Introduction: This paper describes building a high performance data repository system with Objectivity/DB. Specifically, this paper will address how to provide extremely high throughput (high ingest and delete rates) while simultaneously allowing many users to query the data and quickly receive results.

Overview: Since 1996, Objectivity/DB has been the platform for the largest publicly acknowledged database in the world at the Stanford Linear Accelerator Center in Palo Alto, California. The Babar system currently employs about 2000 CPUs in 100 servers to ingest and process approximately a Terabyte of data a day and as of December, 2003 has grown to over 800 Terabytes in online and near-online storage.

In June of 2003, Objectivity participated in a benchmarking activity with Silicon Graphics in which a sample application was proven to ingest well over a Terabyte of data an hour using a single Origin server with 32 processors. A separate 32-processor Origin server was used to simultaneously query the database using over 200 concurrent threads without any degrada-
tion to the ingest performance.

The basic architecture of this system is shown in Figure 1, except that the query processors are shown as separate workstations. The details of this sample application and its specific requirements cannot be made public, since the benchmark was performed under non-disclosure agreements. We can, however, describe quantitatively how to design an Objectivity/DB system to achieve results that have been considered either out of reach or prohibitively expensive to attain.

The Problem: The largest data management system vendors provide a "one size fits all" solution to database management based on relational database theory. Over the last 15 years, there have been many arguments about the advantages and disadvantages of the relational model and these arguments will continue for many more years. The biggest proponents of the relational model say vehemently that the relational model has only been partially implemented by the largest vendors. Criticism of "relational" databases gets confused with criticism of the relational model.

This paper will not attempt to argue the merits of the relational model, except in how it has been generally implemented. "Relational" databases became very popular in the 1970's, not just because they were based on the relational model, but also because they supported the SQL interface.

Although SQL is a very convenient language for users to access data and has become a standard query language, it's actually an extremely limited language for software developers and leads database management systems and their users into very bad constraints.

Ironically, very few large applications built on top of SQL databases still offer SQL interfaces for their end-users. Instead, end-users interact with the database through an application user interface. If SQL is no longer the interface that the end-users see, the query language just becomes a development and administration choice.

Consequently, instead of referring to these DBMS products as "relational", which is a theoretical model, this paper will refer to these products as “one
size fits all” or “SQL” databases. The solutions that these products provide are optimized for the most lucrative applications of the 1970’s. These applications generally have the following characteristics:

**Relatively static data** – E.g., products and customers are not created in large quantities every day. Once the database is built, the amount of data added on an hourly or daily basis is relatively small.

**Relatively simple data** – The data fits well into tables and there are few many-to-many or recursive relationships like tree or network structures. Data is organized into tables by type.

Binary “unstructured” data is almost always stored external to the database or as “BLOB”s. Data with complex or dynamic structure is simply considered unstructured.

**“Manageable” cardinalities** – E.g. hundreds of products, thousands of customers, less than 100’s of gigabytes in total.

**Query by attribute** – Data is fetched based on values, rather than following relationships between one object (or row) and another. Relationships are represented using special values (primary and foreign keys), but there are few “deep” queries that require traversal through many relationships.

**Small, short transactions** – Each transaction changes a small amount of data, e.g. a customer id, product id(s), payment and date, and is committed to the database quickly.

**Centralized processing** – Updates and queries are done through a server, either online or in batches.

**Highly administered** – Database administrators are available to manage the data behind the scenes.

Applications that have these characteristics fit well into the SQL database
management model, but many applications, especially modern ones, have more complex requirements. For these applications, one cannot configure the RDBMS solutions to meet the requirements without making severe compromises and sacrifices.

Consequently, rather than choosing the appropriate database management solution, requirements of the applications are often limited to what the database and the available hardware will support. If the database can’t be changed, the requirements of the application must. For many applications this is undesirable, but for some it is impossible.

This is not because the RDBMS products are inferior, but rather that they’re an incorrect tool for the problem, just like using a hammer is a terrible way to drive a screw and vice versa. Much of this “impedance mismatch” comes directly from organizing data into tables, but much of it comes indirectly from the constraints that tables and SQL put on the RDBMS architecture.

Some requirements that preclude the use of a “one size fits all” solution are:

**High ingest rates:** Inserting new records into most data management systems is usually very slow. It’s often multiple orders of magnitude slower than writing the data directly to disk and very difficult to make completely parallel because of central server architecture.

This is a direct result of organizing data into tables. Since data of the same type ingested from different sources must be organized into the same table, there is a natural bottleneck. Although tables can often be physically distributed, any query on a particular table must be able to access all the data in that table or at least an index to all the data in that table.

When indexes are applied to the table the bottleneck becomes even tighter, since indexes are very difficult to maintain with multiple parallel updaters.

Consider an application whose single data type is a message, but must ingest thousands of messages from thousands of sources simultaneously. Forcing all messages into a single “Message” table provides no value. Instead, it makes much more sense to group the messages, both logically
and physically, by their source and/or the time each message was received.

**High delete/archival rates:** Applications that are constantly streaming in new data often have a need to archive older data and keep newer data online.

For most SQL databases this requires a query on all tables by time and a complex mechanism to copy the data (atomically) to another storage repository and then delete it from the database.

The overhead in terms of processing and I/O involved in this operation is very large. Reversing the process and restoring archived data into the original repository can be equally difficult.

Again, this is a direct result of organizing data into tables. It may make much more sense to reorganize the data from several tables by time, rather than by type, especially if most of the user queries to fetch the data will have a time constraint as well.

**Complex relationships:** SQL is a very rich language for finding objects by attribute using a single statement. An example query by attribute might by “SELECT FROM CUSTOMERS WHERE LAST_NAME = ‘Jones’”. Query engines and b-tree indexes can make queries like this perform very well. Unfortunately, querying by attribute is generally the only mechanism offered by SQL databases to identify and fetch objects. If an application needs to perform multiple queries based on relationships, fetching objects by attribute becomes excruciatingly slow and cumbersome. An example query by traversal would be to find all the orders that a particular customer has placed.
In a software application, a Customer object would simply contain pointers or references to a group of Order objects. But a Customer table in a SQL database can’t have a variable number of columns. Instead, identifiers (called primary keys) are added to the Customer table to uniquely identify the customers. Foreign keys are added to Order table that reference these rows.

One can then join the two by querying by attribute (“SELECT FROM CUSTOMERS,ORDERS WHERE ORDERS.CUSTOMER_ID = CUSTOMERS.CUSTOMER_ID”).

This is such a common and well-known practice that it can be taken for granted. In fact, joins like this are quite expensive. First, it requires an extra column in both the CUSTOMERS and ORDERS table. Also, to perform properly, they generally require that the primary key column be indexed, which can be a very high cost in terms of storage, memory, concurrency and insert performance.

The cost of joins becomes even more expensive when one considers many-to-many associations, like orders can include many products and products can be included in many orders.

In a SQL database, these require a whole separate table (called a “join” or “junction” table) to represent the relationships. These intermediate tables become much larger than either of the two tables at the ends of the associa-
Not only do the primary keys of the two original tables require indexes, an index is required on at least one column, if not both columns, of the join or junction table.

Now consider a network of objects or elements, a network being a tree where each child can have multiple parents. Examples of a network of elements might be a road network or a network of inter-related messages. These kinds of complex structures are generally implemented in “connectivity” tables. These tables are like join tables, but even more inefficient, inadequate and unnecessary.

**Flexibility:** Modern software applications require much more flexibility than applications have in the past. In particular, data types may change to reflect an ever-changing application domain, or to reflect an ever-improving understanding of that domain. Similarly, the computing environment may change to take advantage of newer, more cost-effective hardware. The leading database management products fail to meet the new flexibility demands in several ways.

Schema changes and the associated data migration are difficult, especially for large amounts of data. Upgrading hardware is difficult as well and often requires that the original platforms must be replaced – not added to. Re-hosting a SQL database can be expensive and time consuming.

Both schema changes and hardware upgrades can require either lengthy
Performance Optimization: Part of the reason that schema changes are difficult in a SQL database is because the schema design greatly affects performance.

Standard SQL database design dictates that an architect first normalizes a database schema to maximize storage efficiency, and then de-normalizes the areas that require it to optimize performance. This creates an undesirable trade-off of performance versus efficiency, but it also imposes a trade-off of either performance or efficiency versus complexity of the object model that the schema is going to support.

Despite the array of tools and methods apparently available to a database administrator for performance optimization, they tend to just be new structures (views, indexes, stored procedures, etc.) layered on top of the tabular structure.

The best ways to improve performance are eliminating unnecessary work, distributing necessary work across the most appropriate resources (e.g. using memory effectively) and maximizing parallelism.

Traditional database optimizations are really very limited when trying to approach the real capabilities of the hardware, don't address the largest areas of resource consumption and often create new problems of their own.

Very Large Cardinalities: When the quantity of rows in tables grows to very large numbers, the numbers of rows in the join tables get even larger. “Modern” databases still have trouble with tables that approach even a billion records.

As cardinalities increase, fitting complex applications into the SQL model of joins and indexes becomes inefficient, uneconomical and ultimately unfeasible. Memory consuming indexes become numerous and very large. The amount of disk overhead required (known as data explosion) can grow to four times the size of the original data.

Many of these problems can be managed with a combination of limiting the system requirements and purchasing resources (faster servers, more disk
and more memory). Not only is this expensive, it also doesn’t really solve the problem, because the problems will come become even more serious as the application wants to scale either in cardinalities or complexity.

A Solution: In designing a data storage solution and looking at products to help, it makes sense to start with first principles. After all, different applications in different environments with different requirements usually need different solutions.

So what capabilities do we need in a database management system? SQL databases tend to emphasize querying the database, which is not surprising, since SQL is a query language. In fact, many books on databases suggest that the role of a database is almost exclusively to provide a query engine. However, query is often not the driving reason to adopt a particular database management system. You may need a database management system to provide many other features that are very difficult to implement well in an application without a database.

Let's look at all the basic functions one expects from a database management system:

- Persistence
- Retrieval / Query
- Cache
- Concurrency
- Schema Control and Data Migration
- Transactions (Commit / Rollback)
- Single Consistent View of Data

Each of the following sections describes one of these basic DBMS functions, and how Objectivity/DB handles it.

Persistence: Persistence provides three main benefits. First, an application can stop and restart and retrieve the data that it needs to function. Second, an application can work with larger data sets than will fit in memory. Third, an application can share its data with other processes. This has traditionally meant the transfer of data from volatile memory to disk.
Some of today’s applications have little need for persistence, which is why there are “inmemory” database management systems. These applications generally have a relatively small amount of data, but need to be able to update and/or access it very quickly. Most DBMS applications, however, require some form of saving data to disk.

This section will discuss several ways persistence can be implemented, and then describe how Objectivity/DB persists data at length and in detail, since persistence has a large impact on how all the other database management features are implemented.

Let’s first think of how we would implement persistence if we didn’t have a database. One obvious way to save the data would be to simply write it to a file. As the Java language became popular, and with it the concept of object “serialization”, a concept developed that each object could simply write itself to disk and read itself back. Nothing seems simpler or more efficient.

There are two problems with this simplistic approach. The first is that the objects need to be found again, which requires some kind of query or navigational mechanism. This will be addressed in the “Retrieval / Query” section.

The second problem is that objects in memory often have references to other objects, which will become invalid as soon as the memory is released.

**The Hierarchical Solution:** Early hierarchical databases would maintain references between objects (or data structures) by storing the actual physical location of the referenced object. This was convenient because you could stream a large amount of data from memory to disk and simply replace the memory pointers with the physical pointers.

This solves the problem of volatility in the address, but exposes a new one: The physical location of the referenced object may have to change if the object changes. If the physical location changes, the references that use that physical location have to change as well.

In a hierarchical database, this means that when an object’s physical
location changes, some process must find all the references and modify them. The penalty for physically moving an object makes working with a hierarchical database extremely awkward for both the database designers and for the end users, except for very constrained applications.

The Relational Solution: Relational database theory promised to end the problem of moving data by eliminating physical dependence. Instead, logical “keys” maintain relationships between objects, completely independent of physical location. Furthermore, to keep the size of each object (row) more static, objects that have a relationship to many other objects do not keep a list of those keys; instead each referenced object will keep the key of its referencer as described earlier in the “Customer” and “Order” relationship.

The indirect primary and foreign key strategy won’t be as fast as a direct physical pointer to the location of the orders, but by putting all the customer IDs in an index so that one can perform a binary search, the performance will not suffer much compared to the benefit of physical independence.

This is a good general solution, but it does present a trade-off of convenience against performance and scalability. As described previously, indexes and “joins” are not free. In general, the biggest problem is that the indexes become too large to manage and too slow to update as the amount of data and users grows.

As the numbers of rows in the tables (e.g. orders and customers) grow, it takes longer to look up the orders for a particular customer because the index is getting bigger. As the indexes get larger, they have trouble fitting into memory. The database ends up consuming large amounts of memory to keep performance reasonable.

An Object-oriented approach to persistence: Objectivity/DB takes an object-oriented approach, which preserves the advantages of the previous solutions and avoids the disadvantages. That is, Objectivity/DB allows objects to keep lists of the objects they reference. However, the referenced objects are identified logically, and not with physical addresses.

The problem with maintaining physical addresses is really the same problem
that software developers have with memory management. For years, software developers have solved the problems by replacing pointers with handles.

Handles are pointers to pointers that uniquely identify an object, but the layer of indirection means that the handles don’t directly point to a location in memory (or in Objectivity/DB’s case – on disk). This allows software to re-allocate memory without modifying the object ID, so the address can be changed in a single place instead of modifying all the references to the object.

Because each element has its own list of elements it’s connected to, traversing from one element to another is fast, lightweight and relatively painless with no need for extra memory-consuming indexes that are slow to update.

Managing concurrency while updating an object’s list of references is much simpler as well, because only the updated object needs to be locked. There is no need for locking an entire segment of an index, so Objectivity/DB can use pessimistic locking (described further in the section titled Concurrency).

**A closer look at object identifiers:** Objectivity/DB’s “persistent handles” are called Object Identifiers, or OIDs. Each OID is assigned when a new object is first placed into the database. In C++, this is almost always when the object is first created. In Java, this would be when the object is “clustered”.

![64 Bit OID (Object ID)](image)

Each OID consists of four 16-bit fields, where each field represents a level in the logical organization of objects in Objectivity/DB. This logical organization
consists of a hierarchy of storage structures. An “Objectivity/DB database” is actually a federation of databases, where each database is normally a single file in which objects are stored. Each database is subdivided into containers, and each container is subdivided into pages. A page is Objectivity/DB’s unit of transfer from disk to application memory. Individual objects are stored in slots on a page.

The first field in the OID represents the database ID, the second represents the container within the database, the third the logical page within the container, and the fourth is a logical slot within the page. Once the OID is assigned, it’s permanent for the life of the object, regardless of how often its attributes change.

An Objectivity/DB container is a group of objects within a database. Since a container can have 16-bits worth of pages, a container can hold 65,536 pages worth of objects (minus a few for Objectivity/DB’s internal use). Depending on the page size, a container can hold almost 4 gigabytes of objects.

Objectivity/DB containers provide to a database what shelves provide to a closet and what directories provide to a file system. They’re an extremely flexible way to organize information as objects.

At the beginning of each container, Objectivity/DB stores a page map, represented in Figure 5. The page map is the first layer of indirection that allows Objectivity/DB to avoid the rigidity of hierarchical databases by assigning the logical page (the page ID in each OID) to a physical page.

When a page is updated, Objectivity/DB can simply write the new page to disk, rather than overwriting the old one. When a user “commits” a change to the object, Objectivity simply changes the page map to point the constant logical page to the new physical page. This basic architecture has many advantages that we’ll discuss further in the following sections.
Query/Retrieval: Although query and retrieval are distinct concepts, they are related enough to be discussed together. In a SQL database, these concepts tend to blend together, because a SQL statement doesn’t differentiate between fetching a known object and finding an object by attribute.

Query by Attribute: Objectivity/DB has mechanisms similar to SQL for finding objects based on attribute. One can call “scan” functions that use predicates to find objects that fit specified criteria. The predicate language is similar to SQL and other query languages and the Objectivity/DB kernel will maintain user-definable b-tree indexes to optimize performance of these scans.

A fundamental difference from SQL, however, is that Objectivity/DB allows you to scope your search to a particular database or container, like looking for a file within a folder instead of looking through the entire disk.

You can narrow down the search because Objectivity/DB allows you to group objects logically rather than requiring that you group them by type in tables. Many objects have a principal organizing attribute beyond what kind
of object it is. A bolt in a piece of machinery would likely be organized by its component. A company document would be organized by its project. A message would be organized by the date and time it was created.

Query by Traversal: A second kind of object retrieval is to traverse from one object to a related object. This kind of retrieval is really just a fetch from the database. Finding objects by traversal doesn't require an actual search, because they're readily identifiable by their unique IDs, in this case Objectivity/DB OIDs.

SQL databases have no concept of a fetch by ID; any row must be queried by attribute. Objectivity/DB can retrieve an object by its OID, without any knowledge of its type, just by resolving the 64-bit address.

Cache: Objectivity/DB provides a client-side cache for its Java, C++ and Smalltalk applications. Having the cache on the client side has the following advantages:

1. Allows more flexibility – each client can have one or more caches for its own data
2. Faster – avoids network slowdown when using cached objects
3. Reduces traffic on the network

Server-side cache is less important for Objectivity/DB than for other architectures, since Objectivity/DB is less dependent on large indexes that need to be kept in cache.

Figure 6 shows how pages are used both persistently in containers and in the memory cache of an application accessing the federation.
An application first makes a connection to the federation and then creates at least one session. Each session has a memory cache associated with it whose size (initial and maximum number of pages) is set from an application parameter.

When a persistent object is accessed in an Objectivity/DB application, Objectivity/DB’s kernel (provided as a shared library linked to the client application) will check to see if the object’s page is already in the session’s cache. If it is, Objectivity/DB simply provides the pointer in memory to the object and the operation is accomplished in a matter of a millisecond or less.

If the object’s page is not already in the cache, the object’s entire page is brought into memory. Since Objectivity/DB allows clustering objects of different types together on a page, it’s very likely that related objects – ones likely to be needed by the application - will be fetched with the same read.

Concurrency: One of the best features of third-party database management systems is their ability to help users and applications work with commonly accessible data while providing predictable behavior to all clients.

Concurrency is usually managed by “locking” data and by using transactions to define when the locks are released. The transactions almost always must adhere to the ACID properties.

In order to support the ACID properties, standard locking models include at least two obvious types of locks – read and update. Read locks can allow other readers to read data, but must restrict updaters from changing the data as they are reading it. Update locks must exclude access to the data that the updater is modifying.

Despite the seeming simplicity of the concept, there are many ways to implement concurrency that can have a dramatic effect on performance and confidence in the data.

1 Most disk drive controllers will read at least as much from the disk anyway, so there’s no performance advantage in reading single objects.

2 See the section “Objectivity Terminology” for a description of the ACID properties of transactions.
One of the important implementation details is when the data is locked and how conflicts are resolved. Another important implementation detail is the granularity of the locks. It can be very important to know (if not control) how much data is being locked when an attribute is being read or modified. Here again, physically organizing data by type can create severe inefficiencies. Almost all database management systems have coupled locking with the physical layouts of data for efficiency. This introduces another trade-off of poor choices.

If page-level locking is implemented, rows that have no relationship with each other than a common type will be locked together. For example, locking one customer record will effectively lock many customers. Not only does this prevent access to the other customer records, knowing which customers are locked together is impossible because users and applications generally have no control of which customers are on the same page.

This unpredictability can be disastrous not only because one must handle cases where locks aren’t granted, but also because it can lead to deadlocks. The other choice is row-level locking, but this has obvious performance ramifications. If an application is going to read or update hundreds of thousands of rows in a second, it can mean that the same number of locks must be evaluated per second. Since the lock manager is by nature centralized, it also becomes a bottleneck.

The one-size-fits-all solutions generally deal with this trade-off with two strategies: Optimistic locking and short transactions. Although the release of an update lock must occur at or after the end of a transaction to maintain atomicity and isolation, when the update lock is initiated is much more open to implementation choice.

Optimistic locking assumes that the likelihood of a transaction reading a piece of data and that piece of data being updated at the same time is very small. Optimistic locking allows for the rows to be read without locking other updaters out for the length of the transaction. This strategy depends on short
transactions, since the shorter a transaction is, the more likely this assumption will be true.

There’s a severe danger with this approach, however. Although the database will prevent “dirty” reads (reads of partially modified data), this does allow a user or application to violate the spirit, if not the letter, of the “Isolation” and “Durability” parts of the ACID properties by allowing updates on “stale” reads.

A common example is an employee (say Mary) who is given a $5 an hour raise at around the same time the whole company is given a 10% raise. Let’s say that transaction A (entered by one HR person) is responsible for the $5 raise and transaction B (entered by another HR person) is responsible for the 10% raise.

If transaction B reads Mary’s current salary after transaction A reads it, but before transaction A commits the $5 an hour change, transaction B will be using stale, obsolete data as the basis for the update. Not only will Mary lose the 10% increase on the extra $5, she might lose the $5 raise entirely, because transaction B may overwrite transaction A’s changes. This is known as “last one in wins” case and can cause serious problems.

The typical cure for this is a “SELECT FOR UPDATE” SQL statement. This causes pessimistic locking, where the read locks are obtained in advance of updating. Another cure for this is to set up a stored procedure that timestamps data so that it can be checked for staleness the time it’s committed. If the data is stale, the transaction is aborted.

If the database is supporting row level locking and there are a lot of updates, both of these strategies will lead to severe performance problems. This is compounded when users don’t want to or can’t keep their transactions short. Users may not be able to keep their transactions short if they need to ensure atomicity over a complex set of operations.

A simple example is that a bank does not want to transfer money from one account to another in two transactions just to keep them short. If the withdrawal from one account fails, the bank certainly doesn’t want the
money deposited anyway. They want both operations (and possibly many more) in the same transaction.

This is where Objectivity/DB containers and pessimistic locking help solve this dilemma. Most applications want to lock more than an individual row at a time, not only for efficiency, but also logically. If you are locking a customer, it makes sense that you’d also lock his accounts. Standard database page-level locking doesn’t allow you to do this because pages are generally homogenous according to type.

Objectivity/DB containers, however, give users and applications a clear way to cluster objects together according to how they’re related. An Objectivity/DB user can simply cluster a customer and his accounts in the same container. If there’s a lock conflict, the application learns of it immediately and is able to handle the exception before more work is done.

Some SQL DBMS vendors offer “clustering”, but only in a very limited way and consequently rarely used. For example, Objectivity/DB allows clustering on an object-by-object basis, regardless of type. Other clustering features require objects to be of a limited number of specific types. Comparing the documentation on a vendor-by-vendor basis shows other substantial differences.

Not only does Objectivity/DB provide standard read and update locks, it also offers MROW (Multiple Readers, One Writer) mode, which is a form of “stale” reads except far safer and easier than using the optimistic locking approach.

As one may remember from the “Persistence” section, Objectivity/DB updates pages by writing the modified page, changing the page map, then releasing the old page to a free list.

To provide MROW mode, Objectivity/DB simply allows readers to read the old pages as the new ones are being created. By keeping reference counts on the old pages, Objectivity/DB can release them when the last reader has committed or refreshed their view.
This technique allows very high concurrency applications the ability to read existing data without waiting for updaters to finish. Because Objectivity/DB still maintains pessimistic locking and keeps track of the modified pages, the user will get notified immediately if the application tries to update data that has been modified. The user or application can then refresh the data and re-request the update privilege.

This fully prevents the “last one in wins” problem, since transaction B would be able to read Mary’s old salary, but would know immediately if transaction A updated the salary when transaction B attempted to make its change. The effect would be that transaction B would simply refresh the view (getting the updated salary), then apply the 10% increase. The application code is clean, there are no complicated stored procedures, the transaction completes quickly and no operation is lost to a race condition.

The end result is that Objectivity/DB users and applications have tremendous control and confidence in the integrity of their data and the ability to choose the length of their transactions based on what’s most logical and efficient for the application – not what’s least likely to cause errors.

**Schema Control and Data Migration**: SQL databases support fixed numbers of columns in tables. When the number of columns changes, generally all the rows of the table must be migrated immediately to the new form. Simple operations like deleting a column can even require burdensome migrations that include copying the entire table. During this operation, the table will be unavailable to users.

Objectivity/DB users are given more choice for schema evolution and data migration. In each class. Each type has a set of shape numbers each of which is unique in the schema across all the classes.

Each object in Objectivity/DB has a shape number to describe its class including its version. Consequently, the type can evolve into a new shape without immediately affecting any of the objects.

As objects are used, Objectivity/DB will detect if the shape number of the object has been evolved. If it has, Objectivity/DB will evolve (migrate) the
object into the new shape and change its shape number to the evolved one. If the migration requires adding primitive attributes, those attributes will be added with default values that can be defined in the schema.

If the migration requires deleting attributes, Objectivity/DB will simply drop the data from the object. More complex migrations can be defined with callbacks defined in the application that allow you to read the object in the old shape and migrate it to the new shape yourself.

Objectivity/DB applications can therefore defer all data migration until the migration is necessary and when the data is already being read into the system for other reasons. It may be undesirable to ever migrate data, since it may be archived or there may be a desire to always store the objects in their original form. With Objectivity/DB, this is not only possible, it’s a natural capability of Objectivity/DB’s architecture.

**Transactions (Commit / Rollback):** Objectivity/DB provides standard transaction semantics. Objectivity/DB provides session classes and API that provide begin, checkpoint and commit functions. The begin and commit functions are self-explanatory. The checkpoint function commits the data to the federation, but keeps all the locks intact.

**Single, Consistent View of Data:** Another fundamental benefit of a database management system is providing a single, consistent view of data. A natural inference from this is that the data is centralized, but this isn’t necessarily a requirement and often it’s undesirable. Centralization leads to bottlenecks and single points of failure.

SQL databases are by nature centralized because the query engine tends to be centralized. This is depicted in Figure 7. Joins between tables are already inefficient. Separating data across a network or other slower connection aggravates the cost of joins. Without flexible clustering, it’s difficult for a SQL database to split tables across multiple servers in an efficient way.

Objectivity/DB OIDs provide a natural ability to easily distribute a federation’s data across servers, clients or any data storage accessible to clients. As described in the section titled Persistence, the first 16 bits of the...
OID map to the database of the object. Each database is represented by a single file.

When Objectivity/DB fetches an object based on its OID, it first looks up the database file in a database catalog using that 16-bit number. This is a very fast operation, since the database list is generally not very volatile and will easily fit in a client's cache. To move data from one server to another or even to a client, all one has to do is run an Objectivity/DB utility called oochangedb. No application code must be changed.

The oochangedb utility simply moves the database file to the new location and updates the database catalog. Each client discovers that the catalog has changed and refreshes the list.

Once the database of the desired object(s) has been located, the file is opened and the specific pages are found by finding the container offset, looking up the physical page in the page map and fetching the page(s) by reading from the file.

If the page has been read from a different architecture than the client, the appropriate conversion (endian, byte-swapping, etc) is done in the client's memory.
Objectivity/DB therefore supports the centralized client-server architecture, and many Objectivity/DB customers choose centralization because it fits the application.

Objectivity/DB also supports many distribution models that can greatly improve the performance and availability of data in the federation.

Common examples are:

- Easily adding servers without federation downtime to increase performance using parallel processing.

- Since Objectivity/DB supports server heterogeneity, new servers can be purchased by choosing the best value at the time. Original hardware investments are not lost, nor do the original investments bind the system to any specific vendors.

- Moving local data to local machines to reduce network traffic and increase performance of providing that data to the local clients.

- Often objects are placed into databases according to locality (geography, for example). This means that Denver’s data can be stored in a database hosted in Denver and San Francisco’s data can be stored in a database hosted in San Francisco.

- San Francisco can still read Denver’s data and vice versa, but San Francisco does not have to communicate over a wide area network to access its own data.

- The lock server maintains the consistent view across the federation, including concurrency control and integrity of transactions. In order to decentralize the lock server, Objectivity/DB also provides a “Fault Tolerance Option” that allows separate lock servers to manage groups of databases.

Figure 8 shows an example of a somewhat distributed architecture. Most of the data is stored on a dual-server system, so it’s still basically a client-server architecture. Two workstations, however, have local databases. If these clients request objects located in these databases, the access can be provided directly.
Objectivity/DB also provides a “Data Replication Option” that allows database images to be replicated as many times as necessary. Consequently, one or more clients can have a copy of a database that’s on the server. This obviously has large performance advantages for fairly static databases that have data that’s shared across multiple clients.

Objectivity/DB’s distribution model can even be taken to the point where there is essentially no single server at all, as shown in Figure 9. This is a pure “pier-to-pier” distribution model - any client can access any other client’s data. Again, either a single or multiple lock servers will enforce consistency, concurrency control and transaction integrity of the federation.
A Practical Example – A Message Storage and Analysis System

**Description:** To examine the practical application of Objectivity/DB’s features, let’s design a message system designed for an internet service provider (ISP) to monitor very large quantities of e-mail and instant messages and detect undesirable advertisements (spam) and viruses.

The requirements of such a system might be as follows:

- The system will receive an average of 10,000 messages per second, but at peak times may receive as many as 50,000 messages per second.
- The average size of a message (with attachments) is 5KB.
- The system must be able to store all incoming messages – none may be lost.
- Each valid message must be made available within 2 minutes.
- Two weeks worth of messages will be kept online.
- All messages (even spam) will be stored in a long-term archive that will be queried.
- If a virus is detected in a message, the system will:
  - Start an electronic investigation to attempt to determine the source of the virus.
  - Determine how the virus spreads itself and use that information to add to the virus detection capabilities.
- Attachments will be saved in the database so that queries can be performed. Those attachments in a recognized format will be stored as structured data. Those in an unrecognized format will be stored as unstructured binary or text objects.

**Logical Schema:** Let’s propose a schema to define the structures of the data we want to manage. The schema will be composed of classes and their attributes. The most obvious classes that we’ll need are Message, Person, Address and Attachment. The schema proposed here is just a simple example.

**Message Class:** Basic Attributes

The message class will have the following attributes:
Message Format and References: Another attribute might be “format”. Rather than simply define this as an integer and assign different formats a unique number, let’s create a new class “Format” and refer to the message’s format using a reference.

There are currently many different formats for messages and there will be many more created in the future. For e-mails, the standards are defined by different “Request for Comment” documents called RFCs. One of the original baselines was RFC822, but there have been many new RFC documents that are new versions of the format.

By defining a class called format, we can include the attributes that are common and stable (if any) over all the versions and use this as a base class. We can then define specific formats and add new attributes by deriving new classes from the base class.

As new formats are created, we can either create a new instance of an existing format with different attribute values or we create a whole new class derived from Format if no existing formats fit.

Unlike SQL tables, the object-oriented approach gives us the flexibility to fit the data structures to the existing structures that already exist and also be adaptable to new structures that may be defined.

If we simply assigned a number that matched an entry in a format table, we’d have to assume that the SQL query would include knowledge of the table(s) to search to find that entry (in the FROM clause). By using a reference, we can go directly to the format object, regardless of what specific type of format it is. Changes to the schema are less likely to require changes to the data.
Attachments and Relationships: We also want to use references for Attachments, but since a message can have any number of attachments (as opposed to only one format), we’ll want to create either an array of references or a one-to-many relationship.

An attachment can be just about anything, so we’ll simply define the list of attachments as a list of ooObj objects, Objectivity/DB’s base persistent-capable class. By doing this, any class we define can be used as an attachment, including other messages. We’ll still create a class called FileAttachment, from which we can derive specific classes for specific kinds of files. For now, we’ll just have two specific classes, TextData and BinaryData.

Obviously, if we find a virus in an attachment we’ll want to know quickly which message it has come from. If we find attachments by searching the attachments of messages, the virus-scanner can simply keep the reference to the “current message”. But if we search attachments independently, we’ll want each attachment to have a reference to the message it came from.

To do this, we’ll simply define an Objectivity/DB one-to-many bi-directional relationship between Message and Attachment. Each time we add an attachment reference to a message, Objectivity/DB will ensure that attachment OID will be added to the message and the message OID will be set in the attachment.

Sender, Recipients, CC Lists and Addresses: The sender, recipients and CC lists are all composed of Address objects. In a real system, we might make the Address class a base class for all the different kind of addresses the system will use.

For now, we’ll assume that the Address class simply contains a String for the actual address (e-mail or instant message id), a many-to-one bi-directional relationship to a Person object, and a one-to-many bi-directional relationship to other Address objects.

We’ll add a one-to-many Address relationship to the Address class. This recursive relationship defines a tree where an address (like group@objectivity.com) can be an alias for more addresses. When we
actually populate the relationship with values, we can choose to leave this empty if it’s a simple address or populate it if it’s an alias for multiple addresses.

![Diagram of MessageManager Application]

**Physical Data Placement:** Now that we have our schema, we can now think about how we want to organize our objects on our physical storage. With almost all SQL databases, it’s assumed that the physical placement of the data will follow the table structure and be organized by type.

For this application and many others, however, it doesn’t make sense to store the data purely by type for several reasons.

For example, messages have a very strong temporal component, but there may be many different kinds of messages. In our initial schema design, we assume that there is only a single type of message, but we’d like to be able to expand the application to subtype message into many different kinds without affecting our queries.

On the other hand, message and address objects are of different types, but are tightly related. Rather than separate these objects, we’d like to group these as “composite” objects and possibly organize them by attributes appropriate to these, like last name.
The difference between Objectivity/DB and other solutions is not that Objectivity/DB requires a physical design. The difference is that Objectivity/DB allows more customization and is less restricted to defining the physical strategy according to type.

**Using Databases and Containers for Fast Archival and Deletes:** In order to sustain the rate of incoming messages, we’ll need to be able to archive older messages and delete them as fast as the incoming rate. We discussed earlier how this can be a problem if one is using queries to find the messages and delete them.

The preferred design partitions data according to time so that large numbers of messages and attachments can be archived at once. This can be accomplished easily with Objectivity/DB by using the physical database and container concepts.

In the diagram to the left, each blue cylinder represents a database and each yellow block represents a container. Each database can represent a time period (e.g., an hour) and each container a smaller time period (e.g., a second).

In this case, each database would have 3600 containers of data, one for each second in its hour. Two weeks of data would require 14*24 = 336 databases.

If we run parallel threads or processes for ingesting the data, we’ll want to provide separate databases (and consequently separate containers) for each process or thread. This would multiply the number of databases required by the number of threads. We’ll examine how to optimize the number of concurrent threads later in the paper.

Since Objectivity/DB uses a separate file for each database, copying a specific hour’s worth of data to an archive and deleting it requires little more than using the operating system calls to copy or move the appropriate file(s). Deleting an hour’s worth of data could actually be accomplished in a matter of seconds or less.
This does not prevent optimizing queries that are not time-based, nor does this design require us to organize all the federation’s data by time. Logical query optimizations can be employed using indexes and customizable optimizations like hash maps.

**Using Databases and Containers for Fast Queries:** Furthermore, finding data by time will be much easier, since it’s logically and physically organized this way. This is a huge benefit for this application, because time is a major search criterion for almost all queries.

We’ll accelerate queries that have a time filter by creating and maintaining time metadata for all the databases. We’ll simply create a special database and populate it with a list of databases we’ve created for the data. Each time we create a new message database, we’ll add the entry to our metadata and include the time range for it.

When we query the database for a given time period, we’ll first look at the list of databases and find those that match the time period we’re looking for. We can then avoid searching databases that we know will have no relevant data.

We can carry this further by maintaining metadata for all the containers in a database. Each time we create a new database, we’ll create a special metadata container that will contain a simple list of all the message containers in the database and the time period each one represents.

Each time we create a new container of messages, we’ll add that container...
and its time period to the list. In this example, we’ll assume that each database and container has a discrete and predictable time period. For example, each database could represent a particular hour and contain 3600 containers, each representing a second’s worth of messages.

These ranges lists are very similar to the skeletons of a b-tree index, but they have several advantages. Let’s look at the costs of these range lists.

Each entry in a list requires 24 bytes – 8 bytes for the start time, 8 bytes for the stop time and 8 bytes for the OID to the database.

If we assume each database represents an hour of data, the requirements for the two-week online storage can be calculated as follows:

DB List Metadata Storage Requirement
= Number of Days * databases/day * number of streams * 24 bytes
= 14 days * 24 databases/day * 60 streams * 24 bytes
= less than half a megabyte

Container List Metadata Storage Requirement
= Number of Seconds/Day * 24 bytes * number of databases
= 3600 * 24 * 24 databases/day * 14 days * 60 streams
= approximately 1.74 GB

Using Databases, Containers and Range Lists for Parallel Queries: As described in the section titled “Query by Attribute”, Objectivity/DB allows users and applications the ability to scope queries within databases and containers.
In our system, if a query has a time constraint – i.e., we’re looking for objects that were received within a given time period – we can choose to perform the query only on the databases and containers that fall within or overlap that time range, using the range lists to quickly discover the appropriate “candidate” containers.

We can also use the range lists to provide acceleration to other attributes that may be used as query constraints.

Not only does this greatly reduce the amount of data that needs to be searched, it also provides a simple, but extremely effective, opportunity for performing the query across parallel threads, as shown in Figure 13.

Hardware Architecture Design and Sizing: We can now start to scope the hardware requirements to build such a system. If we average 10,000 messages per second and we need to store 7 days worth (604800 seconds), we’ll have an average of about 6 billion messages in storage. If the average size of a message is 5Kbytes, we’ll need at least 30 Terabytes of disk space to store it.

Let’s assume a 50% “data explosion” rate, which represents the overhead for Objectivity/DB, including any indexes or other extra metadata. This will be analyzed in more detail, but is a fairly safe estimate. Let’s also assume...
that the total size and quantity of messages may fluctuate as much as 25%. We’ll add another 25% for any other factors we may not have considered as well, so we’ll need a total storage area of 60 Terabytes.

The storage hardware must also be able to sustain writing at the peak rate of 50,000 messages, or 250 MB, per second and be able to support simultaneous reading of the data at a similar rate. Here again, the ability to delete data by deleting files at a time is very important, since it will require very little I/O.

A very small benchmark was performed to determine the relationships of object size and page size against overall ingest throughput on an Objectivity/DB system.

To perform the benchmark, a system with the following specifications was used:

- 1.2 GHz Athlon processor
- Windows XP Professional
- Objectivity/DB /C++ 8.0
- C: Drive - 60GB Quantum FireballP 7200 RPM (ATA)
- F: Drive - 80GB Maxtor DiamondMax Plus 7200 RPM (ATA)

The objects created were instances of a class called Message with the following members:

- uint64 mReceived;
- uint64 mSent;
- ooVString mBody;
- ooRef(Format) mFormat;
- ooRef(Address) mTo[] : copy(drop);
- ooRef(Address) mCC[] : copy(drop);
- ooRef(Attachment) mAttachments[] : copy(drop);

The objects were created in the database in single transactions. The size of the object was varied by assigning different sized strings to the mBody member variable. The total size of the object was estimated by adding 16
bytes to the size of the mBody string for the mReceived and mSent variables. There was also overhead for the reference and associations that was not counted.

The number of objects created in each transaction varied according to the object size.

The calculation used was:

\[
\text{number of objects created} = \frac{50,000,000}{\text{size of data string in object}}.
\]

The actual range of number of objects created per transaction was 1,000,000 to 59.

The first object of each transaction was clustered using the container OID. All following objects were clustered using the previously created object. Each object’s mBody variable was set by passing a pointer to a pre-defined string using a member function call. The member function called ooUpdate() each time, so some optimization might be gained for smaller objects by eliminating this redundancy.

For each run a new database and container were created and the container was pre-sized with 20,000 pages and a 50% growth rate. The creation of the database and container were not included in the timings.
Just from observation of the Windows' performance monitor, the CPU seemed to be running approximately 33% to 50% busy during object creation. The lock server was sharing the CPU, but could not have been very busy, since there was only a single container being used.

Multi-threading or running parallel processes with both disks would almost certainly increase throughput within the same configuration.

Given parallel input streams, one can reasonably assume that many such systems can be used in parallel to achieve very large overall ingest rates. For example, let's assume that we can sustain 10MB/sec. Using 30 such systems in parallel should provide an overall throughput of approximately 300 MB/sec or over a Terabyte per hour (300 * 3600 sec/hour). If such systems cost $1000 each, the total cost would be $30,000.

The most expensive part of this system is the physical disk storage. Based on suggested retail prices published on websites as of February 7, 2003 (from http://www.apple.com/xserve/raid/ on November 11, 2003), these are some prices for high-speed RAID storage:

<table>
<thead>
<tr>
<th></th>
<th>Apple Xserve RAID</th>
<th>Dell EMC CX200</th>
<th>IBM ProFibre DF4000R</th>
<th>HP 7100</th>
<th>Sun StorEdge T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>2.52TB</td>
<td>2.2TB</td>
<td>2.2TB</td>
<td>2.64TB</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>3U</td>
<td>2 x 3U</td>
<td>2 x 3U</td>
<td>3 x 3.5U</td>
<td></td>
</tr>
<tr>
<td>Price</td>
<td>$10,999</td>
<td>$30,000</td>
<td>$43,974</td>
<td>$109,968</td>
<td>$144,300</td>
</tr>
<tr>
<td>Price per GB</td>
<td>$4.36</td>
<td>$13.63</td>
<td>$20.08</td>
<td>$50.21</td>
<td>$54.66</td>
</tr>
</tbody>
</table>

This implies that providing the storage capacity for a 60TB system could cost between $264,000 and $3,300,000. Clearly, the cost of hardware to support the database (beyond the basic disk storage) is a small portion of the total cost of the system.

Each of these systems, however, supports much more I/O bandwidth than...
the single disk drives used here.

To scale it further, the first likely Objectivity/DB bottlenecks would be the lock server and the journal file access. Based on larger benchmark experiments, the lock server shouldn’t become a bottleneck until scaled at least by a factor of 80 or so if it’s running on a reasonably fast CPU and network connection.

The journal file directory/disk would be a more serious problem, but fortunately Objectivity/DB’s journal files are very small, since they only have to store the old page maps (the old pages aren’t freed until the end of the commit process). These performance measurements also assume fairly long transactions, so the amount of communication with the journal files will be very small.

**Objectivity/DB Terminology:** Since Objectivity/DB is an object-oriented data management system, we will introduce a few new terms to those familiar with other data management systems. These are:

- **Federation**: An Objectivity/DB federation can be thought of in the same way other data management systems use “database”. Objectivity/DB’s federation is a collection of one or more Objectivity/DB databases.

- **Object**: An Objectivity/DB object is similar to the data portion of an object as used by programming languages. Each object is an instance of a defined class and consists of member variables and relationships, as defined in an Objectivity/DB schema.

Objectivity/DB is not an “active” data management system in that it does not store the member functions. Functions are applied by code written in one of our language bindings (e.g. C++, Java, Smalltalk). The objects can also be modified using an Objectivity/DB programming interface called Active Schema.
An Objectivity/DB object is also similar to a “record” in other data management systems and object members can be thought of as the record’s columns.

On the other hand, an object is more flexible than the traditional concept of a record. For example, an object can have embedded members of other classes and variable-length arrays.

**OID**

An OID is an object identifier. Each object has a unique 64-bit OID made up of four 16-bit fields. These four fields represent the object’s database, container, page and slot, which are described below.

The OID of an object is assigned when the object is first persisted and remains the same for the life of the object, even if the object is modified. This allows objects to contain references to other objects by using these OIDs.

Like pointers or references in object-oriented languages (and unlike people), both the referring object and the referenced object can change without affecting the relationship between them.

For example, a Message object can be associated with a Person object. Either object can be renamed or otherwise modified without changing the OID(s) or pointers between them.

**Page**

An Objectivity/DB page is a group of objects stored physically together on disk. A page is the unit of transfer from the federation to the application. The page size is definable by the creator of the federation and is usually matched to the page size of the operating system.

The objects on a page can be of different types. For example, a Person object can have its Message objects
Building a High-throughput Data Repository with High Query Performance

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>A class is defined as persistent capable if it has Objectivity/DB methods</td>
</tr>
<tr>
<td>OID</td>
<td>A Person object can also fetch their associated Messages with a single disk</td>
</tr>
<tr>
<td>Page</td>
<td>group on the same page. Consequently, fetching a Person object can also fetch</td>
</tr>
<tr>
<td>Slot</td>
<td>The page of an object's OID refers to which page in a given container the</td>
</tr>
<tr>
<td>Container</td>
<td>object is located. The slot of an object's OID refers to where in the page</td>
</tr>
<tr>
<td>Database</td>
<td>Both the page and slot values are “logical”, not “physical”. There is a slot</td>
</tr>
<tr>
<td>Persistent</td>
<td>map on each page that allows Objectivity/DB to move an object on a page</td>
</tr>
<tr>
<td>OID</td>
<td>without changing the object’s OID.</td>
</tr>
<tr>
<td>Slot Map</td>
<td>There is a slot map on each page that allows Objectivity/DB to move an</td>
</tr>
<tr>
<td>Page Map</td>
<td>object on a page without changing the object’s OID.</td>
</tr>
<tr>
<td>Directory</td>
<td>A container is a group of pages of objects. It is very analogous to directories</td>
</tr>
<tr>
<td>Page Map</td>
<td>in a file system. An application developer can place an object in a particular</td>
</tr>
<tr>
<td>Page Map</td>
<td>container when the object is first persisted in the same way a user can put</td>
</tr>
<tr>
<td>Container</td>
<td>a file in a particular folder or subdirectory when the file is first saved.</td>
</tr>
<tr>
<td>Page Map</td>
<td>An Objectivity/DB container can contain a single object or many thousands.</td>
</tr>
<tr>
<td>Database</td>
<td>These objects can be of mixed types and sizes. Each container stores a page</td>
</tr>
<tr>
<td>Database</td>
<td>map that allows Objectivity/DB to move pages within a container without</td>
</tr>
<tr>
<td>Database</td>
<td>changing any object’s OID.</td>
</tr>
<tr>
<td>Page Map</td>
<td>Consequently, database files can be moved and copied through Objectivity/DB</td>
</tr>
<tr>
<td>Database</td>
<td>commands with similar speeds as basic move and copy operating system commands</td>
</tr>
<tr>
<td>Persistent</td>
<td>without changing or otherwise affecting object OIDs.</td>
</tr>
</tbody>
</table>
federation. This usually means that the class was derived from Objectivity/DB’s ooObj class.

**Persistent**  
An object that is stored in the federation is considered persistent.

**Transient**  
An object that is in an application’s memory, but not stored in the federation is considered transient.

**Reference**  
Objects in Objectivity/DB can have references to other objects. These references are stored in an object simply using the OIDs of the other objects.

A reference or array of references can be defined in the schema of a class, just as one would declare a pointer in a transient class. A reference or array of references can then be populated with values, just as pointers in transient classes can be assigned values.

**Relationships**  
Also known as associations, these are Objectivity/DB structures of references that have special meaning to the Objectivity/DB kernel.

Relationships can have different cardinalities (one-to-one, one-to-many, many-to-one or many-to-many).

Relationships can have properties that Objectivity/DB will maintain, such as bi-directionality, delete propagation and others.

**ACID Properties**  
The ACID properties of transactions are not specific to Objectivity/DB; they are expectations described in ISO/IEC 10026-1:1992 Section 4 for the behavior that all transactions should provide.

Atomicity:  
An entire sequence of actions defined by the beginning and end of the transaction must be either completed or aborted. A transaction cannot
<table>
<thead>
<tr>
<th>Consistency:</th>
<th>The state of the data after an aborted transaction must be the same as the state was before the transaction was started. The state of the data after a committed transaction must include and only include the work done during the transaction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isolation:</td>
<td>A transaction's effect is not visible to other transactions until the transaction is committed.</td>
</tr>
<tr>
<td>Durability:</td>
<td>Changes made by the committed transaction are permanent and must survive system failure.</td>
</tr>
</tbody>
</table>