WHITE PAPER

Sybase® Unwired Platform 2.1

Mobile Business Object Best Practices

TABLE OF CONTENTS

Best Practices for Developing an Mobile Business Object Data Model

Define Mobile Business Objects so they can be efficiently consumed by native applications.

PRINCIPLES OF MBO MODELING

Understand key concepts required to develop an efficient data model.

⚠	Design the MBO model based on mobile application requirements, not on the EIS model.
✓	Design and implement an efficient data-retrieval API for your EIS to populate the MBOs in the cache, and return only what is required by the MBO.
⛔	Using existing EIS APIs for data retrieval simply because they already exist is inefficient because the EIS-model APIs were likely created for other purposes, such as desktop applications, making them inappropriate for mobile applications.
ℹ	Each MBO package is a client-side database. See *MBO Packages*.
⛔	Do not put more than 100 MBOs in a single package. Instead, use multiple packages.
✓	When modeling the MBO, remove unnecessary columns so they are not loaded into the Unwired Server cache (also called the cache database, or CDB). If you cannot remove these columns, use result-set filters to remove columns from the EIS read operation and to customize read-only data into a format more suitable for consumption by the device.

MBO Consumption

MBO data is consumed by the mobile application and has a direct impact on its performance. The mobile application operates around the MBO data model, and an inappropriate MBO data model impacts not only mobile application development and maintenance, but synchronization performance and reliability.

When defining the MBO data model:

1. Understand the requirements of the mobile application.
2. Ensure that it allows the mobile application to efficiently satisfy functional requirements.
3. Keep in mind that mobile devices, including tablets, are limited in terms of resources and capability. In most cases, it is inappropriate to make the EIS business object model available to the mobile application to use. While doing so may save time during MBO development, it can lead to extended testing and tuning, and potentially frustrate users.

MBO Read Definition

An MBO definition is derived from the result of a read API provided by the EIS, for example, a SQL SELECT statement. This API is usually developed specifically for mobilizing the data to be consumed by the mobile application. The API must be as efficient as possible to minimize impact to the EIS. While Unwired WorkSpace provides an easy way to consume existing back-end APIs, define the MBO based on application need instead of what is already exposed by the EIS. For example, in most cases, reusing an API developed for a desktop application is not a good choice for the mobile application.

To evaluate if an existing API is sufficient, consider whether it:

1. Returns as close to what the mobile application requires for the MBO.
2. Fills the CDB with as few interactions as possible that satisfies the data freshness requirements of the mobile application.

Result-Set Filters

If the EIS API returns more than what the MBO requires, use a custom result-set filter to remove unnecessary columns (vertical) or rows (horizontal). While a filter adds a certain amount of overhead, it is more efficient than moving redundant data through the mobilization pipeline and consuming resources on the device.

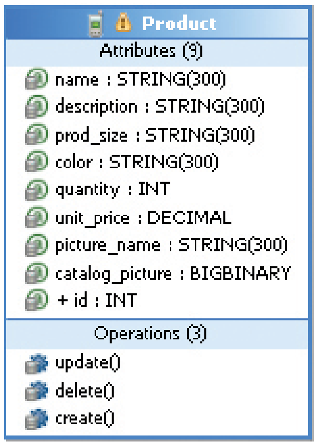
A result-set filter does more than remove columns; you can customize it to format read data, making the data more suitable for mobile application consumption. For example, some EISs store information as strings, which when converted to numeric values, reduces synchronization size and causes a mismatch between datatype and usage.

MBO Packages

A package translates to a database on the device with MBOs as tables. Updates to MBOs within the package can occur within a transaction on the device. In other words, a package defines a functional unit with multiple object graphs of MBOs. For convenience, the developer can put unrelated MBOs into a single package, however when the number of MBOs becomes large, it creates a maintenance problem. Changes to one set of MBOs impacts all others as the package needs to be redeployed. During deployment, the cache must be refilled.

MBO ATTRIBUTES

Understand key MBO attribute concepts, before you define them.

ℹ	An MBO instance equates to a database row, MBO attributes map to database columns, and in the database, the database row must be less than the page size. See *MBO Persistence on the Device.*
ℹ	If the application will be localized, increase row size to accommodate multibyte encodings. See *MBO Persistence on the Device.*
ℹ	A smaller page size generally produces better overall database and synchronization performance. See *MBO Persistence on the Device.*
⚠	If computed maximum rowsize is larger than specified page size, promotions of VARCHAR to LONG VARCHAR and BINARY to LONGBINARY occur until the row fits into the page.
✓	Keep MBOs as lean as possible to keep page size small.
✋	Do not define an MBO with more than 50 attributes.
✋	 Product Attributes (9) name : STRING(300) description : STRING(300) prod_size : STRING(300) color : STRING(300) quantity : INT unit_price : DECIMAL picture_name : STRING(300) catalog_picture : BIGBINARY + id : INT Operations (3) update() delete() create() For EIS operations where the maximum length information is not provided, the default is STRING(300). Always change this default value to match the expected maximum length using STRING(*n*), where *n* is the actual length of the STRING.

MBO Persistence on the Device

Every MBO instance is stored as a row in a table, and each attribute is represented by a column. Row size depends on the number and type of attributes in the MBO — the larger the MBO, the bigger the row. Since a row must fit within a database page, the page size is influenced by the largest MBO in the package. During code generation, Unwired Server computes the maximum row size of all MBOs to make sure they fit within the specified page size. The computation also takes into consideration the use of any multibyte encoding. In normal usage, the actual row size is often a fraction of the maximum row size. For example, an attribute that holds notes may be more than a thousand characters. Unless the MBO developer pays attention, it is easy for the maximum row size to be unreasonably large.

In the event that the maximum row size computed during code generation is larger than the specified page size, VARCHAR columns are promoted to LONG VARCHAR to move storage out of the row until the row fits in the page. This results in an indirection to access the data, and may impact query performance. To avoid indirect access:

1. Eliminate attributes not required by the application.
2. Reduce the maximum size. For example, the EIS may have a notes field with a 1000 character limit. Use a smaller size for the mobile application if possible.
3. Consider splitting the MBO if it contains many attributes.
4. Analyze data to determine the normal row size. Use a page size large enough to hold the row. Code the mobile application to configure the mobile database to run with this page size. During code generation, a much larger page size is used to avoid promotion, but the application runs using a small page size. A drawback to this approach is a possible synchronization failure when the actual data exceeds the page size specified during runtime. The developer must determine the minimal page size with the lowest probability that the actual data exceeds it.

Why Page Size Matters

Sybase testing indicates that a page size of 1 – 4KB provides the best overall performance. Start testing with these sizes, unless the MBO model requires a larger page size. The main issue with large page size is related to slow write performance of Flash-based file system/ object storage used by the device. The impact is most evident during synchronization, when the database applies the download transactionally.

MBO INDEXES

Understand how to maximize index efficiency use in mobile applications.

✓	Use the minimum number of indexes.
?	If findByPrimaryKey is not required to locate the MBO via the business key, disable generation.
?	If FindAll is not required, disable generation, except for MBOs with a low number of instances on the device.
?	Determine if an index is required for user-defined object queries.

Impact of Indexes on the Device

When performing updates, or during initial or large subsequent synchronizations, index maintenance is a significant performance consideration, especially on device platforms where all root index pages stay in memory. Even a small number of indexes impacts performance, even when they belong to tables that are synchronized. For a very small table, you may not need to use an index at all. When synchronization performance is slower than expected, evaluate how many indexes are deployed in the package.

Reducing Indexes

By default, two queries are generated for each MBO: findByPrimaryKey and FindAll. The primary key is the business key in the EIS. If the mobile application does not need to locate the MBO via its business key, disable generation of the findByPrimaryKey query. This is especially true for child MBOs that can navigate to the parent MBO and need not locate the parent via the primary key.

FindAll does not require any index as it scans through the table, instantiates, and returns all MBO instances as a list. Unless the number of MBO instances is small, the FindAll query is inefficient, and Sybase recommends that you disable its generation by unselecting the Generate FindAll query checkbox.

By default, developer-defined object queries create indexes. If the number of instances of the MBO is small, an index may not make much difference as far as performance, however, you should disable index creation for these object queries by unselecting the Create an index checkbox when synchronization performance is an issue.

MBO KEYS

Understand the purpose of primary and surrogate keys in Unwired Server.

ⓘ	Unwired Server uses a surrogate key scheme to identify MBO instances. See *Surrogate Keys.*
ⓘ	The CDB uses a primary key (EIS business key) to uniquely identify MBO instances. The primary key locates instances in the CDB to compare with the corresponding instance from a cache refresh or Data Change Notification (DCN). See *Primary Keys.*
ⓘ	The CDB creates an association between the surrogate and primary keys. See *Associating the Surrogate Key with the Primary Key.*
✋	Do not define a primary key for an MBO that is different from the EIS business key.
✋	Do not define an MBO without a primary key, or an implicit composite primary key that consists of all generated columns is assigned.
✓	Create an operation that returns an EIS business key (primary key) so the CDB can associate the newly created instance with the surrogate key.

Surrogate Keys

Each MBO instance is associated with two keys: surrogate and primary. The surrogate key scheme allows creation of instances on the device without having to know how the EIS assigns business keys. While business keys can be composite, the surrogate key is always singular, which provides a performance advantage. Unwired Server uses a surrogate key as the primary key for synchronization, and to keep track of device data set membership. It also serves as the foreign key to implement relationships on the device.

Primary Keys

Each MBO must have a primary key that matches the EIS business key. During data refresh or Data Change Notification (DCN), CDB modules use columns designated as the business key to locate the row in the CDB table for comparison to see if it has been modified. If the defined primary key does not match the business key, errors may result when merging the new data from the EIS with data already in the CDB. Additionally, an MBO without a primary key is assigned an implicit composite primary key consisting of all columns.

Associating the Surrogate Key with the Primary Key

The CDB associates the surrogate key with the EIS primary key:

1. A row (MBO instance) from the EIS is assigned a surrogate key to associate with the business key or primary key by the CDB.
2. An instance created on the device is assigned a surrogate key locally, and the CDB uses the primary key returned from the creation operation to form the association.
3. In the event that the creation operation does not return the primary key, the CDB module identifies the newly created instance as deleted to purge it from the device.
4. Eventually, whether through DCN or a data refresh, the created instance from the EIS is delivered to the CDB and subsequently downloaded to the device.

A drawback to this approach is that the newly created MBO instance on the device is deleted and replaced by a new instance with a different surrogate key. If DCN is used to propagate the new instance from the EIS, the mobile application may not be aware of it for some time. Depending on the use case, the momentary disappearance of the instance may not be an issue.

MBO RELATIONSHIPS

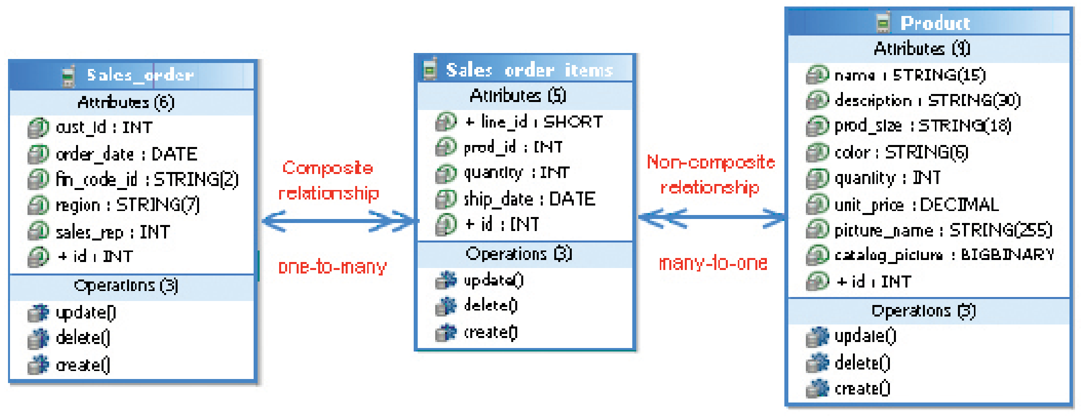
Understand the role of relationships in MBOs.

ℹ	Relationships provide navigation and subscription inheritance. See *Relationship Modeling*.
ℹ	Composite relationships provide navigation, subscription inheritance, and cascading create, update, and delete capabilities. See *Relationships and Primary Keys*.
🛑	Relationships should not span multiple synchronization groups.
🛑	Relationships should not span multiple cache groups.
⚠	Map relationships using all primary key attributes of the source in one-to-many and one-to-one relationships.
⚠	Map relationships using all primary key attributes of the target in many-to-one and one-to-one relationships.
⚠	One-to-many relationships when "many" is large may be expensive to navigate on the device.

Relationship Modeling

There are two types of relationships: composite and noncomposite.

- In a noncomposite relationship, targets are automatically subscribed to. That is, all instances related to the source are downloaded, for example, all products referred to by sales order items. In the same way, when the source is no longer part of the device data set, all instances of the target no longer referred to are removed. For many-to-one relationships like sales order items to product, only those products that have no referring sales order items are removed.
- In modeling terminology, composite means a whole-part relationship. Composite relationships offer cascading create, update, and delete capabilities. For delete operations, if the parent is deleted, all the target/child instances are deleted as well. For update and create operations, when the submitPending API is invoked by the mobile application, all modified/new instances are sent with the parent as part of the operation replay.



Both types of relationships provide bidirectional or unidirectional navigation through generated code with lazy loading, which means children/targets are not loaded unless they are referenced. However, due to instantiation costs, navigating to a large number of children can be expensive, especially on devices with limited resources. If the instantiated object hierarchy is very wide, the mobile application should consider other means to access the children.

Relationships and Primary Keys

A relationship is implemented on the device database and the CDB through foreign keys. In Unwired Server, the foreign key contains a surrogate key instead of an EIS business key or primary key. In a one-to-many relationship, the foreign key is in the target and it is impossible for it to reference more than one source. Therefore, the restriction of using a primary key of the source is to ensure there is only one source.

MBO SYNCHRONIZATION PARAMETERS

Understand the role of synchronization parameters in MBOs.

ℹ	A synchronization parameter is also referred to as synchronization key. See *Data Subscription*.
ℹ	A set of synchronization parameters is sometimes referred to as a subscription. See *Data Subscription*.
ℹ	Synchronization parameters allow the user/developer to specify the data set to be downloaded to the device. See *Subscribed Data Management*.
✓	Use synchronization parameters to retrieve and filter data for the device.
⚠	Understand how multiple sets of synchronization parameters or subscriptions impact the cache module to make sure it is a scalable solution.
✓	To increase the membership of the download data set, use multiple sets of synchronization parameters.
✓	To reclaim storage during runtime via exposed APIs, if needed, purge the collection of synchronization parameter sets.
✓	To retrieve data from the EIS, define synchronization parameters and map them to all result-affecting load parameters.
✓	Use synchronization parameters not mapped to load parameters to filter the data retrieved into the CDB.

Data Subscription

The mobile application uses synchronization parameters to inform Unwired Server of the data it requires. Instances of qualified MBOs that correspond to the synchronization parameters are downloaded to the device. The set of synchronization parameters creates a subscription that constitutes the downloaded data. Any changes to the data set are propagated to the device when it synchronizes. In other words, synchronization parameters are the facility that determines membership of the downloaded data set.

Data Retrieval

Synchronization parameters are used for data loading and filtering. When a synchronization parameter is mapped to a load parameter of the load/read operation of the MBO, it influences what data is to be retrieved. In this capacity, the synchronization parameter determines what is loaded into CDB from the EIS. Not all load parameters influence the data to be retrieved. For example, a Web service read operation of getAllBooksByAuthor(Author, userKey) uses the userKey load parameter only for authentication. Whoever invokes the operation retrieves all the books by the specified author. In this case, do not map a synchronization parameter to userKey. Instead, use a personalization parameter mapping to provide the user identity.

The key principle when mapping synchronization parameters to load parameters is to map all result-affecting load parameters to a synchronization parameter. Failure to do this results in constant overwriting or bouncing of instances between partitions in the CDB. Unwired Server uses the set of synchronization keys mapped to load parameters to identify the result set from the read operation.

Data Filtering

Synchronization parameters not mapped to load parameters are used to filter the data in the CDB. If the data in the CDB is valid, cache refresh is not required and Unwired Server simply uses the unmapped synchronization parameters to select the subset of data in the CDB for download. Consider this data filtering example that uses the product MBO:

Data Filtering Example

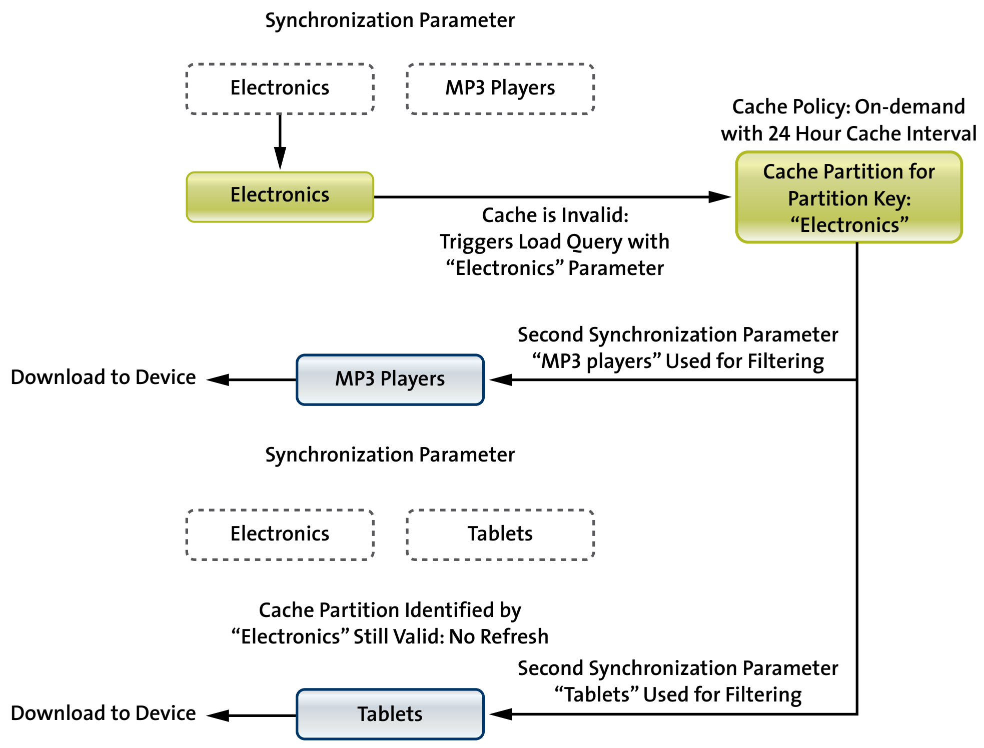
Read Operation: `getProducts(Category)`
Synchronization Parameters: `CategoryParameter, SubCategoryParameter`
Mapping: `CategoryParameter -> Category`

Events:

- Jane Dole synchronizes using(“Electronics”,“mp3player”) as parameters. CDBinvokes: getProducts(“Electronics”).
- John Dole synchronizes using (“Electronics”, “tablets”) as parameters.

CDB:

1. Uses an on-demand cache group policy with a cache interval of 12 hours.
2. For invocation one, a partition identified by the key "Electronics" is created with all electronics product. Unwired Server uses "mp3 player" as the selection criteria for the subcategory attribute and downloads only all mp3 players in the catalog.
3. For the second synchronization using "Electronics" + "tablet", since the partition identified by "Electronics" is still valid, Unwired Server uses "tablet" as the selection criteria for the subcategory attribute and downloads only all tablets in the catalog.


Synchronization Parameter
Electronics
MP3 Players
Cache Policy: On-demand with 24 Hour Cache Interval
Electronics
Cache Partition for Partition Key: "Electronics"
Cache is Invalid: Triggers Load Query with "Electronics" Parameter
Second Synchronization Parameter "MP3 players" Used for Filtering
Download to Device
MP3 Players
Synchronization Parameter
Electronics
Tablets
Cache Partition Identified by "Electronics" Still Valid: No Refresh
Second Synchronization Parameter "Tablets" Used for Filtering
Download to Device
Tablets

Subscribed Data Management

Synchronization parameter sets are cumulative; every new set submitted by the application/ user produces additional members in the download data set. You can take advantage of this to lazily add data on an as-needed basis. For example, a service engineer usually does not require all the product manuals associated with the assigned service tickets. Modeling the product manuals as an MBO that takes the manual identifier as a synchronization parameter enables the user to subscribe to a particular manual only when needed. The drawback to this approach is that it requires network connectivity to get the manual on demand.

It is important to understand the impact to the CDB for each additional subscription. Performance suffers if each subscription results in an invocation to the EIS to fill the CDB whenever the user synchronizes. For example, assume that the product manual MBO is in its own cache group with a cache interval of zero. The API to retrieve the manual uses the product manual id as the load parameter which is mapped to the synchronization parameter. If the service engineer has subscribed to five product manuals, during each synchronization, the cache module interacts with the EIS five times to get the manuals. The solution is to use a large non-zero cache interval as the product manuals seldom change. Always consider what the cache module has to perform in light of a synchronization session.

Over time, the number of product manuals on the device can grow to consume significant storage. Through the exposed API, the application can purge the entire set of subscribed manuals to reclaim resources.

MBO CACHE PARTITIONS

Understand how to effectively use cache partitions.

ℹ	By default, the CDB for the MBO consists of a single partition. See *Cache Partitions*.
ℹ	Partitions are created when synchronization parameters are mapped to load parameters. All such load parameters use the partition key to identify the partition. See *Partition Membership*.
✓	To increase load parallelism, use multiple partitions.
✓	To reduce refresh latency, use multiple partitions.
✓	Use small partitions to retrieve “real-time”data when coupled with an on-demand policy and a cache interval of zero.
✓	Use partitions for user-specific data sets.
✓	Consider the partitioning by requester and device ID feature when appropriate.
🛑	Do not create overlapping partitions; that is, a member (MBO instance) should reside in only one partition to avoid bouncing.
⚠	Avoid partitions that are too coarse, which result in long refresh intervals. Avoid partitions that are too fine-grained, which require high overhead due to frequent EIS/Unwired Server interactions.

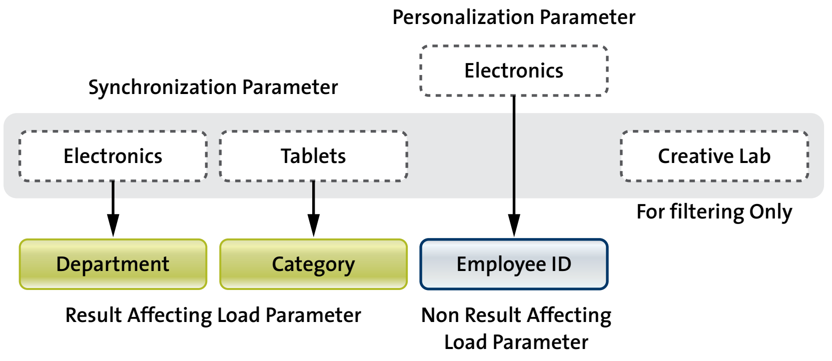
Cache Partitions

The CDB for any MBO consists of one or more partitions, identified by their corresponding partition keys:

- A partition is the EIS result set returned by the read operation using a specific set of load parameters.
- Only the result-affecting load parameters form the partition key.
- Using non-result-affecting load parameters within the key causes multiple partitions to hold the same data.

All result-affecting load parameters must be mapped to synchronization parameters to avoid this anomaly in the CDB:

```
Set of synchronizations mapped to load parameters = set of result affecting load
parameters = partition key
```


Personalization Parameter
Electronics
Synchronization Parameter
Electronics
Tablets
Creative Lab
For filtering Only
Department
Category
Employee ID
Result Affecting Load Parameter
Non Result Affecting Load Parameter

Partition Key = Department + Category

Partitions are independent from one another, enabling the MBO cache to be refreshed and loaded in parallel under the right conditions. If the cache group policy requires partitions to be refreshed on access, multiple partitions may reduce contention, since refresh must be serialized. For example, you can model a catalog to be loaded or refreshed using a single partition. When the catalog is large, data retrieval is expensive and slow. However, if you can partition the catalog by categories, for example, tablet, desktop, laptop, and so on, it is reasonable to assume that each category can be loaded as needed and in parallel. Furthermore, the loading time for each partition is much faster than a full catalog load, reducing the wait time needed to retrieve a particular category.

Partition granularity is an important consideration during model development. Coarse-grained partitions incur long load/refresh times, whereas fine-grained partitions create overhead due to frequent EIS/Unwired Server interactions. The MBO developer must analyze the data to determine a reasonable partitioning scheme.

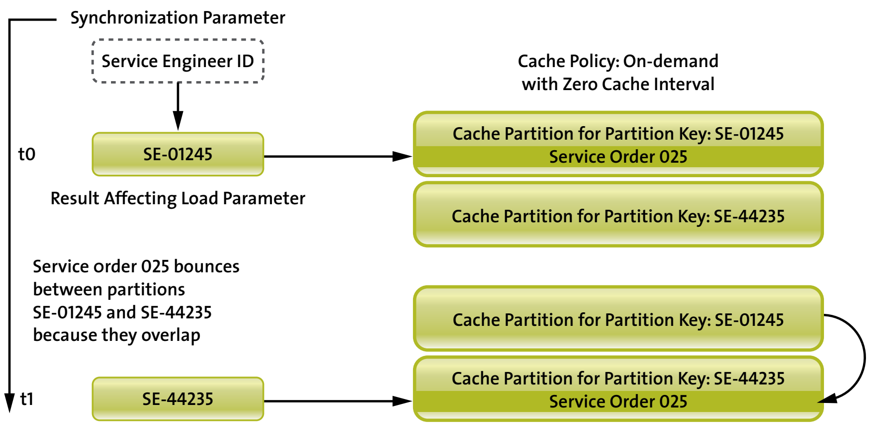
Partition Membership

Partitions cannot overlap. That is, a member can belong only to a single partition. Members belonging to multiple partitions cause a performance degradation, known as partition bouncing: when one partition refreshes, a multi-partition member bounces from an existing partition to the one currently refreshing. Besides the performance impact, the user who is downloading from the migrate-from partition may not see the member due to the bounce, depending on the cache group policy to which the MBO belongs.

An MBO instance is identified by its primary key which can only occur once within the table for the MBO cache. A partition is a unit of retrieval. If the instance retrieved through partition A is already in the cache but under partition B, the instance's partition membership is changed from B to A. It is important to understand that there is no data movement involved, just an update to the instance's membership information. As such, the migrated-from partition cache status remains valid. It is the responsibility of the developer to understand the ramification and adjust the cache group policy if needed.

Avoiding Partition Bouncing

In the following use case, where a service order is purposely assigned to multiple users and running on an On-demand policy with a zero cache interval, partition bouncing occurs when service engineer SE-44235 synchronizes at a later time. The service order is now on the devices of both engineers. However, consider the scenario where engineer SE-01245 also synchronizes at time t1. The synchronization process may find that service order 025 is no longer in the partition identified by its ID and results in a deletion from the data store on the device.


Synchronization Parameter
Service Engineer ID
Cache Policy: On-demand with Zero Cache Interval
t0
SE-01245
Cache Partition for Partition Key: SE-01245
Service Order 025
Result Affecting Load Parameter
Cache Partition for Partition Key: SE-44235
Service order 025 bounces between partitions SE-01245 and SE-44235 because they overlap
Cache Partition for Partition Key: SE-01245
t1
SE-44235
Cache Partition for Partition Key: SE-44235
Service Order 025

To avoid this problem, the primary key of the approval MBO can be augmented with the user identity. The result is that the same approval request is duplicated for each user to whom it is assigned. There is no partition bouncing at the expense of replication of data in the cache. From the partition's point of view, each member belongs to one partition because the cache module uses the primary key of the MBO to identify the instance and there can only be one such instance in the cache for that MBO.

To avoid partition bouncing:

1. Propagate the load parameter as an additional attribute (salesrep_id) to the Activity MBO.
2. Designate the primary key to be a composite key: activity_id and salesrep_id. This causes the cache module to treat the same activity as a different instance, avoiding bouncing at the expense of duplicating them in multiple partitions.

The previous example illustrates an MBO instance bouncing between partitions because they are assigned to multiple partitions at the same time. However, partition bouncing can also occur if load parameters to synchronization parameters are not carefully mapped. Consider this example:

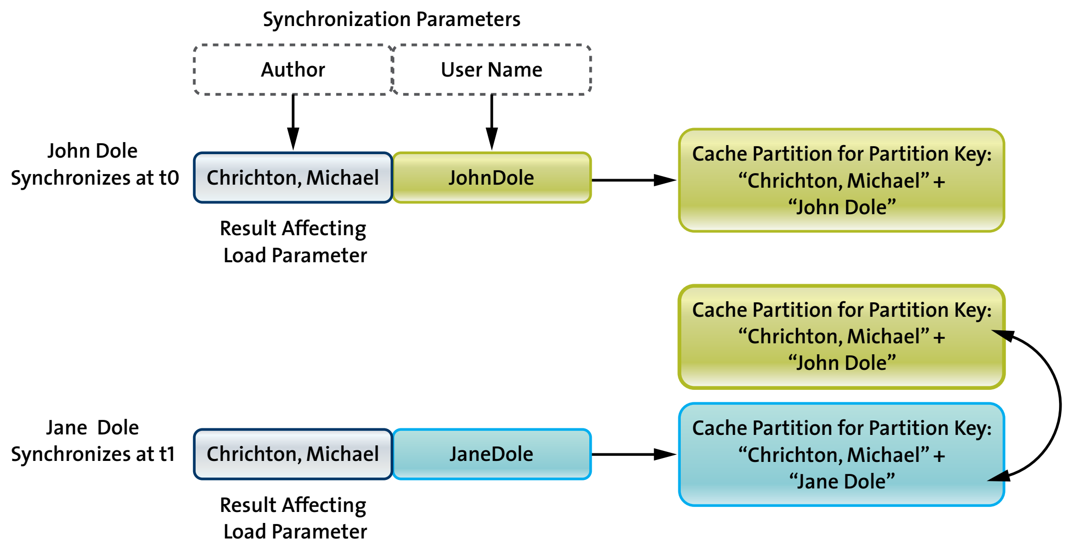
```
Read Operation: getAllBooksByAuthor(Author, userKey) Synchronization Parameters:
AuthorParameter, userIdParameter Mapping: AuthorParameter Author, userIdParameter
userKey
```

Events:

- Jane Dole synchronizes using ("Crichton, Michael", "JaneDole") as parameters Cache invokes: getAllBooksByAuthor("Crichton, Michael", "JaneDole")
- John Dole synchronizes using ("Crichton, Michael", "JohnDole") as parameters Cache invokes: getAllBooksByAuthor("Crichton, Michael", "JohnDole")

Cache:

1. For invocation one, a partition identified by the keys "Crichton, Michael"+ "JaneDole"is created.
2. For invocation two, a second partition identified by the keys "Crichton, Michael" + "JohnDole" is created.
3. All the data in the partition: "Crichton, Michael" + "JaneDole" moves to the partition: "Crichton, Michael" + "JohnDole".


Synchronization Parameters
Author
User Name
John Dole Synchronizes at t0
Chrichton, Michael
JohnDole
Cache Partition for Partition Key: "Chrichton, Michael" + "John Dole"
Result Affecting Load Parameter
Cache Partition for Partition Key: "Chrichton, Michael" + "John Dole"
Jane Dole Synchronizes at t1
Chrichton, Michael
JaneDole
Cache Partition for Partition Key: "Chrichton, Michael" + "Jane Dole"
Result Affecting Load Parameter

Cache Partition Overwrite

Partition overwrite is due to incorrect mapping of synchronization parameters and load parameters, and greatly hinders performance.

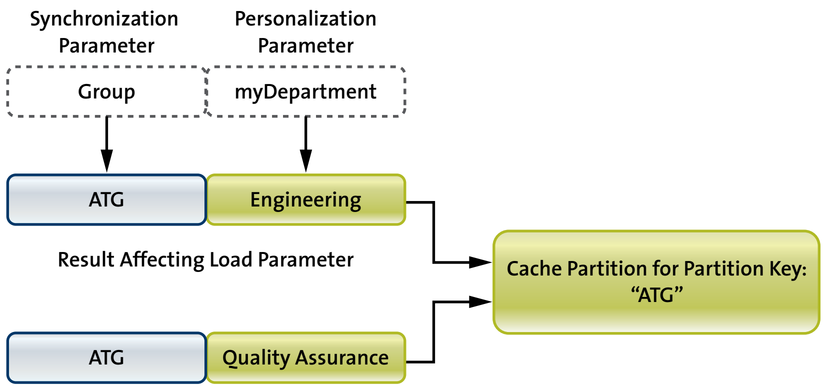
```
Read Operation: getEmployees(Group, Department) Synchronization Parameters:
GroupParameter Mapping: GroupParameter Group, myDepartment (personalization key)
Department
```

Events:

- Jane Dole synchronizes using ("ATG") as parameters and her personalization key myDepartment Cache invokes: getEmployees("ATG", "Engineering")
- Jane Dole synchronizes using ("ATG") as parameters and his personalization key myDepartment Cache invokes: getEmployees("ATG", "Quality Assurance")

Cache:

1. For invocation one, a partition identified by the key "ATG"will be created with employees from ATG Engineering department.
2. For invocation two, the same partition identified by the key "ATG" is overwritten with employees from ATG Quality Assurance department.
3. Not only is the cache constantly overwritten, depending on the cache group policy, one may actually get the wrong result.


Synchronization Parameter
Personalization Parameter
Group
myDepartment
ATG
Engineering
Result Affecting Load Parameter
Cache Partition for Partition Key: "ATG"
ATG
Quality Assurance

Partition key consists of load parameters that are mapped to synchronization parameters.
In this example, only one synchronization parameter is mapped to one of two load parameters, and the second load parameter maps to a personalization key.

EIS returns two distinct data sets identified by the same cache partition key, causing cache overwrite.

User-Specific Partitions

Partitions are often used to hold user-specific data, for example, service tickets that are assigned to a field engineer. In this case, the result-affecting load parameters consist of user-specific identities. Unwired Server provides a special user partition feature that can be enabled by selecting the "partition by requester and device ID " option. The EIS read operation must provide load parameters that can be mapped to both requester and device identities. The result is that each user has his or her own partition on a particular device. That is, one user can have two partitions if he or she uses two devices for the same mobile application. The CDB manages partitioning of the data returned by the EIS. The primary key of the returned instance is augmented with the requester and device identities. Even if the same instance is returned for multiple partitions, no bouncing occurs as it is duplicated.

Developers can easily define their own user-specific partition by providing appropriate user identities as load parameters to the EIS read operation.

Partitions and Real-Time Data

For some mobile applications, current data from the EIS is required upon synchronization. This implies that Unwired Server must retrieve data from EIS to satisfy the synchronization.

If the amount of data for each retrieval is large, it is very expensive. Fortunately, real-time data is usually relatively small so you can employ a partition to represent that slice of data. It is important that the MBO is in its own synchronization and cache groups, which allows a user to retrieve only the data required from the EIS and download it to the device upon synchronization. This use case requires an on-demand cache group policy with zero cache interval.

MBO SYNCHRONIZATION GROUPS

Understand how to effectively use MBO synchronization groups.

ⓘ	Synchronization groups allow MBOs with similar characteristics to be synchronized together. See *Synchronization Groups*.
ⓘ	Consider the cost of multiple synchronization sessions with a single synchronization group, versus a single session with multiple groups. See *Synchronization Sessions*.
✓	For flexibility, separate MBOs into appropriate synchronization groups.
✓	To implement synchronization priority, use synchronization groups; indicate which group to synchronize first.
✓	Use synchronization groups to break up large synchronization units into smaller coherent units to deal with low-quality connectivity.
✓	Use one synchronization session for multiple synchronization groups during runtime to reduce overhead if appropriate.
✋	Relationships should not span multiple synchronization groups.
⚠	Too many MBOs within a synchronization group defeats the purpose of using a group; limit groups to no more than five MBOs.

Synchronization Groups

A synchronization group specifies a set of MBOs that are to be synchronized together. Usually, the MBOs in a group have similar synchronization characteristics. By default, all MBOs within a package belong to the same group.

A synchronization group enables the mobile application to control which set of MBOs is to be synchronized, based on application requirements. This flexibility allows the mobile application to implement synchronization prioritization, for example, to retrieve service tickets before retrieving the product information associated with those tickets. Another advantage is the ability to limit the amount of data to be synchronized in case of poor connectivity. A large synchronization may fail repeatedly due to connectivity issues, whereas a smaller group may succeed.

Placing too many MBOs in a synchronization group may defeat the purpose of using groups: the granularity of the synchronization group is influenced by the data volume of the MBOs within the group, urgency, data freshness requirement, and so on. As a general guideline, limit synchronization groups to no more than five MBOs.

Synchronization Sessions

You can use multiple synchronization groups within a single synchronization session. If no synchronization groups are specified, the default synchronization group is used. A session with multiple groups is more efficient than multiple sessions using one group at a time. More sessions means more transactions and more overhead. Therefore, the mobile application developer should determine or allow the user to choose what to synchronize through an appropriate user interface. For example, when WiFi connectivity is available, the user can choose an action that synchronizes all MBOs together in one session. Specifying a synchronization session provides flexibility to both the application developer and user. This code snippet shows how to synchronize multiple groups within a single session.

```
ObjectList syncGroups = new ObjectList(); syncGroups.add
(CustomerDB.getSynchronizationGroup(“orderSyncGroup”) ); syncGroups.
add(CustomerDB.getSynchronizationGroup(“activitySyncGrou p”)); CustonmerDB.
beginSynchronize(syncGroups, “mycontext”);
```

Relationships and Synchronization Groups

Relationships that span multiple synchronization groups can cause inconsistent object graphs on the client database, which may in turn cause application errors and user confusion. In general, the only reason to have an object graph that spans synchronization groups is to implement a deferred details scenario. For example, in a field service use case, the engineer wants to review his latest service ticket assignments without worrying about all the relevant details associated with the ticket (product manuals). One obvious solution is to forgo the relationship since it is an association (in UML terminology). Although the navigation ability provided by the relationship has been sacrificed, it can easily be addressed by using a custom function that locates the product manual.

MBO CACHE GROUPS

Understand how to effectively use MBO cache groups.

ℹ	A cache group contains MBOs that all have the same load/refresh policy. See *Cache Groups in the Context of MBOs*.
ℹ	All MBOs within a cache group are refreshed as a unit for on-demand and scheduled cache policies. MBOs in DCN cache groups are updated via DCN messages in an ongoing basis. See *Cache Groups and Synchronization Groups*.
✓	Use cache groups to break up expensive data retrievals from the EIS.
✓	To avoid refreshing and retrieving MBOs not related to a synchronization group, map synchronization groups to cache groups.
🛑	Avoid circular dependencies between cache groups.
✓	Use one synchronization session for multiple synchronization groups during runtime to reduce overhead if appropriate.
🛑	Do not model relationships spanning multiple cache groups — in some cases, references may resolve incorrectly.

Cache Groups in the Context of MBOs

All MBOs within a cache group are governed by the specified cache group policy. A cache group defines how the MBOs within it are loaded and updated from the EIS to the CDB. With on-demand and schedule policies, all MBOs are loaded and updated as a unit. In general, the larger the unit, the longer it takes to complete. If data is not required all at once, you can employ multiple cache groups to break up an expensive load operation. This allows loading to occur in parallel with a lower overall wait time when compared to a large load. For example, load products and the product manual in separate cache groups. For very large data volume, the continual refresh cost may be too expensive, in which case a DCN cache group policy allows the MBOs in the cache group to be updated as changes occur in the EIS through notification messages.

Cache Groups and Synchronization Groups

Cache and synchronization groups are not related. While a synchronization group specifies which MBOs to synchronize, a cache group defines which MBOs are loaded into the CDB, and when. However, Sybase recommends that you coordinate a synchronization group with a corresponding cache group so only the synchronized MBOs are loaded or refreshed. If the MBOs in the synchronization group are members of a much larger cache group, the devices performing the synchronization may wait a long time for the entire cache group to refresh.

Circular Dependencies

Circular dependencies spanning cache groups are not supported.

Relationships and Cache Groups

The MBO developer must be cautious when defining relationships that span cache groups as this can cause references to be unresolved. For example, sales order has a one-to-many composite relationship with sales order items. Both MBOs are in separate cache groups. Assuming that both are loaded in-bulk (all instances retrieved through a single read). If they have different cache intervals, the sale order items cache group could refresh with orphaned instances but without corresponding sales orders. As a general rule, keep relationships within the same cache group.

SHARED-READ MBOS

Understand how to effectively use shared read MBOs.

ℹ	"Shared-read MBO" refers to MBOs that have caches that are populated by a shared read operation. See *Populating the Cache Via a Shared Read Operation*.
ℹ	All MBOs sharing the common read also share the same partition key — in other words, they are always loaded and refreshed together. See *Updating Shared Read MBOs*.
✓	Use shared read whenever possible to increase load efficiency.
✓	To implement transactional update, use client parameters for child objects with create and update operations on root MBOs.
✓	Use multilevel insert for create operations if the EIS does not support an interface where child objects can be sent as client parameters.
✓	For operations, use the "Invalidate the cache" cache update policy to force a refresh if the application requires a consistent object graph at the expense of performance.
✓	Use multiple partitions, if possible, to alleviate the cost of "Invalidate the cache," as only the affected partition needs to be refreshed.
✓	Always enable "Apply results to the cache" for a create operation to maintain surrogate key to (just returned) business key association.
⚠	Use the "Apply results to the cache" cache update policy for operations that are applicable only to the MBO instance on which the operation is executed. In case of an object graph, child MBOs are not updated.

Populating the Cache Via a Shared Read Operation

Shared read is an optimization that loads multiple MBOs with a single read operation. The more MBOs that share the common read, the better the performance. All the MBOs that share a common read operation also share the same partition key — the implication is that all MBOs within a partition refresh and load as a unit.

Updating Shared Read MBOs

There is no shared write or composite write operation to transactionally update shared read MBOs in an object hierarchy. However, a transactional update or create operation is still possible if the EIS provides an operation that takes dependent objects as client parameters. The application passes the children as client parameters prior to invocation. However, even if the result object hierarchy is returned, only the root object can be applied to the CDB. The CDB is inconsistent with the EIS until refreshed.

If the shared read MBOs are in a composite relationship and each one provides create and update operations, the results of all the operation replays can be applied to the CDB.

Note: A create operation uses multilevel insert support to update child attributes that require the parent's primary key. JDBC EISs support this type of interaction and may even be able to enroll in two-phase commit for a transactional update.

Addressing Inconsistency

Since it is not always possible to apply results to the CDB for all shared read MBOs in an object hierarchy, the developer must decide whether the temporary inconsistency is an issue for the user. Regardless of whether the CDB needs to be invalidated to force a refresh, the create operation's cache update policy should always enable "Apply result to cache" to allow the CDB to associate the returned business key with the surrogate key from the device. This enables the CDB to avoid having to delete the new object on the device in preparation for a new one from the EIS.

If the business requirement is such that the application must immediately have a consistent object graph, select the "Invalidate the cache" option for the operation's cache update policy to force an immediate refresh. To alleviate the impact of a refresh, use multiple partitions if possible. Instead of invalidating an entire MBO cache, only the partition involved is affected. This assumes that the CDB can detect the target partition based on the partition key of the returned results.

MBO AND ATTACHMENTS

Understand how to effectively include attachments in your MBO Model.

ℹ	Attachments can be large (photos, product manuals, and so on) and are not always required by the application, as is often the case with e-mail attachments. See *The Choice of Synchronization*.
ℹ	It is expensive to always synchronize large objects that are only occasionally required. See *Inline Attachments are Expensive*.
ℹ	Use a separate MBO to carry an attachment. See *Consider the Attachment as an MBO.*
⚠	Inline attachments can result in redundant data transfer to and from the device. In most wireless networks, upload is only a fraction of the advertised bandwidth, further exacerbating long synchronization time.
☑	Subscribe to required attachments on-demand through synchronization parameters.
ℹ	Use initial synchronization in a favorable environment to load required attachments (reference data) for full offline access. See *Offline Access Pattern*.
🛑	Do not include the attachment as an MBO attribute if the mobile application or EIS can update the MBO.

The Choice of Synchronization

Synchronizing data that is not required on the device impacts multiple components of data mobilization. However, there is no definitive solution for data that is only used occasionally, since you must take into account connectivity and demand patterns. In general, Sybase recommends that you defer transfers until required. The exception is in an offline mobile application. The developer must analyze business requirements and the environment when making the decision of when to synchronize, and how much data to synchronize.

Inline Attachments are Expensive

Regardless of the decision on synchronization, attachments should not be embedded inline with the MBO as an attribute. Attachments do not generally change, and having it inline results in high data transfer overhead. Updating the MBO can cause transfer of inline attachments even though they are not modified. The cost of uploading and downloading a large attachment can be significant. Updating the status of a service downloads the attachment again if it is handled inline. In most wireless networks, uploads are slower than downloads, so it is not advisable to upload attachments. The same is true for downloads. If the EIS updates regular attributes of the MBO instance, the attachment is downloaded again if it is handled inline. The convenience of having the attachment inline is rarely worth the cost of moving them through the wireless network.

Consider the Attachment as an MBO

Using a separate MBO to hold the attachment provides flexibility through synchronization parameters and synchronization groups. Modeling the attachment MBO to employ a synchronization parameter allows the application to subscribe to the Attachment when required. A separate synchronization group can hold the attachment MBO, which then can be prefetched or pulled on demand. Prefetching can usually be performed asynchronously without impacting usability of the mobile application. In addition, this pattern enables timely delivery of transactional information, for example, a service ticket by separating or deferring reference data.

Offline Access Pattern

For mobile applications that run in offline mode, on-demand access of attachments is not possible. In this case, it is better to bulk download all attachments during initial synchronization in a high quality connected environment. For example, through device cradle or WiFi connectivity. This approach is possible because attachments rarely change and occasional changes can be downloaded at specific times of the day. The cost of this approach is a more complex and longer application roll out cycle.

Best Practices for Loading Data From the Enterprise Information System to the Consolidated Database

Define MBOs so they efficiently load data from the enterprise information system (EIS) to the consolidated database (CDB).

UNDERSTANDING DATA AND DATA SOURCES

Designing an efficient data loading architecture for your MBOs requires a good understanding of the data to be mobilized and the data sources that provide that data.

While you can use Unwired WorkSpace to quickly create a working prototype, developing a production environment that is scalable requires careful planning and detailed knowledge of the data movement between the CDB and the EIS.

You must understand the characteristics of the data that is to be mobilized:

- Read/Write ratio — read-only, read/write, mostly read, mostly write
- Sharing — private versus shared
- Change source — mobile only, EIS only, mobile and EIS
- Change frequency
- Freshness expectation
- Access pattern — peak/valley or distributed
- Data volume

REFERENCE DATA	TRANSACTIONAL DATA
• Mostly read or even read-only	• Read and write
• Usually shared between users	• Usually private but can share with other users
• Generally updated by EIS	• Updated by both mobile application and back end possible
• Infrequent or scheduled changes	• Moderate change frequency
• Able to tolerate stale data	• High freshness expectation
• May be concentrated during initial deployment, occasional thereafter	• Access pattern varies depends on use case: morning/ evening,or throughout the day
• Large to very large data volume is possible	• Moderately low data volume (not including historic data which is considered as reference)

Table 1. Common data characteristics

Data Sources

It is important to understand how, what, and when data can be obtained from the EIS to fill the CDB. What are the characteristics of the EIS to consider for data loading?

- Efficiency of the interface:
  - Protocol — JCO, Web Services, JDBC
  - API — number of interactions required to retrieve the data
- Push or pull
- Reaction to peak load
- Availability of internal cache for frequently accessed data

GUIDELINES FOR DATA LOADING

Understand the guidelines for efficiently loading data from the EIS to the CDB.

ⓘ	Poor performance is often due to the manner in which data is loaded into the CDB. See *Data-Loading Design.*
ⓘ	MBOs that use DCN as the cache group policy have only one partition. See *DCN and Partitions.*
⚠	Use caution when recycling existing APIs provided by EIS for use by the mobile application. Adopt only after careful evaluation.

✓	Use an efficient interface (protocol) for high data volume, for example, JCO versus Web Services.
✓	Use DCN to push changes to avoid expensive refresh cost and differential calculation for large amounts of data.
✓	Use multiple partitions to load data in parallel and reduce latency.
✓	If initial data volume is very large, consider a scheduled cache group policy with an extremely large interval to pull data into CDB and then update via DCN. However, do this only with careful orchestration to avoid lost updates.
✓	Use cache groups to control and fine-tune how the manner in which MBO caches are filled.
✓	Use shared-read MBOs whenever possible.
✓	Improve update efficiency by using small DCN message payloads.
🛑	Do not mix DCN with scheduled or on-demand cache policies, except for one-time initial data load operations — thereafter, use only DCN for updates.

Data-Loading Design

Successful data-loading design requires careful analysis of data and data source characteristics, usage patterns, and expected user load. A significant portion of performance issues are due to incorrect data loading strategies. Having a poor strategy does not prevent a prototype from functioning, but does prevent the design from functioning properly in a production environment. A good design always starts with analysis.

Recycling Existing Artifacts

The most common mistake is to reuse an existing API without understanding whether it is suitable. In some cases, you can make the trade-off of using a result-set filter to clean the data for the MBO if the cost is reasonable. This filtering does not eliminate the cost of retrieving data from the EIS and filtering it out. Every part of the pipeline impacts performance and influences data loading efficiency. The best interface is always based on your requirement rather than a design intended for a separate purpose.

Pull Versus Push

Since push-style data retrieval is performed by HTTP with JSON content, optimized interfaces like JDBC or JCO are often more suitable for high-volume data transfer. Pull-style data retrieval requires the same amount of data to be transferred during refresh, and then compares changes with what is currently in the CDB. If data volume is large, the cost can be overwhelming, even with an optimized interface. DCN can efficiently propagate changes from the EIS to the CDB. However, mixing DCN and other refresh mechanisms is generally not supported. When refresh and DCN collide, race conditions can occur, leading to inconsistent data.

You can load data using a pull strategy, then switch to DCN for updates. The key is to make sure the transition between pull and push is orchestrated correctly with the EIS so updates are not missed between the time the pull ends and the push begins. Initial loading can be triggered by device users by way of the on-demand cache group policy, or with a scheduled cache group policy that has a very small interval, which then changes to an extremely large interval once data loads.

It is not advisable to use a very large DCN message for updates. Processing a large DCN message requires a large transaction, significant resources, and a reduction in concurrency.

Cache Group and Data Loading

Cache groups are the tuning mechanism for data loading. Within a package, there can be multiple groups of MBOs that have very different characteristics that require their own loading strategy. For example, it is common to have transactional and reference data in the same package. Multiple cache groups allow fine-tuning which data in a package is loaded into the CDB independent of other cache groups.

Using Cache Partitions

Cache partitions increase performance by enabling parallel loading and refresh, reducing latency, supporting on-demand pull of the latest data, and limiting scope invalidation. You must determine whether a partitioned-cache makes sense for the mobile application. The mobile application may not be able to function without the entire set of reference data, and partitioning is a viable alternative. However, even if a cache partition is not the right approach, it may still be worth considering if you can apply the concept of horizontal partition. A cache partition uses vertical partitioning. In horizontal partitioning, with a hierarchy, you may not need to load the entire object graph to start as long as some levels can be pulled on demand. By using additional cache groups, you can potentially avoid a large data load.

A cache partition is a set of MBO instances that correspond to a particular partition key. The loading of the MBOs is achieved through synchronization parameters mapped to result affecting load parameters.

DCN and Partitions

There is only one partition for the DCN cache group policy. When a synchronization group maps to a DCN cache group, the synchronization parameters are used only for filtering against the single partition. In addition, the single partition of the MBO cache in the DCN cache group should always be valid, and you should not use an "Invalidate the cache" cache update policy for any MBO operations.

REFERENCE DATA LOADING

The strategy for reference data loading is to cache and share it.

- Usually read or read-only
- Shared between users in majority of cases
- Usually updated by the EIS
- Infrequent or scheduled changes
- Ability to tolerate stale data
- May be concentrated during initial deployment, and occasional thereafter
- Large to very large data volume is possible

The more stable the data, the more effective the cache. Once the data is cached, Unwired Server can support a large number of users without additional load on the EIS. The challenge with reference data is size and finding the most efficient method for loading and updating the data.

Load via Pull

Configure Unwired Server components to pull data into the CDB from the EIS:

✓	Partition data, if possible, within an MBO or object hierarchy.
✓	Load partitions on demand to spread the load, increase parallelism, and reduce latency.
✓	Use a separate cache group for each independent object hierarchy to allow each to load separately.
✓	Use a scheduled policy only if the back end updates data on a schedule; otherwise, stay with an on-demand cache group policy.
✓	Use a large cache interval to amortize the load cost.
✓	Use the "Apply results to the cache" cache update policy if the reference MBOs use update or create operations.
⚠	Consider DCN for large data volume if cache partition or separation into multiple cache groups is not applicable or ineffective. This avoids heavy refresh costs.
⚠	Use DCN if high data freshness is required.
⚠	Server-initiated synchronization (SIS) is challenging due to cache interval settings and require user activities to refresh for on-demand cache group policy. Change detection is impossible until the cache is refreshed.
🛑	Do not use a zero cache interval.

Load via Push

Configure Unwired Platform components and the EIS so that the EIS can push data changes to EIS:

✓	Use parallel DCN streams for initial loading.
✓	Use Unwired Server clustering to scale up data loading.
✓	Use a single queue in the EIS for each individual MBO or object graph to avoid data inconsistency during updates. You can relax this policy for initial loading, as each instance or graph is sent only once.
✓	Use a notification MBO to indicate loading is complete.
✓	Adjust change detection interval to satisfy any data freshness requirements.

Parallel Push

DCN requests are potentially processed in parallel by multiple threads, or Unwired Server nodes in case of clustering. To avoid a race condition, serialize requests for a particular MBO or an MBO graph. That is, send a request only after completion of the previous one. This ordering must be guaranteed by the EIS initiating the push. Unwired Server does not return completion notification to the EIS until the DCN request is fully processed.

Hybrid Load — Initial Pull and DCN Update

⚠	Ensure that the end of the initial load and start of the DCN update is coordinated in the EIS to avoid missing updates.
✓	Use parallel loading via multiple cache groups and partitions. Once the DCN update is enabled, there is always a single partition.

PRIVATE TRANSACTIONAL DATA LOADING

Use either pull or push loading strategies for private transaction data.

- Read and write
- Can be updated by both the mobile application and EIS
- Moderate change frequency
- High freshness expectation
- No sharing between users
- Access pattern varies depending on use case: morning/evening or throughout the day
- Moderately low data volume (not including historical data which is considered as reference)

If data freshness and consistency is of high priority, then an on-demand cache group policy is the more suitable approach. DCN, however, allows for change detection and server-Initiated synchronization (SIS) without additional work.

Load via Pull

✓	Partition per user using either the "Partition by requester and device identity" feature in the cache group policy, or a specific identity provided by the developer.
✓	Use an on-demand cache group policy with a zero cache interval for consistency and high data freshness.
✓	Use the "Apply results to the cache" cache update policy if there are create operations that associate a surrogate key and a business key.
✓	Use a notification MBO to implement SIS if required.
⚠	Ensure that all members of a partition belong to only one partition.
⚠	Ensure the EIS can support peak on-demand requests based on expected client load.
🛑	Do not use a scheduled cache group policy.

Load via Push

✓	Use parallel DCN streams for the initial load if there is a large user population.
✓	Use Unwired Server clustering to scale up data loading.
✓	Use a single queue in the EIS for each individual MBO or object graph to avoid data inconsistency during updates. You can relax this policy for initial loading, as each instance or graph is sent only once.
✓	Use a notification MBO to signal that initial loading is complete.
✓	Adjust the change detection interval to satisfy any data freshness requirements.
✓	Always use the "Apply results to the cache" cache update policy for all operations.

SHARED TRANSACTIONAL DATA LOADING

Shared transactional data has a higher chance of being stale until it becomes consistent again.

Multiple users can update the same instance leading to race conditions. While it is possible to provide higher consistency through the On-Demand policy with a zero cache interval, the cost can also be high, depending on the data volume involved in a refresh. This approach is feasible if the use case is such that each user is retrieving 10-20 object graphs shared with other users. A use case that leverages a user identity partition means that an instance belongs to multiple partitions as it is shared. For example, an approval request that is assigned to two managers shows up in two partitions. This condition violates the restriction that each member can only belong to one partition. It also means that the member bounces between partitions. To resolve this, add the user identity as part of the primary key so the cache sees a unique instance for the partition. In this scenario, the load parameter corresponding to the user identity should be propagated to an attribute. The other alternative is to use the "partition by requester and device id" option in the cache group settings. With this setting, the instance identity is automatically augmented with the requester/device ID so there is never partition bouncing.

Load via Pull - High Consistency

✓	Use an On-Demand cache group policy with a zero cache interval.
✓	Use "Apply results to the cache" cache update policy if create operations form the association between the surrogate key and business key.
✓	Use a notification MBO to implement server initiated synchronization (SIS) if required.
✓	Augment the primary key to avoid instance bouncing between partitions at the expense of duplication.
⚠	Ensure the EIS can handle peak on-demand requests based on expected client load.

Load via Pull

✓	Use an On-Demand cache group policy with a non-zero cache interval.
✓	Use "Apply results to the cache" cache update policy for all operations. Users see each others changes.
⚠	SIS is activated by changes made by other users. Changes from the EIS are only detected at expiration of the cache interval.
🛑	Do not use a partition since it creates duplicate copies of valid data until the interval expires. This creates further inconsistency between users.

Load via Push

✓	Use a single queue in the EIS for each individual MBO or object graph to avoid data inconsistency during updates. This is not required for initial loading since each instance/graph is sent once.
✓	Use a notification MBO to indicate initial loading is complete.
✓	Adjust the change detection interval to satisfy data freshness requirements.

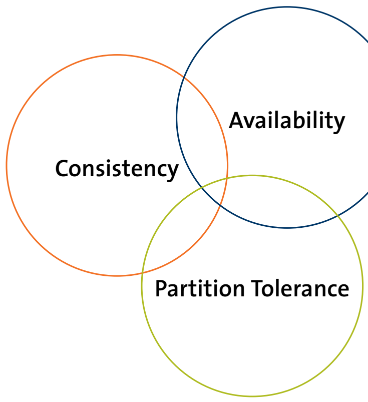
UNWIRED SERVER CACHE

The Unwired Server cache (or cache database CDB) caches data required by device applications.

A typical hardware cache has only two states: valid and invalid. When invalid, the data in the cache row is no long relevant and can be overwritten. The cache contains data needed by the processor at a given time. Data can be brought into and evicted from the cache rows rapidly.

The CDB, however, is filled with data required by the devices. Filling can occur all at once (DCN) or over time (on-demand). There is no eviction. In the case of a cache group policy that uses a pull mechanism, even if the CDB or a cache partition is invalid, the data is still relevant; The policy is used to detect changes when compared with new data from a refresh.

Whereas data in a hardware cache is always consistent with, or even supersedes data in the memory subsystem, data in the CDB does not. In database or application terminology, it is not the system of record, and is not guaranteed to be consistent with the EIS. This is neither a design nor implementation flaw, but is intended to avoid tight serialization and scalability problems. The CDB operates under the principle of eventual consistency. In a distributed environment, it is impossible to provide all three guarantees simultaneously: consistency, availability, and partition tolerance:



When data resides on the device and operates in a partitioned mobility environment (tolerant to network outage), the choice is whether to have consistency or availability. To enable mobile users to perform their tasks even without connectivity, the choice is availability. Hence, there is no consistency guarantee between the CDB and the mobile databases. The relationship between CDB and EIS is somewhat different. In general, there is no expectation of a partition between the CDB and the EIS so there is no need for partition tolerance. However, achieving both consistency and availability means a tight coupling between them and the integration is either too costly or invasive. This is why the CDB is never considered the system of record.

You can configure the cache group, by way of a cache group policy, to function in a high-consistency or flow-through mode where data always has to be fetched from the EIS. The data residing in the CDB is used only to detect data changes, so only the difference is transferred to the device.

CACHE GROUP POLICIES

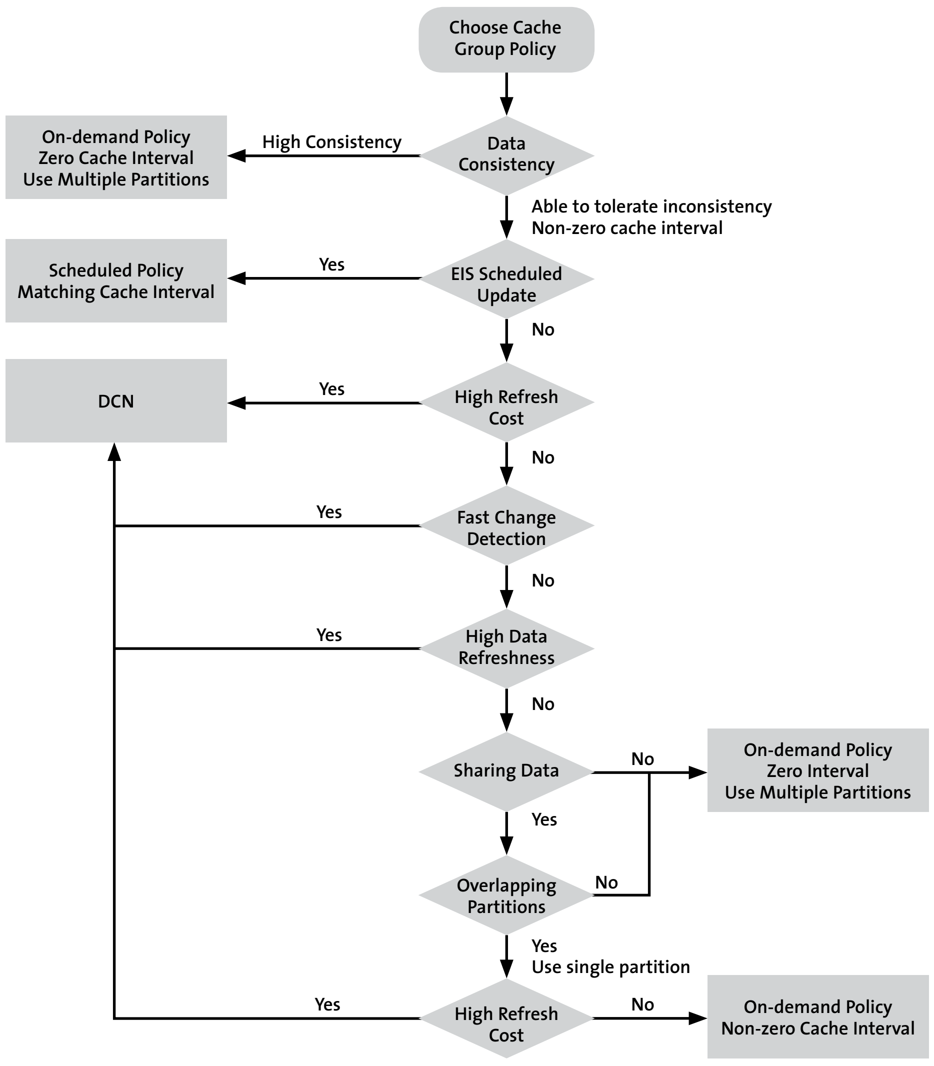
Understand the role of cache group policies and the effect of cache refresh.

Unwired Platform supports four cache group policies, three of which are relevant to loading data into the cache. The Online cache group policy is used only by workflow applications that require access to non-cached data:

- Scheduled and On-Demand policies are based on data retrieval APIs exposed by the EIS.
- DCN utilizes notification messages from the EIS that pushes both initial data and changes to the CDB.

The cache group policy for a particular cache group depends largely on the characteristics of the data contained within. In some cases, you can choose multiple policies. In general, if the EIS can push and data inconsistency can be tolerated, DCN is the appropriate choice.

The following flow chart attempts to capture the logic in choosing an appropriate cache group policy. However, use this as a reference only. Actual requirments vary due to many factors, and you should test your cache groups in a realistic test environment to understand timing, data flow cost, EIS load and user experience.


Choose Cache Group Policy
Data Consistency
High Consistency
On-demand Policy Zero Cache Interval Use Multiple Partitions
Able to tolerate inconsistency Non-zero cache interval
EIS Scheduled Update
Yes
Scheduled Policy Matching Cache Interval
No
High Refresh Cost
Yes
DCN
No
Fast Change Detection
Yes
No
High Data Refreshness
Yes
No
Sharing Data
No
On-demand Policy Zero Interval Use Multiple Partitions
Yes
Overlapping Partitions
No
Yes Use single partition
High Refresh Cost
Yes
No
On-demand Policy Non-zero Cache Interval

Sybase, Inc.
Worldwide Headquarters
One Sybase Drive
Dublin, CA 94568-7902
U.S.A
1 800 8 sybase

www.sybase.com

SYBASE® An SAP Company