouting, etc., are based on Objectivity/DB. These have been licensed to Nortel, Ottawa, CA, the largest broadband telecommunications vendor in the USA, who is extending the system and installing it for Sprint in a $2B contract.



Base Station Manager (GDMO in ODBMS)

The International standard (ITU X.700 series) for managing telecommunication networks, General Definition of Managed Objects (GDMO), is written in terms of objects. QUALCOMM found they could implement these directly as objects, which could send messages to each other, to accomplish the dynamic management of the network. This model is naturally a network of related objects (1:many, many:many), in which the ODBMS bidirectional relationships were far faster and easier. Also, mapping from their C++ language to the RDBMS cost an additional layer of complexity, slowed down the system, and dramatically increased the cost of maintenance, due to the need to change this mapping and the underlying RDBMS. Also, the RDBMS lacked inheritance, which is used heavily in the model, and which allows extending the system easily by adding new (sub-) types of network elements. The ODBMS, they found, provided the necessary distributed single-logical view over all the base stations and the full network. Further, it allowed the programming model of objects to be directly represented as ODBMS objects, easy implementation, maintenance, and improving performance. Other advantages include support for versioning and scalability, plus proven success in 24x7 production use.

COM21
(Cable Modem)

COM21 sells and supports a cable modem system, providing telecommunications (including internet access) via existing cable TV coaxial cables with throughputs up to 1Mbps, compared to today's maximum of 56Kbps via telephone wiring (~20x faster). To support the elements in this network, Com21 developed their NMAPS (Network Management and Provisioning System) using Objectivity/DB. Their system maintains information about a network of cable modems (STUs) and their hub controllers (HCXs).

Their Object Model is an abstraction/mapping of the actual physical hardware involved in the cable network. Classes include such real-world abstractions as HCX (the cable hub), HCX-slot (slot for card within the hub), HCX-card (card that fits within slot), STU (the modem attached to the hub), Subscriber (individual subscriber to the service), QosGroup (quality of service group for set of subscribers _ different levels of quality of service are available.) An object-database is a natural fit to such real world constructs.

Objects are divided into databases as a function of their usage. The topology database is the primary database and represents the physical state of the network (i.e. which modems are connected to which hub). Polled statistical data lives in its own database, and a database also exists for recent alarm data.

The NMAPS databases are accessed simultaneously by any number of GUI network management applications, which display information and support queries. The GUI application is in Motif and presents a meaningful view of the information stored in the databases to the human observer. Using Objectivity dictionaries greatly improves the performance of named lookup queries from the GUI.

The polled statistical data grows quite rapidly, generating up to a gig of polling data per day. These objects are created by background daemon processes. The daemons also populate the polled data database.

Their choice of ODBMS over (O)RDBMS, and from which vendor, was based on the following criteria:

- Quality of the product _ According to their spokesperson, other DBMS systems Com21 experimented with did not have the same level of quality and reliability as Objectivity. They've since advanced through two more versions of Objectivity, with greater reliability, scalability, and capability.
- Distributed Architecture - Com21's application is a natural fit to Objectivity/DB distributed architecture which allows the NMAPS databases to be spread across several file systems, on different servers, and to be accessed transparently by different monitoring and control stations.
- MROW read feature on read transactions provides the highest possible level of concurrency. When two applications update the container simultaneously, they must serialize. However, all readers and up to one concurrent writer run physically in parallel, with no blocking, resulting in much higher throughput.
- The natural fit of containers to HCX Objects provided better support for multilevel clustering and caching, and hence easier modeling and development, and much faster performance. Recall the HCX is the hub and is attached to multiple modems (STUs). Modems are placed within the HCX container corresponding to the HCX hub to which they are attached. Their HCX object actually derives from Objectivity container class and the STUs it contains represent the modems connected to the hub.
- Support for dynamic moving of STU to different HCX (online re-clustering).
- Multiple containers within the polling database allow daemons to concurrently update information in the database, without concurrency blockages, allowing, again, much higher throughput.
- Support for mapping and indexing for fast lookup; e.g., all Subscriber objects are indexed by name and account number and Com21 also maps from account number to Subscriber object. This allows fast generation of billing information.
- Ease of development with Objectivity allowed Com21 to develop their own custom event notification system fine tuned to their application needs.
- DBMS is completely transparent _ embedded inside product.
- Scalability and Distribution. Com21 cable networks are growing rapidly. Their cable modem networks have encountered no performance problems despite rapid growth.

Nortel, Sprint
(Cell Phone Base Station Management)

Nortel, the largest supplier of telecommunications equipment in North America, built the system in use today for the Sprint PCS cell phone network. Starting with the Qualcomm CDMA technology, they were able to use the ODBMS capabilities to extend the system and customize it as needed, something that would be extremely difficult in (O)RDBMS technology. Their application is a base station manager (BSM) that stores configuration and fault information for a set of CDMA cell sites (Base Transceiver Site, BTS). Their key DBMS requirements were:

Performance: The performance of the BSM is important to managing an adequate number of BTS cell sites. Higher performance allows for more cell sites to be managed by a single BSM and adds deployment value to the BSM product.

Time to Market: Wireless service providers are building out their infrastructure to support the growing wireless market. Base station management software availability is a critical factor in gaining market share in this market. Nortel's choice of base station management software components was strongly driven by this criterion. See, also, QUALCOMM, above, to understand the basic use of the ODBMS technology, the starting point from which Nortel extended the system.

Nousoft
(Library Document Management, Web)

Nousoft, Monterey, CA, was founded in 1993 by the developer of the leading installed base (300) library management system with the goal to bring a new generation of capabilities. Its systems support universities and other large libraries for interactive, electronic access to documents, automatic accounting of copyright fees, and easy access dynamically from the World Wide Web. It supports MARC (the library standard), SGML, and HTML, with databases in use up to 10s of GB, up to 100s of users. It's built on Objectivity/DB.

The Nousoft system supports 70 inter-object relationships. This has never been successfully accomplished in RDBMSs, but is straightforward and fast in the ODBMS. A typical installation has 14M transactions/yr, with 17 branches, and sub-second (complex, WAN) transactions. Another ODBMS payoff is due to caching in the client. An 8.5x11" image, 1-bit b&w is delivered to the user in under 1 sec. Even for 16b (256 colors), less than 4 sec is required, which is several times better than any other system. Users couldn't believe it was real. There is no degradation at all for multiple users…except when they happen to simultaneously request the very same image (not likely), in which case, time roughly doubles.

The same implementers built the new system in 1/4 the time of the old. Although the implementers had the benefit of experience, they also implemented significantly more functionality, and credited the 1/4 savings to object technology and the ODBMS. The old system implementation required 3 people x 18 mo to get DB to handle the basic MARC record. The new system required 2 people x 4.5 mo, with about twice as much functionality.

Iridium
(Telecommunications)

Iridium, a $7B consortium headed by Motorola, is a satellite-based, wireless personal communication network designed to permit any type of telephone transmission—voice, data, fax, paging, to reach its destination anywhere on earth, at anytime. Although the full system will not be operational until 1999, several applications are now in production. The system includes over 66 low-earth-orbit (LEO) satellites, in order to work at low power with small (antennae and devices). This means, though, that satellites orbit the earth in less than 90 minutes, so during a typical telephone conversation, messages must be dynamically rerouted from one satellite to another. The developers describe the Iridium switching problem as equivalent to re-configuring an ATM switch every 4 minutes. Objectivity/DB is used in the base stations, for satellite routing and rerouting, and message routing and rerouting, as well as network management.

Iridium Handheld Wireless Communication Anywhere and Anytime

One application involves storing Mission Plans, a hierarchical description of the state of the satellite, from antenna to modem. The plans are for long periods of time (8 hours), with time-series like arrays of data for each time, giving state information for various times. The plan is updated every few hours and constantly referenced other times. Such dynamically varying sized structures are well supported by ODBMS, but not by RDBMS.

They have three facilities, each with ~50 Sparc20 servers and workstations. These facilities, in Phoenix, Washington DC, and Rome, are linked internally and among each other and 20 gateways as one distributed system. The distributed ODBMS single logical view was critical for this. They expect to generate 10 Gigabytes of data/day and will need to keep ~5 days of data online, requiring scalability. Other database problems include network management (node management, fault management), satellite control (commanding, telemetry, orbit management), and mission planning (resource scheduling).

For resource scheduling, modeling issues require complex, multi-versioned objects so they can create plans and alternate plans for satellites, communication paths, etc. This is compute intensive, generating 10-30GB/day, with 100 concurrent clients.

Another, related application chose to use an RDBMS. They experienced first hand the difficulties of attempting to use an RDB with a complex data model. The bottom line was that they found it impossible to keep an RDB schema mapped to an OO design. The schema was always 2 weeks behind. Eventually they abandoned the approach and went to file based persistence in order to make the first release. A decision to use Objectivity/DB in the second release was later made, in order to regain the DBMS facilities of recovery, concurrency, scalability, and distribution.

According to the lead database architect for Iridium:

"We had chosen to go with OO technology early in the program for good reasons that I will not go into. Sybase was chosen as the defacto database. Several of us in the architecture/analysis team felt that for a complex database problem, the benefits of going OO would be negated if the database backend was an RDB.

Management here was already suffering from technology-panic. Consequently we had to explicitly examine every single alternative before making the OODB/Objectivity decision. We looked at Hybrids (UniSQL, OpenODB, Matisse), object gateways (Persistence), glorified C++ DMLs (DBTools.h++) etc. We talked to at least 20 customers, and did about 4 separate benchmarks. Our conclusions were that you either go with RDBs or OODBs. The other choices were neither here nor there.

During the evaluation process, it became clear to me that there seems to be a gross oversimplification on how easy it is to translate OO data models into relations. Having written a developer's guide to do this, having seen how much code is generated by the "Persistence" tool to automatically translate OO schemas to RDB application code, having used DBTools.h++ to write code that views the RDB from C++, I for one have no illusions about this.

I must point out that OODBs are not a panacea. Applications such as billing are fairly relational. However, in things like satellite mission planning, network management and satellite control, one has to explicitly represent mission plan objects, managed objects in the network and their relationships etc. Most of the time, there are a fair number of associations between these objects, and obviously considerable use of inheritance in the model. Translating all this to tables is very cumbersome. Not to mention that large designs tend to change, often quite a bit. Having an RDB is like sticking a

magnifying glass on your OO design. If a change looks scary in your OMT model, watch out for its effect on the RDB schema. Anytime you try to isolate your RDB from schema changes (say by using foreign keys for every association), you pay serious performance penalties.

Going with OODBs for purely performance reasons is probably misguided. However, if you have a non-trivial OO application, your engineering risk in terms of coding and maintaining the application are much greater with RDBs than with OODBs. Out of curiosity, I asked the DBTools.h++ people if they saw a groundswell of OO applications using their product. I was told that most of their customer base is and remains people who are reverse engineering the database to be readable by C++ applications. To me that is the market speaking."

BBN
(Military Logistics and Control)

BBN, the creators of the Internet, have deployed a Crisis Management and Resource Logistics Management system built on Objectivity/DB. It supports distributed usage through multiple nodes in the field, for military and civilian crises that require immediate sharing of knowledge and optimization of resource allocation. For example, it was used after the Kobe, Japan, earthquake to coordinate rescue operations.

The top three requirements that led to the choice of the ODBMS are scalability, flexibility, and survivability. The first is due to the need to support massive operations, without the worry that a central server will overload. Instead, they can simple add more servers as needed.

Flexibility is required to support field deployment with constantly changing distribution models, nodes being attached and removed, while communication stays the same and integrity is ensured. In addition to such physical flexibility, it also requires the logical flexibility to support changing and adding object types online.

Finally, survivability is a necessity in such life-threatening operations. No single failure can disable the system, due to replication and use of multiple parallel servers. Also, there is never a need to disable the system due to the 24x7 operation capability.

Intraspect
(Knowledge Management)

Intraspect developed their namesake Intraspect Knowledge Management System (IKMS) using Objectivity/DB. IKMS allows a company to manage its knowledge resources (group memory) by giving employees the ability to collect information, organize it, collaborate, and discover and reuse colleagues' work. Intraspect originally started using Objectivity/C++, but they are now porting their application to Objectivity for Java 1.0. Objectivity for Java deployment occurred December 15, 1997. Intraspect strategically chose to deploy on Objectivity C++ despite the fact they were really waiting for the Objectivity for Java binding. Almost all of the Objectivity C++ features are available in Objectivity for Java (the features that are not available are language difference features and do not make sense in Java). Intraspect and Objectivity are partnering tightly and working together towards successful deployment of Intraspect on Objectivity Java.

Some users of IKMS include: an oil company for engineering archive and information repository, a financial services company for software architecture planning and coordination, a high-technology company for software configuration knowledge base, some consulting companies for managing virtual teams and their re-useable knowledge repositories, a telecommunications company for competitive analysis, and a software company for RFP responses and field intelligence.

Intraspect chose Objectivity over ORDBMSs and other ODBMSs because it has features that are not available from other object databases. Objectivity's distribution and replication capabilities drove their selection. Objectivity allows them to "grow into" distributed database features and have Objectivity manage the distributed aspects of their application transparently via the Objectivity single logical view. Intraspect supports hundreds of simultaneous users in their first release, eventually supporting thousand of simultaneous users. Objectivity's replication and distribution features will allow them to split the load across multiple servers. The elegance of the Objectivity binding as well as the ability to perform distributed queries with the programmer abstracted from the location of the data is in large part why Intraspect choose Objectivity. "Simultaneous users" are simultaneously active users, not just logged-on users. Using Objectivity, performance scales linearly with database size. IKMS document objects are often composed of hundreds of Objectivity objects, because they are divided into segments. Documents can be of any type. A 10MB PowerPoint presentation, for example, is composed of 640 chunks and one "metadata" object. For every other type of object, each IKMS object is exactly one Objectivity object. Breaking an object into small chunks allows speedy network delivery and Objectivity page server technology complements this document packaging technique. The Objectivity paged memory cache allows sections of a document to be present in memory also increasing performance (a multi-megabyte document does not need to be entirely in memory before it can be used). The IKMS current container strategy is 1 container per document allowing any number of reader processes or threads and 1 concurrent update process or thread using the Objectivity MROW feature. Each user command (e.g. dragging a desktop file and dropping it into IKMS "group memory", sending an email

message to one of the mailing lists IKMS maintains) is a separate transaction. Transactions are short-lived for most commands.

IKMS provides two means of accessing the data in their Knowledge Management system. Clients can use either a Java enabled Web-browser or, if the client is running on Windows, a stand-alone Microsoft Foundation Class Application which provides the same functionality. Objectivity is used only in the server currently. Objectivity for Java will allow IKMS to perform in-process multi-threading on the server. Pure preemptive multi-threading is important for performance (especially for a server application) and Objectivity seamlessly maps Java threads to database transaction threads. In-process multi-threading allows the client to "reuse" an open connection to the federated database. Any thread, even new ones, can "reuse" the open connection because the federated database connection is a property of the process, not the thread.

Due to the complexity of the data model, IKMS clearly needed an object database. Documents in the database are highly interconnected. Whenever data is collected, IKMS finds all the contexts in which it is used and presents this information to the user. This allows the user to see how everyone in the Knowledge Management system takes advantage of the same information. This context management would be incredibly painful with a relational database. In addition, previous search results are saved persistently and presented whenever IKMS can ensure they are still accurate. Automatic persistent threading of messages also contributes to the complexity. IKMS also stores product version information persistently in Objectivity.

IKMS has proven that Objectivity could be integrated with the other products easily. IKMS integrates the Verity search engine, WC3 Webserver, a Microsoft Foundation Classes Windows application, an email server and a server written entirely in Java using Objectivity. In fact the entire application is written in Java, except for the C++ interface to Verity. The use of Objectivity is completely hidden and transparent, embedded inside the IKMS product. The Objectivity schema evolution feature gives Intraspect the flexibility to change their schema. In the Knowledge Management field, loss of information is unacceptable, yet customers must be able to take advantage of the latest engineering innovations of IKMS. Old data is never lost but brought into the new schema.

Online backup is important to IKMS. Intraspect has wrapped Objectivity's online backup facility into their own user interface so that the users can schedule regular backups without ever seeing an Objectivity program. Users can select the days and times when backups should occur, and IKMS makes a backup of the database in a directory in the file system. This happens while the server is running. The flexibility of the Objectivity 24 X 7 online backup facility allows this backup customization.

Fisher-Rosemount
(Manufacturing, Process Control)

Fisher-Rosemount, one of the leading process control equipment and system providers, supplies to a large variety of industries, including pharmaceuticals. Their highly distributed systems include active microprocessors to control every valve and switch, all monitored by workstations networked together, allowing both real-time control of process parameters (such as temperature, pressure, etc.) as well as collection of historical data (for later query to determine actual process conditions during the manufacture of particular batches of drugs, for example, correlating these with the actual device/machinery where they originated).

The lead designer offers this description, motivation for using Objectivity/DB, and comparison to RDBMSs:

We have made a very good decision.

A Brief Overview of Our Development:

Objectivity is being used at FRSI in several areas:

- As the database for an existing Console product
- As the database for our Batch Historian
- As the integrated database for a future product line

In these 3 areas Objectivity is being used to store configuration data and to store production data. In the first release of the future product we anticipate a very complex configuration with over 500 classes with a "web" of over a 1000 associations. The configuration database will store approx. 100 MBytes of configuration. The production database will store several GBytes of data.

The selection criteria for us included:

- Reliability: the database must be able to operate in a 7x24 situation for many months at a time.
- Concurrency: the database must be able to be "pounded" on by several users and many processes at the same time. We have several transaction models in place for dealing with different levels of access.
- Performance: with such a complex schema, i.e., many different objects with very different compositions and many associations between classes, performance is a key consideration.

ODB Selection

A comparison of Object-Oriented Data Base Management Systems and Relational Database Management Systems.

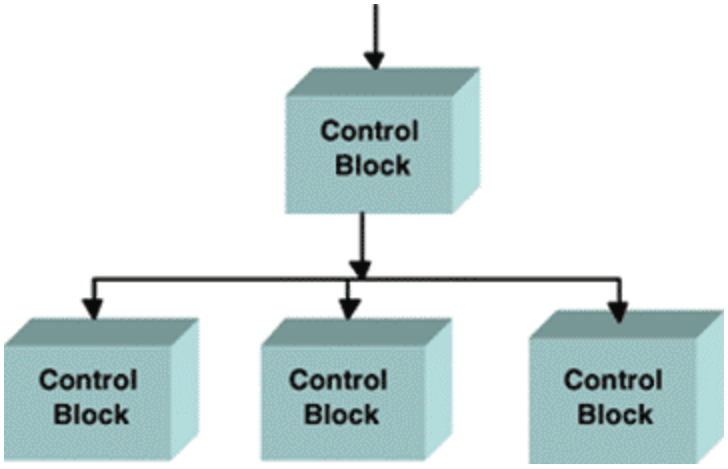
Introduction

Object-Oriented Data Base Management Systems (OODBMS) and Relational Database Management Systems (RDBMS) share a common goal; to manage and protect mission critical data used by applications and individuals. In this regard it appears that any problem that may be solved by one database system may be solved by the other. This conception is not true. There are fundamental technical differences between OODBMS and RDBMS that make certain classes of problems tractable by OODBMS but not by RDBMS. Productivity differences in application development between using an OODBMS or a RDBMS are sourced by these same fundamental technical dissimilarities. OODBMS possess certain features not provided by RDBMS that allow applications to offer benefits that would otherwise not be possible.

OODBMS combine an Object-Oriented metaphor with conventional database management functionality. This combination allows database technology to be used in applications where it was previously not feasible. Historically, applications that were prevented from using database technology managed their own storage of persistent data through flat files.

Performance

RDBMS decompose all data into relational tables. The rules for this decomposition are collectively referred to as the 1st, 2nd, and 3rd normal forms. These rules provide a simple and powerful way to organize data into rows contained within tables. For some problem domains this method of data organization provides quick and convenient access to information. A good example would be a checking account. Individual checks are represented as a row in a table with columns for the check number, date, payee, and the amount. A RDBMS could provide a software application expedient access to this data.



Distributed Control Blocks with 1:M I/O

Another example would be blocks in a Distributed Control System (DCS). Here the RDBMS would setup relational tables to hold information pertinent to a block. Much of the information in this problem domain is hierarchical and may be modeled using 1 to N or N to M sets of relationships. To show these hierarchical relationships in a RDBMS another table(s) would be setup mapping one row from the first table to another row(s) in some other table. As an example consider that the outputs of one block would be mapped to the inputs of other blocks. With a RDBMS a separate table must be inspected at the time of the query to determine which output is routed to which inputs. This operation is an example of selecting a set of rows from a table. If we needed to determine what types of blocks receive input from a

specific output, then multiple tables must be inspected at the time of the query. This operation is referred to as a join, and it is expensive. The expense of a query dramatically increases as more tables are brought into consideration.

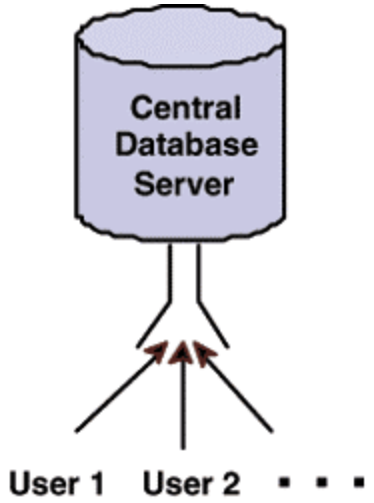
Alternatively, an OODBMS provides support for directly modeling the block and the mappings of block outputs to block inputs. No run-time selection from a table is required to determine which inputs are mapped to a specific output. No run-time joins of multiple tables are required to determine what types of blocks receive input from a specific output.

The OODBMS provide explicit support for 1 to N and N to M relationships. These relationships and the semantic information that they represent becomes an integral part of the data managed by the OODBMS. Not only is the data available to an application, but this semantic information is available with out the expense of a run-time selection or join. A RDBMS does not maintain this semantic information.

The way that an OODBMS stores information on disk is that same as the in memory representation. RDBMS store data in a format on the disk that is different from the in-memory form needed by applications; selects and joins are used to bridge these two formats at run-time. This fundamental dissimilarity causes dramatic differences in run-time performance of applications that deal in problem domains with 1 to N and N to M relationships.

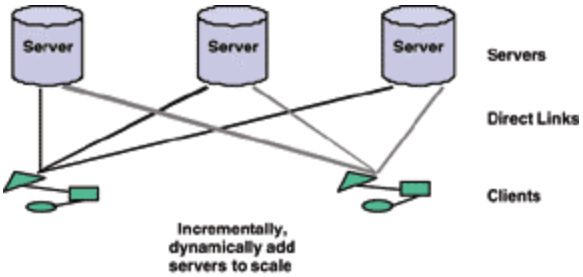
A set of benchmarks run by Sun in 1990 showed that for certain classes of data an OODBMS would out perform a RDBMS by at least one order of magnitude. FRSI applications primarily work with data that fits this pattern.

Scalability



Central Server Bottleneck

Most RDBMS are based on technology using a single server with multiple clients. Some OODBMS follow this pattern, but many are based on peer to peer technology. A peer to peer database system does not have the same potential performance bottleneck that a single server system poses. A peer to peer database system makes extensive use of the compute power and storage capacity on the client's computer.



Scalability by Adding Servers

In a peer to peer system performance problems are often solved by adding more compute power in the form of additional computers. In a client/server system adding more compute nodes makes the bottleneck problem at the server worse; the only option is to increase the server's horsepower.

Peer to peer systems scale better than client server systems. A peer to peer database system may easily and incrementally grow to handle the demands of an expanding application.

A fully distributed OODBMS has the capability to use disk storage any where in the network. This allows much flexibility in adding incremental storage capacity. An additional performance benefit may also be realized by placing persistent objects on the computer's disk where that information is most likely to be used.

Productivity

The data model used by a RDBMS is not natural from a programmer's point of view. A software engineer working in an Object-Oriented language like C++ is use to dealing with data abstractions. These abstractions may be directly modeled by C++ and they bind data and operations on that data together into a single entity. An OODBMS makes C++ data abstractions persistent. From a code view there is almost no difference between a transient instance that lives and dies in RAM verses a long lived instance that is managed by the OODBMS. This means that the persistent entities are made available to the engineer in seamless fashion. The OODBMS is not obtrusive in the application's code.

Alternatively, a RDBMS is very visible to a software engineer working in a C++ environment. Here the engineer is fully cognizant of the fact that two representations of the entities exist, the natural C++ version of an object and it's RDBMS counterpart which may be spread over multiple relational tables. This mapping between the tables in the RDBMS and the natural in memory form of an object must be managed with code provided by the software engineer. There is no alternative here; the mappings must be identified and code must be written to handle the mappings.

The basic types of information stored by a RDBMS are very limited. A RDBMS has support for primitive data such as: [integer + real + text + date]. A RDBMS has no provision for storing complex data abstractions built up through aggregation or inheritance. An OODBMS can directly manage user defined data abstractions based on aggregation or inheritance.

An OODBMS also maintains semantic information that is not available in an RDBMS. Relationships that must be derived in a RDBMS through selection and join operations are explicitly kept in the OODBMS. The operations that may be invoked upon entities managed by an OODBMS are defined by the data abstraction that the engineer designs. The RDBMS has no way to bind behavior to objects. This must be done through convention and represents a large opportunity for introducing errors.

The data model produced by the software engineer during the analysis and design phase of project may be directly and simply mapped to an OODBMS' schema. With a RDBMS, data abstractions must be decomposed into a set of tables that conform to the normal forms of the relational rules.

There's also an impedance mismatch between the way that data is available through a RDBMS verses an OODBMS. An engineer working with a RDBMS works in SQL; that is all access to information managed by the database is framed in the context of SQL. SQL is an excellent way to work with data on an ad hoc basis, but a developer wants the information in terms of the data abstractions that were identified in the analysis and design phase of the project. This is not possible using the SQL access afford by an RDBMS.

A developer working with an OODBMS directly sees the same data abstractions in the database that were identified during analysis and design. No queries are used. This gets back to the fact that objects managed by an OODBMS appear to an application developer in a seamless fashion.

OODBMS directly support reuse through inheritance. This concept is absent from a RDBMS. Commercial grade container libraries and other third party class libraries may be easily stored in OODBMS. This is not possible with a RDBMS.

All of these differences between OODBMS and RDBMS result in a real and sizable productivity gain for developers working with OODBMS.

Fault Tolerance

Conventional RDBMS deal with fault tolerance issues and high data available with hardware based solutions. An example is the use of RAID technology to guard against media failure. These options are also available to an OODBMS.

OODBMS that are based on peer to peer technology are designed without any single point failures. These systems are setup to degrade gracefully. If part of the distributed information store is not available, then the remaining information may still be accessed. These systems are more resilient to catastrophic failures.

Some systems also offer additional protection by replicating important pieces of data across multiple compute nodes. In this situation, failure of a single node does not result in the loss of data that was replicated and therefore available on some other node.

Additional Features and Benefits

OODBMS commonly support certain features that are not available in a RDBMS. These features would be exploited by any FRSI application to offer additional value and benefits to customers.

• Versioning: This feature allows the OODBMS to manage multiple versions of the same object. In its simplest form, linear versioning makes a sequential series of the same object available to an application. Each subsequent version represents the increment change in the object from the previous version to the next. The idea of a "current" version is also supported. These features would be used by many of the entities in FRSI applications. Good candidates for versioning include: [batch recipes + process graphics + alarm configurations + block configurations].

• Schema Evolution: This feature allows the OODBMS to deal with objects as they evolve from one release of FRSI software to the next. Procedures to upgrade an "old" object to a "new" object are handled by the OODBMS.

• Off-Line Data Bases: OODBMS offer the concept of online and off-line databases. To FRSI this means a single application may be written to handle both off-line and online configuration. To our customers this means that they may run their plants with the online data base, but prepare for future plant expansions with off-line databases. The configuration work with the off-line database may actually be performed at a different site. The resulting off-line database containing the "new" configuration for the plant expansion may be transported to the site via network, floppy or laptop. A simple menu selection could merge the off-line database with the customer's online database.

• Long Term Transactions: CAD applications were some the first users of OODBMS. These applications as well as FRSI applications require transaction models that are not supported by conventional RDBMS. A user may require long term access to information managed by the database.

As an example, consider a process graphic that may take days to be finished by a configurer. The database must provide exclusive access to the graphic until the configurer is finished and decides to check the graphic back into the database. Only then would the graphic be available to the rest of users.

OODBMS provide this type of long term transaction in addition to the conventional locking strategy supported by RDBMS.

Summary

We chose Objectivity/DB because:

- Object-oriented (vs. relational)
- High performance
- Good c++ interface
- Reliability
- Good factory support
- Company has a focused direction
- Solid development staff" †

Their customers have database sizes up to several GB, and include pharmaceutical manufactures (e.g., Janssen) and petroleum manufacturers (e.g., Exxon).

† Excerpted from Fisher-Rosemount, Lead Designer customer presentation at a 1997 Objectivity seminar.

Kintetsu
(Transportation, Logistics)

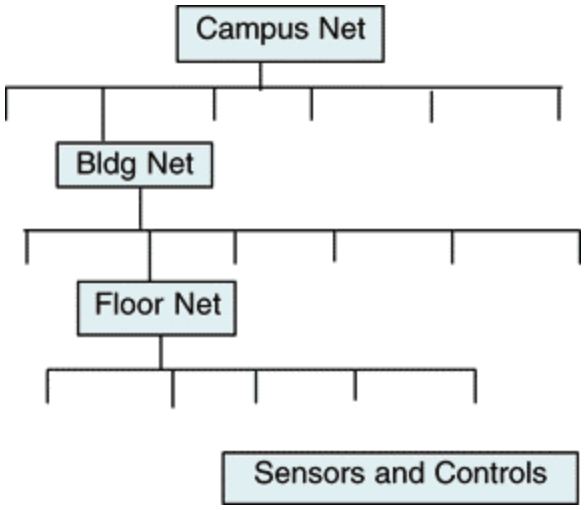
Kintetsu, a $2B company in Japan, models the Japanese railway system using an ODBMS.

Each train is an object, as are platforms, switches, tracks, traffic, etc. The system allows dynamic modeling of changes due to rush hour, holidays, extra load on the electrical supply grid, etc. It's been in production for several years. The developers have reported in published papers that it was developed in only 6 months, a fraction of the time that would have been needed to decompose all the objects into tables. They're now reusing some of their objects for a trucking system.

Siemens
(Process Control)

Landis & Staefa, with worldwide headquarters in Switzerland, is a leading provider of environmental control systems. These are used to control heating, air conditioning, air flowing, power and lighting at the world's busiest airport, Chicago's O'Hare, John Hancock building, the new Chicago Stadium, as well as (high reliability) hospital suites, campus industrial environments, and multi-campus distributed environments, and pharmaceutical manufacturing environments.

Their Insight product, based on Objectivity/DB, is a multimachine, multiuser system used to control and monitor HVAC applications. Graphical monitors show conditions ranging from a temperature variance of a degree up to a fire, with the ability to archive and query history. They store configuration data about the system, and in some cases historical data to be used for later analysis. Key requirements include reliability (e.g., hospitals), flexibility (needs change over time), and scalability (uses grow to very large size in terms of users, number of systems controlled, and information size).



Distributed Control System

The DBMS must be reliable, accurate, flexible, and protect their investment. This last is biggest. After installation, one cannot easily come back later and change a building environment control system. Users need to know that it will always be upgradeable. They can preserve investment in equipment, installation, and training/personnel, which is also important. With an ODBMS, when they add on, they can extend the system — it won't be out of date. They found these to be well supported by the ODBMS.

Other technical DBMS requirements include performance, ease of modification, complex data types, support C++ and NT, multi-platform, shorten development cycle from 32 to 18 mo. Objectivity/DB again met all these. The ODBMS helped with the last requirement (development time) because it is so close to C++ that time to come up to speed was very small, and it has a good toolset.

Architecture of their system is inherently distributed, as a trunk with multiple control stations (using the ODBMS) and proprietary network link to actual controllers. Distribution can be local or WAN, some several miles, some several hundreds of miles, the longest about 1000 miles. Objectivity/DB's support of a single logical view over such distributed environments, with fault tolerance and replication were key advantages.

Other evaluated DBMSs included other ODBMSs and a leading RDBMS with which they were already familiar. The others lacked distribution support, and missed functionality, including time series support as well as immediate-change support. In all cases it was required that the DBMS be open for customer access and extensibility. The SQL/ODBC support here was critical.

SEH
(Semiconductor Manufacturing)

SEH, the world's largest silicon wafer manufacturing company, has been using an ODBMS-based system for process control of all phases of wafer fabrication since 1996. Vector Consulting helped build the system, which now automates and optimizes their operations.

Their requirements that led to the choice of the ODBMS include:

- A database that could manage very complex associations between data, many to many
- Performance had to be very good to manage traversal requirements in the database
- 24 x 7 availability was also a requirement, including online backup and restore
- SQL interface for reporting

NEC
(Semiconductor Manufacturing)

NEC, the second largest semiconductor manufacturer in the world, has deployed its integrated manufacturing (CIM) system, the NEC Object-Oriented and Open (O3) system, based on an ODBMS. Currently used to manage a distributed environment including 16 factories worldwide, O3 represents NEC's migration from the mainframe to open systems, skipping the now declining RDBMS generation and leaping ahead to ODBMS.

According to Shuji Imura, General Manager of NEC's Semiconductor Group Information Systems Center:

"The O3 System is a triumph in implementing object oriented technology in a business critical application. Objectivity/DB has enabled the O3 system to increase our productivity and accelerate time-to-market, giving NEC a significant competitive advantage and increased market share."

American Meter
(Process Control)

American Meter developed a system for brokering and distributing electrical power and gas in real time. With the recent deregulation of these industries, their system has led the way. Originally, they spent two years developing the application with a leading RDBMS, but in beta test they discovered that it was far too slow for production use. Fearing product failure, they decided to try converting to an ODBMS, Objectivity/DB. In three months they were in production, exceeding performance targets. Even better, in the process of conversion, they were able to discard one third of the application code, which had been needed for mapping to the RDBMS. This project won a finalist award at the Object World, San Jose, 1996, shortly after it shipped.

Sloan Digital Sky Survey
(Geographic Information System)

The Sloan Digital Sky Survey (SDSS) is a $32M project run by a consortium, led by Johns Hopkins Space Telescope Science Institute and Chicago's FermiLab, to produce a new, digital survey of all the objects (stars, galaxies, quasars, etc.) in the sky. The previous survey, now over 50 years old, consists of analog glass photographic plates, which dramatically limit distribution, amount of information, and accuracy of information. SDSS has constructed a new 2.5m telescope, based directly on digital (CCD, 4Kx2K chips) imaging, which will gather information including luminosity, spectra and spectral intensities in various bands, and variability, for all objects in space. It is expected to take 5 years to map the northern celestial sphere. Raw information will go into the database, as well as analysis results that collect that information into identified objects, resulting in a catalogue of 100M objects. The database will be accessible to scholars, and parts of it even to schools and the public, via the Internet. Production software entered test usage in 1996, and the final database is expected to be approximately 40 TB, in Objectivity/DB.

The key to use of this enormous archive is supporting multidimensional indexing for queries not only in 3D space, but also in luminosity in several spectral ranges. To make this efficient, SDSS developed a new indexing algorithm, based on modified quad-trees, for n-dimensions. This index partitions all 100M objects into some 40K containers, a clustering device in Objectivity/DB. Coarse-grained maps of the multidimensional space then model the contents of these

containers. Queries, then, quickly index over this coarse grain map to quickly determine which of the containers are needed. This efficiently reduces the search space, and hence query execution time, dramatically. Measured results show it to be 100x faster than traditional databases.

Fermi and Johns Hopkins evaluated other DBMSs, including RDBMSs and ODBMSs, and chose Objectivity/DB for these reasons:

- Objectivity's performance in large-scale applications
- Objectivity's support for platform heterogeneity (they can spread the very large database across multiple hardware from multiple vendors)
- Objectivity provides features (e.g. caching) that Fermi would otherwise have had to code themselves in their application

For more information, see

http://www-sdss.fnal.gov:8000/intro

http://tarkus.pha.jhu.edu/scienceArchive/

LMS
(Healthcare, Fetal Monitoring)

LMS has developed and deployed their Computer Assisted Labor Monitoring (CALM) system for obstetrical care units. It provided real-time, multi-access (bedside, nurse's station, doctor's office, etc.) to comprehensive patient monitoring information, management and analysis of that information, and decision support capabilities. The target is better care including reduced cesareans, fewer critical situations, better handling of those situations, with resulting decrease in injury to mother and child. Their key requirements as outlined by LMS are:

"System Reliability: The CALM system provides obstetricians and nurses with comprehensive patient monitoring, information management and decision support capabilities by more precisely identifying fetal distress events. The critical information is stored in the database. Clinicians requirement to have the data at their fingertip and always available is paramount.

Information Quality: The diagnostic information that the CALM system collects must have high integrity. A corruption in the information could be catastrophic. A clinician administers treatment to the patient based on the information collected in the CALM system. This is why information must be accurate.

Quick Information Access: Clinicians must be able to recall historical information on the patient and quickly react in emergency situations. Also, fast access to statistical information is critical for patient diagnosis."

LMS chose Objectivity/DB, an ODBMS, for the following reasons:

"Code: Unlike others, Objectivity did not appear to require as much training and performance tuning. LMS also liked the guaranteed integrity of objects by using bi-directional object references, as well as safe, transparent indirections (handles) on all references. LMS also liked the transparency of the code.

Caching Mechanism: The ability to configure the size of the cache.

Partnership: LMS felt more comfortable with Objectivity's reliability and performance than that of our competition. These are areas that could not be thoroughly tested. However, in speaking with customers who have deployed applications, they were impressed with the high satisfaction with product and level of support."

Synthesis
(Healthcare, Record Management)

The Synthesis application is Electronic Medical Record (EMRx). It is a repository for medical documents with a powerful search and retrieval capability based upon keywords, synonym, and word proximity. The target market is health care initially, but the technology is general enough to be applied in many different domains.

The following two critical capabilities were key in determining the choice of ODBMS:

Scalability: Maintaining adequate performance under high volumes of data was one of the most critical factors in determining project success and acceptance as a viable solution in the marketplace.

Flexibility: EMRx's sophisticated search engine required storage of complicated data structure. Retrieval and usage of this data in a high performance manner was critical to project success.

Other important criteria influenced the choice of ODBMS over (O)RDBMS, and the choice of particular ODBMS:

Distributed Architecture: Objectivity's distributed architecture was seen as the only way to meet their scalability and performance requirements. Other architectures forced a central server bottleneck that limited the system's scalability. The transparency of the distributed architecture to the application was another big plus.

Container Flexibility: Application-defined locking and clustering through the use of containers gave their designers much flexibility in implementation options and performance trade-offs based on their problem domain.

Scopeable Indexes: The ability to create and use scoped indexes was very important to maintaining high performance. Higher level algorithms could narrow searches down to a portion of the database, and then use scoped indexes to finish the query. Other databases only supported global indexes, which became too large and slow to use effectively (indexes of all documents in their repository vs. documents that were of a c ertain category).

Daikin
(Multi-Media CD-I Authoring)

Daikin Industries, Ltd., is the world leading supplier of a complete range of authoring and premastering tools for DVD-Video and DVD-ROM software production. Daikin's U.S. Comtec Laboratories embeds Objectivity/DB in its SCENARIST™ software authoring system. Scenarist, which allows users to develop titles which are fully-compliant with the DVD-Video specification, is used by over 350 facilities worldwide including Sony Corporation and other multimedia service bureaus, game vendors, post-production companies for television programming and movie production. The Scenarist software enables customers to rapidly bring to market reliable DVD titles compatible with any available DVD player.

Distributed Architecture: Objectivity's distributed architecture provides Daikin customers performance and scalability in a distributed environment where the Scenarist user must complete multiple DVD projects simultaneously in a highly productive, distributed environment.

Platform Heterogeneity: Scenarist is available on both the SGI and NT platforms. Because Objectivity/DB runs heterogeneously on more than 16 hardware platforms, Objectivity allows Daikin to meet the demands of the high-end Scenarist customers running large volumes of complex data on both the NT and SGI platforms.

Object Database Transparency: Software authors using Scenarist share assets with Daikin's Multimedia component server environment called Comtec Future Studio which is connected with computer graphics, digital video, audio stations, MPEG decompression/compression mechanism, all of which are then connected to a CD printer or, via gateway, to a remote studio. The embedded Objectivity/DB provides cross-platform object-oriented database capabilities *seamlessly* _ without any obstruction to the multimedia and printing requirements (both local and remote) of Scenarist. In fact, Objectivity/DB significantly increases performance and seamless workflow productivity.

How to Avoid the Wall

As we have seen in theory, in benchmarks, and in real production applications, the relational wall is real. It occurs dramatically in performance, with execution time rising rapidly, in fact exponentially, with increasing complexity, quickly reaching 100x slower than ODBMS, and beyond. The wall also exists in dramatically increased complexity, causing increased software development and maintenance cost, and in dramatically reduced flexibility and extensibility.

This wall can appear suddenly and unexpectedly when users take prototy pe applications and attempt to deploy them. The resulting growth in information, users, and complexity can bring them face-to-face with the wall. The result is typically failure.

To avoid such failure, users must anticipate where the relational wall will appear, and prepare for it. As long as they hold complexity of information down to very simple, fixed-length, table-structured data, with short, simple, localized operations, central server storage, and limited growth in size and numbers of users, they are safe within the RDBMS world. Exceeding any of these limits will bring them, soon or later, to the relational wall.

The key is to use the right tool for each job. Look for the following telltale signs that the relational wall may be closer than you think:

1. Use of **Objects** (C++, Java, Smalltalk, etc.), which requires mapping to RDBMS tables. Such mapping code may grow to 1/3 to 1/2 of the application, increasing development cost. Worse, with each design change, and maintenance change, those changes are amplified by the mapping code and make maintenance dramatically more expensive, in some cases prohibitively so. While the RDBMS requires you to continually work at the flat, primitive level, the ODBMS allows you to work at any level, up to and including the highest, application level.

2. More than 2 or 3 levels of **Joins**. This is a red flag, since joins are so slow in an RDBMS, and orders of magnitude faster with direct relationships in the ODBMS. The more joins (relationships), the more you'll benefit from an ODBMS.

3. **Many-to-Many Relationships**. The RDBMS model lacks support for any relationships. Simple one-to-one, and to a lesser extent, one-to-many relationships may be modeled, at the cost of extra application code to maintain them, and loss of performance. Many-to-many relationships typically become too complex to be usefully or practically modeled in an RDBMS.

4. **Hierarchy** traversal. If you application includes any hierarchy or multi-rooted network creation, update, or traversal, RDBMS performance will be extremely slow, and ODBMS exponentially faster.

5. **Varying Sized Data** structures. Structures such as time-series and history information, or any other structures that dynamically varying in size, are poorly modeled by fixed size tables. In an ODBMS they're directly supported, making development and maintenance far easier, and performance much higher.

6. **Nested or Complex** data structures. The RDBMS requires and supports only simple, flat, rectangular tables. Any other shape of structure, any complexity, and any nesting will require complex modeling tricks in the RDBMS, making implementation difficult, error-prone, and slow. In and ODBMS, these are supported directly.

7. **Distributed** (non-centralized) **Deployment** environment. The RDBMS borrows its architecture from the mainframe, storing data and performing all operations on a central computer (server). If your environment includes multiples PCs, workstations, servers, the ODBMS will prove far superior by providing a single logical view over all databases on all servers.

8. **Heterogeneous Computers**, operating systems, languages, and networks. RDBMSs typically run in one environment only and, at best, provide only remote thin-client access from different architectures. With the ODBMS, objects may reside and execute transparently on any of the common architectures, allowing you to leverage existing investment in hardware infrastructure and training, to incrementally add new technology equipment along side old, and to support all levels of users, from C++ gurus to end-users clicking buttons on PowerBuilder or Visual Basic forms, all sharing the same objects in the same databases on the same shared servers.

9. **Flexibility and Extensibility**, in both physical and logical design. The RDBMS offers only one physical structure, centralized tables, and logical structures are limited to similar, flat table subsets (views). The ODBMS allows any logical structures, dynamically mapped to the desired physical structures, in multiple databases, with multiple levels of clustering, across multiple types. All of this is hidden from users and applications by the single logical view, so that dynamic, online changes to the physical layout, including moving objects and databases, reclustering, adjusting locking granularity, etc. Similarly, logical changes may also be made online, via schema evolution and instance migration, all new objects and subtypes of old objects to be added, old objects to be modified, with changes made automatically to online production

databases. Features such as versioning can model history and alternatives. Object encapsulation allows fixing one object without breaking the rest of the system. Inheritance allows reusing old objects, even when they don't do exactly what you need, by specifying only the difference.

10. **Integrity and Scalability**. RDBMSs support integrity only at the primitive, flat, tabular level. All data structures above that are left to the application, and integrity is at the whim of each of the applications. With ODBC GUI tools, integrity is at the whim of the end users, that are also forced to work at the primitive, flat level. The ODBMS supports integrity at all levels, with object operations automatically maintaining integrity rules, and enforcing them across all applications. Even ODBC GUI tools, used with an ODBMS, support access to such high level objects and their operations, and the ODBMS can limit end users and groups of users to only certain objects and operations. Scalability in an RDBMS is limited by the single, central-server bottleneck, and by the capacity of the server machine, which must support all users. The distributed ODBMS architecture avoids any single bottleneck, allowing scaling in both clients and servers, adding users without slowing down others, incrementally adding commodity, inexpensive servers to improve performance as needed, supporting unlimited numbers of users, and up to millions of tera-objects, which may be up to many gigabytes.

The final consideration is one of risk. Some, because they are familiar with the older RDBMS companies and products, might deem them to be lower risk. If, however, you're application has any of the above characteristics, or might grow to include any of them, then the risk with the RDBMS is major problems, even failure, in production, while the ODBMS can avoid that. The risk is not in the DBMS product you buy, assuming you buy one that has referencable production users (such as those cited above). Rather, the risk is in the application you develop. If you use objects with an RDBMS, that risk is magnified by the need to add a layer mapping from objects to tables, by the maintenance cost of that layer, by leaving the application-level structures and integrity maintenance to individual applications. With an ODBMS, no mapping layer is required, and the ODBMS subsumes the common high-level model and integrity maintenance across all applications.

Dr. Andrew E. Wade, Ph.D., a founder of Objectivity, Inc., as well as co-founder of ODMG and author of ODMG-93 and many articles.