

Physical Design with Objectivity

Introduction and Background

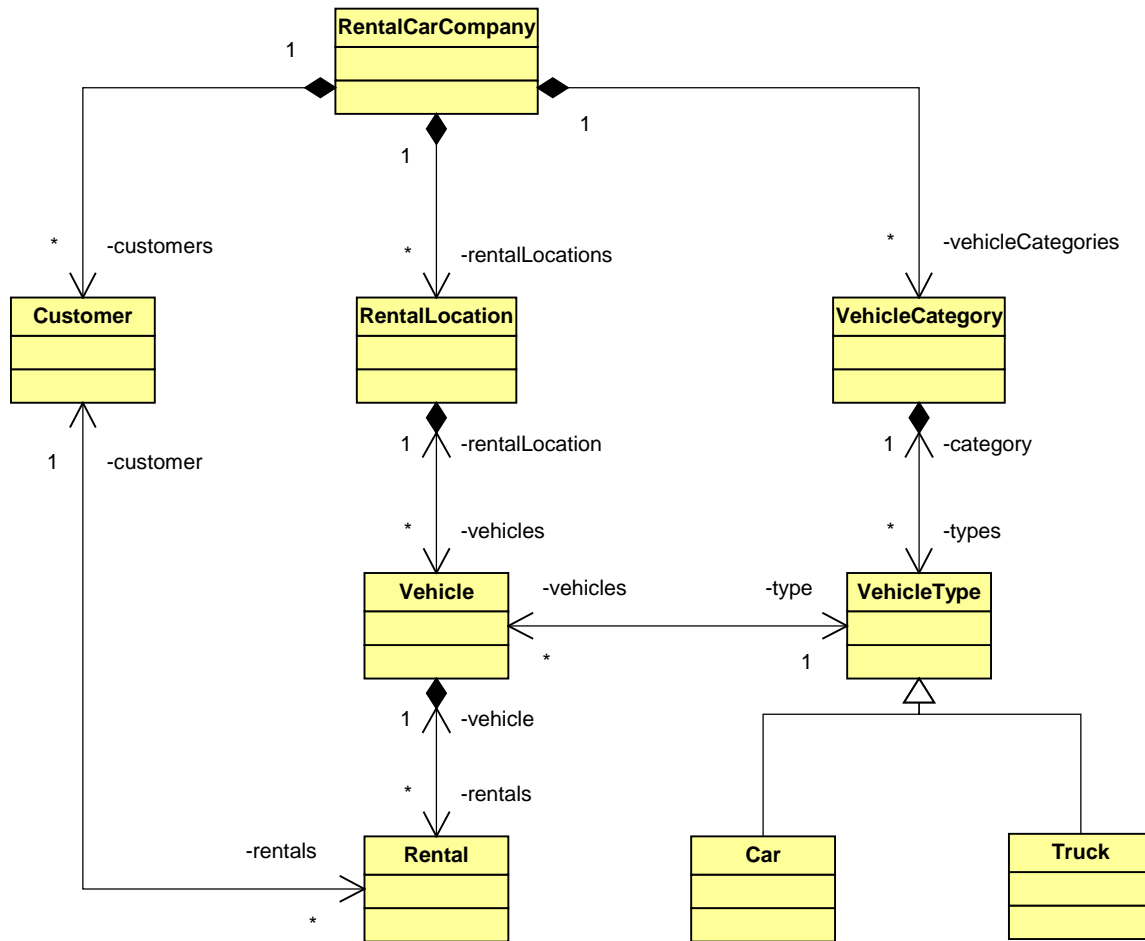
Database application design is often described as a combination of logical and physical design; logical design being the process of creating a logical schema that describes the persistence needs of the application, while physical design being the process of determining how that logical schema is to be represented in the database. Typical physical design for applications built upon relational databases involve creating an internal schema (defining the tables), mapping the logical schema to the internal schema (classes to tables), and choosing various DB specific storage options for the internal schema components (tables and sometimes table fragments).

Because Objectivity/DB is an object-oriented database, typical physical design for applications built upon it is somewhat different. One main difference between Objectivity/DB and relational DBs that in Objectivity/DB the logical and internal schemas the same, thus there is usually no need to develop an internal schema, nor to map between logical and internal schemas. Another significant difference is that the storage options that Objectivity provides are independent of the schema (logical or internal), which leads to greater flexibility (also known as physical independence). The storage options Objectivity provides are page, container, and DB as described in appendix A, and instances of classes in the logical schema (objects) can be placed in any combination in any storage option without regard to type.

Given the above, *Physical design* (PD) in an Objectivity application is best described as the process of determining where objects are placed, what containers and databases are created, and how containers and databases are distributed in order to achieve optimal read, write, and concurrency performance, and when Objectivity/HA is employed, availability.

An Example

Given the following schema (both logical and internal) for a fictitious rental car company application:



(Figure 1 -- Rental Car Company Application Schema)

Reasonable physical design choices might be:

Basic:

- Place the single RentalCarCompany object in a container and then in a database called "Company". This database will contain all company wide information.
- Place Customer objects in a container pool in the owning RentalCarCompany object's DB. A given customer can rent from any number of locations, therefore customer data is best kept with the "Company" database. Concurrency between customer objects too great to place them all in a single container, yet not great enough to justify placing each in their own container, thus the choice of a container pool, where customer objects are randomly spread among several containers.
- Place all VehicleCategory objects near the owning RentalCarCompany object. The category objects are company wide information and thus belong in the "Company" database. They are seldom updated and thus don't need their own container and can go into the RentalCarCompany object's container, and in fact

placed as nearby (on the same page if possible) as the RentalCarCompany object so that browsing the vehicle categories for the rental car company can be done with a minimum of page reads.

- Place all VehicleType objects near their owning VehicleCategory object. Vehicle types are integral to their owning vehicle category, are seldom updated, and by placing them nearby their owning category, page reads for the common operation of browsing types are minimized. Note that this design choice pertains to both Car and Truck objects such that they don't need their own design choices, as they are VehicleType objects.
- Place each RentalLocation object in its own database given the name of the location. These databases will contain all location specific information.
- Place each Vehicle object in its own container in its owning RentalLocation object's database. Vehicles are assigned to (owned by) a single rental location and thus should be placed in that rental location's database. Concurrency between vehicles can be great enough to justify placing each in their own container.
- Place each Rental object nearby its owning Vehicle object. A vehicle can only be rented out to one customer at a time, thus there is no concurrency between rental objects and therefore they can all be placed into the same container. A given rental object pertains to a single vehicle object and will often be used in conjunction with that vehicle, thus placing the rental objects nearby their owning vehicle object will reduce page reads and writes.

Advanced:

Distribution:

- Place each rental location database on a server at the location site. Since rental locations are geographically dispersed, performance is improved by giving each rental location local access to its rental location specific data instead of forcing access over usually slower remote network connections.

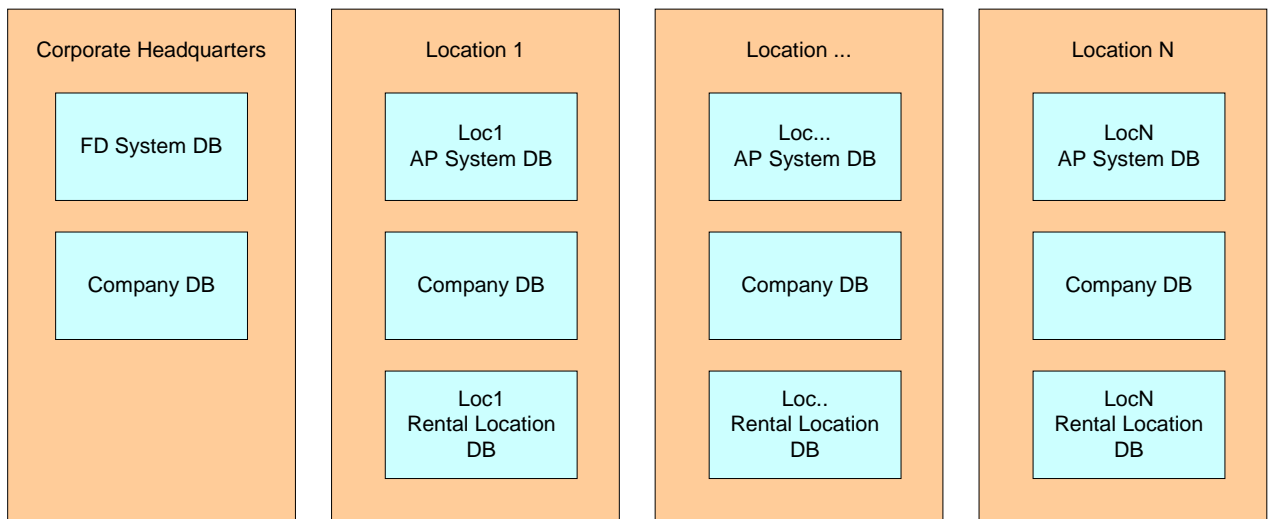
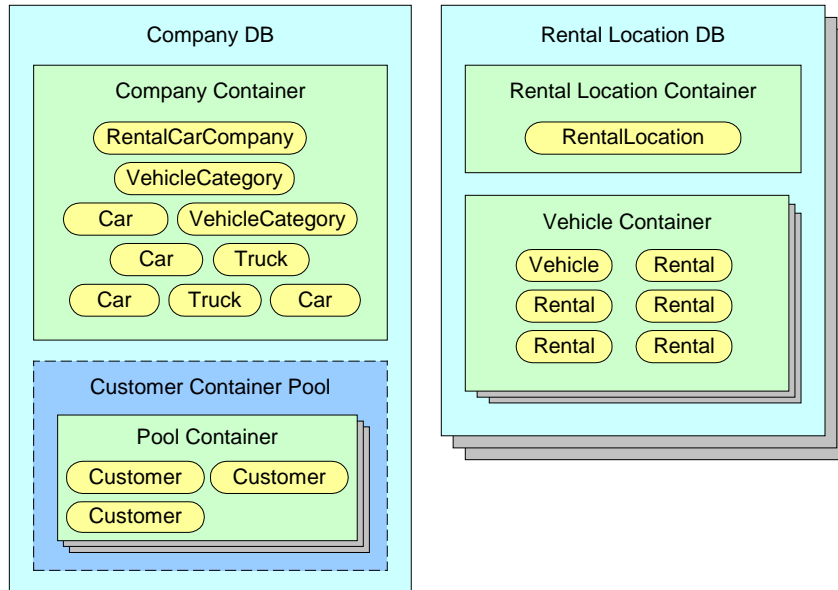
High Availability (HA) Option:

- Give each rental location its own AP. Giving each location its own AP allows them to operate on local data even when remote connections are lost.
- Replicate the "Company" database such that there is a server at each rental location site that has a copy. Giving each location a replica of the "Company" database makes local all data needed for normal operation at a location so that lost remote connections can be fully tolerated.

As can be ascertained from the rationale for each physical design choice, physical design's greatest impact is on application performance. A good physical design uses Objectivity/DB page, container, DB resources optimally and leads to good performance by maximizing benefits of clustering and distribution and minimizing performance hits from excessive locking and lock conflicts, and a poor physical design does not use resources optimally and leads to poor performance due to poor processor/disk utilization,

excessive remote data access, excessive page reading/writing, excessive locking, and/or lock conflicts.

Figure 2 is a depiction of this physical design:



(Figure 2 -- Rental Car Company Application Physical Design)

Object placement Strategies

Object placement strategies determine where objects are to be placed, in relation to each other and into what containers.

Here are the concepts of object placement, which will be followed by a description of the various object placement strategies provided by the PDM.

Goals	Motivations
Place objects that are used together in the same container	Performance is best when an operation only has to open a single container to access data.
	Performance is best when objects that are used together, are locked together. This is what happens for objects in the same container. Once a lock has been obtained for the container, no additional lock server traffic is necessary to lock individual objects within it.
	Query optimizing indexes have better overall performance when placed at container scope, as opposed to DB or FD scope which incur slower updates due to use of micro transactions, which are used to avoid a locking bottleneck for indexes whose objects span multiple containers.
	Because containers own pages and pages are not shared between containers, objects in separate containers cannot possibly be on the same page. Unnecessarily placing objects in separate containers eliminates the possibility that common page reads/writes may be used.
Place objects that are used together as close to one another as possible.	Objects reside on pages within a container. When transferring an object to/from disk, the entire page it is on is transferred. Placing objects that are used together close to one another, such that they are likely to be on the same page, reduces the number of transfers required for those operations that work with the group.
Use separate containers for separately accessed objects when access serialization is unacceptable.	Because locking is controlled at container level, concurrent access (read-update or update-update) of two or more objects within the same container must be serialized. It is considered a "lock-conflict" when one client must either wait for another to release a lock, or abandon their operation. If either the frequency of lock conflicts or the duration of lock waiting leads to unacceptable performance, then the separately accessed objects should be placed in separate containers.

Place independent groups of objects in separate containers.	When a container is created, it is given two physical pages overhead (for the page map) used to keep track of its logical pages. As the container is grown to contain more pages, the page map size is increased, thus increasing the overhead.
	Placing different groups of objects in the same container that are not used together unnecessarily increases the container overhead for working with either. Therefore, it is best to place independently used groups of objects into separate containers to minimize the container overhead that must be read/written for reading/updating objects from any one group.
Place objects that need to be distributed differently into separate containers.	Distribution is configurable per database or per container, not per object. Therefore to distribute one group of objects differently than another, they must be in different containers.
Use containers to segregate objects by for ease of archival and/or search.	Containers used this way capture objects with some common key value or that have a value in some range. This can simplify archival based on those values, as containers or the databases that contain objects associated with those values can be easily archived or otherwise manipulated as a group. Segregating by value can greatly speed queries were these values are provided such that the containers searched can be narrowed down considerably.

The following are common object placement strategies used to meet these goals:

Place By Owner	Description	Place the object nearby its owner. { Available for objects that have owners only. }
	Use	When a parent object (the owner) and its child objects (those being placed) are often worked with together, and/or the parent is navigated through to get to a child.

	Don't use	When it would cause an unacceptable level of lock conflicts to have the parent and child objects all share the same lock, or when children need to be distributed independently of the parent.
In Container	Description	Place all objects into the same container.
	Use	When the objects are all used together.
	Don't use	When it would cause an unacceptable level of lock conflicts to have all objects share the same lock, or when objects need to be distributed independently.
ooEachInOwnContainer	Description	Place each object in a new container created for it.
	Use	When objects are not worked with together, when placing all objects into the same container would cause an unacceptable level of lock conflicts, or when objects need to be distributed independently.
	Don't use	When objects are used together but not concurrently (consider In Container).
		When objects are sometimes used together and sometimes concurrently (consider In Container Pool).
In Container Pool	Description	Place objects in a random or round robin fashion into a group (pool) of containers.
	Use	When seeking a balance between In Container and Each In Own Container. When the Objects may be used concurrently so In Container is not desired, and also sometimes used together so that the spreading incurred from Each In Own Container is not desired either. By controlling the number of containers in the pool you can adjust the behavior from In Container (a pool size of 1) to Each In Own Container (a pool size \geq number of objects).
	Don't use	When concurrency is not an issue.

		When concurrency is such an issue that all lock conflicts are to be avoided (consider Each In Own Container).
		When objects may need to be distributed independently.
In Container Map	Description	Place objects in a map of containers, with the map key provided by the application at run time.
	Use	When it is desired to segregate data according to some key value.
		When it is desired to be able to narrow the scope of scans to those containers matching a provided key.
	Don't use	When key values don't exist or can't be generated that provide the desired segregation.
In Container Vector	Description	Place objects in a vector of containers, with the index of the container to use provided by the application at run time.
	Use	When it is desired to segregate data according to some integer value .
		When it is desired to be able to narrow the scope of scans to those containers within a provided index range.
	Don't use	When index values don't exist or can't be generated that provide the desired segregation.

Container DB-placement strategies

Container DB-placement strategies determine what databases to create, and what containers they contain.

The goals of container placement are:

Place containers that are used	Traversing references or associations between objects is faster if they are in the same database as opposed to being in
--------------------------------	---

together into the same database.	separate databases.
Place containers that need to be replicated independently into separate databases.	Replication is configurable per database, not per container. Therefore to replicate one container independently of another, they must be in separate databases.
Place containers that are highly concurrently, accessed in separate databases.	Having containers in separate databases allows you to place each on separate physical drives. This will improve performance by eliminating an I/O bottleneck and by reducing competition for drive head placement. There is also a performance benefit from using multiple files in a multi-processor environment, as contention at the file level is reduced.
	Even though containers are locked independently, certain operations (adding, deleting, and expanding containers) affect structures at the database level and must be serialized, which can have an adverse effect on performance.
	If the expected number of containers exceeds the available number of databases (an FD is limited to 65535 databases), then put only as many containers per database as is necessary to keep the number of databases within that limit.
Place containers that are to be archived independently into separate databases.	Having containers in separate databases allows you to delete or detach one container's DB without deleting or detaching the other container.

The following are common container placement strategies used to meet these goals:

In Owners DB	Description	Place the container in the same database that contains the object's owner. { Available for objects that have owners only. }
	Use	When a parent object (the owner) and its child objects (those being placed) despite being in separate containers are worked with together, and/or the parent is navigated through to get to a child.

	Don't use	When children need to be distributed independently of the parent.
		When children are placed in multiple containers that are highly concurrently accessed.
In DB	Description	Place all containers into the same database.
	Use	When the containers are all used together.
	Don't use	When the containers need to be distributed independently.
		When the containers are highly concurrently accessed.
Each In Own DB	Description	Place each container in a new DB created for it.
	Use	When containers need to be distributed independently.
	Don't use	When not needed, as this leads to the greatest level of container separation.
In DB Pool	Description	Place containers in a random or round robin fashion into a group (pool) of DBs.
	Use	When the containers are highly concurrently accessed.
	Don't use	When containers need to be distributed independently.
In DB Map	Description	Place containers in a map of DBs, with the map key composed of a single value condition or combination of value conditions
	Use	When it is desired to segregate data according to value conditions (date/time, client, and/or application supplied context). The segregation itself may be to enable independent distribution or selective archival.
	Don't use	When key values don't exist or can't be generated that provide the desired segregation.
In DB Vector	Description	Place containers in a vector of DBs, with the index of the DB to use provided by the application at run time.
	Use	When it is desired to segregate data according to some integer value (int64).
		When it is desired to be able to narrow the scope of scans to those DBs within a provided index range.
	Don't use	When index values don't exist or can't be generated that provide the desired segregation.

You may want to limit the number of containers per DB along with the number of physical pages per container as a means to limit the size of DB files as discussed in the section on container pages.

You also may want to limit the number of containers per DB so that more databases are used, which in turn means that those databases can be distributed across servers and/or disks.

File placement strategies

File placement strategies determine how database or container files are to be dispersed.

The goals of file distribution are:

Spread files of a very large federation across multiple servers.	Each server has its own resources (discs, memory, CPU) that contribute to the overall capacity of the system, extending data size and processing capacity way beyond that possible on a single server.
Reduce the impact of slow and/or unreliable networks by placing data nearest to its most common or critical users.	Nearest is in terms of network connectivity. A nearby server is one that is on a relatively fast and reliable connection, while a faraway server is one that is on a not so fast and/or not so reliable connection.
	Common and/or critical users should have reliable high-speed access to the data they work with most.
	If you find that you have part of a database that is best placed in one location and another part in another location, consider breaking the database apart or using container files.
Place data on separate servers to reduce impact of losing a single server.	The loss of a single server would only result in losing access to that server's data; the remainder of the federated database remains available.
	It is important to understand what type of operations are likely to be performed from where when deciding what data to place on what server. It does not help to have access to some of the data, if nothing can be done without some other unavailable part.

The following are common file placement strategies used to meet these goals:

Single Location	Description	Place files at a single location.
	Use	When locating data nearest to its primary users.
	Don't use	When you have multiple concurrent updaters and you want the performance benefits gained from dispersing files amongst servers and/or disks (consider Location Pool).
		When the location depends upon some value condition (consider Location Map or Location Vector).
Location Pool	Description	Place databases in a random or round robin fashion into a group (pool) of locations.
	Use	When dispersing databases amongst servers and/or disks to improve performance and/or gain capacity.
	Don't use	If you don't have the available resources (servers and/or disks). Though, you can still use this strategy and set all pool locations to the same resource (host and path).
		If you haven't broken your data into multiple databases.
Location Map	Description	Place DBs in a map of locations, with the map key composed of a single value condition or combination of value conditions.
	Use	When it is desired to distribute data according to value conditions (date/time, client, and/or application supplied context).
	Don't use	When key values don't exist or can't be generated that provide the desired segregation.
Location Vector	Description	Place files in a vector of locations, with the index of the location to use provided by the application at run time.
	Use	When it is desired to distribute data according to some integer value (int64).
	Don't use	When index values don't exist or can't be generated that provide the desired segregation.

Image AP-placement strategies

Image AP- placement strategies determine how databases are assigned to autonomous partitions (APs) and how they are replicated.

The goals of image AP-placement are:

Organize databases into APs to reduce the impact of losing access to a server with system resources.	An application must have access to system resources (system database, lock server, and journal directory) to be able to work with databases. Each AP (autonomous partition) has its own system resources, and using APs to group databases that are used together provides applications using one AP's database group independence from system resource failures that should only affect other groups.
Placed databases on multiple APs to gain the ability to have multiple lock servers.	For some applications, the lock server can become a bottleneck. Without using APs there can only be one lock server for the entire FD, but with APs each AP can be given its own lock server.
Replicate critical databases onto multiple servers.	Critical data is that data has a high value and is hard or impossible to reproduce.
	This maximizes the availability of critical data such that individual process, server, or network failures can be tolerated.
	Replication used this way also serves the purpose of a constantly up-to-date backup of the replicated databases. Keep in mind though, unlike backups created by ooBackup, which cover the entire database, this form of backup requires some of your attention to ensure correctness. When deciding to replication some data and not other data, make sure that there are no dependencies between the two sections, or if there are, that they are few and you are willing to manually fix broken relationships. The problem is that upon loss of a non-replicated section, references from the replicated section into the non-replicated section are left dangling.
Avoid replicating data over long distances that must be updated quickly.	The commit call on a transaction does not complete until all available replicas of affected databases have been updated (assuming the updating process has access to a quorum of images). This is termed synchronous replication, and is used to avoid the many pitfalls that can occur from inconsistent data versions. The cost for this safety is that the commit call will take as long as the longest commit. If there is a replica reachable only by a slow connection, then the commit will

	take a long time. If how long the commit takes is of no great concern, then this poses no problem, but if commit times need to be kept as short as possible, then long distance replication should be avoided.
Avoid ambiguous quorum weighting.	When using replication, each replica is given a weight, which is used when determining which side of a network break has update rights and which doesn't. Ambiguous weighting occurs when a break does not result in either side having a quorum, and nobody has update rights.
	Consider using a tiebreaker database image or the two machine hot fail over function if there is an even number of database images and there is no natural master/slave designation.

The following are common image placement strategies used to meet these goals:

Single AP	Description	Place databases on a single AP.
	Use	When using APs to group databases, based on type conditions or state.
	Don't use	When the AP depends upon some value condition (consider setting the image placement strategy for each DB location in the Location Map DB placement strategy).
When replication is needed (consider Replication and Master/Slave Replication).		
AP Pool	Description	Place databases in a random or round robin fashion into a group (pool) of APs.
	Use	When dispersing databases amongst APs to take advantage of multiple lock servers.
	Don't use	If you don't have available resources for multiple APs.
If you are uncomfortable with having multiple points of failure for a group of databases. Often APs are used to group databases that are used together, and only one AP failure can affect the group, but when using this strategy the group is dispersed amongst multiple APs, a failure in any one AP could cause problems for applications using the entire group.		

Replication	Description	Replicate databases across two or more APs.
	Use	To deal with databases that are common to two or more AP groups.
		To provide continuous backup of critical data.
Don't use	If the AP hosts have slow connections between them and update performance is critical.	
Master Slave	Description	Replicate databases across two or more APs with one AP designated as the master and all others designated as slaves. The database images are weighted such that any application with access to the master AP will have update rights to the database, while those that don't have access to the master AP don't have update rights.
	Use	For the same reasons as the Replication strategy, and one AP can be designated the master.
	Don't use	For the same reasons as the Replication strategy, or one AP cannot be designated the master.

Advanced Topics

Container pages

-

A container consists of two types of pages: logical and physical. Logical pages are actually not pages at all; instead they are entries in the container's page map. Physical pages, though, are real pages and are where the container's persistent objects exist. The page map is a map of logical to physical pages, where the key is the logical page number component of a persistent object's OID, and the value is the location of the beginning of a contiguous array of physical pages that all belong to that logical page.

For performance reasons the page map is limited to 64K (65536) logical page entries, which when using the PDM will never be exceeded. Additionally, all object placement strategies allow you to set this limit lower than 64K and place a limit on the number of physical pages, except for Place By Owner, which inherits the limits set on the object placement strategy used to place the owner. Before a placement is returned by the PDM, the logical and physical page limits of the enclosing container are checked, and if reached, a new container is created and returned as the placement.

To control the locking granularity (how many objects are locked together) you may want to limit the approximate number of objects placed in a container. You can do this by setting the maximum number of logical pages according to the following formula:

Logical page limit = number of small objects * average small object size / page size + number of large objects

A small object is one that is smaller than a page, where a large object is one that is larger than a page.

Objectivity supports database files with up to 4G pages, which with an 8K page size equals approximately 35 TB. This is a very large file, and may not be supported by your operating system, fit on your disks, or is simply too large to manage. For the management reason alone many Objectivity customers with large federated databases chose a smaller value (e.g. 10GB) as their maximum database file size.

To limit the size of database files, you can limit the number of physical pages per container and the number of containers per database.

Max database file size = max number of containers * max number of physical pages per container * page size

You can specify the maximum number of physical pages per container on object placement strategies, you can specify the maximum number containers per database on container placement strategies, and finally, you set the page size when you create your FD.

When limiting the number of physical pages per container, it is good to know how physical pages are used.

A single physical page can house many small objects, and many physical pages can be used to house a single large object. If you have large objects, then the number of physical pages in a container is going to be greater than the number of logical pages.

When there is more than one version of the container, there can be more than one version of a physical page, all of which contribute to the physical page count. Multiple versions of a container are used to preserve MROW reader views when there is an updater, to preserve the state of the entire FD when performing an on-line backup, and to store to-be committed data for long transactions that create/update more data than can fit in memory.

Note that if you are limiting both logical and physical pages, it is when the first limit is reached that a new container is created, so the other limit is obscured. So, for example, if you set your logical and physical page limits equal and you have large objects, MROW readers, do on-line backup, or have long transactions, then the physical page limit will be the limiting factor. For this reason it is not uncommon to have a physical page limit larger than the logical page limit, such that depending upon the specific circumstance, one or

the other limit may come into play.

In addition to allowing you to specify limits on logical and physical pages, object placement strategies allow you to specify the initial number of pages and the growth factor (by what percentage a container is grown when needed) for containers they create. There is only one initial number of pages setting, which affects logical and physical pages equally, and there is only one growth factor setting, which again affects logical and physical pages equally.

Growing a container is not the fastest operation, the page map must be extended, free pages in the database have to be allocated to the container, and if there are not enough free pages, the database file extended. How many times a container is grown is affected by the initial number of pages setting and the growth factor setting. In general, a small initial number of pages and a small growth factor will lead to a larger number of growths than a large initial number of pages and a large growth factor will.

While it is important for performance to provide initial number of pages and growth factor settings that will not lead to a large number of container growths, it is also important for performance to not have a large number of unused pages, so simply making these settings very large is not usually the answer. Here are some general guidelines:

- Set the initial number of pages as high as you can and still have less than 25% unused pages.
- Use a smaller growth factor with a large initial number of pages setting to minimize overshoot.
- Use a larger growth factor with a small initial number of pages setting to minimize number of growths.

Note that running ooTidy will remove all empty pages and thus interferes with your initial number of pages setting, so it is usually best to only run ooTidy on those databases which are mostly or completely populated.

One last thing to consider when setting the initial number of pages, growth factor, and logical and physical page limits is how they combine. The way the logical and physical page limits work is that once one is reached, a new container is created to place new objects into and the current one is no longer populated with new objects. If the combination of initial number of pages and the growth factor leads to the logical page limit being exceeded and not just met, then those extra logical pages are not going to be used. The same concern can apply to the physical page limit, but not always, as physical pages are used for many purposes beyond storing newly created objects. So you want to make sure the initial number of pages grown some number of times equals your page limits, or only exceeds them by a small amount.

Maximizing population performance

If your application needs to populate your FD at a rate greater than can be accomplished with a single process/thread, you can take advantage of Objectivity's distribution and multi-user support so that multiple processes and/or threads can populate it at the same time.

The degree of parallelism (thus performance) achieved depends upon the resources you provide and how each process/thread uses those resources. Minimum parallelism is achieved by having all processes/threads use the same CPU, disk and lock server. Maximum parallelism is achieved by giving each process/thread its own CPU, disk and lock server.

Parallelism is compromised by (in no particular order):

- Sharing CPUs.
- Sharing disks.
- Sharing lock servers.
- Sharing databases.
- Sharing containers.

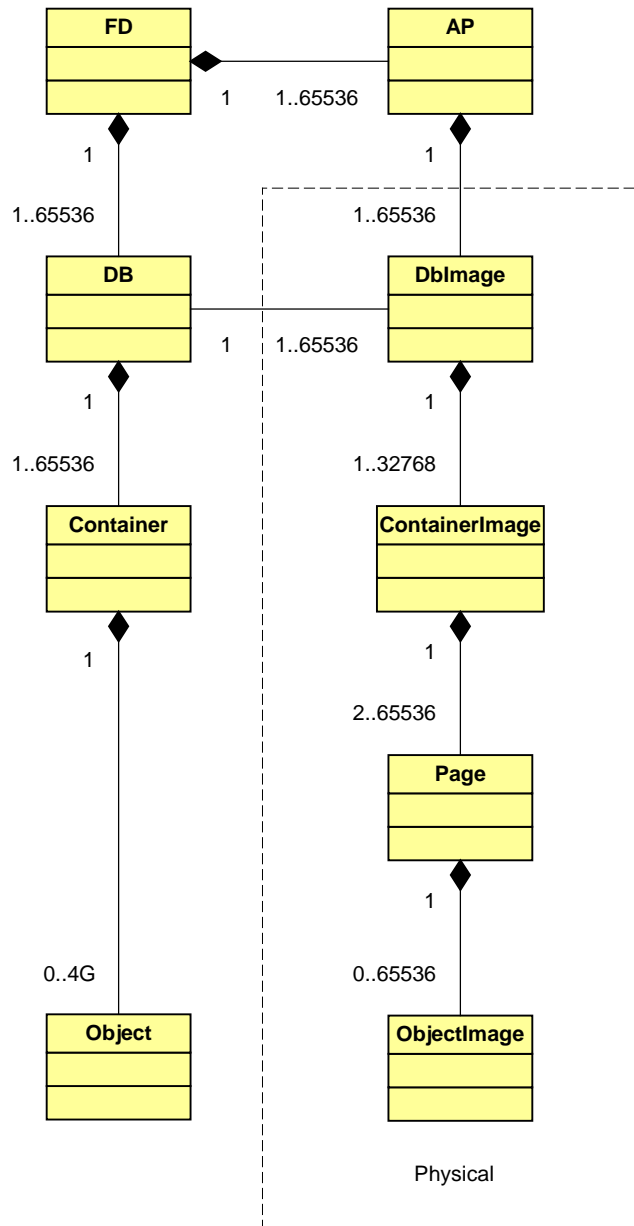
Real life resource constraints will usually lead to some degree of CPU, disk, and lock server sharing. Database and container sharing, though, are the result of physical design decisions. If maximum population rate is the most important performance characteristic of your system, you can then decide to separate concurrent populators into separate containers or databases. Doing so, though, tends to lead to data that is more spread out, which adversely affects read and general update performance for operations that use the data together after being populated. If that is the case, you have the following choices.

1. Organize your used together data for optimum read and single updater performance by placing it all together into the same container. This is done, of course, at complete sacrifice of parallel population performance.
2. Sacrifice some read performance and single updater performance and gain multiple populator and post population multiple concurrent updater performance by giving each populator its own container. Post population read and single updater performance is diminished because data is now more spread out. Post population multiple updater performance is improved by increased concurrency gained from multiple containers.
3. Again with a sacrifice in read performance, use a container pool to get equal separation for populators and multiple concurrent post population updaters.
4. Separate each populator into its own database for the maximum in population performance, sacrificing read and post population update performance.

If once populated your data is mostly or totally read only, then the choice is easier to make. Option 3 can be eliminated because it spreads data amongst containers for increased post population concurrent update performance, which is not a concern for read mostly data.

Going all the way to separating populators into separate databases has the additional benefit that those databases can be placed on separate disks, even hosts. Even if you don't initially have many resources, you can later scale your system effectively by adding more resources and moving databases to them.

Appendix A (Federated Database Storage Hierarchy)



(Figure 2-- Federated Database Storage Hierarchy)

Component	Description
FD (Federated Database)	An FD ties all databases into a logical whole; objects within any database can point to objects within any other, and applications work with a federated database as if it were a single database.
AP (Autonomous Partition)	An autonomous partition (AP) contains the actual database data, referred to as images. An AP is autonomous in that a client working with one AP's data does not necessarily need access to any others; each AP has all of the resources needed (lock server, schema and catalog) to work with its data. A single logical DB can be owned by a single AP or be replicated to multiple APs. A system DB that contains a catalog of user created databases and a language independent schema is automatically replicated across all APs. APs are typically used to represent data at one geographical location and/or for replication.
DB (Database)	A database (an awkward name as logically the FD is the database) contains containers. Each database has a default container that can be useful during early development just as somewhere to put objects before the physical design is devised. A DB has one or more images; when more than one; each is a replica of the other.
DB Image	Database images are the unit of distribution. Each database file can be independently placed on any host on the network – AP catalogs keeps track of where each is, so applications don't have to.
Container	<p>A container contains persisted objects, and is the unit of locking. A lock obtained for an object within the container is really a lock on the container, and all other objects in that container are effectively locked as well. Lock server traffic is greatly reduced, and hence performance improved, by organizing objects that are used together into the same container, as individual locks are avoided.</p> <p>As of Objectivity/DB version 9.0, a container can also be a unit of distribution, as individual containers can be designated to go into their own file.</p>
Container Image	Container images contain the actual container data (a replica or not) divided into logical pages. Each logical page uses one or more physical pages, more than one only when an object does not fit onto a single page.
Page	A logical page contains the persisted objects and is the unit of

	transfer between database hosts and clients.
Object	The logical object, which ultimately your application works with.
Object Image	The physical representation of an object on a container image page.