SYBASE®

An **SAP**® Company

Mobile Data Models: Using Mobile
Business Objects

Sybase Unwired Platform 2.2
SP02

Contents

Introducing Mobile Business Object Data Models

This guide provides information about how to develop mobile business objects (MBOs) to fully maximize their potential, concepts you should know before developing MBOs, and how to use the Sybase® Unwired Platform Server API to customize MBO behavior. The audience is mobility architects and advanced developers who work with APIs, but who may be new to Sybase Unwired Platform.

Companion guides include:

- *Fundamentals* – provides high-level mobile computing concepts, and a description of how Sybase Unwired Platform implements the concepts in your enterprise.
- *Sybase Unwired WorkSpace - Mobile Business Object Development* – provides the procedures required to develop your MBOs.
- *Javadocs* – provide a complete reference to the APIs.
- **Device platform-specific** *Developer Guides* – provide details about how to develop native applications from generated Object API code.
- *Developer Guide: Migrating to Sybase Mobile SDK* – provides information about migrating applications and artifacts from previous versions of Sybase Unwired Platform to the current version.
- *Supported Hardware and Software* – provides information about the various integrated and third party hardware and software components and versions supported by this version of Sybase Unwired Platform.

See *Data Modeling* for information about fundamental data modeling concepts and practices.

Mobile Business Object Overview

The cornerstone of the solution architecture is the mobile business object (MBO). For native Object API applications and Hybrid Apps, mobile business objects form the business logic by defining the data you want to use from your back-end system and exposing it through your mobile application or Hybrid App.

MBO development involves defining object data models with back-end EIS connections, attributes, operations, and relationships that allow filtered data sets to be synchronized to mobile devices. MBOs are built by developers who are familiar with the data and transactional requirements of the mobile application, and how that application connects to existing datasources.

An MBO is derived from a datasource, such as a database server, Web service, or SAP® server. MBOs are deployed to Unwired Server, and accessed from mobile device application client code generated from Sybase Unwired WorkSpace or by using command line tools. MBOs:

- Are created using the Sybase Unwired WorkSpace graphical tools, which simplify and abstract back-end system connections, and provide a uniform view of transactional objects
- Are reusable, allowing you to leverage business logic or processes across multiple device types.
- Future-proof your application; when new device types are added, existing MBOs can be used.
- Provide a layer of abstraction from the Unwired Server interaction with heterogenous back ends/devices, as shown in the following diagram.

SAP Mobile Platform
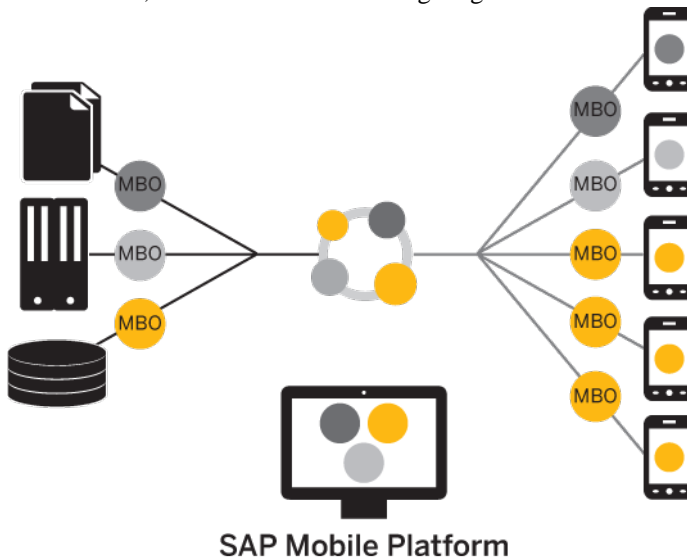
MBOs can include:

- Implementation-level details – metadata columns that include information about the data from a datasource.
- Abstract-level details – attributes that correspond to instance-level properties of a programmable object in the mobile client, and map to datasource output columns. Parameters correspond to synchronization parameters on the mobile client, and map to datasource arguments.

  MBO operations include arguments that map to datasource input parameters. The argument's value that is passed to the enterprise information system (EIS) at runtime can come from an MBO attribute, personalization key, client parameter, or a default/constant value.
- Relationships – defined between MBOs by linking attributes and load arguments in one MBO to those in another MBO.

Developers define MBOs either by first designing attributes and load arguments, then binding them to a datasource; or by specifying a datasource, then automatically generating attributes and load arguments from it.

A mobile application package includes MBOs, roles, datasource connection mappings, cache policies, synchronization related information, and other artifacts that are delivered to the Unwired Server during package deployment.

When the data model is complete, code artifacts are generated. The MBO package, containing one or more MBOs is deployed to Unwired Server. Other MBO artifacts are used to develop a mobile application using Native Object API or Hybrid Web Container API — when the application is deployed to a device, the MBO data model set resides on the device (in API code form). On-device data changes are synchronized to the MBO on the server, then to the EIS backend. Backend changes are in turn communicated to the device via the MBO on the server that sends a notification to the device and updates the MBO data on the device.

Data Modeling

After connecting to datasources using connection profiles from Sybase Unwired WorkSpace, the MBO developer defines the mobile interaction pattern, which typically involves selecting data subsets.

The MBO is developed to define each data subset, describe the data and the operations on the data.

- The data subset is deployed with the MBO package to Unwired Server where the MBO manages synchronization between the EIS and Unwired Server.
- Artifacts generated from the MBO are then used to develop the mobile application, typically further defining the data subset for data representation on the device.
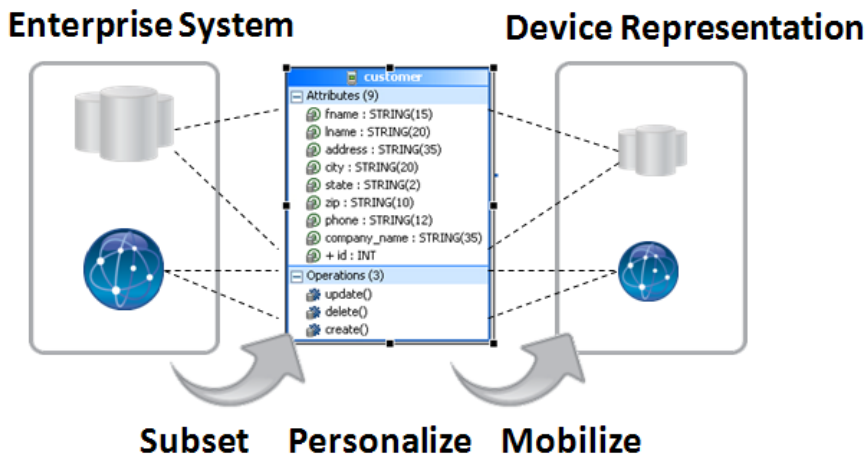
Datasources

A datasource is the enterprise information system (EIS) where data is retrieved from and transactions are executed. A connection profile is a design-time connection to a datasource. Connection profiles are created to specific datasources by providing connection information such as host, port, login, and password among others. The connection profiles are used to define MBOs and operations, and mapped to existing, or used to create new, server connections when the package is deployed to Unwired Server.

Unwired Platform hides the interaction complexity with datasource-specific protocols, such as JDBC™ for database and SOAP for Web services.

Unwired Platform supports multiple EIS connection types. See *Supported Hardware and Software* for information.

MBO Attributes

MBO attributes define the structure for the data associated with the MBO instance on the mobile device.

Attributes define the scope of the device-side data store. Attributes and load arguments in an MBO define the server cache. The server cache and device data store are populated by reading data from the EIS using exposed services or standard protocols and methods, such as a SQL select statement for a database datasource.

Attributes also have additional metadata provided during its Read/Load operation, such as specification of a load argument and synchronization parameter option that can be used to provision user-specific data, that is, the load arguments narrow down the data downloaded to the CDB, and the synchronization parameter filter the CDB data further when downloading to the mobile device. For example, a sales representative in the Eastern region may be interested

only in seeing data for that region. Developers can build the application to map those preferences to MBOs to drive the data filtering specific to the user. The same data obtained from the EIS is then further partitioned and used to serve all users, thereby optimizing requests for data to the EIS and improving performance of the mobile application.

Operations

MBOs may incorporate operations that change the data retrieved from the enterprise information system (EIS).

- Create, Update, Delete (CUD) operations – an operation definition contains arguments that map to the arguments of the EIS operation, and can create, update, or delete data. These operations cause state change.

  Operation replay executes the client-initiated transactions (Create, Update, and Delete operations) and propagates those changes to the EIS.
- Read/Load – an operation that includes optional load arguments that determine initial loading of data from the EIS to the Unwired Server cache. For example, a SQL Select statement for database datasources.
- Entity Read – an operation's cache policy determines how the results of operation replay are applied to the Unwired Server cache. An Entity Read operation can be defined to refresh a single instance of an MBO, including an MBO object graph (all MBOs linked by composite relationships).
- OTHER operation – an operation definition for operations other than create, update, or delete. These operations do not cause state change.

The operation definition supports validation and error handling.

Relationships

Relationships define the data association between two MBOs by linking attributes and load arguments (the read operation's parameters) in one MBO to attributes and load arguments in another MBO.

Relationships help provision related data as one unit, and properly sequence the operations on the related MBOs.

Other Key Concepts

Other key concepts for understanding mobility include object queries, synchronization parameters, result set filters, result checkers, and personalization keys.

- Object queries – a SQL statement associated with a mobile business object (MBO) that runs on the client against data that was previously downloaded to the device. The object query searches the device database and returns a filtered result set (such as a single row of a table). Object queries enable discrete data handling.
- Synchronization parameters – metadata that defines how the values provided by the device application client are used to filter data already existing in the CDB, which is downloaded to the device, to provide data of interest to the user. Synchronization parameters may be

     used to retrieve the cached data, or mapped to load arguments to filter the data that is cached for the MBO and then served to the client application.

- Result-set filters – a Java API used to customize the data returned by an enterprise information system (EIS) operation. Developers can use result-set filters to alter or manipulate returned data and its structure, that is, MBO attributes, before it is placed in the server's cache.
- Result checkers – a Java API that implements operation execution checks, and error handling logic for the MBO operation. Developers can use result set checkers to implement customized error checking for MBOs, since not all MBOs use a standard error reporting technique.
- Personalization keys – metadata that enables users to store their name/value pairs on the device persistent store, the client application session (in memory), or the Unwired Server, and use the personalization keys to provide values to load arguments, synchronization parameters, operation arguments, or device application business logic specific usages. To use a personalization key for filtering, it must be mapped to a synchronization parameter/ load argument.

     The developer can define personalization keys for the application, or use two built-in system defined personalization keys—username and password—to perform single sign-on from the device application to the Unwired Server, authentication and authorization on Unwired Server, as well as connecting to the back-end EIS using the same set of credentials. The password is never saved on the server.

Data Caching

Data caching is initial loading, or filling, the Unwired Server cache with enterprise information system (EIS) data, then continuing to refresh the cache with changes from the EIS or mobile device on an ongoing basis.

Since continual synchronization of the data between the EIS and device puts a load on the EIS, Unwired Platform provides several options for loading and refreshing the data cache.

Options include narrowing the EIS data search so that only specific data is retrieved (based on load arguments), identifying effective policies for handling data updates once operations are performed (based on an operation's cache policy), scheduling periodic updates to occur when system usage is low (based on cache group and refresh policy), updating only changed data in the cache, and so forth.

You can use multiple options to load and refresh the right data at the right time, and to deliver the smallest, most focused payload to the mobile device.

The primary loading schemes provide a trade-off between time and storage space. For example, bulk loading takes more time because data is loaded for all users, but once loaded, the data can be shared between users.

- Bulk loading – data for all users is loaded in bulk.

- Client defined partitioned data loading – only user relevant data, or partition, is loaded, based on MBO load arguments.
- EIS defined partitioned data loading – only user relevant data, or partition, is loaded, based on the DCN `partition` attribute.
- On-demand loading – data is not filled, until the client performs synchronization.
- Scheduled loading – data is filled periodically according to a schedule you set.
- Event notifications – data is pushed by the EIS when specific events occur.

Data change notification (DCN) facilitates propagation of data changes from the back-end enterprise information system (EIS) to the Unwired Server interface for any MBO. The DCN payload containing changed data is applied to the cache and the change gets published to subscribing device users based on a change detection interval time. DCN is used for cases where the cache refresh may be expensive due to volume. This option is the least intrusive and most optimal for addressing high-load environments and optimizing the load on the EIS to keep the Unwired Server cache consistent.

Object API Code Generation

To access and integrate MBOs in a device application, developers generate object code for the target device platform, and then use their IDE of choice to build the native device application. The object code generation step is the bridge from the Unwired Server server-side development (MBOs) to client-side development (device applications).

The generated object code for each MBO follows a standard pattern of properties for attributes, operations, and abstracts persistence and synchronization. Object code generation is supported in the native language for each target platform. Unwired Server client libraries complement and are required for the generated object code, which together are used in the device application.

The generated code is built upon the Mobile client libraries, which combine support for reliable transmission of data and transactions, security of data while in transit or on device, sending notifications when data changes occur in the back-end application, consistent interface on all platforms, all of which abstract developers from mobility related complexities.

Code generation is supported for these platforms: BlackBerry and Android (Java), iOS (Objective-C), Windows Mobile and Windows (C#).

See *Supported Hardware and Software* for the most current version information.

Package Deployment

The last step of mobile business object (MBO) development is to deploy the MBO definitions to Unwired Server as a deployment package.

When you deploy MBOs to the Unwired Server, you are deploying:

- MBO definitions including attributes, operations, connections, synchronization groups, and cache configuration as defined in the package.
- MBO custom code related to result-set filters and result checkers.
- Other functionality captured in the MBO model.

MBOs are deployed using a deployment wizard through which you can make the choices that are appropriate for application requirements. Developers use Sybase Unwired WorkSpace to deploy a package.

The production administrator can deploy from a wizard using the web-based management console, or from the command line. Deployment-time tasks include choosing:

- Target domain – logical container for packages.
- Security configuration – used for authentication and authorization of users accessing the package.
- Server connections mapping – to bind MBOs design-time datasources to production datasources.
- Application ID – the application ID registered for the application. The value differs according to application client type.

Server API

Sybase Unwired Platform includes several interfaces that open specific features and functionality of Unwired Platform for custom development. Customizing mobile business objects (MBOs) allows you to better control behavior of these features.

- **Result set filter –** use a custom Java class to filter the rows or columns of data returned from an MBO read operation. You can write a filter to add, delete, or change columns, or to add and delete rows.
- **Result checker –** use the custom Java class to implement custom error checking for MBO operation results returned from the enterprise information system (EIS) to which the MBO is bound.
- **Data change notification (DCN) and Hybrid App DCN –** a refresh mechanism that uses an HTTP interface to inform Unwired Server of EIS data changes, and to optionally propagate those changes to the specified MBO.
- **DCN filter –** use a DCN filter to preprocess the submitted DCN. The filter converts raw data in the DCN request to the required JavaScript Object Notation (JSON) format. The filter can also postprocess the JSON response returned by the Unwired Server into the format preferred by the enterprise information system (EIS).
- **Custom transforms –** create a transform to modify the structure of generated Web Services message data, so it can be used by an Unwired Platform MBO.

You can program these functions in any order; each class is implemented independently.

Javadocs

The full Unwired Platform runtime installation includes Javadocs. Use the Sybase Javadocs as your API reference.

By default, Javadocs for Result Set Filters, Result Checkers, and Data Change Notifications are installed in `SUP_HOME\Servers\UnwiredServer\APIdocs\index.html`.

Mobile Business Object Development Task Flow

This task flow guides the MBO developer through the decision-making process and steps required to develop mobile business objects in Sybase Unwired WorkSpace, which provide the server-side code in mobile applications, including Hybrid Apps.

This task flow does not include all configurable options, but does guide the MBO developer through the commonly performed tasks required to develop a mobile data model.

1. Determine the data needs of the mobile application.

   For design information, see *Best Practices for Developing an MBO Data Model*.

2. Create and open the Mobile Application Project, which contains all project-related information.
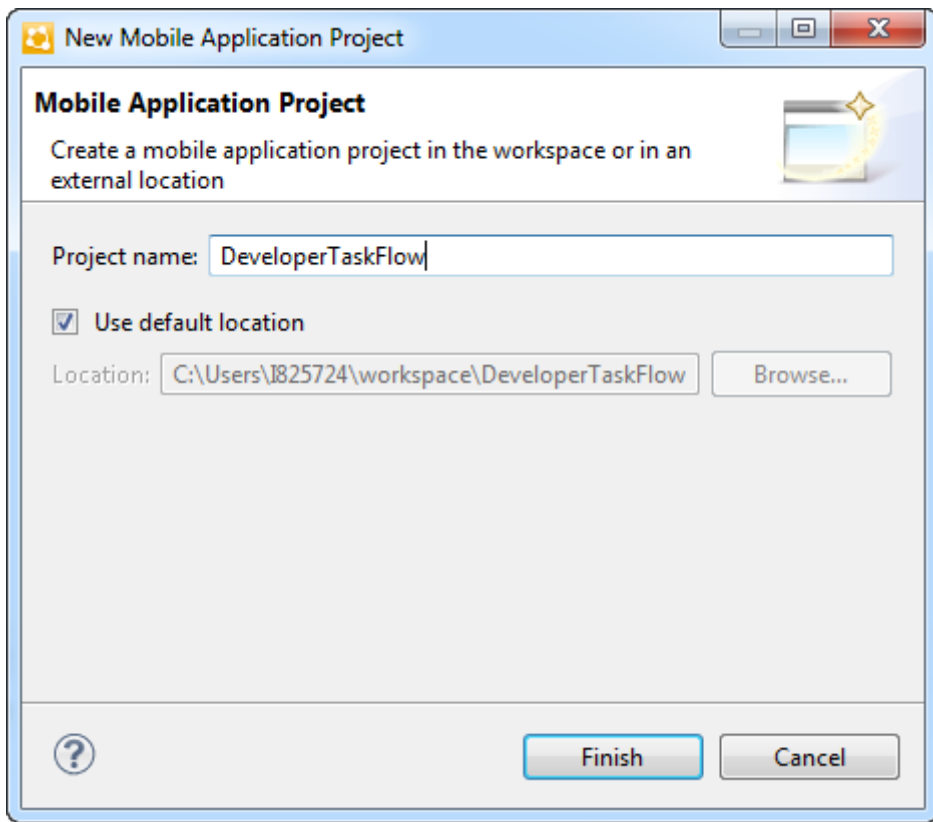
See *Creating a Mobile Application Project* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

3. Create and connect to the enterprise information system (EIS) datasource using a connection profile from which you develop the MBOs.
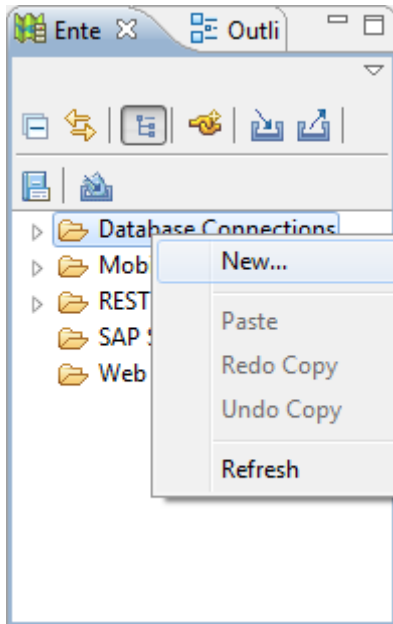
- For design information, see *Best Practices for Loading Data From the EIS to the CDB*.
- See *Creating a Data Source Connection Profile* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

4. Create the MBOs. The simplest method is to drag and drop the datasource (for example, a database table) onto the mobile application diagram and bind the MBO to the datasource.
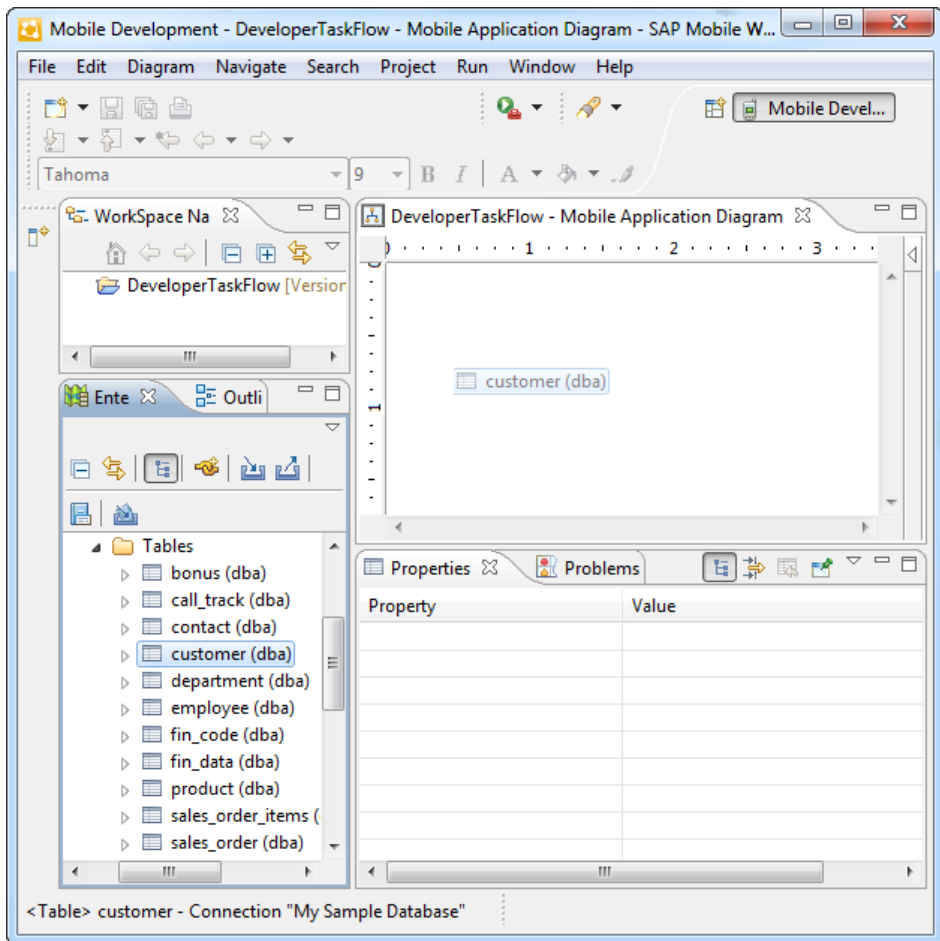
For design information, see:
- Define the attributes – see *MBO Attributes*
- Define the operations – see *Mobile Business Object Operation Properties*, and related topics
- Define other datasource and runtime specific properties during MBO creation, or later from the Properties view – see *Mobile Business Object Properties*

See these topics in *Sybase Unwired WorkSpace - Mobile Business Object Development*:
- *Creating Mobile Business Objects*
- *Binding Mobile Business Objects to Data Sources*
- *Creating Attributes for a Mobile Business Object*
- *Creating Operations for a Mobile Business Object*
- *Mobile Business Object Operation Properties*

5. Define Relationships between MBOs. Most mobile applications include several to many individual MBOs.
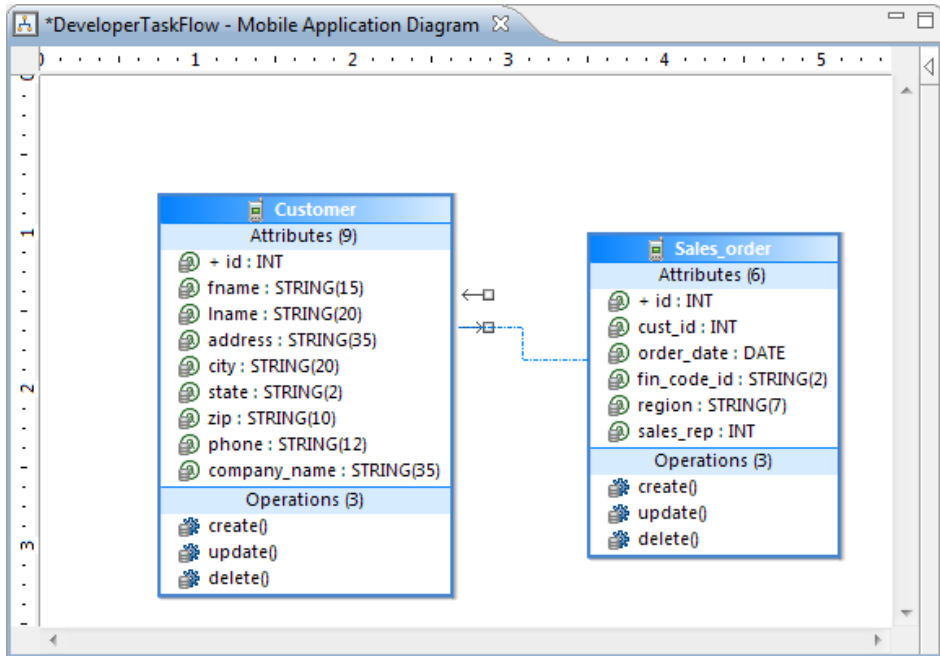
- For design information, see *MBO Relationships*
- See *Creating Relationships Between Mobile Business Objects* in *Sybase Unwired WorkSpace - Mobile Business Object Development*

6. Define cache refresh behavior.

   For design information, see:
   - *Cache Groups* and *Data Change Notification*.
   - *Operation Cache Policies*.
   - *Load Arguments*.

   See these topics in *Sybase Unwired WorkSpace - Mobile Business Object Development*:
   - *Creating Cache Groups*:
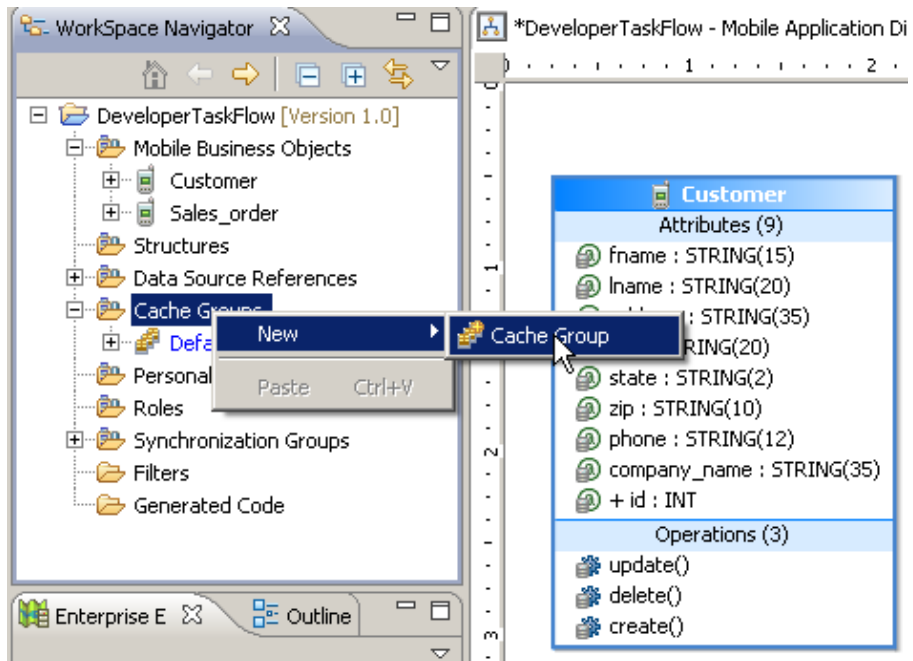
- *Setting an Operation Cache Policy*:
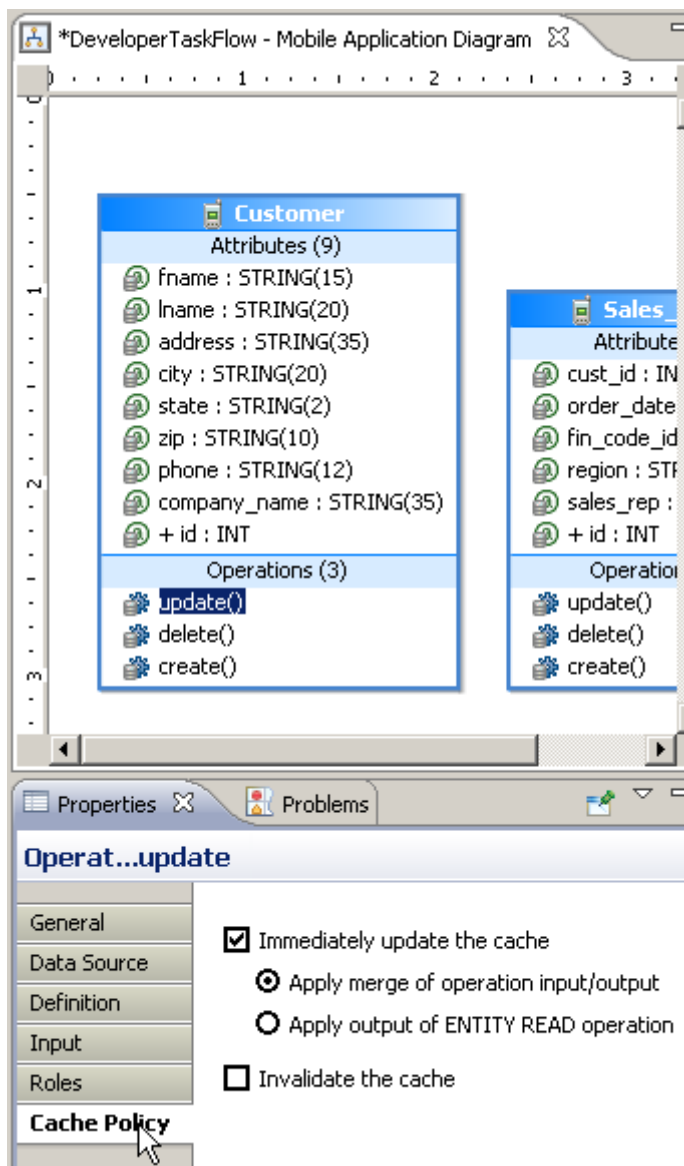
- *Load Arguments*:
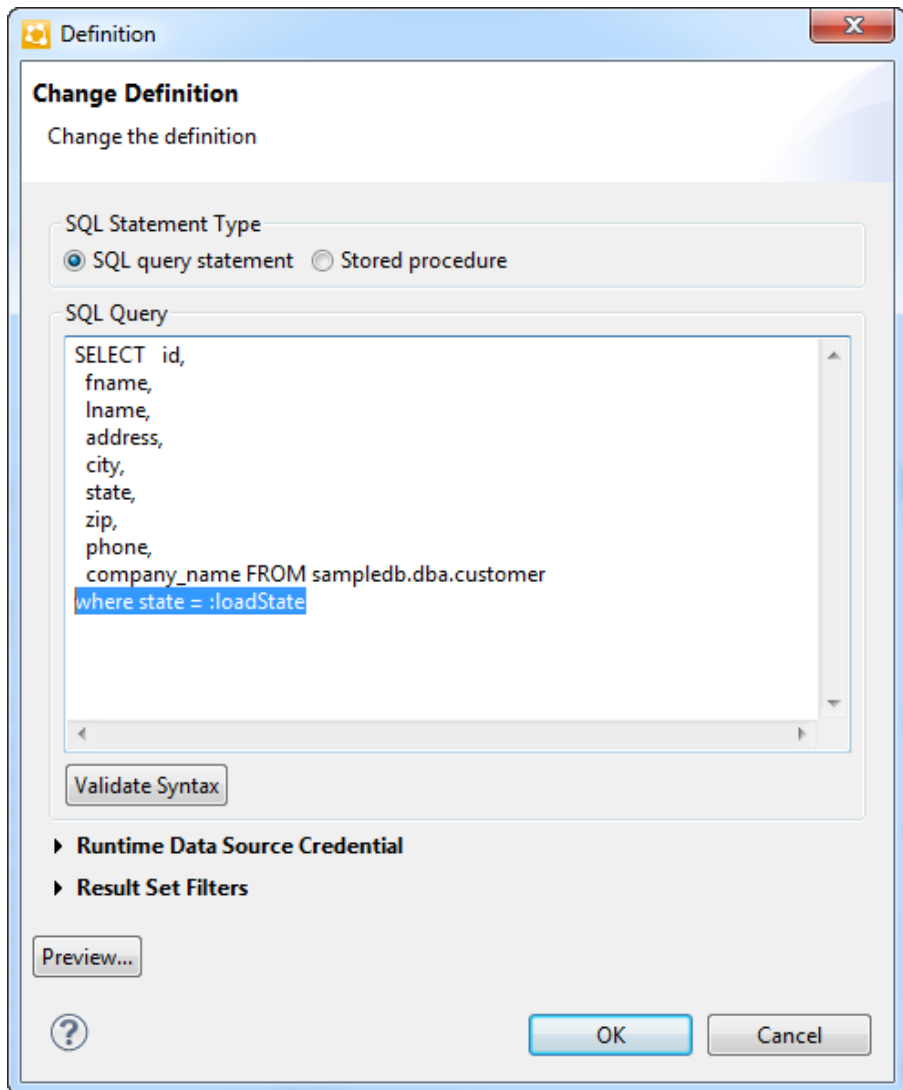
7. Define synchronization behavior.

For design information, see:
- *Synchronization Groups* and *MBO Synchronization Groups*
- *Synchronization Parameter Definition Guidelines* and *MBO Synchronization Parameters*
- *Understanding Synchronization Parameters*

See these topics in *Sybase Unwired WorkSpace - Mobile Business Object Development*:
- *Creating Synchronization Groups*:
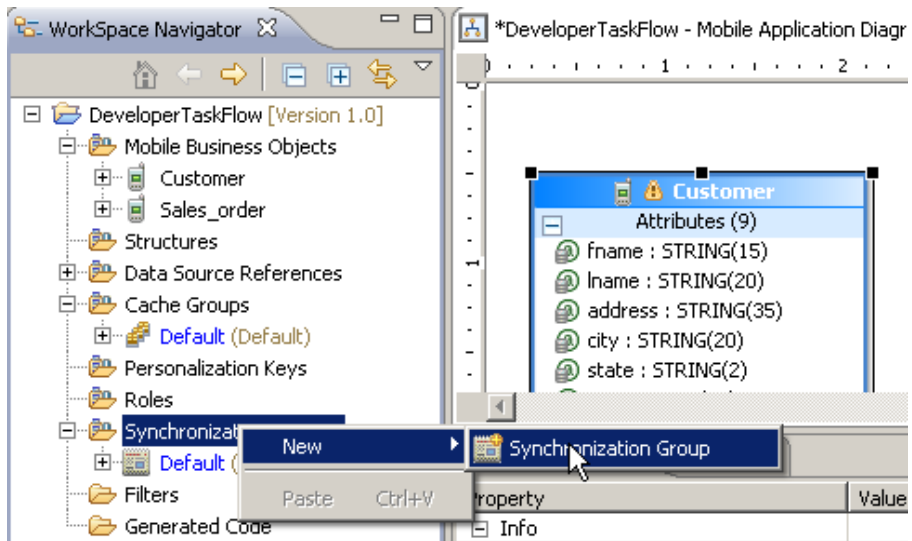
- *Defining Synchronization Properties for Individual Mobile Business Objects*:
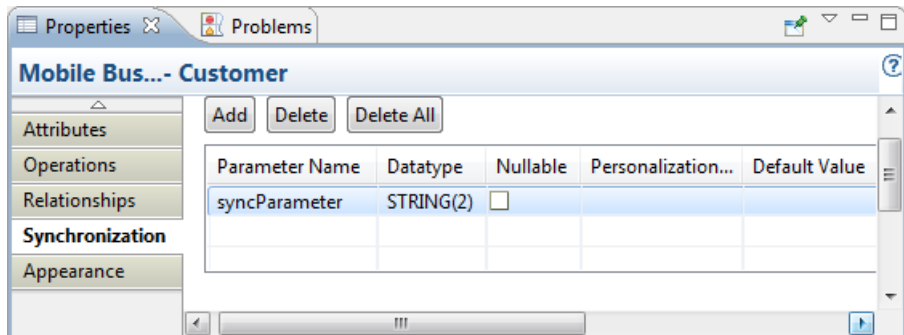
- *Combining Load Arguments and Synchronization Parameters*:
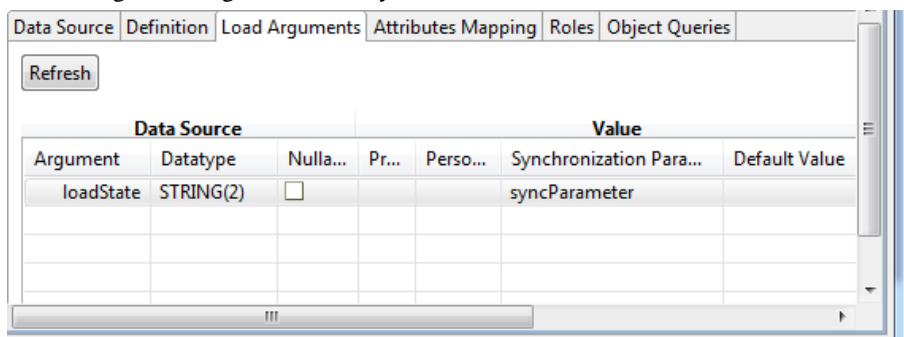
8. Consider additional MBO configuration options when defining the mobile data model.
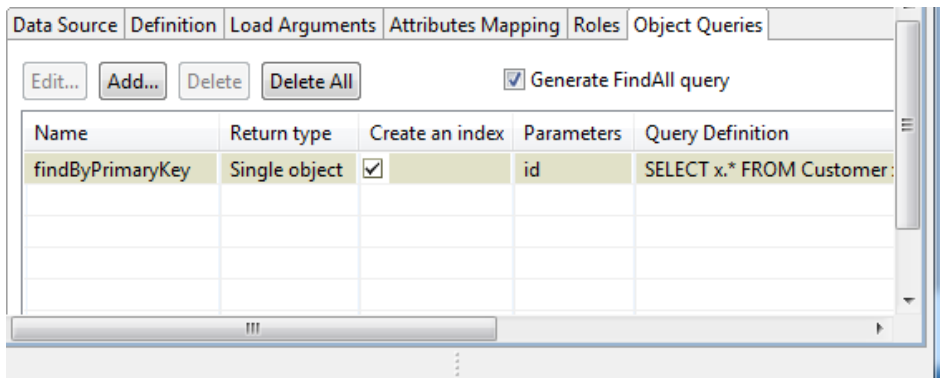
- Use object queries to filter and display data already downloaded to the device:
  - For design information, see *Object Queries*.
  - See *Generating Object Queries from Primary Key Attributes* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.
- Use personalization keys to personalize certain input field values within the mobile application.
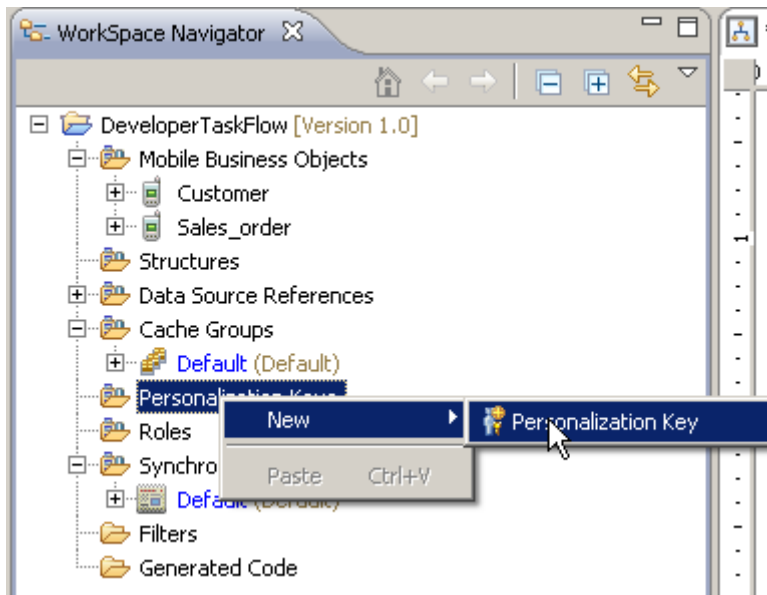
See *Managing Personalization Keys* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.
- Use logical roles to provide authorization to MBOs and MBO operations.
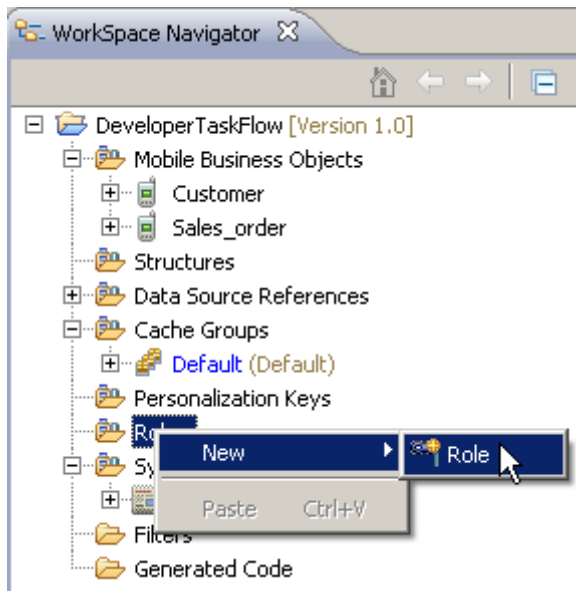
See *Managing Roles and Permissions* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

- Set runtime datasource credentials, which is required for establishing EIS connections on the client's behalf, such as in single sign-on (SSO) configurations.

See these topics in *Sybase Unwired WorkSpace - Mobile Business Object Development*:

- *Implementing SSO for SAP*.
- *Propagating a Client's Credentials to the Back-end Data Source*.
- Use result set filters to add custom logic to manipulate rows and columns of data returned to the Unwired Server cache.
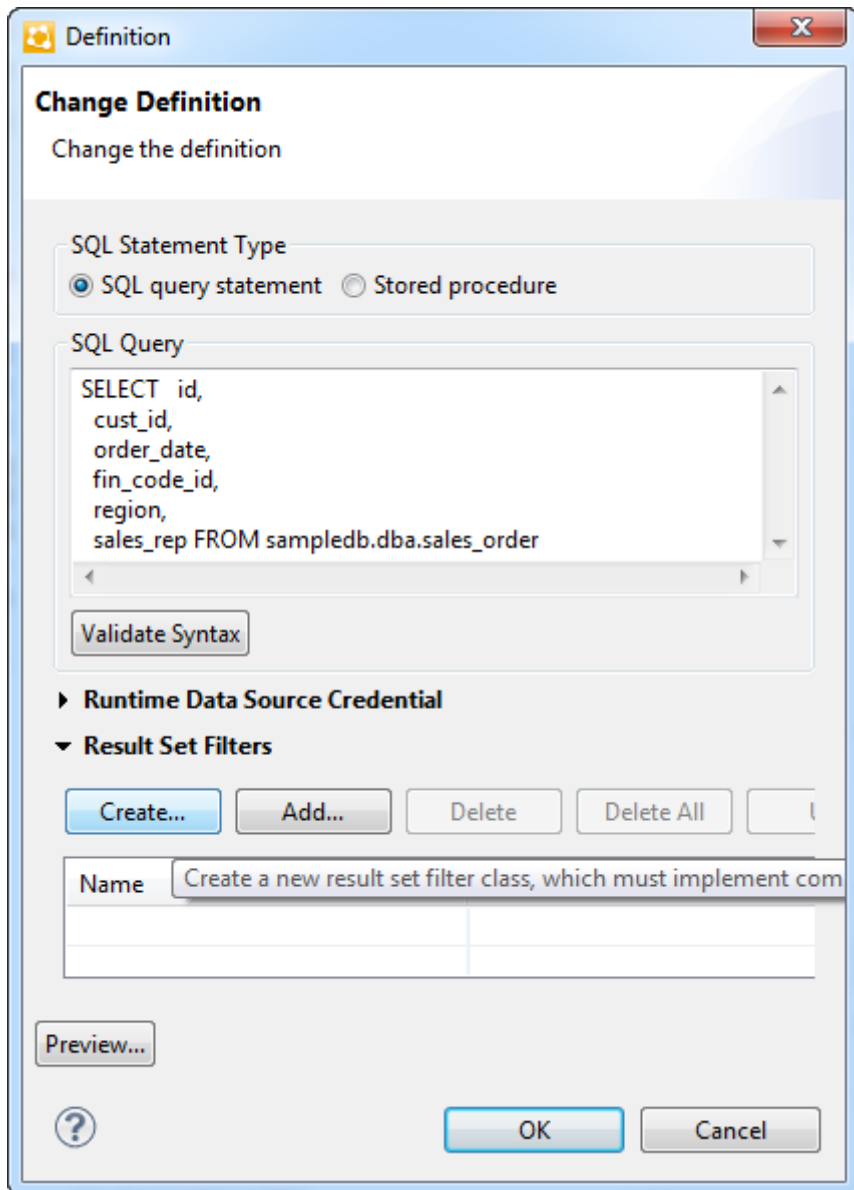
- For design information, see *Result Set Filters*.
- See *Adding a Result Set Filter* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.
- Use result checkers to customize error checking for mobile business object (MBO) operations.

- For design information, see *Result Checkers*.
- See *Adding a Result Checker* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.
- For more information about datatype usage in mobile applications, see *Datatype Support*.

- For more information about, and to learn about planning for MBO and mobile application life cycles, see *Development Life Cycle* in *Fundamentals*.

9. Generate the Object API code and deploy the MBOs.

- Generate the object API code for the mobile platform on which you are developing the application.
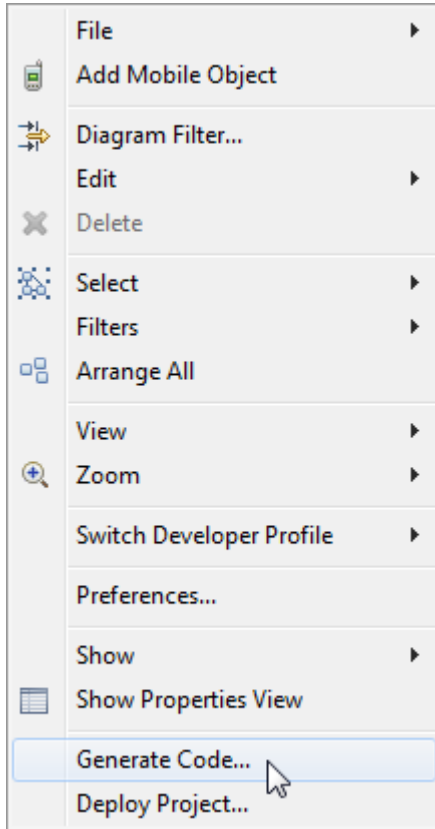
See *Generating Object API Code* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

- When you are ready to test a mobile application, deploy the MBOs to Unwired Server.
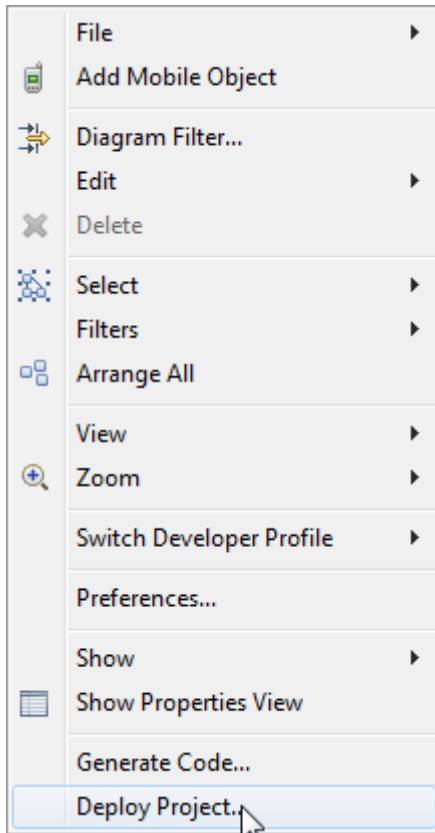
See *Packaging and Deploying Mobile Business Objects* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

See also

- *Best Practices for Developing an MBO Data Model* on page 93
- *Best Practices for Loading Data From the EIS to the CDB* on page 119
- *MBO Attributes* on page 95
- *Operation Cache Policies* on page 37
- *MBO Relationships* on page 99
- *Cache Groups* on page 35
- *Data Change Notification* on page 153
- *Load Arguments* on page 27
- *MBO Synchronization Groups* on page 111
- *Synchronization* on page 30
- *Object Queries* on page 51
- *Result Set Filters* on page 131

- *Result Checkers* on page 139

Mobile Business Object Mobility Properties

To understand how to customize MBOs to meet your device application needs, you must first understand the concepts that affect end-to-end data flow between client, Unwired Server cache (CDB), and the enterprise information system (EIS) to which the MBOs are bound.

Load Arguments

Load arguments control the amount of data refreshed between the enterprise information system (EIS) and the cache database (CDB), and each load argument creates its own client defined partition in the CDB based on load argument value (partition key). Partitions are refreshed concurrently, thus improving performance. In contrast, synchronization parameters filter CDB data downloaded to the mobile device during device application synchronization.

Set load arguments in the Properties view, from the **Attributes > Load Arguments** tab. Set synchronization parameters from the Synchronization tab. It is important to understand both their differences and how they work together to load (data refresh) and filter (synchronize) data. For example, you can use:

- A synchronization parameter and a separate load argument – refresh data based on an argument independent of synchronization, or
- A load argument that maps to a synchronization parameter – use the same value for both refreshing and synchronizing data. Basically, one synchronization parameter induces one client defined partition. This provides more fine-grained CDB partitioning and concurrency, but may introduce more partition refresh overhead and less data sharing across devices when there are too many different values from synchronization parameters.

Figure 1: Synchronization Parameter
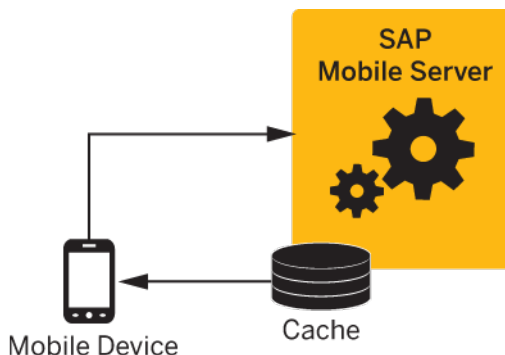
Figure 2: Load Argument
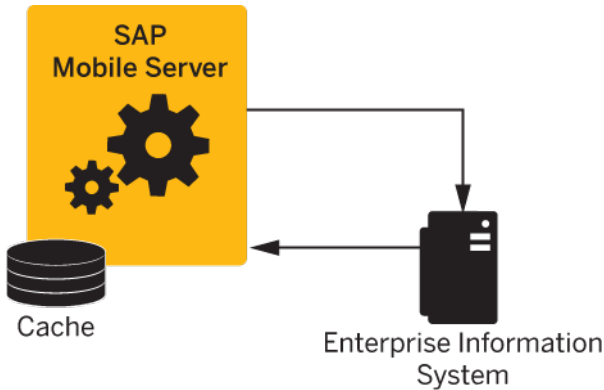
Client Defined Cache Partitions

Partitioning the Unwired Server cache (CDB) divides it into segments that can be refreshed individually, which provides faster system performance than refreshing the entire CDB.

Client defined cache partitioning is determined by one or more partition keys, which is one or more load arguments used by the operation to load data into the cache from the enterprise information system (EIS).

Define a load argument (cache partition key) in the **Attributes > Load Arguments** tab of the MBO's Properties view. A device application user can specify values for synchronization parameters that are mapped to load arguments when synchronizing his or her client application, possibly through personalization keys or default values.

All client defined cache partitions require a load argument:

- Create cache partitions through a load argument specified by the client, for example, a load argument that maps to a synchronization parameter which uses a personalization key or default value.
- Refresh a cache partition if data in the partition is:
  - Expired
  - Invalidated
  - Inconsistent – if a client has multiple partitions, refresh all partitions even if only one partition expires.
- If the MBO is defined with something other than "=" in the **where** clause, manually edit, in the synchronization tab, the SQL code for the customized download data.

Examples: Parameters and Client Defined Cache Partitions

Create client defined cache partitions based on mobile business object (MBO) and load argument definitions.

These examples use the employee table in the My Sample Database connection profile.

_____

Create a mobile business object (MBO) that uses a load argument and personalization key to partition the Unwired Server cache database (CDB). The general process is:

- In Sybase Unwired WorkSpace, create an MBO with a load argument from the employee table, add a personalization key, and map the load argument that defines how the CDB is partitioned to the personalization key.
- Client (device application) partitions are created as new clients connect. Users set the personalization key in their application, and then synchronize and download data.
- The CDB loads data that satisfies the MBO definition using the load argument value, which is the personalization key value in this example, passed by the client, and returns only those rows that matches the client's personalization key value.

For this example:

1. Drag and drop the employee table, and edit the definition to include the state_param parameter (load argument):

```
SELECT    emp_id,
  manager_id,
  emp_fname,
  emp_lname,
  dept_id,
  street,
  city,
  state,
  zip_code,
  phone,
  status,
  ss_number,
  salary,
  start_date,
  termination_date,
  birth_date,
  bene_health_ins,
  bene_life_ins,
  bene_day_care,
  sex FROM sampledb.dba.employee
where state = :state_param
```

2. Create a personalization key with these values:
   - **Name** – state_pk
   - **Type** – string(4)
   - **All other entries** – accept default values

3. From the Attributes Load Arguments tab, map the load argument to the personalization key:
   - **Argument** – state_param
   - **Datatype** – string(4)
   - **Nullable** – no
   - **Propagate to** – state
   - **Personalization key** – state_pk

---

4. Deploy the package to Unwired Server.

Client and CDB behavior is:

- Client 1 sets state_pk to "TX ", while client 2 uses "GA ". Two rows (one for each argument) are added to the parameter table in the CDB. Two trailing white spaces are added to pad the total length to four, since state_pk is defined as string(4).
- The CDB partition table contains values that define the partition key for each partition (TX and GA).
- The partition refresh table tracks the most recent refresh for each partition.

Only the data in the partition of interest refreshes. This is an important performance consideration for large tables.

Creating client defined cache partitions based on compound parameter values
This example shows how to create a partitioned cache for the employee table where the partitions are defined by a compound partition key that uses two attributes: city and state.

Manually edit the SQL definition. For a query that does not require exact matches for the state or city parameters, use this MBO definition as the download query:

```
SELECT    emp_id,
  manager_id,
  emp_fname,
  emp_lname,
  dept_id,
  street,
  city,
  state,
FROM sampledb.dba.employee
WHERE state LIKE :state_param + '%'
AND city LIKE :city_param + '%'
```

Synchronization

Determine the amount of data (filter), and under what conditions (timing and triggers), mobile devices upload MBO data to and download data from the Unwired Server cache (CDB).

Synchronization properties are unavailable for MBOs in cache groups that use an Online policy.

Understanding Synchronization Parameters

Synchronization parameters restrict the rows that are transferred from the Unwired Server cache database (CDB) to the device to match values the client provides.

- A synchronization parameter does not affect enterprise information system (EIS) interaction with the CDB, unless you specify the synchronization parameter setting for a given load argument.

- An Online cache group policy, typically used with Hybrid Apps, does not support synchronization parameters.
- After you bind a mobile business object (MBO) to a datasource, MBO attributes map to datasource fields (for example, database columns). You control the amount of data synchronized (filtered) between the CDB and device application by defining synchronization parameters that map to attributes. For example, mapping a synchronization parameter to the "state" attribute, returns customer records for a particular state based on the value entered by the device application user.
- Another example is an MBO named "sales_order" with a synchronization parameter mapped to the "region" attribute. Executing the query `select * from sales_order` from the device application returns a complete copy of all sales orders in the CDB. If the application user provides a region when synchronizing, for example `select * from sales_order where region=Eastern`, the client sees sales orders only for the Eastern region.
- Using a synchronization parameter to filter results may be particularly useful for MBOs that have large amounts of data that do not change frequently, making periodic bulk loads and a longer cache interval more appropriate. For example, use `select * from customer` to bulk-load all customers. Then design a synchronization parameter that maps to the "state" attribute. To load only California customers, the device application user passes in the "CA" parameter.

Synchronization Parameter Behavior

These examples illustrate how a device application client uses a single synchronization parameter defined in Sybase Unwired WorkSpace to synchronize data on the client with data contained in the MBO instance on Unwired Server.

1. Create a Customer MBO from the customer table in the sampledb.
2. Add a stateParam synchronization parameter and map it to the state attribute.
3. Deploy the MBO.
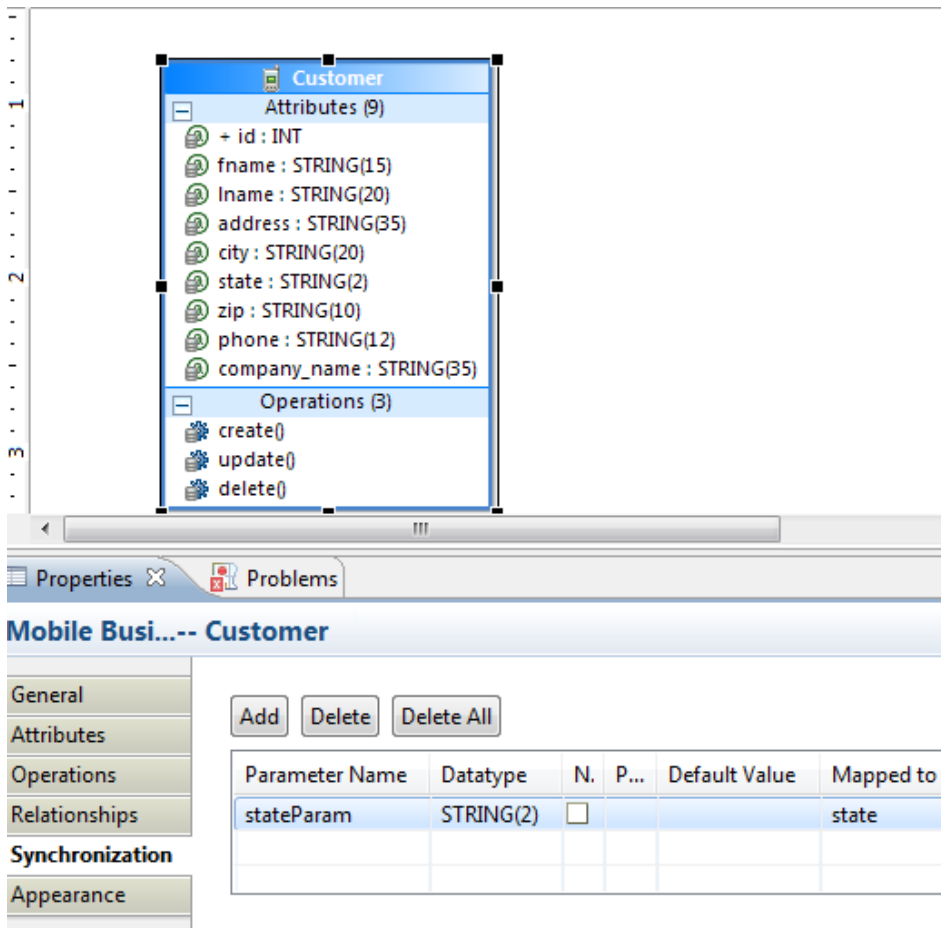
Example device application behavior using this synchronization parameter includes:

1. Retrieve the synchronization parameters object from the MBO instance:

```
CustomerSynchronizationParameters sp =
Customer.SynchronizationParameters;
```

2. Set the stateParam synchronization parameter to "CA", save then synchronize. FindAll retrieves all 10 records that have state="CA".

```
sp.stateParam = "CA";
sp.Save();
SUP101DB.Synchronize();
```

3. Perform a second sync without setting stateParam. FindAll again returns the same 10 records.

```
SUP101DB.Synchronize();
```

Conclusion: You need not always set the sync parameter before a sync. If the same sync parameter value is to be used, it does not need to be set again, since it uses the previous sync parameter value.

4. Set stateParam to "NY", save then synchronize. FindAll retrieves 23 records, the 10 returned using the previous sync parameter state="CA" and 13 where state="NY".

```
sp.stateParam = "NY";
sp.Save();
SUP101DB.Synchronize();
```

Conclusion: When you save a new set of sync parameters the records returned from the synchronization includes all previous sync sets. That is, synchronization parameters are cumulative and previous sync parameters are maintained unless Delete is called on the sync parameters.

5. Call the Delete method, set stateParam to "MN", and save then synchronize. FindAll returns 5 records where state="MN".

```
sp.Delete();
//You must re-retrieve the synchronization parameter instance
sp = Customer.SynchronizationParameters;
sp.stateParam = "MN";
sp.Save();
SUP101DB.Synchronize();
```

Conclusion: Calling Delete on the sync parameter clears the previous sync parameters. On the next sync the previous result sets for those sync parameters are removed.

It is not possible to retrieve all the synchronization parameter values except for those most recently used. When a synchronization is performed it synchronizes all sync parameter sets. That is, it attempts to load data from the Unwired Server cache based on each sync parameter (for on demand this is done during the sync, for scheduled it is based on the configured schedule).

Synchronization Parameter Considerations

Modeling and mapping of synchronization parameters to load arguments implicitly generates client defined data partition keys within the Unwired Server cache. Partition keys define subsections of data within an MBO, enabling parallel data access to large MBO data sets.

For example, you can refresh multiple partitions in parallel, or query one partition while another is being refreshed. In general, partitions prevent serialized access to the cache. Some best practices for defining client defined partition keys include:

- Synchronization parameters should be defined and mapped to all result-affecting load arguments. Failure to do so results in partitions being continually overwritten/deleted which leads to unexpected results in the mobile client.
- Result-affecting load arguments are those arguments of the EIS read operation that affect the results of the operation. Some arguments may be information needed by the EIS but do not actually affect the results of the read operation. For example; suppose an MBO is modeled using a Web Service operation "getAllBooksByAuthor(AuthorName, userKey) where userKey is simply a mechanism to authenticate a user and does not affect the results

of the operation. For a given "AuthorName" the service will return the same list of books regardless of the "userKey" value. In this case "userKey" is not result-affecting and therefore should not be mapped to a synchronization parameter.

Synchronization Parameter Definition Guidelines

Understand guidelines and restrictions when defining synchronization parameters.

Guideline	Description
Datetime and time synchronization parameters	Synchronization may fail if you use SQL Anywhere® as the Unwired Server cache database (CDB) when the synchronization parameter is a datetime or time datatype, since datetime and time columns contain three digits for the fraction portion, making direct comparison incompatible with Sybase Unwired WorkSpace.
	To compare a datetime or time datatype to a string as a string, use the **DATEFORMAT** function or **CAST** function to convert the datetime or time datatype to a string before comparing. For example, this SQL statement is the generated download cursor when attribute c2 (datetime) is specified as the synchronization parameter:
	`SELECT x.* FROM Mydatetime x WHERE ( ( x.c2=:c2 ) OR ( ( x.c2 IS NULL ) AND (:c2 IS NULL ) ) )`
	Change the statement to:
	`SELECT x.* FROM Mydatetime x WHERE ( (dateformat(x.c2, 'yyyy-mm-dd hh:mm:ss') = dateformat(:c2, 'yyyy-mm-dd hh:mm:ss') OR ( ( x.c2 IS NULL ) AND (:c2 IS NULL ) ) )`
Datatype and nullable default values	When a synchronization parameter is mapped to an attribute, to maintain consistency between the two, the parameter datatype and nullable fields follow that of the attribute to which it is mapped.

Synchronization Groups

A synchronization group specifies the synchronization behavior for every mobile business object (MBO) within that group.

Data that is downloaded from Unwired Server to the device is in the scope of a synchronization group. That is, if the device user initializes a download to the device from a specific MBO, Unwired Server transfers the delta data of all MBOs within the same synchronization group to the device.

Cache Groups

A cache group specifies the data refresh behavior for every mobile business object (MBO) within that group.

During development, you can group MBOs based on their data refresh requirements. Some terms and concepts you should be familiar with are:

- **Cache group –** includes a cache policy and the MBOs that share that policy. An MBO can belong to only one cache group.
- **Cache –** MBO data in the Unwired Server cache (CDB) can be refreshed according to a cache policy, along with other mechanisms, such as data change notification (DCN).
- **Cache policy –** defines the cache refresh behavior and properties for the MBOs within the cache group based on the policy:

  - **On demand –** the cache expires after a certain period of time (cache interval) such as 10 minutes. The cache is not updated until a request is made of the cache and the cache has expired. If a request is made of the cache and it is expired, there may be a delay responding to the request while the cache is refreshed.
  - **Scheduled –** the cache is refreshed according to a schedule such as 7:00 am, 1:00 pm, or 6:00 pm. Note that load arguments filled from transient personalization keys cannot be used with a scheduled cache type, because transient personalization key values are stored in the device application session, and unknown to Unwired Server.
  - **DCN –** the cache never expires. Data refresh is triggered by an enterprise information system (EIS) Data Change Notification. The cache interval fields are disabled when DCN is selected.

    You can define MBOs without any load operations (not bound to a datasource), only if the MBO belongs to a cache group that uses a DCN policy.

    See the *Mobile Data Models: Using Mobile Business Objects* for details about implementing DCN.
  - **Online –** only can be used with message-based Hybrid Apps. See *Online Cache Group Policy*.
  - **EIS managed –** DCN controls cache refresh, the ability to define cache partitions, update MBO data within those partitions, and subscribe users to those partitions.

Each cache group contains a cache policy, which in turn contains cache refresh/update properties. When a refresh occurs, the Unwired Server calls the default read operation (for each MBO in the cache group), and all of the rows that are returned from the enterprise information system (EIS) are compared to existing rows in the CDB as follows:

- If the CDB is empty, all rows are inserted.
- If any rows exist in the CDB, Unwired Server processes the row-set and checks (using the primary key) to determine if the row already exists in the cache:

- If it does, and all columns are the same as the EIS, nothing happens. When a client synchronizes to request all rows that have changed since the last synchronization, only rows that have changed are included, which is important for performance and efficiency.
- If the row does not exist, it is inserted and the next synchronization query retrieves the row.

Cache Group Considerations

Define multiple cache groups for data based on specific usage patterns and consistency tolerance. For example, for transactional data that has little tolerance for EIS data that is stale, an On-demand cache policy with a small coherence window is a more appropriate cache solution than an hourly schedule-based refresh.

Some best practices for defining cache groups and allocating MBOs to those groups include:

- Place all MBOs that are modeled with composite relationships in the same cache group.
- Do not place reference data and transactional data in the same cache group. Typically, reference data has different data consistency requirements than transactional data. For example, a SalesOrder MBO that has a composite relationship to a SalesOrderItem MBO, which contains a reference (non-composite) relationship to Product. Since products do not change as often as sales order items, you may want to put the Product MBO in a different cache group.
- Avoid circular dependencies between cache groups.
- Avoid loading of an MBO in one cache group based on the attributes of an MBO in another cache group.

On-demand versus scheduled refresh
Cache policy refresh options affect cache partitions, in that they determine the frequency with which the CDB is updated from the EIS (data refresh). The scheduled refresh option refreshes the cache based on a clock. The on-demand refresh option, which is based on client actions, is discussed here, with an emphasis on how to determine if cache partitions behave as expected.

To validate CDB data partition refresh behavior for client defined partitions:

1. Define multiple partitions and multiple clients.
2. Define a cache group with an On demand policy and confirm cache refresh behavior. Set the cache policy with a cache interval that is long enough to allow you to perform updates to the EIS and synchronize both clients.
3. Wait until the cache interval passes, then resynchronize the clients. The second synchronization should reflect EIS changes, since the cache policy dictates that the CDB must now refresh.
4. Inspect CDB log files for time stamps in the "LAST_REFRESH" and "LMD" columns for your package to confirm that the partitions and rows of data for the associated partitions have refreshed as expected.

To confirm the correct client defined partitions have refreshed:

1. Make updates to data from multiple partitions in the EIS.
2. Synchronize one client so only one partition refreshes. Make sure the values for the length of the cache interval and the last time the partition refreshed indicate that the data in that partition is stale and needs to refresh when you synchronize.
3. The synchronized client retrieves refreshed rows only from the partition of interest. Data from other EIS partitions do not update the CDB, even though data has changed in the EIS (because no client has requested a synchronization of that partition).
4. When the second client synchronizes the second partition, only that partition refreshes.

Operation Cache Policies

Setting an operation cache policy for mobile business object (MBO) operations gives you more control of Unwired Server interactions with the enterprise information system (EIS) to which the MBO is bound, and Unwired Server cache (CDB) updates. Fine-tuning these interactions and updates improves both Unwired Server and device application performance.

MBO operations perform specific functions based on their definition:

- **Read operation (MBO attributes, load arguments, and synchronization parameters)** – the EIS operation used to define and initially populate the CDB (from the EIS) for the MBO. Also called a load operation.
- **Create, Update, Delete (CUD operations)** – modify EIS data depending on the definition of the operation. Unwired Server maintains a cache (CDB) of back-end EIS data to provide differential synchronization and to minimize EIS interaction.

    While this type of bulk-fetch and CDB caching are effective in reducing the number of interactions required with the EIS, and work well in some other cases (where MBO data is occasionally updated in the EIS), performance suffers if changes are initiated from Unwired Server (by way of MBO operations), or if changes are frequent.

    When an operation is submitted from a device application to the EIS, the cache must be refreshed for those changes to be available to a client the next time the client synchronizes.
- **Entity Read operations** – refresh a single instance of an MBO composite graph (MBOs in a composite relationship) in the Unwired Server cache (CDB). This type of operation can only be invoked for Create and Update operations that use an "Immediately update the cache" operation cache policy.

Operation cache policies determine how the CDB is updated after an operation executes. The operation cache policy and option combinations from which to choose to associate with MBO CUD operations include:

- **Immediately update the cache with Apply merge of operation input/output** – this can be used for CUD operations where the MBO attributes map one-to-one with corresponding values in the EIS (no EIS generated IDs or side-effects on MBO attributes). For example:

1. The MBO Developer edits the Contact MBO's officePhoneNumber attribute, and the Update operation requires Unwired Server to send fname, lname, city, state, and so on attributes that have not been changed on the client.

2. Unwired Server recognizes that only officePhoneNumber has changed, and updates the cache correspondingly if the EIS operation executes without error.

- **(Legacy) Apply results to the cache** – used primarily to support Create operations where the EIS generates the business ID for the new entity. Sybase Unwired Platform requires the EIS developer to provide a create method that returns all the attributes of the MBO that the Unwired Server normally sees from the read/load operation. This allows Unwired Server to insert the real business ID into the MBO instead of a temporary one manufactured on the client (generateKey) as a placeholder. When Unwired Server synchronizes the newly created MBO back to the client, it contains the real EIS defined ID field, and the client can update its remote copy by matching the surrogate key.

- **Immediately update the cache with Apply output of ENTITY READ operation** – applies to any operation type, and also applies to graphs of related MBOs and shared read operations. For example:

  1. The MBO Developer wants to change the ShipDate attribute on the SalesOrder MBO for a customer in a SalesOrder -> LineItems object graph, then submit the Update operation.

  2. The EIS recognizes that due to product availability, some of the line items cannot be delivered on that date, so only some of the LineItem MBOs are modified as a result.

  3. Unwired Server recognizes that the Update operation succeeds (but this is the "normal" Update operation that has not been modified to actually return MBO values, much less the whole tree of SalesOrder->LineItem).

  4. Unwired Server recognizes an ENTITY READ operation associated with the Update operation takes the SalesOrder.ID as input and executes the entity read operation that returns the updated SalesOrder and LineItems information from the EIS.

  5. Using the shared read feature, Unwired Server updates the cache for both parent and child MBO values in the graph.

- **Invalidate the cache** – the cache is unaffected in terms of what is in the cache database (CDB), but Unwired Server internally marks the cache partition to which the affected MBO belongs as invalid. The next time a client tries to synchronize values of this partition Unwired Server refreshes the cache by performing the load operation. If the cache is very finely partitioned, this can be the equivalent to the ENTITY READ approach: apply the operation and then perform an EIS read to update the cache. If the load operation takes a specific ID and returns a single MBO record, then these are equivalent. If the MBO partition is larger scale (Customer by State for example), then a lot of MBOs are invalidated and the cache refresh (all customers in State=CA) is a more expensive operation.

- **None** – the cache is not updated after an operation. If there is a one hour refresh policy, changes are not reflected in the cache (or the device) until that hour expires and Unwired

Server refreshes the cache. If the policy is DCN, the cache is not updated until the EIS sends a DCN message reflecting the changes.

When an MBO operation is called, its cache policy determines how operation results are applied to the CDB.

Other mechanisms used to update the CDB that are external to MBO operations, and not associated with operation cache policies include:

1. EIS-initiated DCN – an HTTP request to Unwired Server, in which the DCN request contains information about the changed data, or the changed data itself.
2. Scheduled data refresh – Unwired Server polls the EIS for changes at specified intervals.
3. MBO cache group – every MBO belongs to a cache group that specifies a cache refresh policy for every MBO in that group. Plan carefully to maximize cache group and cache policy efficiency. Examples include:
   - A poorly designed MBO might have an operation with a cache policy that updates only the operation results to the CDB, but the MBO belongs to a cache group with an interval that refreshes the entire MBO on too short a schedule, minimizing the value of the cache policy.
   - This same MBO properly designed might have a cache group that refreshes the MBO nightly, increasing Unwired Server performance by deferring load from peak usage hours.

Operation Cache Policy Data Flow

Understand the data flow of Sybase Unwired Platform operation cache policies.

The diagrams in the following sections illustrate the data flow from an Unwired Server point of view, which is different than runtime. At runtime, the mobile application client can send separate operation and synchronization calls to Unwired Server by calling the Object API Synchronize() method, which does both (call an operation and synchronize).

Invalidate the Cache Data Flow

Understand how the "Invalidate the cache" operation cache policy invalidates the cache after an EIS operation.

In this example, the client defined partition for the Order MBO is defined by the customerId attribute. The REST service exposes an Order GET operation that takes customerId as a parameter, and returns a subset/partition of Order data specific to customerId. The REST service also exposes an Order POST operation that takes an Order graph as a parameter:

1. The "Invalidate the cache" operation cache policy associated with the MBO Update operation invalidates the cache partition after the REST Service POST is invoked.
2. The next client data synchronization for the invalid partition triggers the REST Service GET operation that takes customerId as an input parameter.
3. The REST Service returns all orders for the given customerId (a partition) and the cache manager merges the orders as well as the order/items into the cache.
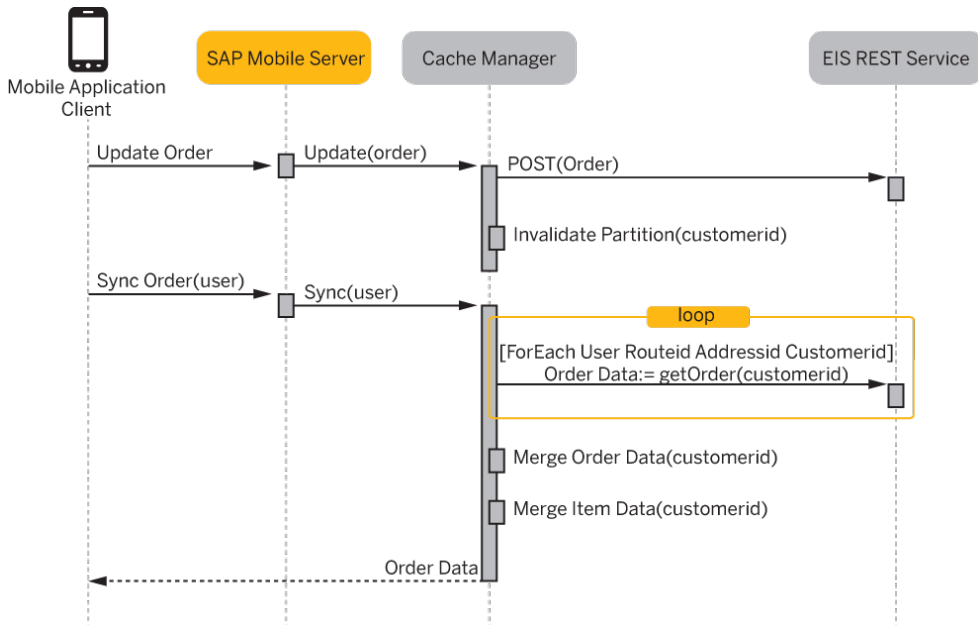
Immediate Refresh with Entity Read Data Flow

The Update and Create operations use the "Apply output of ENTITY READ operation" operation cache policy option with the "Immediately update the cache" cache policy, to immediately apply the results of the Entity Read operation to the Unwired Server cache.

This example:

- Requires the EIS to expose the REST service as a read operation that returns a single instance of the data when given the primary key (Entity Read).
- Reduces the transaction lock scope within the cache when compared with the "Invalidate the cache" option that refreshes a partition, because it remains coupled to the client Update operation and the granularity of the cache refresh is at the primary key/graph ID level rather than the partition ID level.
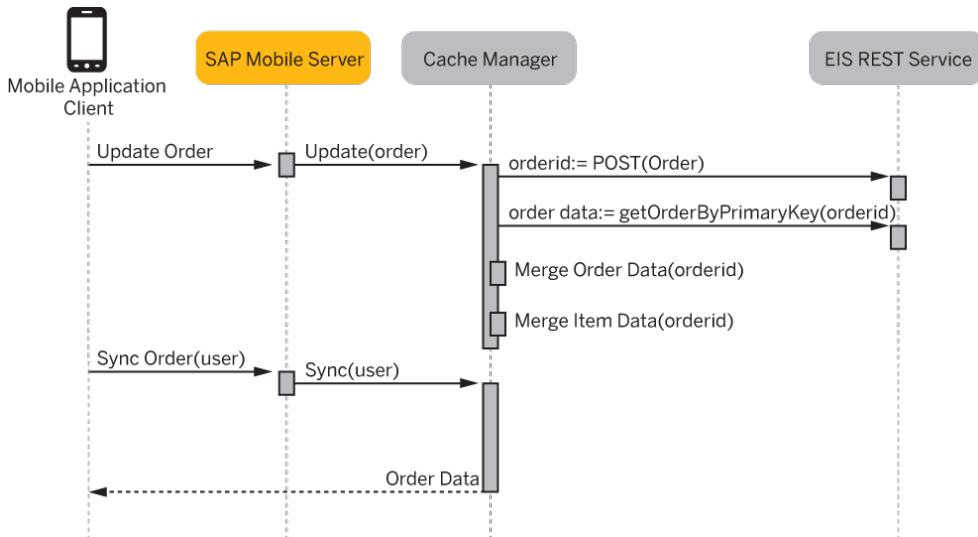
When executing an operation that is configured for object graph-level immediate refresh:

1. The client executes an operation that is configured for graph-level refresh and synchronizes.
2. The operation is replicated to Unwired Server during synchronization.
3. Internally, Unwired Server:
   a. Begins a new transaction and reads the replay operation from the operation replay queue.
   b. Invokes the corresponding data services operation, passing in surrogate key and other client attributes specified by the client application.
   c. Obtains a graph lock to prevent concurrent modification of the graph by other refresh activities such as On-demand refresh, Scheduled refresh, or DCN.
   d. Invokes the EIS operation within the context of the operation replay transaction.
   e. Combines the EIS operation results with the client attributes to determine the business primary key of the affected graph.
   f. Invokes an Entity Read operation on the EIS to read the entire graph.
   g. Merges the results of the Entity Read operation with the contents of the graph currently held in the cache bringing the cached graph up to date.
   h. Commits the operation replay transaction.

Operation Cache Policies and Client Synchronization
Operation cache policies are server-side configurations only and have no impact on specific clients.

For example, the effect of setting the "Immediately update the cache" policy is:

1. A Client invokes the Create, Update, or Delete operation on an MBO and then performs a sync operation.
2. The operation executes on the EIS and the EIS returns a result-set.
3. When "Immediately update the cache" is checked the results from step two are used to refresh the cache.
4. Unwired Server runs the download cursor on the Unwired Server cache and returns the results as part of the client sync.
5. The client database is refreshed with the latest EIS data.

"Immediately update the cache" can be more efficient with regard to cache refresh compared to cache groups - in particular if your client application is the primary updater of the EIS, it may allow a larger coherence window for on-demand refresh or replace it completely.

Composite Operations

The MBO designer can model a create, update, or delete operation on the root node of a composite relationship graph that accepts the entire composite graph as input.

The composite operation allows mapping of EIS operation arguments to any or all of the MBOs contained in the composite MBO graph such that child nodes are not required as explicit parameters to parent operations. Creation, deletion, or modification of child nodes in a composite operation results in the addition of these nodes to the operation replay graph sent to Unwired Server when the composite operation on the root MBO is submitted. Composite operations involving child nodes in a composite graph do not result in separate replay operations for the child nodes. In other words, the device application developer can create a composite MBO tree in one create call from the root MBO.

You can define CUD operations on child MBOs in a composite graph, but those operations are not (and are not part of) composite operations and must be explicitly invoked by the client (with their own replay operations).

The operation replay includes only composite graph nodes which have changed in the operation replay message, which minimizes network traffic from the client device. Unwired Server is responsible for assembling any missing portions of the graph from the cache prior to invoking the EIS operation.

After the EIS operation executes, the operation cache policy determines how those results are applied to the Unwired Server cache.

Composite Operation Data Flow

Understand how a composite operation together with an operation cache policy merges client-provided parameters with merged results and applies them to the Unwired Server cache to create, add, and delete results for a composite object graph.

The data model/use case described here focuses on variations in the way that a composite graph can be modified on the client and the implications of each modification type on Unwired

Server processing. The Order, Item and ItemNote entities that form the data model are used for examples. The client application updates, creates, or deletes one or more nodes in the composite graph and then submits the replay operation to Unwired Server, which constructs the composite MBO graph, binds the graph to the EIS invocation parameters and invokes the REST service. The composite operations are configured to immediately "Apply merge of operation input/output", so the cache is refreshed with the current EIS version of the graph after the EIS operation is invoked.

Create:
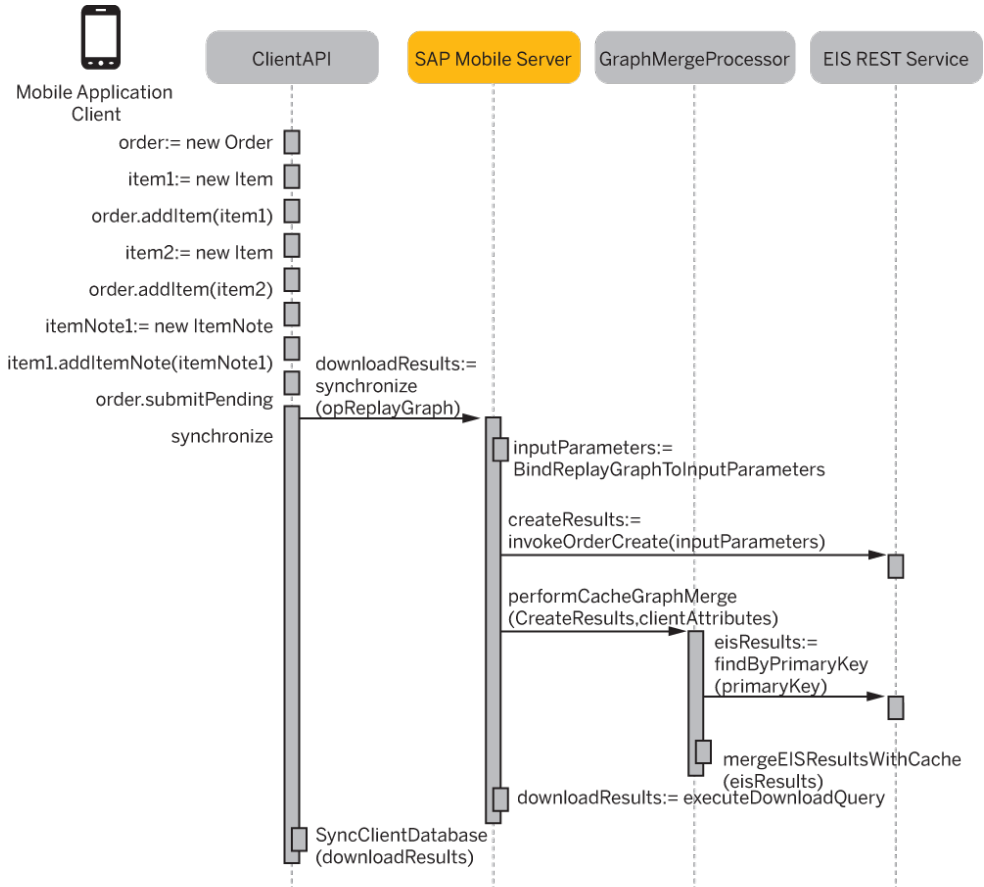
*Line breaks have been inserted into code for reabability.

Add:
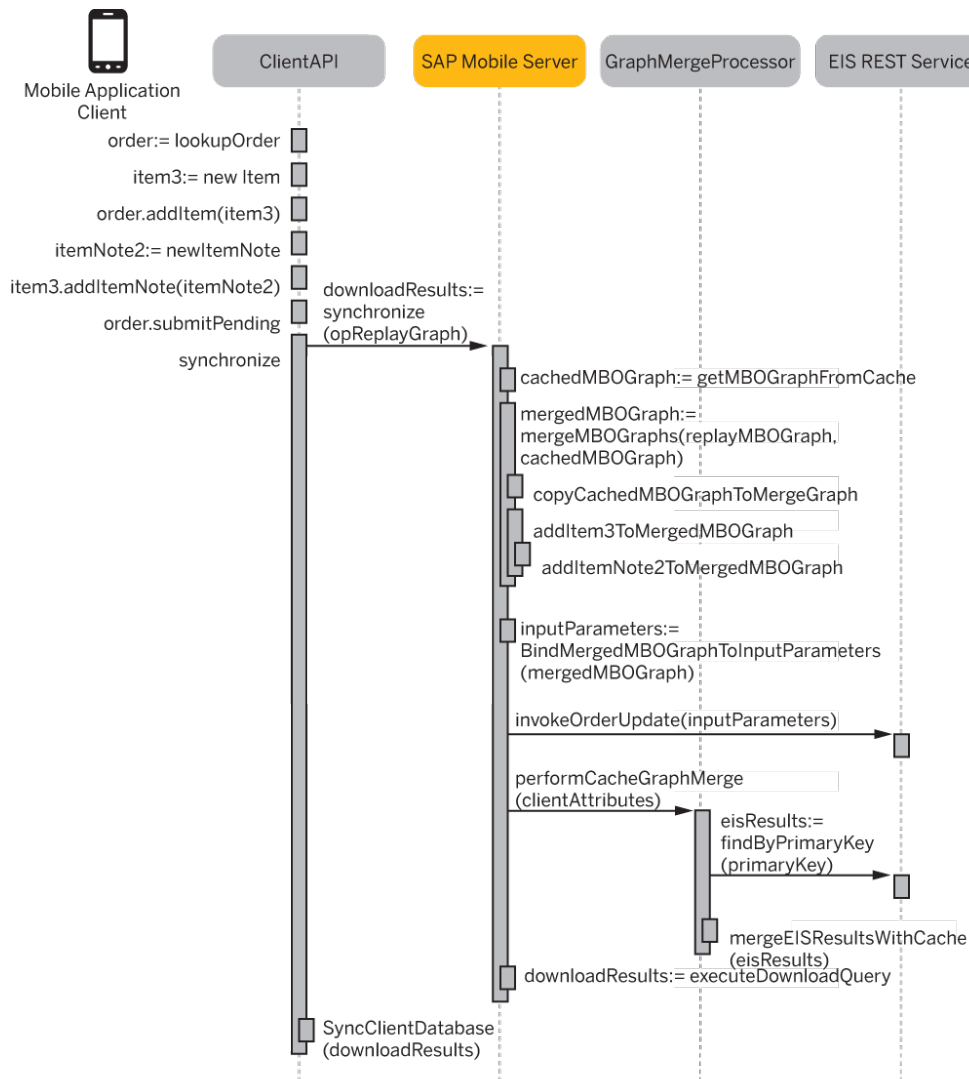
Line breaks have been inserted into code for reability.

Delete:
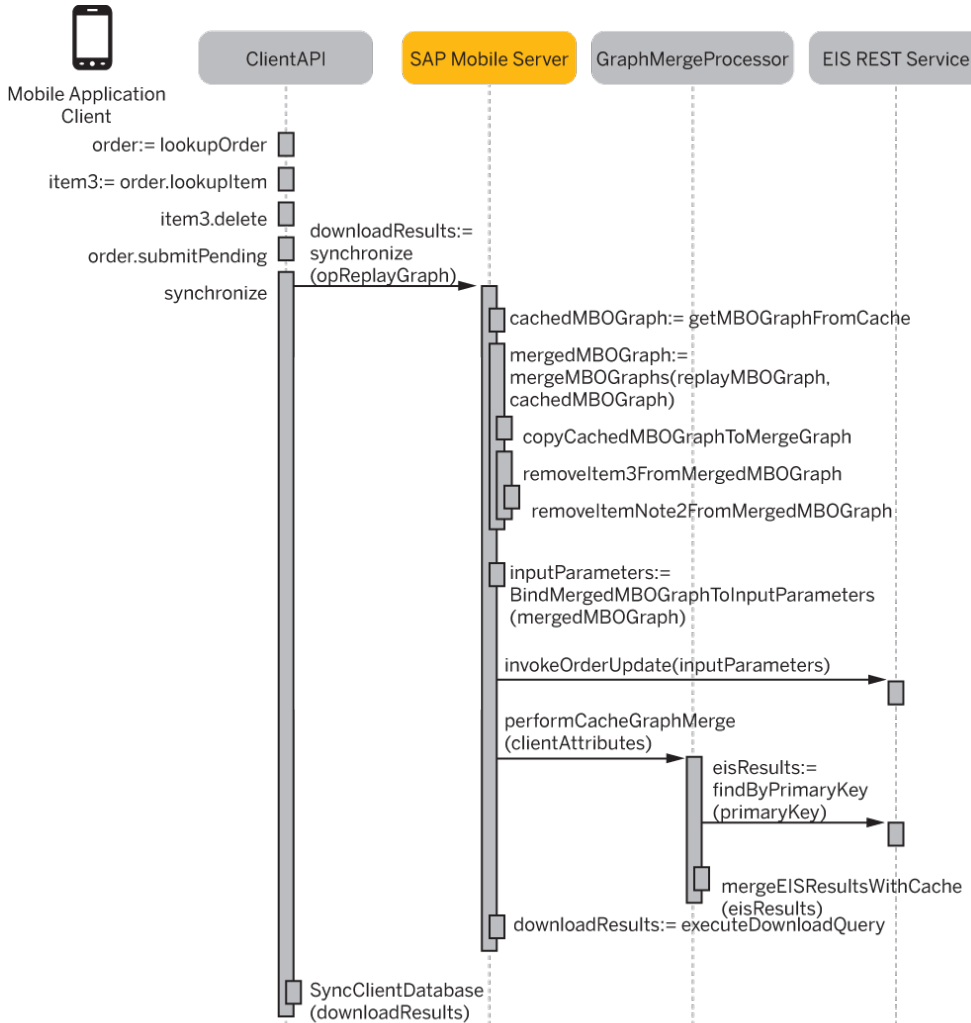
Line breaks have been inserted into code for reability.

Creating a New Composite Object Graph

This use case describes the creation of a new composite relationship consisting of Order<-Item<-ItemNote MBOs.

The following diagram shows the construction of an Order with two Items and one ItemNote:

1. When the pending operation on the Order is submitted a replay operation is sent to Unwired Server which contains the entire composite graph.

---

2. Since the replay operation contains the entire graph, there is no need to merge the replay graph with the cache to pick up missing graph parts.

3. The operation replay graph is bound to the EIS Order create operation's input parameters and the Order create operation is invoked.

4. The primary key contained in the create operation results is used to query the EIS for the current version of the graph which is then merged into the cache.

5. The download query executes and the query results are returned to the client.
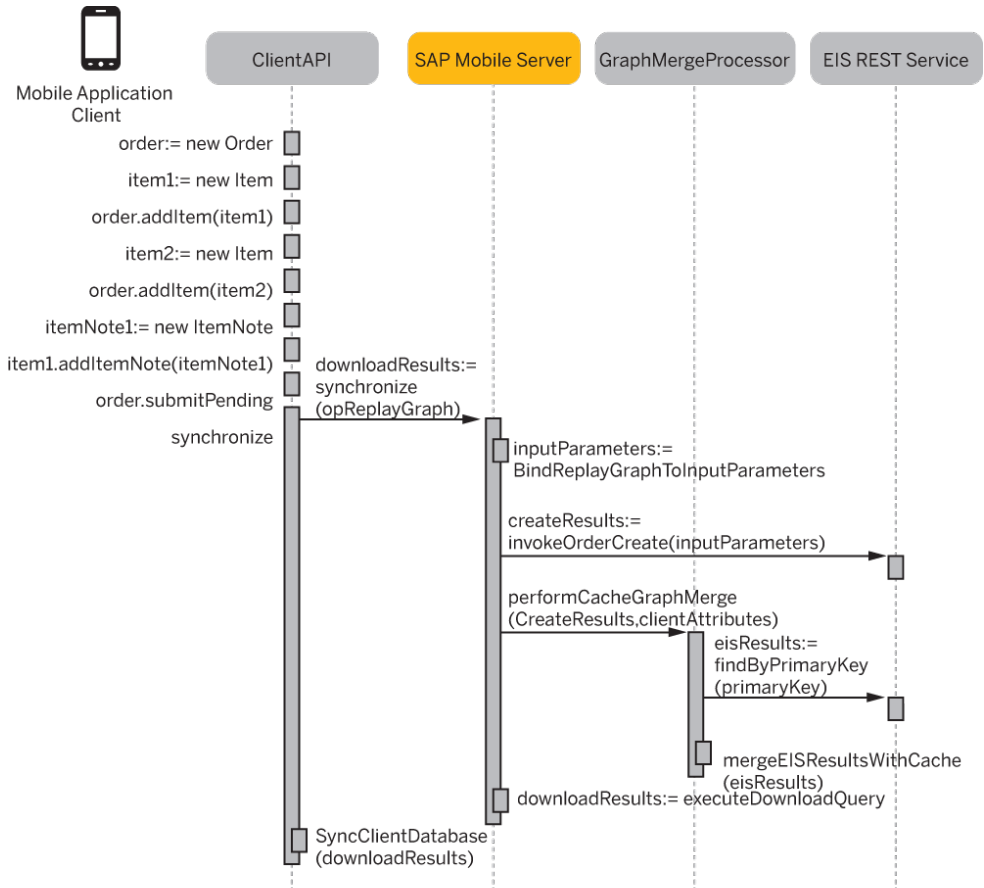
Line breaks have been inserted into code for reabability.

Adding a Composite Child Graph

This use case describes the creation of a new lineItem and ItemNote and attaching it to an existing Order.

The following diagram illustrates the addition of a new Item and Note to the Order created in the previous use case:

1. The client looks up the existing order, creates a new Item with an attached ItemNote and adds the new Item to the existing Order.
2. Since this modifies an existing graph and does not create a new one, the graph contained in the replay message cannot be assumed to be complete, so Unwired Server merges the partial MBO graph contained in the operation replay message with the graph currently contained in the cache.
3. When creating the merged MBO graph, two MBO nodes are identified in the operation replay graph which are not present in the cached graph such that the merged graph consists of the cached graph plus two additional MBO nodes (the new Item and ItemNote nodes).
4. The merged MBO graph is assembled and bound to the parameters of the Order update operation, and the update operation is invoked. If the update operation is configured for entity-level refresh, the findByPrimaryKey operation is invoked to retrieve the current graph from the EIS so that it can be merged into the cache. In this case the primary key required by the findByPrimaryKey operation is derived from the clientAttributes rather than from the results of the update operation.
5. During cache merge processing the surrogate keys provided in the client attributes for the new Item and ItemNote are preserved in the cache.
6. Cache changes are downloaded to the client.
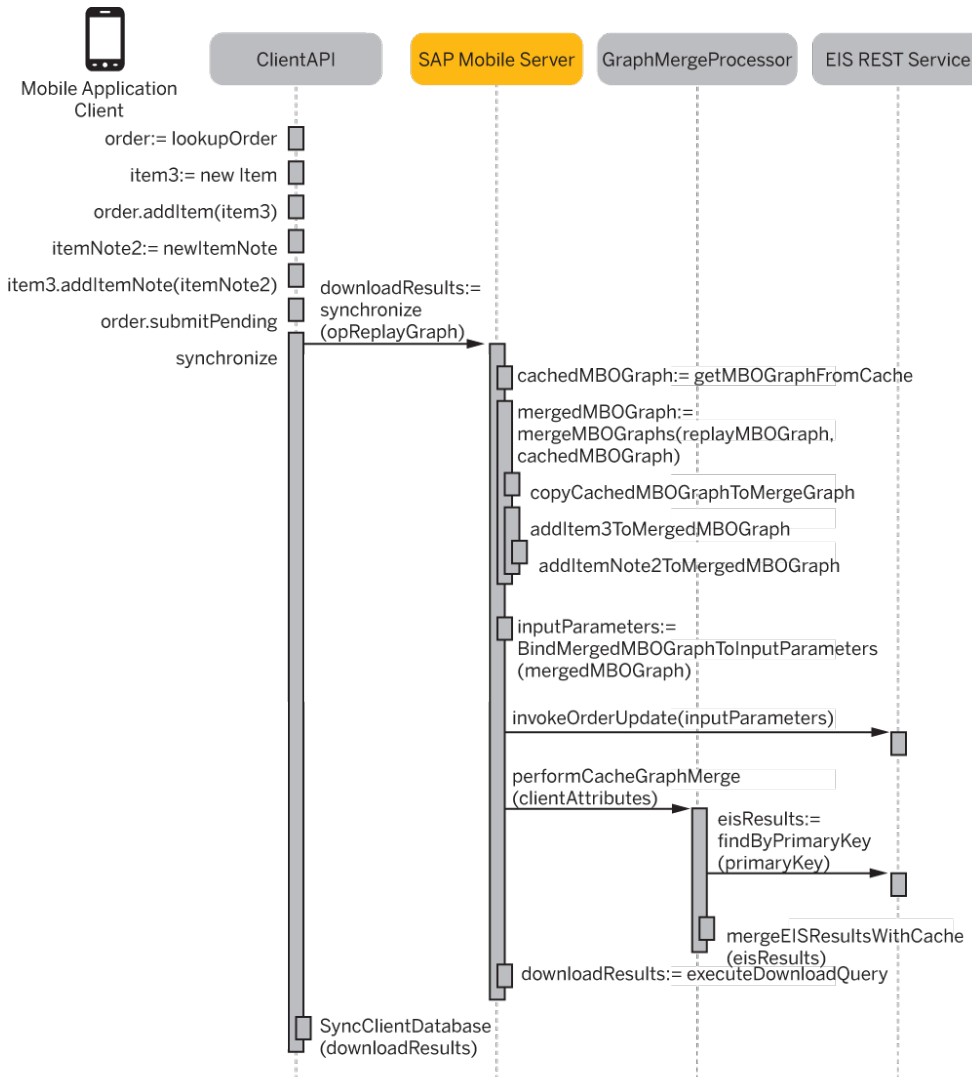
Line breaks have been inserted into code for reability.

Deleting a Composite Child

This use case describes the deletion of an Item from the composite graph.

The following diagram illustrates the removal of the Item that was created and added to Order in the previous example:

1. The client looks up the Order, looks up the Item and invokes the Item's delete operation. In this case the operation replay graph contains the Order node and a child Item node marked as delete pending.
2. When constructing the merged MBO graph, Unwred Server notes that the Item in the replay graph is marked as delete pending and removes it and its ItemNote child from the merged MBO graph.
3. The merged MBO graph is then bound to the Order update operation parameters and the rest of the flow is identical to the addition use case example.
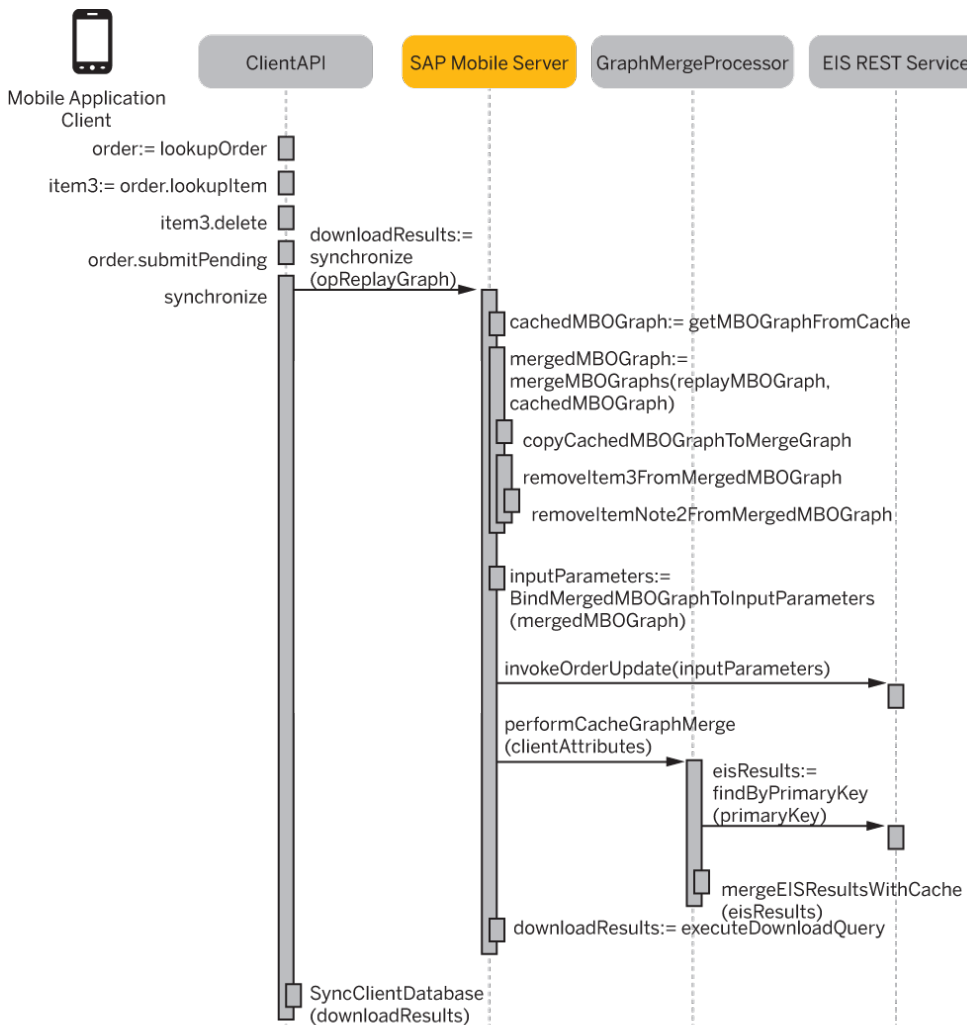
Line breaks have been inserted into code for reability.

Updating a Composite Node

This use case describes the modification of an existing ItemNote.

The following diagram shows an ItemNote created previously having its description attribute changed. In this scenario the operation replay message contains an Order, a child Item and a grandchild ItemNote which is marked as update pending:

1. When constructing the merged MBO graph the server sees that the ItemNote in the replay graph is marked as update pending and replaces the corresponding ItemNote in the merged MBO graph.
2. The merged MBO graph is then bound to the Order update operation parameters and the rest of the flow is identical to the previous use case.
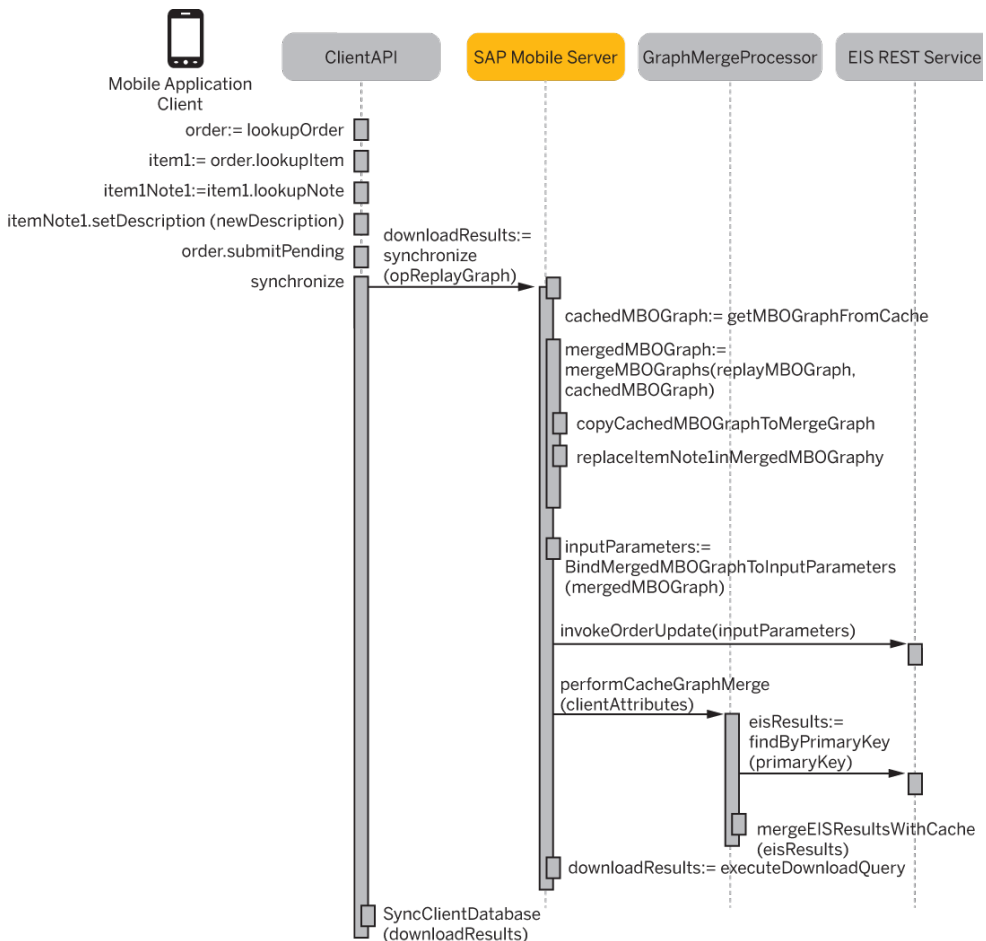
Line breaks have been inserted into code for reability.

Object Queries

Object queries are SQL statements associated with a mobile business object (MBO), against the persistent store on the device that returns a subset of a result set. For example, an object query is used to filter data already downloaded from the CDB to display a single row of a table when triggered.

Define object queries to return a subset of MBO results, either from an MBO deployed to Unwired Server or a local business object.

Table 1. Object Query Usage

MBO type	Usage
Local business object	The requested data must be available on the mobile device. If not, MBO operations must be called to insert the requested data. The query can then continue to return data from the client's local database.
Bound to a data-source	1. Create the query. 2. Call the query from the device application at runtime to display a subset of the results on the device.
Contained in a cache group that uses an Online policy	MBOs that use an Online cache group policy generate a single read-only object query named findByParameter, which is automatically generated by Sybase Unwired WorkSpace. findByParameter query parameter(s) are generated for every load argument that has a **Propagate to attribute**. The findByPrimaryKey object query and any other user defined object queries are removed for MBOs that use an Online cache policy. By default, the return type is **Return multiple objects** and **Create an index** is false, and these are the only values you can modify. If you modify a **Propagate To attribute** of a load argument belonging to an MBO using an **Online** cache group policy, the object query is automatically updated.

See *Improve Object Query Performance* in *Mobile Data Models: Using Mobile Business Objects* for information about optimizing object query performance in Native Object API applications.

Object Query Definition Guidelines

Understand how to define object queries.

Support for various compact databases
Since object queries can run on multiple mobile devices that may run different compact databases (UltraLiteJ® or SQLite® for example), object queries support a subset of UltraLiteJ SQL statements.

Table 2. Object query restrictions

SQL	Restrictions
Select statement	Supported – Order by Unsupported: • Bulk and Math functions • Group by • For • Option • Row limitation • As
Input-parameter	Supported – :name
Comparison operators	Supported: • Date format is *YYYY-MM-DD* and contained in single quotes. • Literal strings must be contained in single quotes.
From clause	Supports multiple MBOs.

See the UltraLiteJ documentation for more information about the *UltraLite SELECT statement clauses*.

Object queries must use aliases

Define the object query using an alias that references the MBO and attribute names (not table and column names from which they are derived). For example, if you have an MBO named Cust with a cust_id attribute (which is a primary key), defining this object query:

```
SELECT c.* from Cust c WHERE c.cust_id = :cust_id
```

and this parameter:

• Name – cust_id
• Datatype – INT

results in an object query that returns a single row from the Customer table. You must use an alias (c and c.attribute_name) in the query definition or an error occurs during code generation.

General behavior

General object query behavior, including assigning parameters a default value/primary key includes:

- For automatically generated object queries derived from "Primary key" settings, Sybase Unwired WorkSpace returns the attributes marked as primary key parameters.
- For manually created object queries, the parameter field allows you to select any of the available attributes, and, when selected, matches the datatype accordingly. You can still manually type in the attribute name.

Validation rules

Follow these rules when defining an object query:

- All columns defined in the MBO must appear in the result set of the object query. If not, an error is generated when running the object query similar to:

  ```
  java.sql.SQLException:JZ008: invalid column index value at 2
  ```
- "findBy" is a reserved word for an object query. If you create an operation starting with "findBy", you receive the warning message: Operation name "{0}" starts with 'findby', which may cause name conflict when generating client code.
- Do not duplicate query or operation names.
- Do not use these reserved names as the query name: "pull", "downloadData."
- Do not use an operation name, query name, or attribute name that is the same as the name of the MBO to which they belong.
- While clicking **Generate** generates a valid query, Sybase Unwired WorkSpace does not parse or validate the generated query. If you modify the query, be mindful that parameters are not validated until code is generated. For example, this error (mixed case between parameter name and query definition) is not detected until code generation or deployment:

  Parameter: `Param1 (INT)`

  Query definition: `SELECT x.* FROM TravelRequest x WHERE x.trvl_Id = :param1`

Object Query Indexes

Indexes improve the performance of searches on the indexed attributes (database columns to which the MBO attributes map), by ordering a table's rows based on the values in some or all the attributes. An index locates rows quickly, and permits greater concurrency by limiting the number of database pages accessed. An index also provides a convenient means of enforcing a uniqueness constraint on the rows in a table.

Object query examples that could serve as indexes.

Object query definition	Description
select x.fname, x.lname from Customer x where x.state := :state and x.city := :city	Create one index on the attributes "state" and "city" of the "Customer" MBO.
select x.fname, x.lname from Customer x where x.state := :state or x.city := :city	One query can only generate one index, all the attributes referenced in the query construct a composite index (state, city).

Object query definition	Description
select x.fname, x.lname, y.prod_id, y.quantity from Customer x, Sales_order y where x.id := y.id and y.prod_id := :prod_id	No index is generated for join queries.

When to Create an Object Query Index

There is no simple formula to determine whether an index should be created. You must consider the trade-off of the benefits of indexed retrieval versus the maintenance overhead of that index.

Consider these factors in determining if you should create an index:

- **Keys and unique columns –** Sybase Unwired WorkSpace automatically creates indexes for findByPrimaryKey object queries. You should not create additional indexes on these columns. The exception is composite keys, which can sometimes be enhanced with additional indexes.
- **Frequency of search –** if a particular column is searched frequently, you can achieve performance benefits by creating an index on that column. Creating an index on a column that is rarely searched may not be worthwhile.
- **Size of table –** indexes on relatively large tables with many rows provide greater benefits than indexes on relatively small tables. For example, a table with only 20 rows is unlikely to benefit from an index, since a sequential table scan would not take any longer than an index lookup.
- **Number of updates –** an index is updated every time a row is inserted or deleted from the table and every time an indexed column is updated. An index on a column slows the performance of inserts, updates and deletes. A database that is frequently updated should have fewer indexes than one that is read-only.
- **Space considerations –** indexes take up space within the database. If database size is a primary concern, you should create indexes sparingly.
- **Data distribution –** if an index lookup returns too many values, it is more costly than a sequential table scan. Also, you should not create an index on a column that has only a few distinct values.
- **Order by –** if you use object queries (also called dynamic queries) with "order by," then you might require indexes for ordering columns to ensure that the database can use an index for ordering, rather than creating a temporary table which can be slow on a mobile device.

FindAll Object Query Guidelines

Understand FindAll query definition guidelines.

By default, a FindAll object query is generated for every MBO and uses these values:

- Name – FindAll

- Parameters – none (A FindAll query without parameters generates a query such as

```
Select * from ...
```

)
- Query Definition – SELECT x.* FROM {Entity} x
- Create an index – false
- Return Type – Multiple Objects (accepts {Single Object, Multiple Objects, Result Set})

Sybase Unwired WorkSpace validates the query name and disallows it if it is a reserved word or restricted in some way, including:

- All MBO operation names
- Standard generated getter/setter methods (Get{Attribute}/Set{Attribute})
- Standard generated relationships (Get{RelationShip}/Set{RelationShip/ Get{RelationShip}Size}
- Standard methods (find, create, delete, update, save, refresh)
- GetMetaData
- GetClassMetaData
- Anything that starts with a underscore (e.g. _init)
- IsDeleted, IsDirty, and so on
- KeyToString
- Equals
- GetHashCode
- xxxFilterBy(...)
- Bind
- Load
- Find
- Find_os
- Merge
- CopyAll
- CreateBySQL
- GetDownloadState
- Set/GetOriginalState
- CancelPending
- CancelPendingOperations
- SubmitPendingOperations
- Internal_{xxx}
- SubmitPending
- FromJSON/ToJSON{List}
- GetSize
- FindWithQuery

- Subscribe_{xxxx}/Unsubscribe_{xxx}
- GetPendingObjects
- GetSynchronizationParameters
- GetLogRecords
- LastOperation
- GetCallbackHandler

Improve Object Query Performance

Investigate and optimize object query performance for Native Object API applications on iOS, Windows Mobile, BlackBerry, and Android platforms.

While most simple queries run with reasonable performance, complex queries require a good understanding of the underlying database schema derived from the MBO data model, as is the case in a client/server computing environment where the DBA is often consulted to develop optimized stored procedures or perform database tuning. The differences with mobile application development are that:

- The developer is the DBA of the device database. A device database engine does not have the complex optimization strategies nor does it keep statistics that an enterprise database server does.
- Resources on a device are limited, and heavy or complex queries often result in significant performance implications. For example, hash joins are not supported on the Ultralite database due to potentially large memory requirements for less capable devices.

The Query API is used on Windows Mobile, BlackBerry, and Android platforms, while SUPQuery is used by iOS. Mobile application developers can use these APIs to query the on device database and return data in the form of a result set.

The SQL statement is generated based on the Query/SUPQuery object constructed by the application and allows the Object API to inject conditions to deal with the object states necessary for offline computing. When an object is modified, there is a pending state (current), a download state (server), and the original state (before first modification). The pending and download states reside in the MBO table. The extra conditions in the SQL statement enable the return of either the pending state (if there is one) or the download state. No query optimization is performed by the Object API during the generation of the SQL statement. See either *Query and Related Classes* or *SUPQuery and Related Classes* in the *Developer Guide: yourPlatform Object API Applications*.

Examining the Database Schema and Data Size

Use Sybase® Central from the Sybase Unwired Platform installation to connect to and examine the device's database file. The UltraLite database udb file is compatible across device platforms.

Keep the following in mind when examining the database schema and data size:

- Query/SUPQuery operates at the database level. The MBO attribute is simply translated into a corresponding table column if different.
- Tuning requires knowledge of the database schema generated by the Object API to back the MBOs. The best way to examine the schema is to make a copy of the UltraLite database after initial synchronization finishes to make sure it contains the data set that the query operates upon.
- Query performance depends greatly on the data set. Mock data without the characteristics of the operational data is not useful for tuning.
- Initial synchronization may require the developer to define various synchronization parameters to download the operational data.
- You can use an actual device or a simulator to create and populate the database.

1. Make a copy of the UltraLite database file.

   The *databaseName*.udb file location depends on the device platform or simulator. It is generally found in the application's data folder.

2. Launch Sybase Central (scjview.exe) to manage SQL Anywhere and Ultralite databases.

   The default installation location of the Sybase Central executable is *SMP_HOME*\Servers\SQLAnywhere12\BIN32\scjview.exe.

3. From Sybase Central, select **View and edit the schema of a database**.

4. Provide the connection information to the UltraLite database file copied earlier.

   Generally, you need not specify either a user ID or password, since the Object API does not set them. However, if the database is encrypted, you must supply the encryption key to open the database.

5. Click **Connect** to connect to the copied database file.

Obtaining the SQL Statement Generated From the Query

Set enableTrace in the client code to trace the SQL statement.

The SUPConnectionProfile (iOS) and ConnectionProfile (Android, BlackBerry, Windows Mobile) class includes APIs used to set tracing with or without payload:

To obtain the parameter bindings for the SQL statement, trace with payload set to true.

1. Set the appropriate trace level in the application code by inserting small fragments of code to bracket the execution of the Query or SUPQuery.

   Note: SAP recommends that you do not enable tracing from Sybase Control Center, as doing so negatively impacts performance, and creates a large trace file that makes it difficult to locate the SQL statement corresponding to the Query/SUPQuery.

2. Use this code to turn on or off tracing of the SQL statement with payload in the trace file.
   - iOS:
     ```
     (void) enableTrace:(BOOL)enable
     (void) enableTrace:(BOOL)enable withPayLoad:
     ```

```
(BOOL)withPayLoad

SUPConnectionProfile *cp = [SMP101DB getConnectionProfile];
   // Enable SQL trace with payload of the query
      [cp enableTrace:YES withPayload:YES];
      QueryResultSet cursor = SMP101DB.executeQuery(query);
   // Disable SQL trace
      [cp enableTrace:NO];
```

- Android, BlackBerry and Windows Mobile:

```
ConnectionProfile cp = SMP101DB.getConnectionProfile();
// Enable SQL trace with payload of the query
   cp.enableTrace(true, true);
   QueryResultSet cursor = SMP101DB.executeQuery(query);
// Disable SQL trace
   cp.enableTrace(false, false);
```

Showing the Execution Plan

Use **dbisql** to read the execution plan of the code that you are tracing.

1. If it has not already been done, register the Ultralite ODBC DLL from the *SMP_HOME* \Servers\SQLAnywhere12\BIN32\ directory by executing:c:\Windows \System32\ regsvr32.exe ulodbc12.dll.

2. From *SMP_HOME*\Servers\SQLAnywhere12\BIN32\, execute **dbisql**. When prompted to connect to a database of a specified type, click **Change database type**, select **UltraLite**, and provide connection information.

3. From the Sybase Central main menu, select **Tools > Plan Viewer**.

4. Copy and paste the SQL statement from the trace file obtained previously into the "SQL" area and press **Get Plan**.

   See *Reading UltraLite execution plan* in the online help for the version of SQL Anywhere installed as part of Sybase Unwired Platform to familiarize yourself with execution plan operations before examining your execution plan.

Interpreting the Execution Plan

Interpret the execution plan and, based on the results, optimize object queries.

Relationship Between Query, Statement, and Plan

This example of a simple join between two MBOs or tables illustrates the relationship between the SUPQuery/Query, generated SQL statement, and the execution plan. The column names may be different from the attribute names, depending on the version of the client SDK used. The table with the suffix "_os" stores the original state of the MBO and eliminates duplicate rows if the object is in pending state.

The SUPQuery:

```
Query query = new Query();
query.select("p.PRODUCTID, p.TYPE, t.DESCRIPTION");
query.from("PRODUCTS", "p");
query.join("PRODUCTSDESCRIPTION","t","t.PRODUCTID","p.PRODUCTID");
```

The SQL statement generates:

```
select p."a",p."b",t."c"
from "slowquery_1_0_productsdescription" t,"slowquery_1_0_products"
p
where p."a"= t."a"
 and (t."pending" = 1 or not exists (select 1 from
"slowquery_1_0_productsdescription_os" t_os where t."d" = t_os."d"))
 and (p."pending" = 1 or not exists (select 1 from
"slowquery_1_0_products_os" p_os where p."p" = p_os."p"))
```

Execution Plan

According to the plan below, the join is implemented using a table scan of the
"productsdescription" MBO. It attempts to look up a row for each row on the "products" MBO
using the index:

```
slowquery_1_0_products_findByPrimaryKeyIndex
```

The filter and subquery operations work together to limit the number of rows returned by the
scan to consist only of the pending (if there is one) or download state. Read-only MBOs do not
require filtering since there is only one state – download.

Note: The column names that appear in the SQL query are short names. The mapping between
the attribute name and the short name is in the generated code. In the internal directory of the
generated code, there is a source file that embeds the metadata information for each MBO. For
example, in iOS, the file is in the form <Package><MBO>MetaData.m. For previous releases,
the metadata information is generated only if you select the **Generate Metadata** option
during code generation.

```
join
     [
      filter[scan(slowquery_1_0_productsdescription)],
      filter[index-scan(slowquery_1_0_products,
                     slowquery_1_0_products_findByPrimaryKeyIndex)]
     ]
 sub-query[index-scan(slowquery_1_0_products_os,primary)]
 sub-query[index-
scan(slowquery_1_0_productsdescription_os,primary)]
```

Table Scans

Small table scans can be quite efficient; however, avoid large table scans as much as possible
due to the linear cost. For attributes in a WHERE clause, avoid a table scan by adding
appropriate indexes. UltraLite uses direct page scans when it is more efficient to access
information directly from the database page, however, such results are returned without order.
The object query must use ORDER BY to guarantee ordering of result sets.

Joins

Join operations are common in a relational database. However, too many joins within a query,
especially queries involving tables with many records, are very expensive to perform. Because
of the limited resources associated with the device database, dividing a query into two or more

---

subqueries provides better performance. Since the database engine is single threaded, allowing another application thread to process the result of a subquery while the application thread in the database engine works on another query in parallel may provide better performance in a multicore environment. This might involve additional work for the application developer, but may also allow complex queries to execute with reasonable performance.

Also, for a join operation, the query optimizer does not use table size information or gather statistics to decide which table to scan. The table order in the SQL statement (derived from SUPQuery/Query) determines the table to scan. Therefore if you are joining a very large table with a small one, ordering is important to limit the amount of work the engine must do.

Temporary Tables
Avoid using temporary tables, if possible, since they are fairly expensive. There is a large discrepancy between read and write speed for flash-based memory and storage. When using UltraLite databases with temporary tables:

- In general, the optimizer always tries to avoid creating temporary tables to return query results, because the entire temporary table must be populated before the first row can be returned. If an index exists, the optimizer tries to use the index first and creates a temporary table only as a last resort.
- A temporary table is used by an access plan to store data during its execution in a transient or temporary worktable. This table exists only while the access plan is being executed. Generally, temporary tables are used when intermediate results do not fit in the available memory, such as when:
  - Subqueries must be evaluated early in the access plan
  - Data in a temporary table is held for only a single connection only
  - A query contains an ORDER BY on a column other than an index
  - A query contains a GROUP BY on a column other than an index
- It is difficult to anticipate whether an index you have created avoids the necessity for a temporary table. Therefore, always check the plans for a query to ensure that the indexes you have created are actually being used by the UltraLite query optimizer.
- You can avoid using temporary tables by using an index for the columns used in the ORDER BY or GROUP BY clauses.

Indexes
The optimizer looks at query requirements and checks if there are any indexes it can use to improve performance. If there is not, then the optimizer uses either a temporary table or a direct page scan instead. Therefore, you may need to experiment with your indexes, and frequently check generated execution plans to ensure that you are:

- Not maintaining indexes that are not being used by the optimizer
- Minimizing the number of temporary tables being created

The Object API uses the surrogate key scheme; the surrogate key is part of a composite primary key with the pending flag. A second index is created for the business key of the MBO,

designated by xxxx_findByPrimaryKeyIndex. These are all the indexes for each MBO that are updatable. For read-only MBOs, the primary key consists of only the surrogate key. If the query to be optimized requires additional indexes, the MBO developer must create such indexes from the Object Queries tab in Sybase Unwired WorkSpace.

Create the index by creating an appropriate object query. The new object query results in a small amount of generated code that does not impact application performance. For example, if you create an index for the attribute "zip", you need only create an object query using "zip" in the WHERE clause.

While a new object query does not impact performance, an additional index may slow down synchronization and update/insert operations. Do not create indexes to prepare for future upgrades that may or may not materialize. Even for a read-only MBO, additional indexes may slow synchronization. If MBO data changes infrequently and is limited in size, then the only concern is initial synchronization time. Faster query performance and synchronization speed is a trade-off. Indexes are an important tuning tool but must be used with caution: they are important for large tables due to the cost of table scans; however, redundant indexes slow down insert/update/delete operations, but provide no benefit.

Materialized Views

For complex queries involving large reference tables that change infrequently, it may be beneficial to construct a materialized view to have results return within user experience limits. You can construct such views using a local MBO, and maintain them in the application code. The change log facility notifies the application of changes to the data set after synchronization completes. Based on the changed/new MBO instance, the application code can update the materialized view. As a result, the query can leverage the materialized view without a join operation and spread materialized view maintenance out over time. SAP does not recommend this approach for volatile data, as maintenance processing is complex and time consuming. Additionally, the most recent view is delayed until processing completes so the user may see only partially updated materialized views.

Datatype Support

Sybase Unwired WorkSpace supports a variety of datatypes, from a simple type to an array of objects.

Mobile business object (MBO) attributes and argument/parameter datatypes map to datasource datatypes. Select the datatype of a given argument/parameter or attribute from the datatype drop-down list, which maps to the datasource's datatype. You define attribute and parameter datatypes in a number of Sybase Unwired WorkSpace locations, depending on the MBO development phase, including:

- Creating MBOs – when creating MBO operations and attributes in these locations:
    - Attributes Mapping wizard
    - Client Parameters page (when deferring binding to a datasource)
    - Attributes page (when deferring binding to a datasource)
- Editing MBOs – when editing MBO attributes and parameters from the Properties view in these locations:
    - Load Arguments tab
    - Attributes Mapping tab
    - Synchronization tab (for synchronization parameters)
    - Object Queries tab and Object Query creation wizard
- Testing MBOs – use the Test Execute and Preview dialogs for testing mobile business object operations or previewing attributes.
- Creating and editing personalization keys – holds the personalization parameters and supports the same datatypes as MBO attributes and parameters.

Since attributes and parameters depend on the datasource to which the MBO maps, not all attributes and parameters support all datatypes. Generally, if a datatype does not display in the drop-down list, it is not supported for that MBO.

When defining the default value for a parameter with the maxlength setting, the maximum length also applies to any localized (i18n) values (Including some double-byte character languages, such as Chinese).

Sybase Unwired WorkSpace supports various categories of datatypes.

Table 3. Datatype Categories

Category	Description
Simple	int, string, date, bigString/bigBinary, and so on.
List of simple types	An array of simple types: int[], string[], date[], and so on.

Category	Description
Structure (complex)	Implemented with a structure, and includes: • SAP input structure or input table • Nested complex types in Web Services that handle type structures as input (repeating and nested elements) **Note:** Parameters, personalization keys, and default values all support complex types. Attributes do not, except for those that represent relationships.
List of structure types	An array of objects: customer[], account[], and so on.

The following table describes how to input values for various simple datatypes when specifying default values of parameters of these types in Sybase Unwired WorkSpace.

Table 4. Simple Datatype Description

Datatype	Description
binary (%n)	Select either: • Input manually – enter a base64 encoded string directly in the input field. • Import from file – browse to a file from which the input string is retrieved. To set the length, click the cell you require in the datatype column and select binary(%n). Press enter and then type the size you require. For example, you could: 1. Click the particular cell in the datatype column, and select binary(%n) from the list of datatypes. 2. Enter the value for the binary length by highlighting %n with your cursor, and replacing it with the size you require. 3. Press enter to set the binary length. For example, if you entered 10 as the binary length, you see binary(10) in the datatype cell. **Note:** The maximum allowable length for binary datatypes is 2G bytes. If the MBO's attribute is a primary key, the maximum allowable length for binary datatypes is 2048 bytes. For MBO parameters, binary default value cannot exceed 16384 bytes.

Datatype	Description
date	Select the day in the provided calendar. By default the local time zone is selected. The **Time zone** field is read only.
dateTime	By default the current date, time and time zone display in the calendar.
time	Enter the time, and enter the local time zone in the **Time zone** field.
string(%n)	To set the string length, click the cell you require in the datatype column and select string(%n). Press enter and then type the size you require. For example, you could: 1. Click the particular cell in the datatype column, and select string(%n) from the list of datatypes. 2. Enter the value for the string length by highlighting %n with your cursor, and replacing it with the size you require. 3. Press enter to set the string length. For example, if you entered 10 as the string length, you see string(10) in the datatype cell. **Note:** The maximum allowable length for string datatypes is 2G bytes. If the MBO's attribute is a primary key, the maximum allowable length is 512 bytes.

Datatype	Description
BigString/BigBinary	Allows you to transfer large binary or string data from/to Unwired Server by using the ObjectAPI streaming methods to optimize memory consumption. For example, you can set an MBO's attribute/arguments as BigString/BigBinary and get/set the data by a streaming method such as seek/write/read/flash in the client code.
	To use either BigString or BigBinary types and the associated streaming objectAPI, an MBO operation argument must be mapped to a Fill from attribute that uses the large object type. An error is generated if an MBO operation argument has a BigString or BigBinary type but does not have the associated Fill from attribute with the same large object type. Not all attribute/arguments support BigString/BigBinary:
	1. Only operation argument/structure object attributes can be set as BigString/BigBinary, which have no length limitation. Use a streaming I/O mechanism to optimize memory consumption at runtime.
	2. Personalization key/sync parameter/load argument/object query parameter/client parameter are not allowed to set BigString/BigBinary datatype.
	3. In the attribute mapping section of properties view, primary key attributes are not allowed to have BigString/BigBinary data type. If you set a primary key attribute as BigString/BigBinary, SAP Mobile WorkSpace displays a validation error. The BigString/BigBinary options are still available since you can unselect the primary key checkbox.
	4. To indicate an operation argument needs to use BigString/BigBinary, change the mapped attribute's datatype.
	Note: Ensure that your Unwired Server host is a 64-bit machine with a heap size larger than 512 MB to avoid "java.lang.OutOfMemoryError: Java heap space" errors when processing BigString or BigBinary datatypes.
All others	Enter the appropriate value in the **Value** field.

Time Zone Datatype Behavior

Because enterprise resources are frequently located in different time zones, you need to understand the restrictions of using time-related datatypes when developing mobile business objects (MBOs).

Zone-offset independent field based time
Unwired Platform date, time, and dateTime datatypes hold time zone independent field-based time, as defined in *Incremental versus Field-Based Time*, making zone offsets invalid. For example, if you specify a default value for a synchronization parameter, it is not valid to include the zone offset:

```
2009-08-28T00:00:01+08:00
```

If a device application needs an attribute to store zone offset or zone name data, then define another MBO attribute to contain it. When previewing or testing date, time, and dateTime datatypes, values related to zone-dependent fields in Web services undergo the conversions described in this document. You need to account for any adjustments in the expected results, and may need to adjust any MBO default values you set.

Receiving values from Web services
Zone-offset behavior of date, time, and datetime datatypes:

- xsd:date – if Unwired Server receives a date value from a Web service that includes a zone offset (ending with "Z", "+XX:XX", or "-XX:XX"), Unwired Server ignores and drops the zone offset suffix. DATE values stored in the Unwired Server cache database (CDB), and sent to the client, do not include any zone offset. For example, if Unwired Server receives an xsd:date value:

  ```
  2000-01-01+12:00
  ```

  it converts that value to a DATE:

  ```
  2000-01-01
  ```

- xsd:time – if Unwired Server receives a time value from a Web service that includes a zone offset (ending with "Z", "+XX:XX", or "-XX:XX"), Unwired Server ignores and drops the zone offset suffix. TIME values stored in the CDB, and sent to the client, do not include any zone offset. For example, if Unwired Server receives an xsd:time value:

  ```
  14:00:00+12:00
  ```

  it converts that value to a TIME:

  ```
  14:00:00
  ```

- xsd:dateTime – if Unwired Server receives a dateTime value from a Web service that includes a zone offset (ending with "Z", "+XX:XX", or "-XX:XX"), Unwired Server adjusts the fields to convert the value to UTC (+00:00) and then drops the zone offset suffix. DATETIME values stored in the CDB, and sent to the client, do not include any zone offset. For example, if Unwired Server receives an xsd:dateTime:

```
2000-01-01T14:00:00+12:00
```

it converts that value to a DATETIME:

```
2000-01-01 02:00:00
```

If Unwired Server receives a value from a Web service that does not include a zone offset, then
Unwired Server uses the unchanged value.

Sending values to Web services
By default, Unwired Server appends a "Z" to any date/time value that it sends to a Web service.
If a Web service expects to receive zone-independent field based time, then use a derived
simpleType with a pattern restriction in the XML schema description to indicate to the XML
parser that only zoneless representation is accepted by the Web service. For example, you
could define these simpleTypes:

```
<s:simpleType name="ZonelessDate">
        <s:restriction base="s:date">
                <s:pattern value="[0-9]{4}-[0-9]{2}-[0-9]{2}"/>
        </s:restriction>
</s:simpleType>

<s:simpleType name="ZonelessTime">
        <s:restriction base="s:time">
                <s:pattern value="^[0-2][0-9]:[0-5][0-9]:[0-5][0-9]
                (.([0-9]{3}))?$"/>
        </s:restriction>
</s:simpleType>

<s:simpleType name="ZonelessDateTime">
        <s:restriction base="s:dateTime">
                <s:pattern value="^[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-2]
                [0-9]):[0-5][0-9]:[0-5][0-9](.([0-9]{3}))?$"/>
        </s:restriction>
</s:simpleType>
```

Then use the corresponding derived simpleType instead of using "xsd:date", "xsd:time", or
"xsd:dateTime" as an element's type. These patterns are only examples and not "prescriptive."

Special consideration for client applications
As an example, you are writing a device application, and know that the enterprise information
system (EIS) Web service always expects to receive values with zone offset. You also know
(from above) that Unwired Server always sends a value with a "Z" suffix to the Web service.
How do you then pass the appropriate values (for MBO attributes and/or operation arguments)
from the device application?

In this example, the client device is located in New Zealand (12 hours ahead of UTC), and an
event occurs at device-local date/time "2010-05-12T11:24:00+12:00". Since the client using
the Object API can only pass zoneless values to MBO attributes or operation arguments, the
client application must convert the fields to UTC, for example,
"2010-05-11T23:24:00+00:00", and drop the zone offset to pass "2010-05-11T23:24:00" into
an MBO attribute or operation argument. When the client uploads this value to Unwired

Server, it appends "Z" which results in "2010-05-11T23:24:00Z", which is then sent to the EIS Web service. Since "2010-05-11T23:24:00Z" is an equivalent point in time to "2010-05-12T11:24:00+12:00", no information is lost.

In other words, if the EIS expects to receive values with zone offsets, the client application might need to do zone offset conversions to UTC. Conversely, if the EIS expects to receive zoneless values, then the client application does not need to perform any conversions from device-local time, other than dropping the zone offset.

Load arguments and timezone support for Web service MBOs
Avoid using date/datetime datatypes as load arguments, since you could get unexpected results. If you want to use Web service MBO operations that have time zone offsets, convert the date/datetime value to UTC, before sending it to Unwired Server. When the date/datetime value is returned from Unwired Server, change it back. Other considerations to be aware of:

* Date/Datetime datatype personalization keys for Web services should be avoided – if the client is in a different time zone, the same time change to UTC may be different, requiring a conversion to the personalization key whenever entering a different time zone.
* Default values for Date/Datetime datatypes in Web service operations with time zone offset should be avoided – since the default values are set in the MBO at design time, the developer cannot determine which time zone the client uses, so the UTC conversion is impossible.

Datatype Default Values and Limitations

This topic provides information about datatype default values that can be set for mobile business object (MBO) attributes, arguments, and parameters.

You can provide a default value for attributes and parameters that are compatible with their datatype (and used by the device application to pass to the MBO), whereever you specify a datatype (Properties view, Preview dialog, and so on).

Note: When possible, the default value is retrieved with an appropriate value from the datasource when you bind to the datasource, which you can then modify.

NULL and empty default values
It is important to understand the differences between the default values NULL and no default (leaving the default value empty):

* NULL – datatypes that do not support NULL or the load argument/operation argument property 'Nullable' is not selected, typically do not list it as an option from the drop-down list. After an MBO is created, NULL may be an available default value, but should not be selected if NULL is invalid for that datatype or is otherwise problematic (for example, you would not allow NULL for a primary key). If NULL is selected, and is invalid for the datatype, errors occur either when you deploy the MBO to Unwired Server, or when a device application interacts with the deployed MBO. These examples illustrate how a

device application behaves when an MBO contains a synchronization parameter equal to NULL:

- Where NULL is supported – the device application receives the rows where the attribute in the MBO is NULL. If a synchronization parameter or load argument is NULL, then data refresh is performed using the value NULL.
- Where NULL is not supported – if associating synchronization parameter X with attribute X, the download cursor is similar to:

```
select ... from my_table t where t.last_modified >= ... and t.x
>= :X
```

  If X is NULL, no rows are returned.

- empty default value – an empty string is not the same as no default. For string and binary datatypes, an empty string is a valid default value. For other datatypes, an empty string is invalid and generates an error.

The default value is set according to the nullability and datatype of the argument, synchronization parameter, or personalization key. For nullable types, the initial default value is set to NULL, for non-nullable types, a valid value is set according to the datatype (for example, string "", boolean "false", decimal "0", integer "0", float "0", and so on).

The default datatype length, if you do not specify one, including migrated datatypes, is:

- STRING – 300
- BINARY – 32767

Note: When STRING and BINARY are set to default values, a warning displays indicating the possibility that data truncation may occur during runtime if the length of the EIS column and the associated data are greater than the aforementioned default settings. As the MBO Developer, use your datasource knowledge and judgement to avoid data truncation and at the same time, maximize efficiency and performance by modifying the default to a suitable length. For example, instead of using a string datatype, use string(30) if it meets the needs of your mobile application.

The total length of an MBO synchronization parameter cannot exceed its pagesize
If the total length of a synchronization parameter (sum of maxLength) exceeds the MBO pagesize, an exception is thrown during code generation. When calculating how many BYTEs it takes compared to pagesize, consider that for:

- String type, the length is 4*maxLength. In the MBO model, string maxLength is by character.
- Byte[] type, the length is just maxLength.
- Decimal and Integer types, the length is its precision.
- Other types(int, long, and so on), length can be safely ignored when giving a bit larger margin.

Ensure that the MBO developer uses a larger pagesize or makes the synchronization parameter maxlength smaller.

Setting default values for String, BigString, Binary, and BigBinary datatypes
When editing the default value of parameters/arguments, a dialog allows you to set the default value if the data type of the parameter/argument is String, BigString, Binary or BigBinary. The maximum length for the default value is 16K:

1. For BigString,Binary, and BigBinary, open up a dialog to edit the default value.
2. For String of length greater than 300, open a dialog to enter the default value, otherwise, edit it directly in the cell.

   You can enter default values directly, or select the radio button **Import from file** and **Browse** to retrieve the default value from a file.

Valid, supported value range for DateTime datatypes
When a DateTime value is stored in the database, it will only be represented accurately if it is within the range 1600-02-28 23:59:59 to 7911-01-01 00:00:00. Attempting to store dates outside this range may result in incomplete and inaccurate information.

Negative values
BYTE datatype range is 0-127 if targeted for .Net (C#) device platforms. Negative values are not supported for BYTE datatypes, otherwise synchronization may generate `PersistenceException` errors.

Remote Operation Default Value Precedence

Understand runtime precedence of default values during remote operation execution. Set default values in Sybase Unwired WorkSpace, from **Properties view > Input > Default Values** for a given operation.

- **Structure/complex operation arguments** – if the parent's structure value is empty and not mapped to MBO attributes, the parent's default value is used and all child default values are ignored. If a child element of a parent default value is empty and not mapped to MBO attributes, the child's default value or personalization key value is used.
- **Structure/complex load arguments** – where the parent's structure default value is not empty and at least one child element has a default value, if the parent's structure value is empty, the parent's default value is used. If the default value of the parent's sub-element is null, the corresponding child's default value is used.

  The parent's default value is ignored and the child's default is used in cases where the child element's value is empty.

  Note: Map structure load arguments to personalization keys of the same structure type.
- **Non-nullable operation arguments** – where a simple datatype or a simple datatype contained in a structure type and mapped to an attribute or a client parameter, if the client does not set any value to server, the real processed value is the default value of the type, for example: empty string for string type, 0 for int type.

Structure wrapper node behavior

Using Items-->Item as an example. In this case Items is a structure wrapping only Item. Sybase Unwired WorkSpace does not allow the MBO developer to set default values at the Items level (and the other child Notes if it contains only a single structure element). This is consistent with personalization key behavior. In summary:

1. If a project is created prior to version 2.3 SP03, during migration, Sybase Unwired WorkSpace:
   a. Removes the default value of the parent (wrapper level) argument and disables it.
   b. Moves the original default values at the wrapper level to the structure fields of its single child structure node.
2. If newly created MBOs and operations have a wrapper level load/input argument, Sybase Unwired WorkSpace sets its default value empty and disables editing. The MBO developer can only set the default value of the wrapper node's child structure.
3. For MBO Preview and MBO Operation Test execute, Sybase Unwired WorkSpace:
   a. Uses the wrapper argument's child element's default value.
   b. Saves the wrapped child argument's default value when the MBO developer selects 'Save as default value'.

Default value precedence examples

The default value precedence can be summed up as:

1. Within a complex input argument, the first node found with a default value is applied.
2. If MBO attributes are bound into a node of a complex input argument, any default values defined for the node are ignored.

This example focuses on the "items" input argument:

- If a default value is defined for "items" it is used to invoke the remote operation.
- If no default value is defined for "items" and a default value for "item" is defined, then the "item" default value is used.
- If a default value for both "items" and "item" is defined, the default value for "items" is used:

The "item" input argument is mapped to the "Item" MBO:

- If the modeler defines a default value for "items" or "item" the default value is ignored if the modeler has mapped the input argument node "item" to MBO attributes.
- If the client does send "Item" MBO data, but some attributes are empty or not mapped (for example "item.discount"), then a defined attribute level default is used .
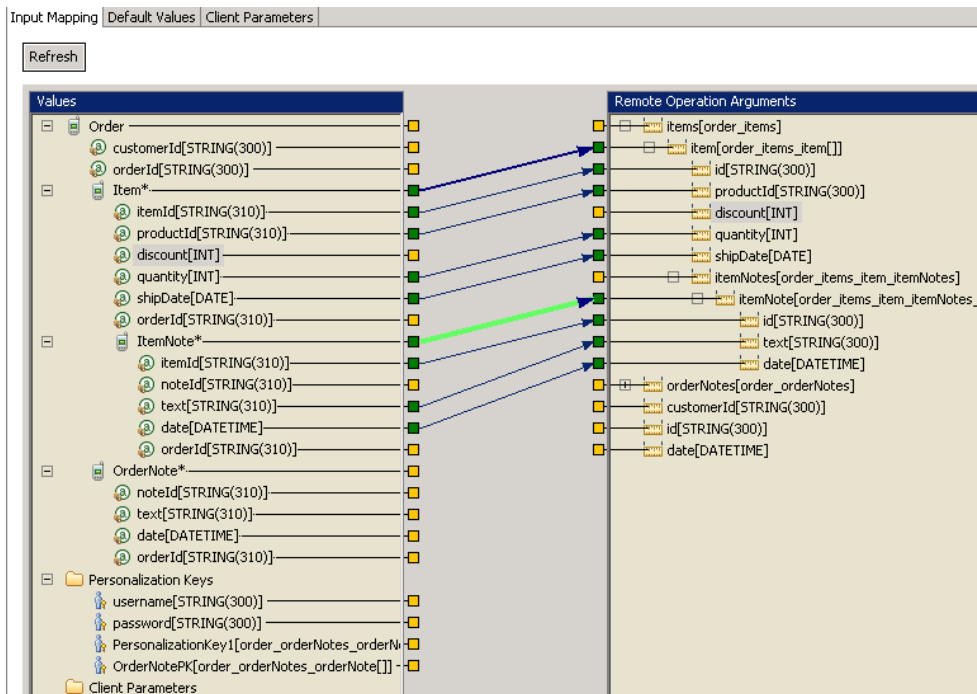
- If a default value has been defined for "itemNotes" or "itemNote" and the client does send "Item" MBO data, then the default value for "itemNotes" or "itemNote" is used:
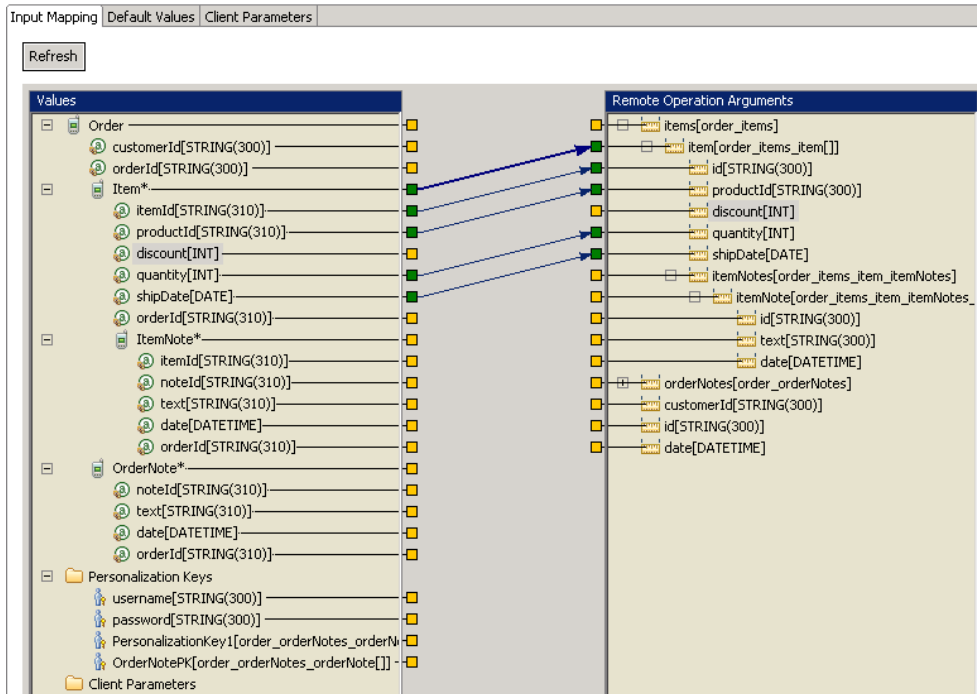
The "item" input argument is mapped to the "Item" MBO and the "itemNote" subnode is mapped to the "ItemNote" MBO:

- If the modeler has defined default values for "itemNotes" or "itemNote" the default values are ignored :

Personalization key example

Personalization keys behave similarly to default values.

In this example the "item" input argument is mapped to the "Item" MBO, and the "itemNote" subnode is mapped to a personalization key:

- If the client sends "Item" data and has set a value for "PersonalizationKey1" then the values for "PersonalizationKey1" are used for "ItemNote".

    Note: If a default value for "itemNotes" is defined then the default value takes precedence over the personalization key and will be used even if the personalization key contains data.

- If a default value for "itemNote" is defined and "PersonalizationKey1" is empty then the default value on "itemNote" will be used.
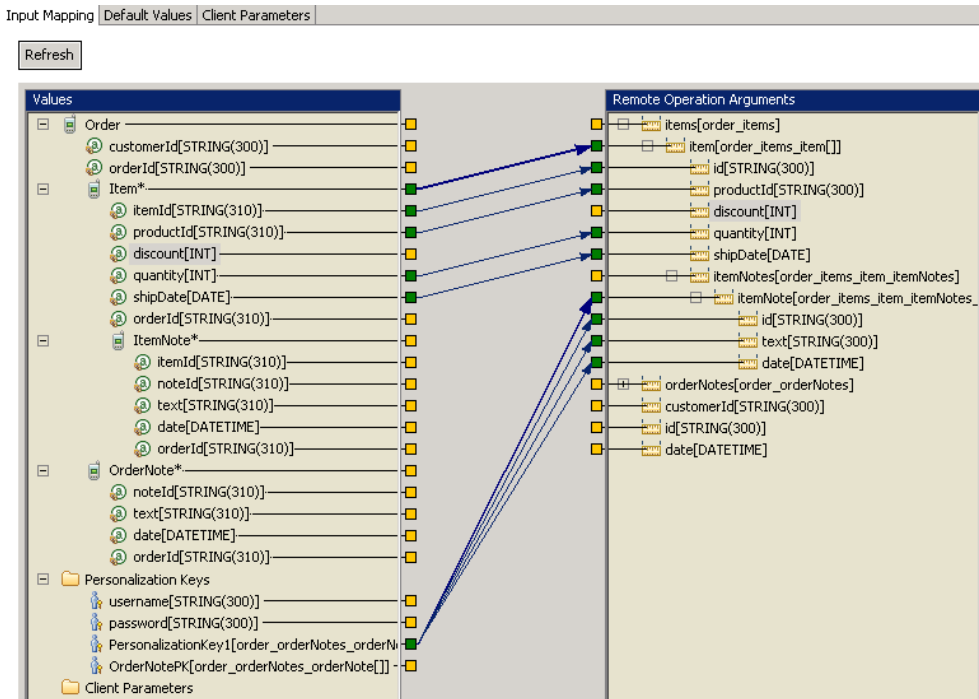
Structure Objects

Structure objects represent complex object datatypes.

Complex Datatypes

Structure objects hold complex object types (data structures), for example, an SAP input structure or input table.

When created, structure objects (or complex types) are generated into a class. The complex type contains one or more attributes. Every attribute contains type and name information.

Type	Valid element
Simple type	String
	BigString
	Char
	Double
	Binary
	BigBinary
	Integer
	Date
	DateTime
	Time
	Boolean
	Long
	Float
	Decimal
	Byte
	Short
List of simple types	An array of any of the supported simple types. For example: String[]

The name of the complex type attribute is used as the generated class name and should follow attribute naming conventions. The complex type is used in many places so naming is important for identification. For example, the complex type:

```
Address
    State
    City
    Street
```

could have this value, represented by this structure:

```
[State="Ca":City="Dublin":Street="Sybase Drive"]
```

Complex Datatype Limitations

Understand complex datatype (structures) restrictions and limitations.

Complex datatype default value limitations
If multiple parameters refer to the same structure object, and individual default values are set for the parameters, only the first value to the structure is passed and other values are ignored. To avoid this situation:

Instead of having multiple parameters referencing the same structure, copy and paste the structure object so that each reference is to a single structure object.

Complex datatype limitations

- Complex datatypes which are not bound to any MBO operation or component are not included in the generated code.
- If you create a nested tree with three levels (structures) and you delete the last node in the tree (level three), no error message displays indicating that the attribute type of one of the attributes on level two is non-existent.
- If there is a type mismatch between a synchronization parameter type and personalization key type, the error is not visible in the header area of the properties view.
- If you attempt to model a structure with a field/attribute of the same type you either get a StackOverflowError or if you try to model an attribute type as a list type of the same structure, the change is ignored.
- If you attempt to model two structures each with attributes of the other type (for example, structure5 contains an attribute of type structure6, and strcuture6 contains an attribute of type structure5) a StackOverflowError occurs.

Deleting structures

A structure can be referenced by a personalization key or a parameter or other structure's attribute. It can be deleted only if it's not referenced by any entity. If a personalization key references the structure, the deletion of the structure (either from object diagram or from workspace navigator) generates an error similar to:

```
"Structure type:'' < structure_name > '' can't be deleted,
because it is still referenced by personalization key : '' < PK_name
> ''".
```

A Cut or Delete of a structure in the object diagram generates a similar error message if the structure is still referenced by a parameter or other structure .

Creating Structure Types That are Compatible With Large Object Types for Client Parameters and Personalization Keys

Client parameters and personalization keys cannot include large object (BigString or BigBinary) types, and cannot have a structure (or list of structures) type that contains large object fields.

If an operation argument has a:

- Large object type – map a client parameter or personalization key into the argument, and use a compatible type (String or Binary) for the client parameter/personalization key.

- Structure (or list of structures) type that contains large object fields – map a client parameter or personalization key into the argument if the type of the client parameter/ personalization key is a structure (or list) with compatible fields.

For example, if the operation argument structure is:

```
MYSTRUCT
mystr[STRING(10)]
mybigbin[BIGBINARY]
```

this is a compatible structure type:

```
CP_MYSTRUCT
mystr[STRING(10)]
mybigbin[BINARY(32767)]
```

To easily create a client parameter or personalization key that matches the type of the argument and automatically add a mapping from the argument to the new client parameter/ personalization key, drag and drop an argument into the Personalization Keys or Client Parameters folder of the Input Mapping figure. When the argument is a structure containing large object fields, and you drag and drop it on the Personalization Keys or Client Parameters folder, a new compatible structure type is created:

- For client parameters – a new client parameter is added with the compatible type, and the mapping added.
- For personalization keys – the New Personalization Key wizard is invoked, with the initial type set to the new compatible structure type.

Unwired Platform to Enterprise Information System Datatype Mappings

These tables provide mapping information for various EIS types into Unwired Platform data types.

Table 5. JDBC types

MBO datatype	Generic JDBC type	Java type
BINARY	java.sql.Types.BINARY java.sql.Types.VARBINARY java.sql.Types.LONGVARBINA-RY java.sql.Types.BLOB	java.lang.Byte[]
BIGBINARY	java.sql.Types.LONGVARBINA-RY	java.lang.Byte[]

MBO datatype	Generic JDBC type	Java type
BIGSTRING	java.sql.Types.LONGVARCHAR	java.lang.String
BOOLEAN	java.sql.Types.BOOLEAN java.sql.Types.BIT	java.lang.Boolean
BYTE	java.sql.Types.Byte, byte	java.lang.Byte
CHAR	java.sql.Types.Char, char	java.lang.Character
DATE	java.sql.Types.DATE	java.sql.timestamp
DATETIME	java.sql.Types.TIMESTAMP	java.sql.timestamp
DECIMAL	java.sql.Types.DECIMAL java.sql.Types.NUMERIC	java.math.BigDecimal
DOUBLE	java.sql.Types.DOUBLE	java.lang.Double
FLOAT	java.sql.Types.FLOAT java.sql.Types.REAL	java.lang.Float
INT	java.sql.Types.INTEGER	java.math.BigInteger
INTEGER	java.sql.Types.INTEGER	java.math.BigInteger
LONG	java.sql.Types.BIGINT	java.lang.Long
SHORT	java.sql.Types.BIGINT	java.lang.Short
STRING	java.sql.Types.CHAR java.sql.Types.NCHAR java.sql.Types.VARCHAR java.sql.Types.NVARCHAR java.sql.Types.LONGVARCHAR java.sql.Types.LONGNVARCH-AR	java.lang.String
TIME	java.sql.Types.TIME	java.lang.String

Table 6. Web service types

MBO datatype	XSD type	Java type
BOOLEAN	xs:Boolean	java.lang.Boolean

MBO datatype	XSD type	Java type
BYTE	xs:Byte	java.lang.Byte
BINARY	xs:Base64Binary xs:HexBinary	java.lang.Byte[]
BIGBINARY	xs:Base64Binary	java.lang.Byte[]
DOUBLE	xs:Double	java.lang.Double
FLOAT	xs:Float xs:Int	java.lang.Float
CHAR	xs:UnsignedShort	java.lang.Character
LONG	xs:Long xs:UnsignedInt	java.lang.Long
SHORT	xs:Short xs:UnsignedByte	java.lang.Short

MBO datatype	XSD type	Java type
STRING	xs:String	java.lang.String
	xs:Duration	
	xs:GYearMonth	
	xs:GYear	
	xs:GMonthDay	
	xs:GDay	
	xs:GMonth	
	xs:NOTATION	
	xs:Token	
	xs:NormalizedString	
	xs:Language	
	xs:Name	
	xs:NMToken	
	xs:NCName	
	xs:ID	
	xs:IDREF	
	xs:ENTITY	
	xs:NMTokens	
	xs:IDREFS	
	xs:ENTITIES	
DECIMAL	xs:Decimal	java.math.BigDecimal
INT	xs:Integer	java.math.BigInteger
	xs:NonPositiveInteger	
	xs:NonNegativeInteger	
	xs:NegativeInteger	
	xs:UnsignedLong	
	xs:PositiveInteger	
	xs:AnyURI (java.net.URI.class)	

MBO datatype	XSD type	Java type
INTEGER	xs:Integer xs:NonPositiveInteger xs:NonNegativeInteger xs:NegativeInteger xs:UnsignedLong xs:PositiveInteger xs:AnyURI (java.net.URI.class)	java.math.BigInteger
DATETIME	xs:DateTime	java.sql.timestamp
TIME	xs:Time	java.lang.String
DATE	xs:Date xs:QName	java.sql.timestamp

Table 7. SAP RFC types

MBO data-type	JCo version 3 type code	ABAP type	Java type
BINARY	JCoMetaDa-ta.TYPE_BYTE JCoMetaDa-ta.TYPE_XSTRING	X Y	java.lang.Byte[]
BOOLEAN	JCoMetaDa-ta.TYPE_BYTE JCo-MetaDa-ta.TYPE_XSTRING	X Y	java.lang.Boolean
BYTE	JCoMetaDa-ta.TYPE_INT1	b	java.lang.Byte
CHAR	JCoMetaDa-ta.TYPE_CHAR	C	java.lang.Character
DATE	JCoMetaDa-ta.TYPE_DATE	D	java.sql.timestamp
DATETIME	JCoMetaDa-ta.TYPE_DATE	D	java.sql.timestamp

MBO data-type	JCo version 3 type code	ABAP type	Java type
DECIMAL	JCoMetaData.TYPE_BCD	P	java.math.BigDecimal
DOUBLE	JCoMetaData.TYPE_FLOAT	F	java.lang.Double
FLOAT	JCoMetaData.TYPE_FLOAT	F	java.lang.Float
INT	JCoMetaData.TYPE_INT	I r	java.lang.Integer
INTEGER	JCoMetaData.TYPE_INT	I	java.math.BigInteger
LONG	JCoMetaData.TYPE_INT	I	java.lang.Long
SHORT	JCoMetaData.TYPE_INT2	s	java.lang.Short
STRING	JCoMetaData.TYPE_STRING TYPE_NUM TYPE_FLOAT TYPE_DECF34 TYPE_DECF16	g N F e a	java.lang.String
TIME	JCoMetaData.TYPE_TIME	T	java.sql.time

Mobile Business Object to Mobile Device Platform Datatype Mappings

This table provides mapping information for various MBO datatypes to those of the mobile device target language.

The optional "?" suffix indicates the datatype supports nullability. In some cases, a nullable target language type might be used for a non-nullable MBO type. In either case, the nullability indicator should always be specified if the target type must support nulls.

Any referenced type name that does not appear in the table is expected to be one of the following:

- The name of a class defined within the same package.
- The fully qualified name of a class defined in a previously compiled package.
- The name of an imported class.
- The name of an external class.

Table 8. MBO to Java RIM datatype mappings

MBO datatype	Java RIM types
BOOLEAN	boolean
BOOLEAN?	java.lang.Boolean
STRING	java.lang.String
STRING?	java.lang.String
BINARY	byte[]
BINARY?	byte[]
CHAR	char
CHAR?	java.lang.Character
BYTE	byte
BYTE?	java.lang.Byte
SHORT	short
SHORT?	java.lang.Short
INT	int
INT?	java.lang.Integer
LONG	long
LONG?	java.lang.Long
INTEGER	java.math.BigInteger (javamx.math.BigInteger)
INTEGER?	java.math.BigInteger (javamx.math.BigInteger)
DECIMAL	java.math.BigDecimal (javamx.math.BigDecimal)
DECIMAL?	java.math.BigDecimal (javamx.math.BigDecimal)
FLOAT	float

MBO datatype	Java RIM types
FLOAT?	java.lang.Float
DOUBLE	double
DOUBLE?	java.lang.Double
DATE	java.util.Date
DATE?	java.util.Date
TIME	java.util.Date
TIME?	java.util.Date
DATETIME	java.util.Date
DATETIME?	java.util.Date

Table 9. MBO to Java Android datatype mappings

MBO datatype	Java Android types
BOOLEAN	boolean
BOOLEAN?	java.lang.Boolean
STRING	java.lang.String
STRING?	java.lang.String
BINARY	byte[]
BINARY?	byte[]
CHAR	char
CHAR?	java.lang.Character
BYTE	byte
BYTE?	java.lang.Byte
SHORT	short
SHORT?	java.lang.Short
INT	int
INT?	java.lang.Integer
LONG	long

MBO datatype	Java Android types
LONG?	java.lang.Long
INTEGER	java.math.BigInteger
INTEGER?	java.math.BigInteger
DECIMAL	java.math.BigDecimal
DECIMAL?	java.math.BigDecimal
FLOAT	float
FLOAT?	java.lang.Float
DOUBLE	double
DOUBLE?	java.lang.Double
DATE	java.sql.Date
DATE?	java.sql.Date
TIME	java.sql.Time
TIME?	java.sql.Time
DATETIME	java.sql.Timestamp
DATETIME?	java.sql.Timestamp

Table 10. MBO to C# device datatype mappings

MBO datatype	C# type
BOOLEAN	bool
BOOLEAN?	bool?
STRING	string
STRING?	string
BINARY	byte[]
BINARY?	byte[]
CHAR	char
CHAR?	char?
BYTE	byte

MBO datatype	C# type
BYTE?	byte?
SHORT	short
SHORT?	short?
INT	int
INT?	int?
LONG	long
LONG?	long?
INTEGER	decimal
INTEGER?	decimal?
DECIMAL	decimal
DECIMAL?	decimal?
FLOAT	float
FLOAT?	float?
DOUBLE	double
DOUBLE?	double?
DATE	System.DateTime
DATE?	System.DateTime?
TIME	System.DateTime
TIME?	System.DateTime?
DATETIME	System.DateTime
DATETIME?	System.DateTime?

Table 11. MBO to ObjectiveC device datatype mappings

MBO datatype	iOS ObjectiveC type
BOOLEAN	bool
BOOLEAN?	NSNumber*
STRING	NSString*

MBO datatype	iOS ObjectiveC type
STRING?	NSString*
BINARY	NSData*
BINARY?	NSData*
CHAR	unichar
CHAR?	NSString*
BYTE	signed char
BYTE?	NSNumber*
SHORT	short
SHORT?	NSNumber*
INT	int32_t
INT?	NSNumber*
LONG	Int64_t
LONG?	NSNumber*
INTEGER	NSNumber*
INTEGER?	NSNumber*
DECIMAL	NSNumber*
DECIMAL?	NSNumber*
FLOAT	float
FLOAT?	NSNumber*
DOUBLE	double
DOUBLE?	NSNumber*
DATE	NSDate*
DATE?	NSDate*
TIME	NSDate*
TIME?	NSDate*
DATETIME	NSDate*
DATETIME?	NSDate*

Table 12. MBO to VB.NET datatype mappings

MBO datatype	VB.NET type
BOOLEAN	boolean
BOOLEAN?	nullable(of boolean)
STRING	string
STRING?	string
BINARY	byte()
BINARY?	byte()
CHAR	char
CHAR?	nullable(of char)
BYTE	byte
BYTE?	nullable(of byte)
SHORT	short
SHORT?	nullable(of short)
INT	integer
INT?	nullable(of integer)
LONG	long
LONG?	nullable(of long)
INTEGER	decimal
INTEGER?	nullable(of decimal)
DECIMAL	decimal
DECIMAL?	nullable(of decimal)
FLOAT	single
FLOAT?	nullable(of single)
DOUBLE	double
DOUBLE?	nullable(of double)
DATE	date
DATE?	nullable(of date)

MBO datatype	VB.NET type
TIME	date
TIME?	nullable(of date)
DATETIME	date
DATETIME?	nullable(of date)

Datatype Support

Best Practices for Developing an MBO Data Model

Define MBOs 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.
ℹ️	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. *Managing Project Size.*
☑️	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 Sybase 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.

Managing Project Size

An Sybase Unwired WorkSpace project that contains many MBOs (hundreds), or trying to access the project from poor performing environments, for example from a virtual machine (VM) can lead to poor Sybase Unwired WorkSpace performance when editing the project.

A best practice is to separate the MBOs into different projects then deploy them into a single application. This practice also makes MBO development easier. For example, the developer need not search hundreds of MBOs to find a particular MBO to view or edit. In cases where it is not possible to split large numbers of MBOs into separate projects, try either:

- Accessing the project from a personal computer that provides good performance, or
- Modifying Eclipse startup options to increase the stack and memory (heap) size. For example:

```
-Xms1024M -Xmx1152M -XX:MaxPermSize=512M
```
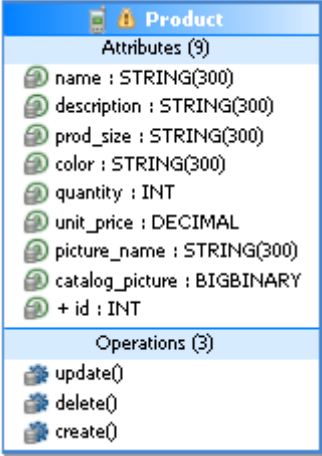
MBO Packages

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.

ℹ️	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.*
ℹ️	Consider row size requirements for localized applications that use multibyte encodings. See *MBO Persistence on the Device.*
ℹ️	A smaller page size generally results in better overall database and synchronization performance. See *Why Page Size Matters.*
⚠️	If computed maximum row size 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.
🚫	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.
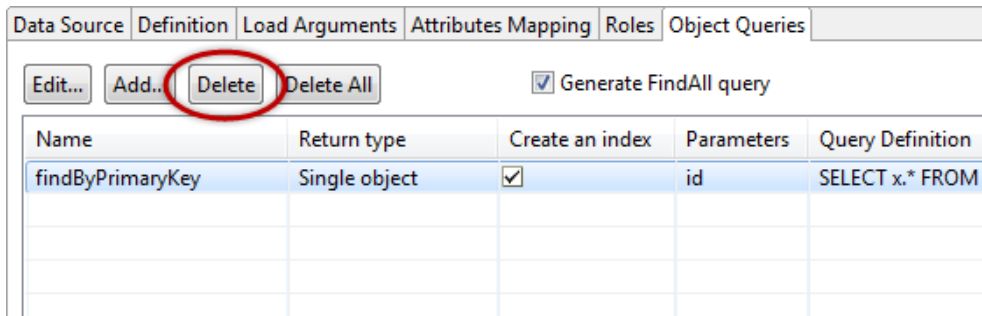
✅	Use the minimum number of indexes.
❓	If findByPrimaryKey is not required to locate the MBO via the business key, disable generation.
❓	If FindAll is not required, disable generation, except for MBOs with a low number of instances on the device.
❓	Determine if an index is required for user-defined object 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.
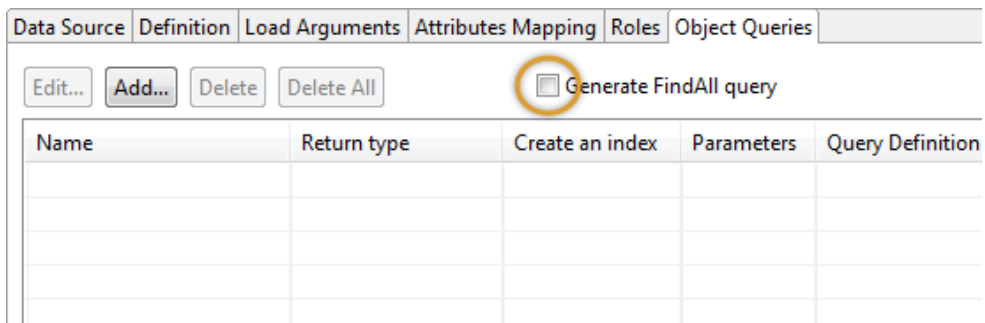
Reducing Indexes

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.

Data Source	Definition	Load Arguments	Attributes Mapping	Roles	Object Queries

Edit...   Add..   Delete   Delete All   ☑ Generate FindAll query

Name	Return type	Create an index	Parameters	Query Definition
findByPrimaryKey	Single object	✓	id	SELECT x.* FROM

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.

Data Source	Definition	Load Arguments	Attributes Mapping	Roles	Object Queries

Edit...   Add...   Delete   Delete All   ☐ Generate FindAll query

Name	Return type	Create an index	Parameters	Query Definition

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 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 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.

🛈	Relationships provide navigation and subscription inheritance. See *Relationship Modeling.*
🛈	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.

ℹ️	A synchronization parameter is also referred to as synchronization key. See *Data Subscription.*

ⓘ	A set of synchronization parameters is sometimes referred to as a subscription. See *Data Subscription.*
ⓘ	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 arguments.
✅	Use synchronization parameters not mapped to load arguments 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 Loading from EIS

A synchronization parameter can also be used to help determine what data to load from the EIS. When a synchronization parameter is mapped to a load argument 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 arguments influence the data to be retrieved. For example, a Web service read operation of getAllBooksByAuthor(Author, userKey) uses the userKey load argument 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 arguments is to map all result-affecting load arguments 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 arguments to identify the result set from the read operation.

Data Filtering

Synchronization parameters not mapped to load arguments are used to filter the data in the CDB. If the data in the CDB is valid, cache refresh is not required and Unwired Platform 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", "MP3 player") as parameters. CDB invokes: 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" + "Tablets", since the partition identified by "Electronics" is still valid, Unwired Server uses "Tablets" as the selection criteria for the subcategory attribute and downloads only all tablets in the catalog.
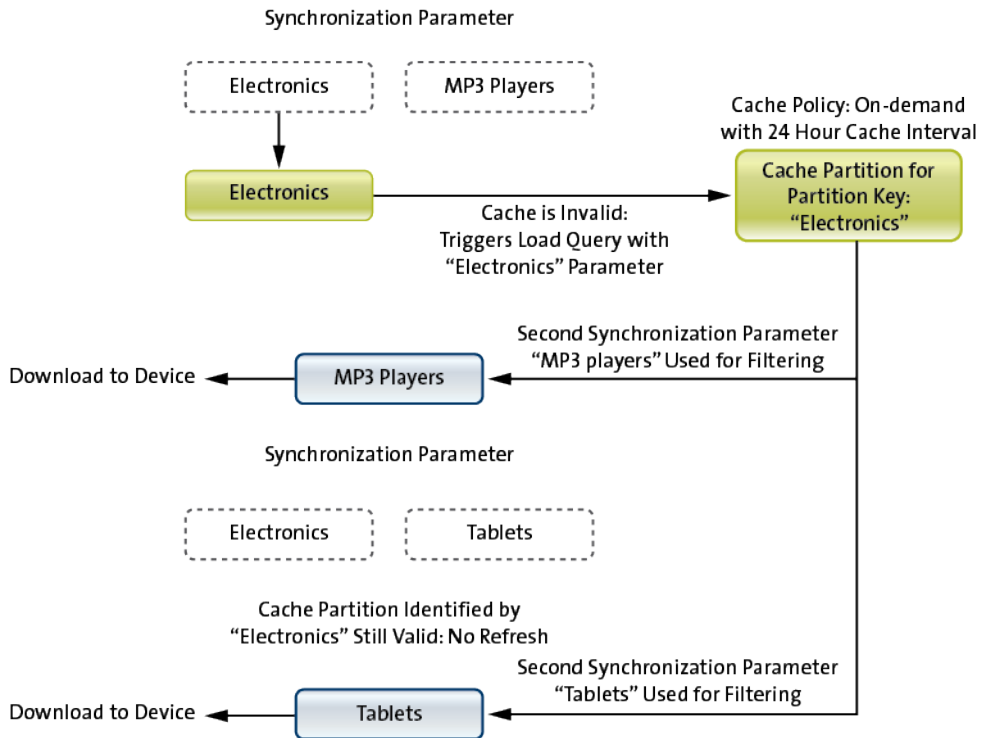
Subscribed Data Management

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 argument 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.

Client Defined Cache Partitions

Understand how to effectively use client defined cache partitions.

🛈	By default, the CDB for the MBO consists of a single partition. See *Client Defined Cache Partitions.*
🛈	Client defined partitions are created when synchronization parameters are mapped to load arguments. The set of load arguments that are mapped to synchronization parameters comprise the partition key. The partition key is used to identify the horizontal subset of the cache which corresponds to an EIS load operation invocation. If a synchronization parameter is not mapped to a load argument, a partition can still be defined by mapping the load argument to a personalization key. See *Partition Membership.*
☑	To increase load parallelism, use multiple client defined partitions.
☑	To reduce refresh latency, use multiple client defined 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 interactions.

Client Defined Cache Partitions

The cache contents 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 arguments.
- Only the result-affecting load arguments form the partition key.
- Using non-result-affecting load arguments within the key causes multiple partitions to hold the same data.

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

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

Personalization Parameter

myEmployeeID

Synchronization Parameters

Electronics   Tablets   Creative Lab

For filtering Only

Department   Category   Employee ID

Result Affecting Load Arguments   Non Result Affecting
Load Argument
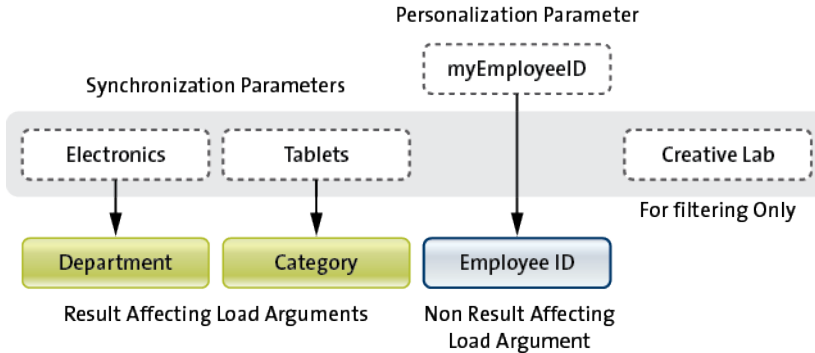
Partition Key = Department + Category

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 interactions. The MBO developer must analyze the data to determine a reasonable partitioning scheme.

Partition Membership
Partitions should not overlap. A member belonging to multiple partitions causes 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
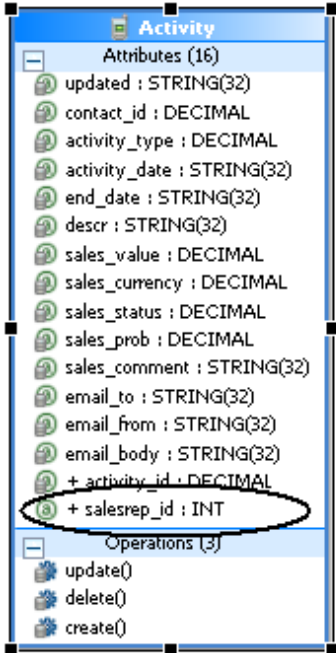
In the following use case, where a service order is purposely assigned to multiple users and using an On-demand policy with a zero cache interval, partition bouncing occurs when service engineer SE-44235 synchronizes at a later time. The service order is now on the devices of both engineers. However, consider the scenario where engineer SE-01245 also synchronizes at time t1. Service order 025 may no longer be in the partition identified by its ID, resulting in a deletion of the service order from the data store on the device when the client synchronizes.
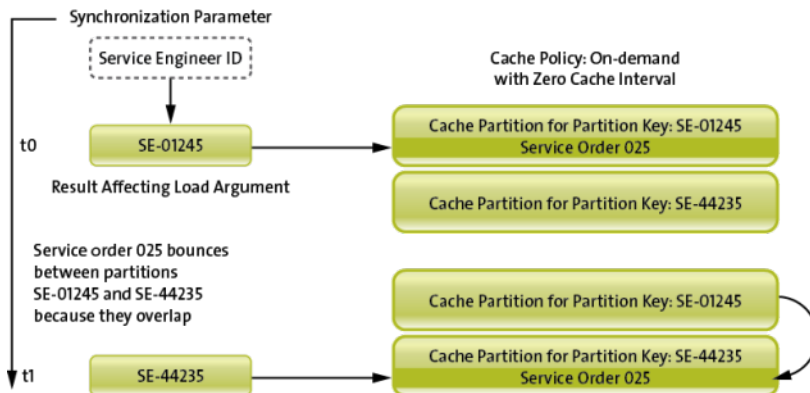
To avoid partition bouncing in this example, use a requester-partitioned cache, which does not require an MBO model change. Another solution is to augment the primary key of the approval MBO with the user identity, which does require an MBO model change. The result is that the same approval request is 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.

The diagrams below illustrate the primary key augmentation approach. The MBO can be assigned to multiple sales representatives in the EIS. MBO definition requests all activities for a particular sales representative using the SalesRepID as a synchronization parameter, which is mapped to the load argument that retrieves all activities for that user.

1. Propagate the load argument 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.

Data Source	Definition	Load Arguments	Attributes Mapping	Roles	Object Queries

Add	Delete	Delete All	Up	Down	Refresh	Remap		Show

Attributes					**Data So**
Name	Datatype	Nullable	Primary Key	Map to	Datatype
salesrep_id	INT	☐	☑		
activity_id	DECIMAL	☐	☑	activity_id	DECIMAL
updated	STRING(32)	☐	☐	updated	STRING(32)
contact_id	DECIMAL	☐	☐	contact_id	DECIMAL
activity_type	DECIMAL	☐	☐	activity_type	DECIMAL
activity_date	STRING(32)	☐	☐	activity_date	STRING(32)
end_date	STRING(32)	☐	☐	end_date	STRING(32)
descr	STRING(32)	☐	☐	descr	STRING(32)
sales_value	DECIMAL	☐	☐	sales_value	DECIMAL
sales_currency	DECIMAL	☐	☐	sales_currency	DECIMAL
sales_status	DECIMAL	☐	☐	sales_status	DECIMAL
sales_prob	DECIMAL	☐	☐	sales_prob	DECIMAL
sales_comment	STRING(32)	☐	☐	sales_comment	STRING(32)
email_to	STRING(32)	☐	☐	email_to	STRING(32)
email_from	STRING(32)	☐	☐	email_from	STRING(32)
email_body	STRING(32)	☐	☐	email_body	STRING(32)

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 arguments 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".
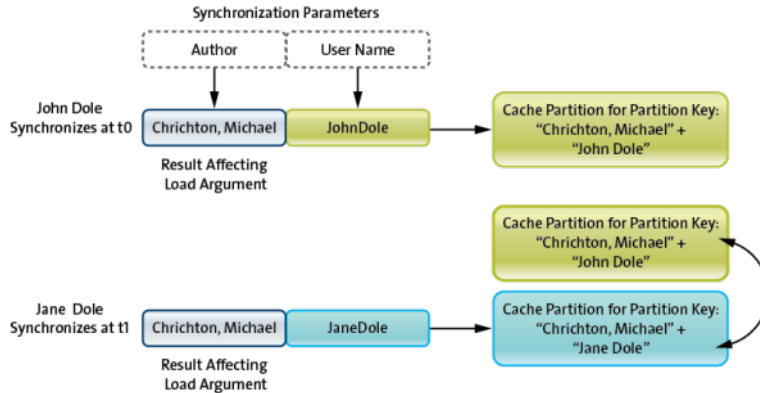
Client Defined Cache Partition Overwrite

Partition overwrite is due to incorrect mapping of synchronization parameters and load arguments, 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")
- John 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.

The solution to this particular problem is to include department in the partition definition.
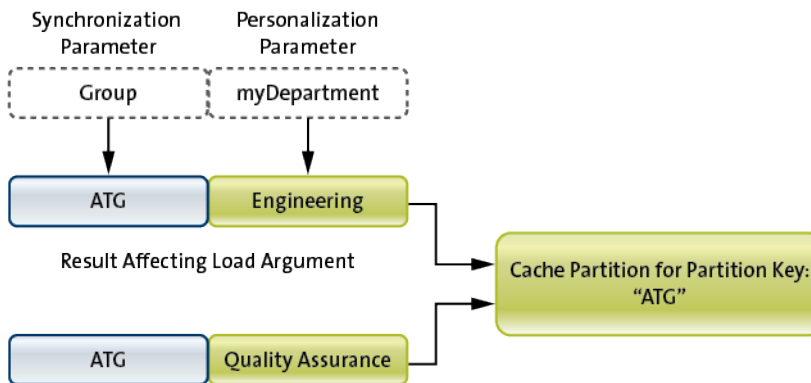
Partition key consists of load arguments that are mapped to synchronization parameters. In this example, only one synchronization parameter is mapped to one of two load arguments, and the second load parameter maps to a personalization key.

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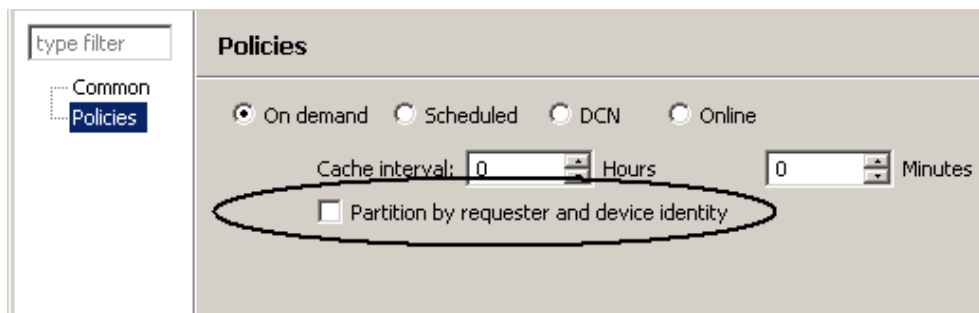
Developers can easily define their own user-specific partition by providing appropriate user identities as load arguments to the EIS read operation.

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. And keep in mind that a package can contain no more than 30 synchronization groups.

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 Wi-Fi 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"));
CustomerDB.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.

> ⓘ 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.
⛔	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, however it is possible to have MBOs forming a cycle as long as they are all within the same cache group. There is a performance impact when processing updates whether it is by DCN or applying the results of cache affecting operations. Developers should create circular relationships only when necessary. Consider if it is reasonable to forgo the relationship and instead use custom code to handle the navigation.

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. 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.

Shared-read MBOs share an EIS read operation to load MBO data into the Unwired Server cache. That is, they belong to the same load group.

🛈	"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.*
🛈	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.
✅	Use multiple partitions, if possible, to alleviate the cost of "Invalidate the cache," as only the affected partition needs to be refreshed.
✅	Always enable "Immediately update the cache" for a create operation to maintain surrogate key to (just returned) business key association.
✅	Use the "Immediately update the cache" cache policy to update an MBO object graph.

Edit Operation

Edit the operation

| Operation name: | create |
| Operation type: | CREATE |

Comment:

| Data Source | Definition | Input | Default Values | Client Parameters | Output | Roles | **Cache Policy** |

☑ Immediately update the cache
 ◉ Apply merge of operation input/output
 ○ Apply output of ENTITY READ operation
☐ Invalidate the cache

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

Use the "Immediately update the cache" operation cache policy and either the "Apply output of ENTITY READ operation" or the "Apply merge of operation input/output" option to transactionally update MBOs in an object hierarchy.

Entity Read may be useful when:

- The remote operation cannot return all the attributes of a composite graph (MBOs in a composite relationship). If the remote operation can return all the attributes, "Apply merge of operation input/output" provides better performance than "Apply output of ENTITY READ operation."
- Using stored procedures, since the MBO developer can use the Entity Read operation without needing to modify any existing stored procedure to get the results into the cache.

Addressing Inconsistency

If 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 policy should always enable "Immediately update the cache" to allow the CDB to associate the returned business key with the surrogate key from the device. This allows the CDB to avoid deleting the new object on the device in preparation for a new one from the EIS.

MBO and Attachments

Understand how to effectively include attachments in your MBO Model.

ⓘ	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.*
ⓘ	It is expensive to always synchronize large objects that are only occasionally required. See *Inline Attachments are Expensive.*
ⓘ	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.
ⓘ	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.
ⓘ	Use Big data types for large attachments to avoid loading on instantiation and use offset based access patterns. See *BigString and BigBinary Datatypes.*

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 analyse 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.

BigString and BigBinary Datatypes

Another cost of inline attachments is object instantiation and resource consumption. Unwired Platform handles this by providing a set of special attributes: BigString and BigBinary. Not only are these attributes not loaded when the object containing them is instantiated, they include a special API to access only the segment in which the application is interested. In other words, very much like the access of a large file, the mobile application can seek to the proper offset to avoid bringing the entire resource into memory.

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 Wi-Fi 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 EIS to the CDB

Define MBOs so they efficiently load data from the EIS to the CDB.

Understanding Data and Datasources

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

While you can use Sybase 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

Table 13. Common Data Characteristics

Reference data	• Mostly read or even read-only • Usually shared between users • Generally updated by EIS • Infrequent or scheduled changes • Able to tolerate stale data • May be concentrated during initial deployment, occasional thereafter • Large to very large data volume is possible

Transactional data	- Read and write - Usually private but can share with other users - Updated by both mobile application and back end possible - Moderate change frequency - High freshness expectation - Access pattern varies depends on use case: morning/evening, or throughout the day - Moderately low data volume (not including historic data which is considered as reference)

Datasources

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.*
🛈	Determine if the EIS can efficiently return a sorted data set based on the primary key when retrieving very large data sets. See *Optimizing Loading of Large Data Sets.*
⚠	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 datasource 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.

Optimizing Loading of Large Data Sets

During a cache fill operation, Unwired Server retrieves a data set from the EIS. The Data Services component performs a sort/merge algorithm upon the data set to detect what has been modified since the last refresh and updates the cache. If the data set is sorted by the primary key, Data Services adds new items, compares and updates matching items, and removes missing items from the cache by traversing the records in the data set. However, if the retrieved data set is not sorted, Data Services must perform an in-memory sort. If the retrieved data set is very large, the sorting may require significant resources, causing the Unwired Server memory footprint to expand and CPU usage to spike.

To optimize performance, change the LOAD ALL or appropriate load queries to include an ORDER BY <primary key> clause to the SELECT statement in cases where the EIS is a relational database.

1. In the Properties view of Sybase Unwired WorkSpace, go to the **Attributes > Definition** tab.
2. Click **Edit**, then adjust the existing SQL definition with an additional ORDER BY clause followed by the primary key column(s).

For a detailed example of using ORDER BY to optimize data refresh of large data sets, see this document on the SAP Community Network *http://scn.sap.com/docs/DOC-32091*.

Alternatively, determine if the EIS can use DCN to send changes as they occur to Unwired Server. This is a superior method to update Unwired Server with deltas of a very large data set. Regardless if the EIS can return a sorted data set or not, CPU usage, network bandwidth, and time to process a very large data set can still be a performance concern.

Sybase Unwired Platform architecture is most optimal when:

1. The mobile client is highly selective in the data required from the EIS and uses synchronization parameters to pull down a small subset of data from the EIS (i.e. cache partitioning). Refreshing cache partitions on an as needed basis is more optimal than pulling down a large number of rows from the EIS. See *Client Defined Cache Partitions*.
2. Effective use of applying operation results to the cache (cache write through/behind using) are used. If the client application performs the majority of data updates then writing through or behind, the Unwired Server cache is more efficient and requires less synchronization with the EIS. See *Operation Cache Policies*.
3. DCN pushes deltas from the EIS into the Unwired Server cache or pushes large unpartitioned data sets into the Unwired Server cache. See *Data Change Notification*.

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 arguments.

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 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 Platform 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 EIS 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 "Immediately update the cache" cache 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.
⚠️	Targeted change notification (TCN), previously called 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)

Pushing data from the EIS via DCN is preferable to pulling data from the EIS, since DCN with payload performs better than an on-demand/zero coherency window where changes are immediately applied to the cache. When using DCN, cache locking is done outside the scope of the synchronization so overall resource utilization decreases. DCN also allows for change detection and targeted change notification (TCN), previously called 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 "Immediately update the cache" cache policy if there are create operations that associate a surrogate key and a business key.
✅	Use a notification MBO to implement TCN/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 "Immediately update the cache" cache 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 argument 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 "Immediately update the cache" cache policy if create operations form the association between the surrogate key and business key.

☑	Use a notification MBO to implement targeted change notification (TCN), previously called 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 "Immediately update the cache" cache policy for all operations. Users see each others changes.
⚠	SIS/TCN 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. As stated in Brewer's CAP Theorem "In a distributed environment, it is impossible to provide all three guarantees simultaneously: consistency, availability, and partition tolerance":
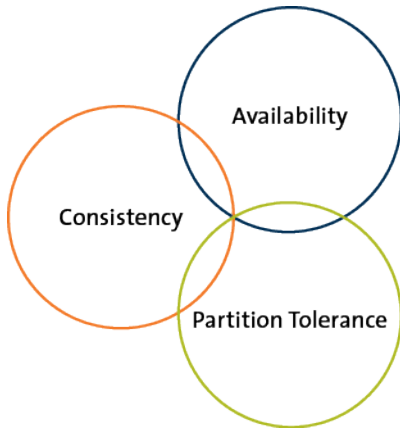
When data resides on the device and operates in a partitioned mobility environment (tolerant to network outage), the choice is whether to have consistency or availability. To enable mobile users to perform their tasks even without connectivity, the choice is availability. Hence, there is no consistency 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 five cache group policies, four of which are relevant to loading data into the cache. The Online cache group policy is used only by Hybrid Apps that require access to non-cached data:

• Scheduled and On-Demand policies are based on data retrieval APIs exposed by the EIS.

- DCN policies (DCN and EIS managed) utilize notification messages from the EIS that push 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 general, if the EIS can push and data inconsistency can be tolerated, a DCN policy 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 requirements 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. The DCN block represents either DCN or EIS managed.
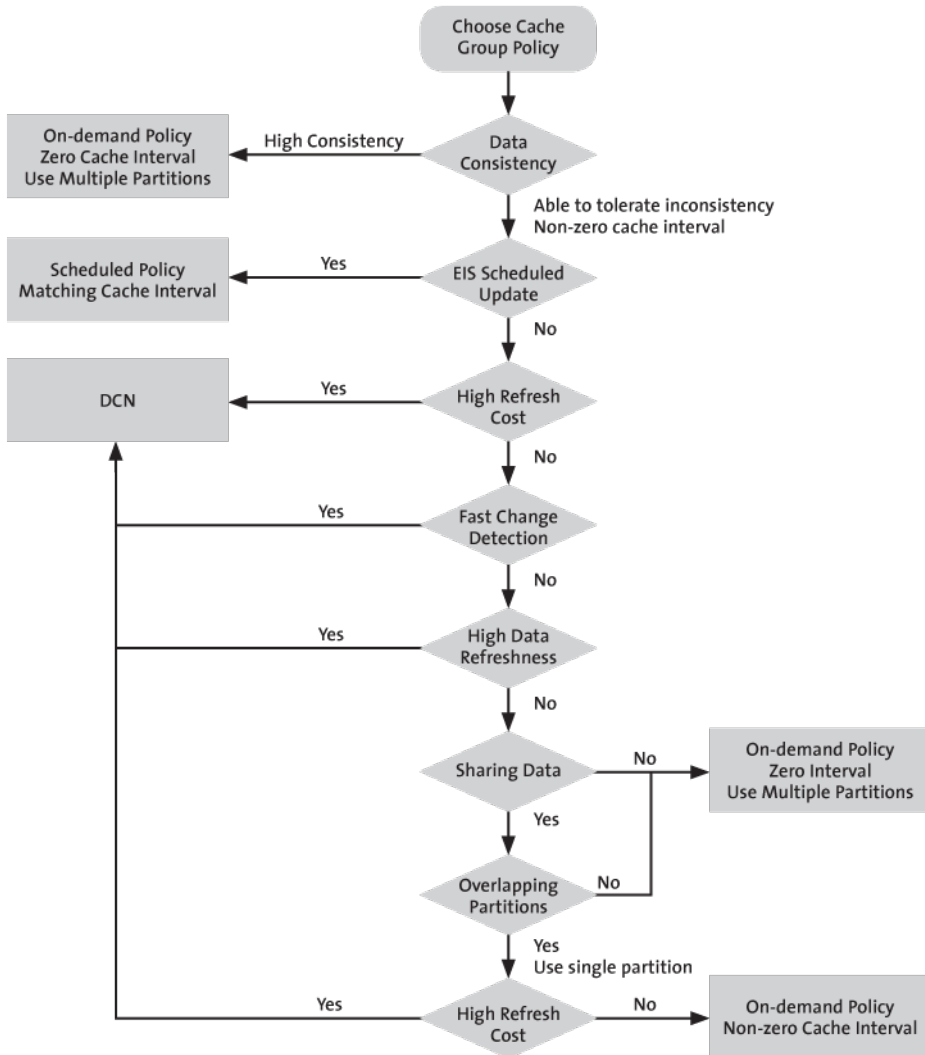
Result Set Filters

A result set filter is a custom Java class an experienced developer writes in order to specifically manipulate the rows or columns of data returned from a read operation for an MBO.

When a read operation returns data that does not completely suit the business requirements for your MBO, you can write and add a filter to the MBO to customize the data into the form you need. You can chain multiple filters together. Multiple filters are processed in the order they are added, each applying an incremental change to the data. Consequently, Sybase recommends that you always preview the results, taking note that the MBO has a different set of attributes than it would have had directly from the read operation. You can map and use the altered attributes in the same way you would do so for a regular attribute from an unfiltered read operation.

Note: The filter interfaces are defined in terms of `java.sql.ResultSet` and `java.sql.ResultSetMetaData`, but these standard JDBC interfaces tend to be read-only implementations. To change data, use a `CachedRowSetImpl` object instead. This object implements ResultSet but also allows you to modify row data.

Example: a simple SELECT statement filter

Suppose you have an MBO based on this query that returns customer information, and you do not want first name and last name divided between two columns (fname and lname) :

```
SELECT * FROM sampledb.customer
```

Instead, write a filter that replaces these columns with a single concatenated "commonName" column.

Note: You could also implement the above example with a more advanced SQL statement with additional computation in the MBO definition:

```
SELECT id, commonName=fname+' '+lname, address, city, state,
zip, phone, company_name FROM customer
```

Example: two separate data sources filter

Suppose you have customer data in two data sources: basic customer information is in an SAP® repository, and more complete details are contained in another database on your network, for example, SQL Anywhere®. You can use a result set filter to combine the SAP customer data with detailed customer data from the database, so that the MBO displays a complete set of information in a single view. You can accomplish this by:

1. Creating a filter for the SAP backend and add it to an SAP MBO.
2. Add a JDBC connection for the SQL Anywhere backend in the filter, then use the SQL Anywhere data to filter the SAP result.

3. Validate the results are what you expect upon completion. When you synchronize the SAP MBO, you should see data from both SAP backend and SQL Anywhere backend.

Result Set Filter Data Flow

A `ResultSetFilter` is a custom Java class deployed to Unwired Server that manipulates rows and columns of data before synchronization.

Result set filters are more versatile (and more complicated to implement) than an attribute filter implemented through a synchronization parameter, since you must write code that implements the filter, instead of simply mapping a parameter to a column to use as the filter.
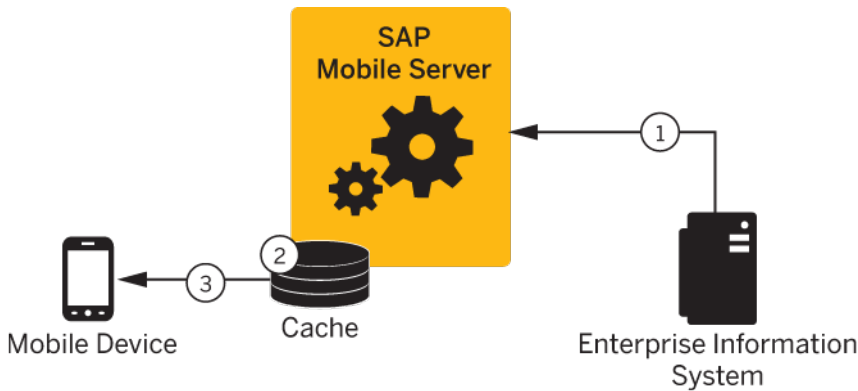
1. Enterprise information system (EIS) data is sent to Unwired Server.
2. The result set filter filters the results, and applies those results to the CDB for a given MBO. For example, the result set filter combines two columns into one.
3. The device application synchronizes with the results contained in the CDB.

The client cannot distinguish between MBOs that have had their attributes transformed through a `ResultSetFilter` from those that have not.

Implementing Custom Result Set Filters

Developers can write a filter to add, delete, or change columns as well as to add and delete rows.

Prerequisites

To write a filter, developers must have previous experience with Java programming — particularly with the reference implementations for `javax.sql.RowSet`, which is used to implement the filter interface and described in the *JDBC RowSet Implementations Tutorial* at *http://docs.oracle.com/javase/tutorial/jdbc/basics/jdbcrowset.html*.

Note: Sybase strongly encourages developers to initially create filters in Sybase Unwired WorkSpace: a wizard assists you by autogenerating required imports and methods so the implementation compiles and runs. Then to customize the code, you can cut and paste fragments from the sample, and make the required changes to get the desired end result.

Task

Once the filter has been implemented and deployed to Unwired Server as part of an MBO package, the MBO developer can apply the filter to other MBOs from Sybase Unwired WorkSpace. See *Adding a Result Set Filter* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

Note: Validate the performance of any custom result set filters, before deploying packages to Unwired Server.

Writing a Custom Result Set Filter

Write a custom result set filter to define specific application processing logic. Save the compiled Java class file to a location that is accessible from Sybase Unwired WorkSpace.

In the custom filter, configure attribute properties so that the returned record set can be better consumed by the device client application. Sometimes, a result set returned from a datasource requires unique processing; a custom filter can perform that function before the information is downloaded to the client.

Data in the cache is shared by all clients. If you need to identify data in the cache to a specific client, you must define a primary key attribute that identifies the client (such as remote_id or username).

1. (Required) Create a record set filter class that implements the
    `com.sybase.uep.eis.ResultSetFilter` interface.

    This interface defines how a custom filter for the data is called.

    For example, this code fragment sets the package name and imports the required classes:

    ```
    package com.mycompany.myname;
    import java.sql.ResultSet;
    import java.util.Map;
    ```
2. (Recommended) Implement the
    `com.sybase.uep.eis.ResultSetFilterMetaData` interface as well as the
    `com.sybase.uep.eis.ResultSetFilter` interface on your filter class.

    If you choose not to implement this interface, Sybase Unwired WorkSpace will have to execute a chain of mobile business object operations and filters and fetch real data before you can see the actual output column names and their datatypes. By first implementing these interfaces, the operation does not need to be executed first. Instead, the `getMetaData()` method obtains the necessary column or data type information.

This example sets the package name but uses a different combination of classes than in the example for step 1:

```
package com.mycompany.myname;
import java.sql.ResultSetMetaData;
import java.util.Map;
```

3. Call the appropriate method, which depends on the interfaces you implement.

`ResultSetFilter` filters the data in the first option documented in step 1. Each filter defines a distinct set of arguments. Therefore, use only the arguments with the appropriate filter that defines these arguments in `getArguments()`, rather than use all filters and datasource operations.

The result set passed in contains the grid data, which should be considered read-only—do not use operations that change or transform data. The return value cannot be NULL, otherwise an execution error occurs.

```
public interface ResultSetFilter {
    ResultSet filter(ResultSet in, Map<String, Object> arguments)
throws
    Exception;
    Map <String, Class> getArguments();
}
```

Next, use `ResultSetFilterMetaData` to format the data from step 1. Use this interface to avoid executing an extraneous datasource operation to generate a sample data set.

```
public interface ResultSetFilterMetaData {
    ResultSetMetaData getMetaData(ResultSetMetaData in, Map<String,
        Object> arguments) throws Exception;
}
```

Note: If the filter returns different columns depending on the argument values supplied, the filter may not work reliably. Ensure that any arguments that affect metadata have constant values in the final mobile business object definition, so the schema does not dynamically change.

4. Implement the class you have created, defining any custom processing logic.
5. Save the classes to an accessible Sybase Unwired WorkSpace location. This allows you to select the class, when you configure result set filters for your mobile business object.
6. In Sybase Unwired WorkSpace, refresh configured MBO attributes, to see the result.

MBO load operations can take parameters on the enterprise information system (EIS) side. These load parameters are defined from Sybase Unwired WorkSpace as you create the MBO. For example, defining an MBO as:

```
SELECT * FROM customer WHERE region = :region
```

results in a load argument named "region".

As an example, if you want a filter that combines fname and lname into commonName, add `MyCommonNameFilter` to the MBO. When `MyCommonNameFilter.filter()` is called, the "arguments" input value to this method is a `Map<String, Object>` that has an entry with the key "region". Your filter may or may not care about this parameter (it is the backend database that requires the value of region to execute the query). But your filter may need some other information to work properly, for example the remote user's zipcode. The `ResultSetFilter` interface includes `java.util.Map<java.lang.String,java.lang.Class> getArguments()` that you must implement. In order to arrange for the remote user's zipcode (as a String) to be provided to the filter, write some custom code in the body of the `getArguments` method, for example:

```
public Map<String, Object> getArguments {
      HashMap<String, Class> myArgs = new HashMap<String, Class>();
      myArgs.put("zipcode", java.lang.String.class);
      return myArgs;
   }
```

This informs Sybase Unwired WorkSpace that the "zipcode" parameter is required, and is of type String. Sybase Unwired WorkSpace automatically adds the parameter for the load operation, so this MBO now has two (region and zipcode). Your filter gets them both when its `filter()` method is called, but can ignore region if it wants.

Validating Result Set Filter Performance

After you deploy the filters to Unwired Server, synchronize data and ensure that filters are performing as you expect.

1. Confirm that the columns appear correctly after the filter has been added to the mobile business object.
   a) Refresh the object.
   b) In the Properties view, select the **Attributes > Attributes Mapping** tab.
   c) Verify that columns are correctly listed in the **Map to** column.
2. From the mobile application running on a device or simulator, open the mobile object, and check that the new column appears.
3. Synchronize the object from the device client or simulator.
4. Troubleshoot filters if issues arise:

   • During synchronization, all `System.out` statements are printed to the Unwired Server log.
   • If you started Sybase Unwired WorkSpace with the -consoleLog in java.exe, `System.out` statements are also printed to the console window.

Filter Class Debugging

Sybase Unwired Platform supports various debugging models: instrumented code, and JPDA (Java Platform Debugger Architecture).

You can also log your own output by including **System.out.println()** in the filter class, output from the class is captured in the Unwired Server log when the filter executes in the server.

Alternatively, you can use the standard Java debugger to debug the filter class.

Enabling JPDA

Set up JPDA and attach the Java standard debugger to Unwired Server.

1. Stop Unwired Server.
2. Add JPDA information from Sybase Control Center:
   a) Select **Servers > ServerName > Server Configuration**.
   b) Select the **General** tab.
   c) Add this information to the value of the User options property. In this example 5005 is the port to which the Java debugger connects:
   ```
   -Xdebug -Xnoagent -
   Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005
   ```
3. Restart Unwired Server.
4. Once Unwired Server is restarted, verify that JPDA mode is working and available at port 5005 by running:

   `netstat -ano | findstr 5005` from a command prompt.

   Look for these results:
   ```
   TCP  0.0.0.0:<JPDAport>   0.0.0.0:0  LISTENING
   ```
5. Use a standard Java debugger and attach it to Unwired Server by specifying the correct host and the JPDA port used.

Begin debugging the result filter class with the Java debugger.

Setting Debug Breakpoints in Result Set Filter Classes

Set breakpoints in the result set filter classes from the Sybase Unwired WorkSpace project that contains the filters.

Prerequisites
Add a result set filter to an Sybase Unwired WorkSpace project.

Task

1. From WorkSpace Navigator in Sybase Unwired WorkSpace, expand the project Filters to access the result set filter Java class.
2. Double-click the Java class to open it in the Java editor.

   The default generated code is a filter that does nothing until you add your filter code.
3. Right-click in the grey vertical bar to the left of the actual code and select **Toggle Breakpoint** to set breakpoints.
4. Compile and deploy the classes to Unwired Server. Redeploy the MBO package if it has changed as part of the filtering.

Setting Up the Debug Session

Debug the deployed result set filter from Sybase Unwired WorkSpace using breakpoints.

Prerequisites

1. Set debug breakpoints in your filter class.
2. Open the Debug perspective from Sybase Unwired WorkSpace by selecting **Window > Open Perspective > Other > Debug**.

Task

1. From Sybase Unwired WorkSpace click the down-arrow next to the Debug toolbar icon, and select **Debug Configurations**.
2. Right-click **Remote Java Application** and select **New**.
3. Name the configuration and change the JPDA port to match that of Unwired Server:
   - Use a standard connection (Socket Attach)
   - Use host 0.0.0.0
   - Set the the port to match the one enabled in Unwired Server (by default 5005)
4. Click **Debug** to save the configuration and launch the debugger.
5. Select **Window > Open Perspective > Other > Debug** to open the Debug perspective.

   In the left pane you can view the threads running inside Unwired Server. When your filter is called, one of these threads suspends, and the code window highlights where the debugger has stopped in the server. The right pane displays the values of local variables inside your filter.

Result Set Filters

Result Checkers

A result checker is a custom Java class that can be implemented to customize error checking for mobile business object (MBO) operations.

Not all MBO operations can use a "standard" error reporting technique as the EIS system may return error codes or failures through various fields; In such cases, you may want to implement your own custom result checker. Doing so allows you to check any field for errors, or implement logic that determines what constitutes an error, and the severity of the error.

A custom result checker can throw errors for both a scheduled cache refresh as well as an on demand cache refresh:

* For a scheduled refresh – the result checker writes a log message that describes the nature of the error to the Unwired Server log. As a consequence of this error, the transaction for the entire cache group is rolled back. The device client user is not notified of these errors; no client log records are generated.
* On demand refresh – instead of writing the error to the server log, the log message is written to the Unwired Server. Services in the server handle the exception. As a consequence of this error, the transaction for the cache group is rolled back. But in this case, a client log record is generated, which is visible to the client application after synchronization.

Both cases send the `OperationStatusEvent`. This event indicates that an operation failed to execute properly. The server uses `OperationStatusEvent` to populate a statistics repository that tracks the success or failure of EIS operation invocations. An administrator can review these statistics in Sybase Control Center, by clicking the Monitor node in the left navigation pane. See *Reviewing System Monitoring Data* in *Sybase Control Center for Sybase Unwired Platform*.

Implementing Customized Result Checkers

Implement a custom result checker with the required Java class to implement custom error checking for EIS-specific business objects.

This section describes how to write and add a custom result checker. See *Adding a Result Checker* in *Sybase Unwired WorkSpace - Mobile Business Object Development* for additional information.

Writing a Custom Result Checker

Use a custom Java class to implement custom error checking.

Provide a Java class that extends the appropriate class for the enterprise information system (EIS).

- **SAP –**

```
package com.sybase.sup.sap3;

import com.sap.conn.jco.JCoFunction;
import com.sybase.vader.core.OperationHandlerBase;

public abstract class SAPOperationHandler extends
OperationHandlerBase {

 /**
  *
  * @param f - JCO function that has already been executed.
  * Use the JCO API to retrieve returned values and
  * determine if the RFC has executed successfully.
  *
  * Throw OHException to return an error.
  * Call OHLog.log() as often as needed to return
  * warnings and other informational messages.
  */
 public void resultCheck(JCoFunction f) {
 }

 /**
  * @param cause Exception thrown by the call to the EIS
  *
  * @param f - JCO function that has already been executed.
  * Use the JCO API to retrieve returned values and
  * determine if the RFC has executed successfully.
  *
  * Throw OHException to return an error.
  * Call OHLog.log() as often as needed to return warnings
  * and other informational messages.
  */
 public void onError(Throwable cause, JCoFunction f) {
 }
}
```

- **Web service (SOAP) –**

```
package com.sybase.sup.ws.soap;

import com.sybase.vader.core.OperationHandlerBase;

public abstract class SoapOperationHandler extends
OperationHandlerBase {

    /**
     *
     * @param response the SOAP Envelope response from
     * a web service execute.
```

```
     * Use the SOAP API to retrieve values and determine
     * if the SOAP request has executed successfully.
     *
     * @param request the SOAP Envelope request sent to
     * the web service.
     * Use the SOAP API to retrieve values.
     *
 * Throw OHException to return an error.
 * Call OHLog.log() as often as needed to return warnings
 * and other informational messages.
     *
     */
public void resultCheck(
     javax.xml.soap.SOAPEnvelope response,
     javax.xml.soap.SOAPEnvelope request) {
}

    /**
     * @param cause Exception thrown by the call to the EIS
     *
     * @param request the SOAP Envelope request sent to the
     * web service.
     * Use the SOAP API to retrieve values.
     *
 * Throw OHException to return an error.
 * Call OHLog.log() as often as needed to return warnings
 * and other informational messages.
 *
     */
public void onError(Throwable cause,
  javax.xml.soap.SOAPEnvelope request) {
}
}
```

- **RESTful Web service –**

```
package com.sybase.sup.ws.rest;

import java.util.List;
import com.sybase.vader.core.OperationHandlerBase;

import java.net.URL;


public abstract class RestOperationHandler extends
OperationHandlerBase{


 public RestOperationHandler () {
  super();
 }

 /**
  * REST Result Check.
  *
  * @param responseBody HTTP response body.
  *
```

```
 * @param responseHeaders HTTP response headers in the form
 * {{header1,value1}, {header2,value2}, ...}.
 *
 * @param httpStatusCode HTTP status code.
 *
 * @param url HTTP URL.
 *
 * @param requestBody HTTP request body.
 *
 * @param requestHeaders HTTP request headers in the form
 * {{header1,value1}, {header2,value2}, ...}.
 *
 * Throw OHException to return an error.
 * Call OHLog.log() as often as needed to return warnings
 * and other informational messages.
 *
 **/
public void resultCheck( String responseBody,
    List<List<String>> responseHeaders, int httpStatusCode, URL
url,
        String requestBody, List<List<String>> requestHeaders) {
}

    /**
     * REST On Error
     * Note that errors are virtually never thrown by REST.
     * Instead, HTTP Error codes are returned.
     * The Result Check method can handle these codes
     * before any exception is thrown by Data Services.
     *
     * @param cause Exception thrown by the call to the EIS
 *
 * @param url HTTP URL.
 *
 * @param requestBody HTTP request body.
 *
 * @param requestHeaders HTTP request headers in the form
 * {{header1,value1}, {header2,value2}, ...}.
      *
 * @return Result Checker Return
 *
 * Throw OHException to return an error.
 * Call OHLog.log() as often as needed to return warnings
 * and other informational messages.
 *
     */
public void onError(Throwable cause, URL url,
        String requestBody, List<List<String>> requestHeaders) {
}
}
```

Result checkers depend on the Unwired Server sup-ds.jar file. For example, SUP_HOME
\Servers\UnwiredServer\lib\ext\sup-ds.jar.

Result Checker Logging

Use OHLog to trap warnings but not halt execution of the result checker.

You can influence the error or warning code and message in the result checker by throwing a DSException, which produces errors and halts execution, or by calling OHLog, which is used for warnings and does not halt execution.

Use OHLog.log() to write to the client log. This method returns true if it successfully wrote the log entry, and false if no client is defined. For example, no client is typically defined for a scheduled refresh.

The following code examples illustrate implementation of OHLog for various enterprise information systems.

Datasource: SAP

```
package com.sybase.vader.test.mms;

import com.sap.conn.jco.JCoFunction;
import com.sybase.sup.sap3.SAPOperationHandler;
import com.sybase.dataservices.OHException;
import com.sybase.dataservices.OHLog;

public class TestSAPOperationHandler extends SAPOperationHandler {

 public void resultCheck(JCoFunction f) {
  OHLog.warn(901, 101, f.toXML());
  //throw new OHException(901, 101, "Throwing test OHException with
901 and 101.");
  }

 public void onError(Throwable cause, JCoFunction f) {
  if(f !=null) {
   OHLog.warn(901, 101, f.toXML());
  } else {
   OHLog.warn(902, 102, "JCoFunction null");
  }
  throw new OHException(904, 103, "old exception said
'"+cause.getMessage()+"'");
 }
}
```

Datasource: Web Service (SOAP)

```
package com.sybase.vader.test.mms;

import java.io.StringWriter;

import com.sybase.dataservices.OHException;
import com.sybase.dataservices.OHLog;
import com.sybase.sup.ws.soap.SoapOperationHandler;

import javax.xml.transform.OutputKeys;
```

```
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;

import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import javax.xml.soap.SOAPFault;
import javax.xml.soap.SOAPEnvelope;

public class TestSoapOperationHandler extends SoapOperationHandler {

 public void resultCheck(
      javax.xml.soap.SOAPEnvelope response,
      javax.xml.soap.SOAPEnvelope request) {

  OHLog.info(901, 101, toXML(request));
  OHLog.warn(902, 102, toXML(response));

  try{
   SOAPFault fault = response.getBody().getFault();
   if(fault!=null) {
    throw new OHException(900,
Integer.valueOf(fault.getFaultCode()), fault.getFaultString());
   }
  } catch (Exception e) {
   // ignore
  }
 }

 public void onError(Throwable cause,
   javax.xml.soap.SOAPEnvelope request) {
  if(request != null) {
   OHLog.info(901, 101, toXML(request));
  }
  throw new OHException(904, 103, "old exception said
'"+cause.getMessage()+"'");
 }

 private String toXML(javax.xml.soap.SOAPEnvelope env) {
  String xmlString="";
  try {
   TransformerFactory transfac = TransformerFactory.newInstance();
   Transformer trans = transfac.newTransformer();
   trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
   trans.setOutputProperty(OutputKeys.INDENT, "yes");

   StringWriter sw = new StringWriter();
   StreamResult result = new StreamResult(sw);
   DOMSource source = new DOMSource(env);
   trans.transform(source, result);
   xmlString = sw.toString();
  }
  catch(Exception e) {

  }
  return xmlString;
```

```
 }
}
```

Datasource: RESTful Web Service

```java
package com.sybase.vader.test.mms;

import java.net.URL;
import java.util.List;

import com.sybase.sup.ws.rest.RestOperationHandler;
import com.sybase.dataservices.OHLog;
import com.sybase.dataservices.OHException;

public class TestRestOperationHandler extends RestOperationHandler {

 public void resultCheck( String responseBody,
    List<List<String>> responseHeaders, int httpStatusCode, URL url,
        String requestBody, List<List<String>> requestHeaders) {

  OHLog.info(901, 101, url.toString());
  if(requestBody != null) {
   if(requestBody.isEmpty()){
    OHLog.debug(902, 102, "request body empty");
   }
   else {
    OHLog.debug(902, 102, requestBody);
   }
  } else {
   OHLog.debug(902, 102, "request body null");
  }
  int i=0;
  for(List<String> list : requestHeaders) {
   String msg = "" + list.get(0) + "=" + list.get(1);
   OHLog.warn(903, i++, msg);
  }

  OHLog.info(905, 105, "httpStatusCode="+httpStatusCode);
  if(responseBody != null) {
   if(responseBody.isEmpty()){
    OHLog.debug(906, 106, "response body empty");
   }
   else {
    OHLog.debug(906, 106, responseBody);
   }
  }
  else {
   OHLog.debug(906, 106, "response body null");
  }
  i=0;
  for(List<String> list : responseHeaders) {
   String msg = "" + list.get(0) + "=" + list.get(1);
   OHLog.warn(903, i++, msg);
  }

  if(httpStatusCode>=300) {
```

```
   throw new OHException(500, httpStatusCode, "HTTP status code
["+httpStatusCode+"] too high");
  }
 }

 public void onError(Throwable cause, URL url,
         String requestBody, List<List<String>> requestHeaders) {

  if(url != null) {
   OHLog.info(901, 101, url.toString());
  }
  if(requestBody!=null){
   OHLog.debug(902, 102, requestBody);
  }
  int i=0;
  if(requestHeaders != null) {
   for(List<String> list : requestHeaders) {
    String msg = "" + list.get(0) + "=" + list.get(1);
    OHLog.warn(903, i++, msg);
   }
  }
  System.out.println("onError called with following stack trace:");
  cause.printStackTrace();
  System.out.println("^^^^^^");
  throw new OHException(904, 103, "old exception said
'"+cause.getMessage()+"'");
 }
}
```

Default SAP Result Checker Code

Default Result Checkers are built-in result checkers that are applied automatically on MBO
operations by Unwired Server. They can be replaced by implementing and deploying a
Custom Result Checker. This is the default result checker used to check results in SAP®
datasources.

```
package com.sybase.sap3;

import java.util.HashSet;
import java.util.Set;

import com.sap.conn.jco.JCoFunction;
import com.sap.conn.jco.JCoParameterList;
import com.sap.conn.jco.JCoRecord;
import com.sap.conn.jco.JCoTable;
import com.sybase.vader.utils.logging.SybLogger;
import com.sybase.sup.sap3.SAPOperationHandler;
import com.sybase.dataservices.OHLog;
import com.sybase.dataservices.OHException;

public class DefaultSAPOperationHandler extends SAPOperationHandler
{
```

```
    private static Set<String>                 nonErrorMessages;
    static
    {
        nonErrorMessages = new HashSet<String>();
        nonErrorMessages.add("No data found");
        nonErrorMessages.add("Data was not found for the document");
        nonErrorMessages.add("No customer was found with these
selection criteria");
    }

    public void resultCheck(JCoFunction f) {
        JCoRecord returnStructure = null;
        JCoParameterList jpl = f.getExportParameterList();
        String errorMsg = null;
        int errorNumber = 0;
        String errorDebugMsg = null;
        boolean success = true;
        if ( jpl != null )
        {
            try
            {
                returnStructure = jpl.getStructure("RETURN");
                if ( returnStructure != null )
                {
                    SybLogger.debug("JCoRecord = '" +
returnStructure.toXML() + "'");
                    String type = returnStructure.getString("TYPE");
                    String message =
returnStructure.getString("MESSAGE");
                    // generally TYPE is S for success, I for
informational,
                    // or empty
                    if ( type.equals("") || type.equals("S") ||
type.equals("I") )                 {
                        SybLogger.debug("Success");
                        //sendDebugMsg(returnStructure);
                        OHLog.info(OHLog.EIS_SUCCESS,
returnStructure.getInt("NUMBER"),
message.isEmpty()?"Success":message);
                    }
                    else
                    {
                        SybLogger.debug("TYPE: <<" + type + ">>,
MESSAGE: <<" + message + ">>");
                        if ( type.equals("W") ||
nonErrorMessages.contains(message) )
                        {
                            SybLogger.debug("Success");
                            //sendDebugMsg(returnStructure);
                            OHLog.warn(OHLog.EIS_SUCCESS,
returnStructure.getInt("NUMBER"),
message.isEmpty()?"Success":message);
                        }
                        else
                        {
                            SybLogger.debug("Error");
```

```
                                  sendDebugMsg(returnStructure,
OHLog.INTERNAL_SERVER_ERROR);
                                  SybLogger.debug("Throwing OHException.
NUMBER="+ returnStructure.getInt("NUMBER"));
                                  throw new
OHException(OHLog.INTERNAL_SERVER_ERROR,
returnStructure.getInt("NUMBER"),
returnStructure.getString("MESSAGE"));
                            }
                        }

                }
            }
            catch (OHException ohe)
            {
                throw ohe;
            }
            catch (Exception e)
            {
                SybLogger.debug("Unable to retrieve RETURN structure -
Will try to retrieve RETURN table next.", e);
            }
        }
        // if there is no RETURN structure, look for RETURN table
        if ( returnStructure == null )
        {
            jpl = f.getTableParameterList();
            if ( jpl != null )
            {
                try
                {
                    JCoTable returnTable = jpl.getTable("RETURN");
                    SybLogger.debug("JCoTable = '" +
returnTable.toXML() + "'");
                    for (int i = 0; i < returnTable.getNumRows(); i++)
                    {
                        returnTable.setRow(i);
                        String type = returnTable.getString("TYPE");
                        String message =
returnTable.getString("MESSAGE");
                        // generally TYPE is S for success, I for
                        // informational, or empty
                        if ( type.equals("") || type.equals("S") ||
type.equals("I") )
                        {
                            SybLogger.debug("Success");
                            //sendDebugMsg(returnTable);
                            OHLog.warn(OHLog.EIS_SUCCESS,
returnTable.getInt("NUMBER"), message.isEmpty()?"Success":message);
                        }
                        else
                        {
                            SybLogger.debug("TYPE: <<" + type + ">>,
MESSAGE: <<" + message + ">>");
                            // throw an exception on error, but need to
discover if other rows exist first
```

```
                                if ( type.equals("W") ||
nonErrorMessages.contains(message) )
                                {
                                        SybLogger.debug("Success");
                                        //sendDebugMsg(returnTable);
                                        OHLog.warn(OHLog.EIS_SUCCESS,
returnTable.getInt("NUMBER"), message.isEmpty()?"Success":message);
                                }
                                else
                                {
                                // If we previously discovered an error we
can log this one and throw the other, later
                                    if(!success)
                                    {
                                            SybLogger.debug("Error");
                                            sendDebugMsg(returnTable,
OHLog.INTERNAL_SERVER_ERROR);

OHLog.error(OHLog.INTERNAL_SERVER_ERROR,
returnTable.getInt("NUMBER"), returnTable.getString("MESSAGE"));
                                    }
                                    else
                                    {
                                            SybLogger.debug("Error");
                                            success = false;
                                            errorMsg = message;
                                            errorNumber =
returnTable.getInt("NUMBER");
                                            errorDebugMsg =
makeDebugMsg(returnTable);
                                    }
                                }
                            }

                    }
                }
                catch (Exception e)
                {
                    success = false;
                    errorMsg = e.getMessage();
                    errorNumber = 0;
                    if (errorMsg == null || errorMsg.isEmpty())
                    {
                        errorMsg = e.toString();
                    }
                }
            }
        }
        if(!success)
        {
            if(errorDebugMsg != null)
            {
              OHLog.debug(OHLog.INTERNAL_SERVER_ERROR, errorNumber,
errorDebugMsg);
            }
            throw new OHException(OHLog.INTERNAL_SERVER_ERROR,
```

```
errorNumber, errorMsg);
        }
    }

    // JCoTables are JCoRecords, so this works
    private String makeDebugMsg (JCoRecord r) {
        String s = "";
        // Some fields may not be present, which will throw an
exception we want to ignore
        try{s+="ID='" + r.getString("ID") + "'";}
        catch(Exception e){}

        try{s+=",LOG_NO='" + r.getString("LOG_NO") + "'";}
        catch(Exception e){}

        try{s+=",LOG_MSG_NO=" + r.getInt("LOG_MSG_NO");}
        catch(Exception e){}

        try{s+=",PARAMETER='" + r.getString("PARAMETER") + "'";}
catch(Exception e){}
        try{s+=",ROW=" + r.getInt("ROW");}
        catch(Exception e){}

        try{s+=",FIELD='" + r.getString("FIELD") + "'";}
        catch(Exception e){}

        try{s+=",SYSTEM='" + r.getString("SYSTEM") + "'";}
        catch(Exception e){}
        return s;
    }

    private void sendDebugMsg (JCoRecord r, int supCode) {
        String s = makeDebugMsg(r);
        OHLog.debug(supCode, r.getInt("NUMBER"), s);
    }
    private void sendDebugMsg (JCoRecord r) {
        sendDebugMsg(r, OHLog.EIS_SUCCESS);
    }
}
```

Default SOAP Result Checker Code

Default Result Checkers are built-in result checkers that are applied automatically on MBO operations by Unwired Server. They can be replaced by implementing and deploying a Custom Result Checker. This is the operation handler code used to check results in SOAP Web service datasources.

```
package com.sybase.sup.ws;

import javax.xml.soap.SOAPFault;

import com.sybase.dataservices.OHException;
import com.sybase.sup.ws.soap.SoapOperationHandler;
```

```
public class DefaultSoapOperationHandler extends
SoapOperationHandler {

    public void resultCheck(
                        javax.xml.soap.SOAPEnvelope response,
                        javax.xml.soap.SOAPEnvelope request) {

       SOAPFault fault = null;
       try
       {
           fault = response.getBody().getFault();
       }
       catch (Exception e)
       {
           //If we're in here, no fault was found.
       }
       if ( fault != null )
       {
         throw new OHException(OHException.INTERNAL_SERVER_ERROR, 0,
fault.getFaultString());
       }

    }
}
```

Default REST Result Checker Code

Default Result Checkers are built-in result checkers that are applied automatically on MBO operations by Unwired Server. They can be replaced by implementing and deploying a Custom Result Checker. This is the default operation handler code used to check results in REST Web service datasources.

```
package com.sybase.sup.ws;

import com.sybase.sup.ws.rest.RestOperationHandler;

/*
 * Default is a no-op
 */
public class DefaultRestOperationHandler extends
RestOperationHandler {

}
```

Result Checkers

Data Change Notification

Data change notification (DCN) is an update mechanism that allows an enterprise information system (EIS) to send data changes to Unwired Server over an HTTP or HTTPS connection using JavaScript Object Notation (JSON).

Two steps are required to send DCN all the way from the EIS to the device: EIS to Unwired Server (DCN), and Unwired Server to device (synchronization). DCN is independent of device synchronization and can be used with or without push synchronization.

This guide describes DCN only. For information about Hybrid App DCN (previously called WF-DCN) see the *Developer Guide: Hybrid Apps*.

All DCN commands support both GET and POST methods. The EIS developer creates and sends a DCN to Unwired Server through HTTP GET or POST operations. The portion of the DCN command parameters that come after `http://host:8000/dcn/DCNServlet`, can all be in POST; any *var=name* can be in either the URL (GET) or in the POST. The HTTP POST method is more secure than HTTP GET methods; therefore, Sybase recommends that you include the authenticate.password parameter in the POST method, as well as any sensitive data provided for attributes and parameters.

You must be familiar with the EIS from which the DCN is issued. You can create and send DCNs that are based on:

- Database triggers
- EIS system events
- External integration processes

For detailed DCN with payload and without paylod examples using POST commands from a SAP ABAP EIS using a Firefox REST Client, see *How-To Use Data Change Notification for CDB Update* at *http://scn.sap.com/docs/DOC-28185#*. While the example is based on Sybase Unwired Platform version 2.1, the basic procedures apply to other versions as well.

Data Change Notification Data Flow

This diagram illustrates DCN data flow.
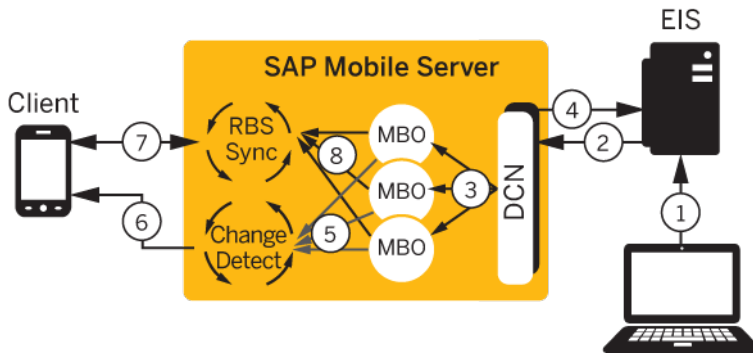
Steps one through four describe DCN, while steps five through eight provide an example of how the change itself is synchronized with the client using targeted change notification (TCN), previously called server-initiated synchronization (SIS), with a client using synchronization parameters.

1. EIS update – a program or some other process updates data in the EIS which is associated with a DCN.
2. HTTP(S) push – the EIS pushes a DCN message with new or changed MBO data contained in the message on the configured HTTP(S) port.
3. DCN operation – the DCN service receives the message and performs the upsert/update to the CDB tables of the corresponding MBOs. DCN upsert/delete operations also set the changed flag of an MBO package to true.
4. Unwired Server response – Unwired Server sends a response message back to the EIS that contains the status of each DCN in the submitted message.
5. Change detection – for a particular device that needs the new data, Unwired Server generates a message for the client indicating that it should synchronize.
6. Change notification – the message is pushed from Unwired Server to the device.
   The flow assumes that the device is registered for notifications and comes online to receive the notification:
   - If the device is online (as viewed in Sybase Control Center, Application Connections), then the notification message is immediately delivered to the device and the application handles the notification accordingly (the developer must code for this scenario).
   - If the device is offline, then the notification is queued and the "Pending Count" for that application connection increases and can be observed to be non-zero in the same Sybase Control Center screen (Application Connections). Optionally, for iOS, APNS can be configured to provide notification to the device/application which ultimately

results in the user launching the application, causing it to reconnect to Unwired Server to retrieve the notification.

See *Subscribing Applications to Push Synchronization Notifications* in *Mobile Application Life Cycle* and *Push Synchronization Applications* in *Developer Guide: <Device Platform> Object API Applications* for additional information about notifications.

7. RBS synchronization – the client receives the message and issues a synchronization request to Unwired Server.

8. Synchronization – Unwired Server retrieves the new/updated data based on the client synchronization parameter and returns it to the client.

Data Change Notification's Effect on the Unwired Server Cache

Supported DCN types and their effect on the Unwired Server cache (CDB):

- **DCN with operation replay (without payload)** – as an example, the EIS contains information for the Customer MBO, with an EIS primary key cust_id, and the MBO model includes a READ operation that retrieves all Customer details and populates the CDB using cust_id. In DCN without payload, if the EIS notices that customer data for cust_id=123 has changed in the EIS, it can send a DCN to Unwired Server with essentially:

```
{MBO=customer, operation=read, cust_id=123}
```

to inform Unwired Server to execute the read operation that updates the CDB with cust_id=123 changes. This form of DCN involves a lot of interactions with Unwired Server and the EIS to pull data changes into the CDB.

- **DCN with payload** – is more efficient since the EIS sends a DCN containing the data:

```
{MBO=customer, operation=upsert, {cust_id=123, fname=John,
lname=Smith, address=...}}
```

when the EIS notices that cust_id=123 has changes, and Unwired Server applies these changes directly to the CDB without additional calls to the EIS.

- **Hybrid App DCN** – used to push notifications triggered by the EIS to the Hybrid App and has nothing to do with the CDB.

Data Change Notification With Payload and Without Payload

Understand the differences between DCN with payload and DCN without payload.

- DCN without payload – calls MBO operations, where the name used in the DCN request matches that of the MBO definition.
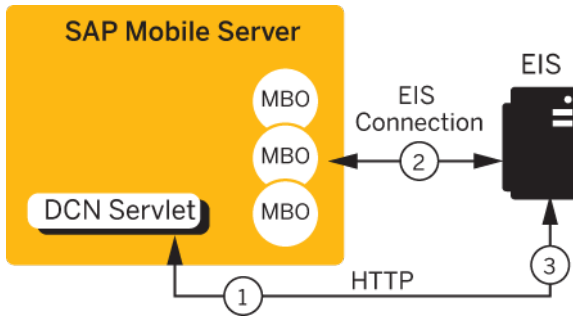
1. The DCN requester sends an MBO operation execution request, along with operation parameters, to Unwired Server.
2. Unwired Server executes the operation, (effectively calling the EIS operation), and updates the cache database (CDB), if needed, depending on the operation's cache policy.
3. Unwired Server returns a DCN status message to the requester.

Note: Be careful when naming MBO operations in Sybase Unwired WorkSpace, for example, an EIS-affecting operation named "delete" may be easily confused with the direct cache-affecting operation named **:delete**.

- DCN with payload – calls only the two direct cache-affecting operations (**:upsert** or **:delete**), which always exist for an MBO, and are not related to user-defined MBO operations.
  - **:upsert** – the message must contain name/value pairs for every required attribute, and the name must exactly match the MBO attribute name.
  - **:delete** – provide only the name/value pairs for the primary key column(s).

These operations respectively insert or update, or delete a row in the CDB. Calling either of these operations does not trigger any other refresh action. A typical sequence of events might be:
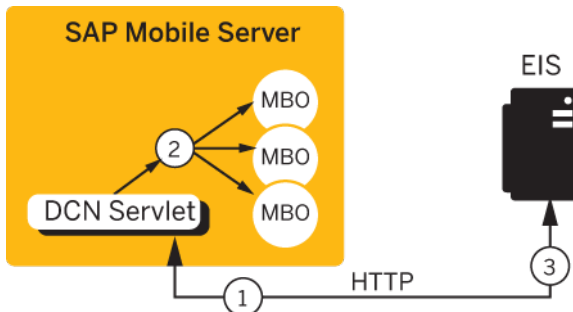
1. Some event initiates the DCN request (a database trigger for example).
2. The Unwired Server cache could be updated directly from the EIS, or the DCN request could originate from a source other than the EIS. The actual data (payload) is applied to the cache, through either an **:upsert** (update or insert) or a **:delete** operation.

    3. Unwired Server returns a DCN status message to the requester.

Performance Considerations for DCN With Payload Versus Without Payload

Performance is one factor to consider when determining whether to implement DCN with or without payload.

If your application requires business data details from the EIS, including those business objects inside your DCN message reduces the number of calls made to the EIS. In these cases, DCN with payload generally provides better performance. But keep in mind that when using DCN with payload, the EIS spends more time retrieving the object data and converting it into the JSON format.

Related DCN Developer and Administrator Tasks

DCN is a feature that is performed in different runtime and development components of Unwired Platform. Review the tasks that must be performed by different roles to effectively implement DCN and Hybrid App DCN in an end-to-end environment.

Management and Monitoring of Data Change Notifications

Various Sybase product documentation guides provide information about managing DCNs and monitoring DCN statistics and performance.

Goal	Topic
Use DCN with other cache management features.	*Cache Data Management* in the *Mobile Application Life Cycle*
Configure synchronization groups to notify device users when a DCN event has occurred. The notification tells the user to synchronize and receive those updates.	In *Sybase Control Center for Sybase Unwired Platform*, see *Configuring Synchronization Groups* in *Sybase Control Center for Sybase Unwired Platform*.
Monitor DCN activity.	In *Sybase Control Center for Sybase Unwired Platform*, see *Checking System Statistics*, *Related Data Change Notification Information*.

MBO Development for Data Change Notification

Sybase Unwired WorkSpace - Mobile Business Object Development contains details about configuring MBOs to enable DCN to refresh cached MBO data.

While an MBO belongs to a single cache group, MBOs in the same project are not necessarily in the same cache group. The cache group policy determines the data refresh behavior of all

MBOs within the group. DCN can be used as the sole mechanism of refreshing cached data in Unwired Server by specifying the DCN cache refresh policy. See *Best Practices for Loading Data from the EIS to the CDB* in *Mobile Data Models: Using Mobile Business Objects*.

Implementing Data Change Notification

Follow the syntax described in this section to implement DCN with payload and DCN without payload.

Invoking upsert and delete Operations Using Data Change Notification

Data change notifications (DCNs) with payload directly update the Unwired Server cache, either with the built-in, direct cache-affecting operations **:upsert** (update or insert), or with **:delete**.

Syntax

DCN with payload requires a JavaScript Object Notation (JSON) string (dcn_request) that contains one or more **:upsert** and **:delete** operations that are applied to the Unwired Server cache (CDB).

```
http://SAP_mobile_server_host:SAP_mobile_server_port(default 8000)/
dcn/DCNServlet
? cmd=dcn
&username=userName
&password=password
&domain=domainName
&package=SAP_mobile_server_PackageName
&dcn_request={"pkg":"dummy","messages":
[{"id":"1","mbo":"CustomerWithParam", "exp":"2012-09-29 00:20:00",
"op":":upsert","cols":{"id":"10001","fname":"Adam"}}]}
&dcn_filter=fully_qualified_name_of_dcn_filter
```

Parameters

- **SAP_mobile_server_host** – Unwired Server host name to which the DCN is issued.
- **SAP_mobile_server_port** – Unwired Server port number. The default port is 8000.
- **username** – authorized Unwired Server user with permission to modify the MBO and permission to submit DCN requests (controlled by possessing the "SUP DCN User" logical role).
- **password** – authorized user's password.
- **domain** – Unwired Server domain that contains the package.

- **package –** Unwired Server package that contains the MBO. The format is package:version. For example, e2e_package:1.0. This is the package name and version as it appears in the Packages folder as viewed from Sybase Control Center.
- **dcn_request –** the JSON string that contains operation name and parameters, which must include:
  - Package name (pkg) – this package name is required to support backwards compatibility but ignored. The package value supplied in the header is the package value used by DCN.
  - A list of messages (messages). Each message includes:
    - A unique message ID (id) used to report back the status. The values provided for the "id" element of each DCN statement within a DCN request message are used only to identify the corresponding status message in the DCN response, which means you can select any value, including nonnumeric characters. Use unique values, so that responses to the correlated requests can be clearly identified.
    - Mobile business object name (mbo).
    - An optional expiration date (exp) sets the expiration time for the cached data using UTC time format (YYYY-MO-DD HH:MM:SS, for example 2012-09-29 00:20:00), which is converted to local time by Unwired Server.
    - Operation name (op): either one of the direct cache-affecting operations (**:upsert** or **:delete**) or one of the user-defined MBO operations.

      Note: The specified cache policy property of the operation still applies when DCN is used to invoke a user-defined MBO operation.

    - Bindings (cols): name and values of operation arguments which are mapped to MBO attributes.
  - (Optional) ppm – personalization parameters (for either the server or client side) that need to be explicitly defined in the DCN request. The format must conform to the JSON messaging synchronization format, which is a Base64-encoded map of personalization parameters. For example, for runtime credentials sent via DCN, the PPM might be:

    ```
    base64encode("{\"username\":\"supAdmin\",\"password\":\"test\"}");
    ```

    See *Data Change Notification Requirements and Guidelines* for guidelines including how to build the ppm.
  - (Optional) partition commands – see *EIS Defined Cache Partitions* for information about partition management.
- **dcn_filter –** (optional) the custom filter used to pre-process the DCN request and post-process the DCN status message. By default, Unwired Server requires the value of the dcn_request field to be a valid JSON string. A DCN filter is used to convert the dcn_request field from a client-specific format to a valid JSON string, before processing in the SUP server. The filter can also reformat the status message returned in the DCN response into a custom format defined by the user.

Examples

- **Upsert example with header** – In the following examples, *supAdmin* represents the Unwired Server Administrator, and *supPwd* represents the Administrator's password defined during Unwired Platform installation.

  this DCN contains a single **:upsert** operation that updates or inserts (upserts) data in the Unwired Server cache for the Department MBO.

  ```
  http://dsqavm5:8000/dcn/DCNServlet?cmd=dcn&username=
  supAdmin&password=supPwd&package=dept:
  1.0&domain=default&dcn_request=
  {"pkg":"dummy","messages":
  [{"id":"1","mbo":"Department","op":":upsert",
  "cols":{"dept_id":"2","dept_name":"D2","dept_head_id":"501"}}]}
  ```

- **Upsert example without header** – this JSON string included in a DCN contains a single **:upsert** operation that updates or inserts (upserts) data in the Unwired Server cache for the Department MBO.

  ```
  dcn_request={"pkg":"TestPackage",
  "messages":
      [{"id":"1","mbo":"Department",
          "op":":upsert",
          "cols":{"DepartmentID":"3333",
                  "DepartmentName":"Test Value",
                                      "DepartmentHeadID":"501"}}]
  }
  ```

- **Delete example with header** – this DCN example deletes a row of data from the Unwired Server cache for the Department MBO:

  ```
  http://dspevm5:8000/dcn/DCNServlet?cmd=dcn&username=
  supAdmin&password=supPwd&package=dept:
  1.0&domain=default&dcn_request=
  {"pkg":"dummy","messages":
  [{"id":"1","mbo":"Department","op":":delete",
  "cols":{"dept_id":"2"}}]}
  ```

- **Delete example without header** – this example JSON string included in the DCN sent to Unwired Server, deletes a row of data from the Unwired Server cache for the Department MBO:

  ```
  dcn_request={"pkg":"TestPackage",
      "messages":[{"id":"1","mbo":"Department",
  "op":":delete",
      "cols":{"DepartmentID":"3333"}}]}
  ```

Usage

Follow these guidelines when constructing a DCN:

- For timestamp values use this format YYYY-MM-DDTHH:MM:SS where:

- YYYY is the four digit year, MM is the two digit month, DD - day, HH - hour, MM - minutes, and SS -seconds
- - is the literal dash ('-') character, : is the literal colon (':') character, and T is the literal T character.

  For example, 2009-03-04T17:12:45.

Note: Time zone information should not be included since it is ignored by the server. Convert timestamps to the corresponding UTC value before submitting them.

- The **:upsert** operation requires:
  - All MBO primary key attributes to be present in the payload.
  - Any other MBO attributes used in the upsert.
  - All columns in the operation use attribute names (not the column names to which they are mapped).
- The **:delete** operation requires:
  - The MBO primary key attribute be present in the payload.
  - All columns in the operation use attribute names (not the column names to which they are mapped).

Controlling Notifications for Native Applications With Cache Partitions

Unwired Platform sends out change notifications based on Subscriptions. MBOs are assigned to synchronization groups and when the device application registers for change notifications, they become part of a subscription. Whenever Unwired Platform detects a change in an MBO, it sends out a notification to all subscribers.

You can further control notifications through the use of cache partitions:

- client defined – if MBO data within a synchronization group is partitioned by synchronization parameters, then only subscribers who have subscriptions to data in an affected partition are notified.
- EIS defined – if MBO data within a synchronization group is partitioned by the DCN partition attribute, then only subscribers who have subscriptions to data in an affected partition are notified.

In the Unwired Platform client API, SynchronizationGroup includes setEnableSIS(boolean) so that a client application can enable/disable push notifications, and the same class has setInterval(int minutes) to specify the minimum frequency over which Unwired Platform attempts to send notifications for a given synchronization group.

These settings can also be controlled through Sybase Control Center:

1. After you deploy a package, navigate to **Domains > <DomainName> > Packages >
    <PackageName> > Subscriptions > Replication**.
2. Select the **Template** tab and create subscription templates for the synchronization groups.

The template creates a subscription for each client the first time they synchronize that
synchronization group. You can use it to set defaults for the Notification Threshold (equivalent
to the `SynchronizationGroup.setInterval` client API). If you select the **Admin
lock** radio button, the template is applied and the client's synchronization group settings are
ignored.

After a client subscribes, you can view their subscriptions on the Devices tab and make manual
modifications to them as needed.

EIS Defined Cache Partitions

Data Change Notification (DCN) includes the ability to define cache partitions, update MBO
data within those partitions, and subscribe users to those partitions.

The EIS defined cache partition is a collection of data from multiple MBOs and provides an
alternative subscribe/unsubscribe model compared to the per-MBO synchronization
parameter model (client defined partition). A user can be assigned to more than one EIS cache
partition and an EIS cache partition can be assigned to more than one user.

The basic flow for defining and subscribing to an EIS cache partition is:

1. The MBO developer models the MBOs.
2. The MBO developer assigns the MBOs that are part of the EIS cache partition to a cache
    group that uses the **EIS Managed** policy.
    (Optional) The MBO developer sets the **Notify EIS to fetch operation** option. The
    notification endpoint is set on the EIS cache group before deployment, and cannot be
    added to the EIS cache group (or otherwise managed) in Sybase Control Center if it is not
    specified before deployment.
3. The MBO developer deploys the project to Unwired Server.
4. Using Sybase Control Center, the Unwired Server Administrator manages the cache group
    to which the EIS cache partition belongs, including:
    *   Defining the DCN and Notification listeners on which Unwired Server communicates
        with the EIS.
    *   Managing the Web service endpoint for operation upload notification. This endpoint is
        used to notify the EIS that replay messages are ready for upload, not for pushing replay
        messages to the EIS, and is available to map to a server endpoint only if the **Notify EIS
        to fetch operation** option was set during MBO development.
    *   Viewing cache group information.
    *   Setting subscription purge threshold and unsubscribed partition purge threshold for the
        "Synchronization Cache Cleanup" scheduled task, which purges expired entries from
        the cache.

5. The EIS Administrator includes JSON messages within the DCN with payload to manage the partition, including updating data within the partition and user subscription.

An example business case that benefits from an EIS managed cache partition is a daily delivery service:

1. Each evening the MBOs are updated by the EIS through DCN with the next day's deliveries and services to be provided. Several MBOs contain the RouteID attribute which identifies various aspects of the customer: location, services or products required, customer notes, and so on.
2. MBO information also includes the user (partition) assigned to each route.
3. In the morning, each user synchronizes and retrieves their schedules based on the data to which they subscribe.
4. Throughout the day each user enters information on the device regarding each customer visited.
5. At the end of the day, the team member updates the EIS through Unwired Server and logs out. For example, if the **Notify EIS to fetch operation** option was selected for the EIS managed cache group, operation replay records are sent to Unwired Server, Unwired Platform Runtime queues them internally and notifies the EIS through the Web service endpoint for operation upload notification, and the EIS retrieves them.

Creating an EIS Cache Partition
Partitions are created implicitly the first time a DCN upsert is performed to a new partition.

See *Adding or Deleting an MBO Instance To or From an EIS Cache Partition* for details.

Removing an EIS Cache Partition
This JSON message within a DCN request removes an EIS defined cache partition.

The partition and any subscriptions to the partition are removed, even if there are active subscribers. No notifications are sent to active subscribers informing them that the partition to which they were subscribed was removed.

Syntax
```
dcn_request= {"pkg":"pkgName","messages":
[{"id":"1","op":":removePartition",
"partition":"partitionName"}]}
```

Parameters

- **"op":":removePartition"** – Synchronously remove the specified partition from the cache.
- **"partition":"partitionName"** – The name of the partition that is removed.

Examples

- **Remove an EIS cache partition** – This JSON message deletes the TOURID_1 partition:

```
 dcn_request={"pkg":"dummy","messages":
[{"id":"1","op":":removePartition","partition":"TOURID_1"}] }
```

Adding or Deleting an MBO Instance To or From an EIS Cache Partition

This JSON message within a DCN request adds or deletes an MBO instance to/from an EIS
defined cache partition.

Syntax

```
dcn_request={"pkg":"partitiontest","messages":
[{"id":"1","mbo":"Department","op":":upsert/
delete","partition":"partitionName",
"cols":
{"dept_id":"1","dept_name":"Demonstrationdepartment","dept_head_id"
:"501"}}]}
```

Parameters

- **"op":":upsert/delete"** – Use **:upsert** for insert or update operations, and **:delete** for
  delete operations.
- **"partition":"partitionName"** – The partition to which the MBO instance is added or
  deleted. If the partition does not already exist, it is created the first time a DCN upsert is
  performed. An empty partition name (zero length string) is valid.

Examples

- **Add an MBO instance to an EIS cache partition** – This JSON message upserts a row of
  data for the Department MBO into partition1:

```
{"pkg":"partitiontest","messages":
[{"id":"1","mbo":"Department","op":":upsert","partition":"partiti
on1",
"cols":
{"dept_id":"1","dept_name":"Demonstrationdepartment","dept_head_i
d":"501"}}]}
```

- **Delete an MBO instance from an EIS cache partition** – This JSON message deletes a
  row of data for the Department MBO from partition1:

```
{"pkg":"partitiontest","messages":
[{"id":"1","mbo":"Department","op":":delete","partition":"partiti
on1",
"cols":
{"dept_id":"1","dept_name":"Demonstrationdepartment","dept_head_i
d":"501"}}]}
```

Creating a User Subscription to an EIS Cache Partition

This JSON message within a DCN request creates a user subscription to an EIS defined cache partition.

Syntax

```
dcn_request={"pkg":"partitiontest","messages":
[{"id":"1","op":":createSubscription",
"partition":"partitionName","username":"userName"}]}
```

Parameters

- **"op":":createSubscription"** – Creates a user subscription.
- **"partition":"partitionName"** – The name of the partition to which the user is subscribed.
- **"username":"userName"** – The user for which the subscription is created.

Examples

- **Create a user subscription to an EIS cache partition** – subscribe user1 to partition1:

  ```
  {"pkg":"partitiontest","messages":
  [{"id":"1","op":":createSubscription",
  "partition":"partition1","username":"user1"}]}
  ```

Removing a User Subscription From an EIS Cache Partition

This JSON message within a DCN request removes a user subscription from an EIS defined cache partition.

The JSON message can remove a user's subscription from individual partitions or remove all subscriptions for a given user.

Syntax

```
dcn_request= {"pkg":"pkgName","messages":
[{"id":"1","op":":removeSubscription(s)",
"partition":"partitionName","username":"userName"}]}
```

Parameters

- **"op":":removeSubscription(s)"** –

  - **removeSubscription** – indicates that a user subscription to a partition is removed.
  - **removeSubscriptions** – indicates that all subscriptions for a particular user are removed.

- **"partition":"partitionName"** – The name of the partition from which the subscription is removed. The partition is deleted if there are no longer any subscriptions to it. Omitting this parameter removes all subscriptions for the given user.
- **"username":"userName"** – The user for which the subscription is removed.

Examples

- **Remove a user subscription from an EIS cache partition** – This JSON message removes user1's subscription from partition1:

```
{"pkg":"partitiontest","messages":
[{"id":"1","op":":removeSubscription",
"partition":"partition1","username":"user1"}]}
```

- **Remove all subscriptions for a given user** – This JSON message removes all subscriptions for user1:

```
{"pkg":"partitiontest","messages":
[{"id":"1","op":":removeSubscriptions",
"username":"user1"}]}
```

Retrieving Pending Operations From Unwired Server

Understand how the EIS fetches pending operations from Unwired Server using Unwired Server HTTP services.

General Rules

- **Authentication** – the same set of rules used for accessing Unwired Server REST APIs also apply to pending operation access. See these topics in the *Developer Guide: REST API Applications*:
  - *Authentication*
  - *HTTP Headers and Cookies*
- **Authorization** – to access the HTTP services, the user requires the "SUP DCN User" role, and this DCN user is only authorized to access packages that are associated with the security configuration against which the user is authenticated. Anonymous access is not allowed.
- **Headers** – use X-SUP-DOMAIN and X-SUP-SC to override the domain derived from the URL and the default security configuration of the domain. See *HTTP Headers and Cookies* in the *Developer Guide: REST API Applications*.
- **Response code** –
  - If any request parameter format is invalid, Unwired Server responds with `400 Bad Request`.
  - If any request parameter value (domain, package) is not found in the system, Unwired Server responds with `404 Not Found`.
  - If a request is unauthorized, Unwired Server responds with `401 Unauthorized`.

- If any other failures are encountered when processing the request, Unwired Server responds with `500 Internal Error`.

Querying Pending Operations

The HTTP client can query the list of PendingOperations by making a GET request to this URL:

```
http://domain:port/end2end.rdb:1.0/PendingOperations?
username=test&amp;
remoteId=cc134c3a-1e7b-41cf-821c-900ec61eef91&amp;$skip=0&amp;
$top=10
```

All parameters are optional. If *username* and *remoteId* are not provided, Unwired Server returns all pending operations for the package. Successful invocation generates get response `200 OK` with the entity set in JSON format in the response body:

```
[
{"data":"{\"id\":1,\"lname\":\"lname2\",\"fname\":\"fname1\",
\"id1\":3520011}",
"entityName":"topicCustomer","operation":"ChangeNameNeedManager","r
equestId":3520012,
"remoteId":"582b509f-677e-4879-b1fb-
d2e0f11d9deb","username":"test"},
{"data":"{\"id\":2,\"lname\":\"lname4\",\"fname\":\"fname3\",
\"id1\":3520013}",
"entityName":"topicCustomer","operation":"ChangeNameNeedManager","r
equestId":3520014,
"remoteId":"582b509f-677e-4879-b1fb-
d2e0f11d9deb","username":"test"}
]
```

Updating Pending Operations

The HTTP client can update a list of PendingOperations by making a PUT request to this URL:

```
http://domain:port/end2end.rdb:1.0/UpdatePendingOperations
```

with this put string:

```
{"data":[
{"success":false,"requestId":13930007,"errorMessage":"It must
fail!","eisCode":"1234"},
{"success":false,"requestId":13930009,"errorMessage":"It must
fail!","eisCode":"1234"},
{"success":true,"requestId":13930008}]}
```

Basic HTTP Authentication

When you use http://<host>:8000/dcn/DCNServlet, the user authentication is done by Unwired Server extracting the user information from the request parameter.

```
username=<username>
password=<password>
```

Alternatively, you can use HTTP BASIC authentication instead of sending the username and password as part of the URL. To use HTTP BASIC authentication, the URL is `http://<hostname>:<port>/dcn/HttpAuthDCNServlet`, as this example illustrates:

```
URL url = new URL("http://<host>:8000/dcn/HttpAuthDCNServlet?
cmd=dcn&package=<package_name>:<package_version>");
        HttpURLConnection huc = (HttpURLConnection)
url.openConnection();
        huc.setDoOutput(true);
        huc.setRequestMethod("POST");
        final String login = "<login_name_of_user_with_DCN_role>";
        final String pwd = "<password_of_user_with_DCN_role>";
        Authenticator.setDefault(new Authenticator()
        {
            protected PasswordAuthentication
getPasswordAuthentication()
            {
                return new PasswordAuthentication(login,
pwd.toCharArray());
            }
        });
        String dcnRequest = "{\"pkg\":
\"<package_name>:<package_version>"\","
            + "\"messages\":[{\"id\":\"1\",\"mbo\":\"CustomerState
\",\"op\":\":upsert\","
            + "\"cols\":{\"id\":\"1020\",\"fname\":\"Paul\",\"city
\":\"Rutherford\"}}]}";
        StringBuffer sb = new StringBuffer();
        sb.append(dcnRequest);
        OutputStream os = huc.getOutputStream();
        os.write(sb.toString().getBytes());
        os.close();
        BufferedReader br = new BufferedReader(new
InputStreamReader(huc.getInputStream()));
        System.out.println(huc.getURL());
        huc.connect();
        String line = br.readLine();
        while (line != null)
        {
            System.out.print(line);
            line = br.readLine();
        }
```

HTTP POST and DCN

You can also use the URL `http://<hostname>:8000/dcn/HttpAuthDCNServlet` if you do not want to send the DCN request as a request parameter but as an HTTP POST body instead.

If you are using HTTP BASIC authentication, the JSON encoded DCN request is always sent as the HTTP POST body.

Data Change Notification Requirements and Guidelines

Familiarize yourself with data change notification (DCN) requirements before implementing DCN.

Personalization parameters in DCN

Personalization parameters of the MBO need to be specified separately in the **ppm** parameter. The required ppm parameter in the `dcn_request` has to be a `string` which should be a Base64-encoded map of personalization parameters. This example shows how to use `ppmString` to define the value for the **ppm** parameter, (including the `gson.toJson()` method used to convert the parameter to the required JSON format), which is then used in the `dcn_request`:

```
Map<String, String> ppm = new HashMap<String, String>();
ppm.put("myCompany", "Sybase");
String ppmString =
Base64Binary.toString(gson.toJson(ppm).getBytes());
```

DCN upsert operations and MBO relationships

When using the DCN payload mode to upsert rows to MBOs where there is a relationship between rows of data, you must provide the data in the correct order so Unwired Server can properly create the metadata in the cache (CDB) to reflect the data relationship. However, when you are using DCN to insert data into the cache, the concept of child and parent may be different from what is reflected in the graphical model of the package used in the design tooling. Also, one-to-many relationships differ, as noted below.

When using DCN to upsert rows to both the parent and child MBOs in a relationship, the order for the upserts can change depending on the nature of the relationship. This is due to the implementation details of the cache metadata. In these examples, the Department MBO is the parent MBO in both relationships, but notice the order of the upsert operations:

- For a one-to-one relationship between:

  ```
  Dept.dept_head_id - > Employee.emp_id
  ```

  (from a department to the department head) the order in which you upsert a new department and new department head is:
  1. Employee
  2. Department

  The foreign surrogate key reference is contained in the cache table used to hold the data for the Department MBO.

- For a one-to-many relationship between:

  ```
  Dept.dept_id - > Employee.dept_id
  ```

  (from a department to all of the employees in the department) the order in which you upsert a new department and a new employee is:

   1. Department
   2. Employee

The foreign surrogate key reference is contained in the cache table used to hold the data from the Employee MBO.

Message autonomy

Unwired Server expects serialized DCN message updates to MBO instances. Therefore if concurrent DCN clients or processes are used, insure that all updates to all rows of MBO(s) are contained within a single DCN request in order to avoid a possible deadlock condition.

Unwired Server expects an entire graph when sending updates to MBOs within a composite relationship.

DCN and deadlocks

The requirements described in *Message autonomy* is designed to prevent deadlock situations. However, if you do not define an order of operation execution, deadlocks might occur depending on the DCN implementation or the locking mechanism used by the enterprise information system (EIS). In a deadlock situation, the entire transaction is rolled back (if there are multiple operations in a single DCN) and a `replayFailed` result is returned.

DCN upsert operations and binary data

When using DCN to upsert binary data to the cache (CDB), the string used for the value of the binary type attribute of the MBO in the request message must conform to a very specific encoding for the DCN request to be processed correctly. Read the binary data into a byte array, then use the following code to obtain it in the correctly encoded format:

```
byte[] picByteArray = < < user code to read binary data into byte[] >
>
String picStringBase64Encoded =
com.sybase.djc.util.Base64Binary.toString(picByteArray);
String picStringUrlEncoded =
java.net.URLEncoder.encode(picStringBase64Encoded, "UTF-8");
```

Use the **picStringUrlEncoded** string as the value for the binary attribute in the DCN request message.

Note: The com.sybase.djc.util.Base64Binary class is in the `sup-server.jar` from the `SUP_HOME\Servers\UnwiredServer\lib` of the installation directory.

Proper encoding of DCN JSON

DCN JSON information must be properly encoded when sent to Unwired Server:

*   The server/Sybase Unwired Platform DCN Web container accepts UTF-8 as its URL/ x-www-form-urlencoded charset encoding scheme.
*   In addition to the dcn_request parameter/value pair in the URL/ x-www-form-urlencoded form, encode other parameter/value pairs according to W3C/IETF specifications.

Refer to *http://www.w3.org/TR/1999/REC-html401-19991224/interact/ forms.html#h-17.13.4.1* and *http://www.ietf.org/rfc/rfc2045.txt* for details.

For example, only alphanumerics [0-9, a-z, A-Z], the special characters contained in the parentheses ($ - . ! * ' () ,), and reserved characters used for specific reserved purposes may be used unencoded within a URL. This DCN upsert command:

```
http://serverhost:8000/dcn/DCNServlet?
cmd=dcn&username=supAdmin&password=s3pAdmin&
domain=default&package=sup101:1.0&dcn_request={"pkg":"dummy","messa
ges":[{"id":"4","mbo":"Customer",
"op":":upsert","cols":{"id":"10004","fname":"B%B"}}]}
```

contains reserved and special characters { } [] : " % that must be encoded or a parsing error is thrown.

If you use Java to make the DCN request, use `URLEncoder.encode(parameter,"UTF-8")` to encode the URL query parameter or post form data before making the request. For this example, construct the URL using this code:

```
String url="http://serverhost:8000/dcn/DCNServlet?
cmd=dcn&username=supAdmin&
password=s3pAdmin&domain=default";
url+="&package="+URLEncoder.encode("sup101:1.0","UTF-8");
url+="&dcn_request="+ URLEncoder.encode("{\"pkg\":\"dummy\",
\"messages\":[{\"id\":\"4\",\"mbo\":\
"Customer\",\"op\":\":upsert\",\"cols\":{\"id\":\"10004\",\"fname\":
\"B%B\"}}]}","UTF-8");
```

DCN and date, time, and datetime datatypes

DCN accepts date, time, and datetime attribute and parameter values using this format:

* `date` – yyyy-MM-dd
* `time` – HH:mm:ss
* `datetime` – yyyy-MM-dd'T'HH:mm:ss

For example, Unwired Server parses `string` or `long` values and upserts a valid timestamp object:

```
http://localhost:8000/dcn/DCNServlet?
cmd=dcn&username=supAdmin&password=
AdminPassword&package=testdatetime:1.0&domain=default&dcn_request=
{"pkg":"testdatetime","messages":
[{"id":"1","mbo":"TestDateTimeStamp","op":":upsert",
"ppm":null,"cols":
{"testTimestamp":"2009-08-09T12:04:05","testDate":"2009-08-09","c_i
nt":"0",
"testDateTime":"2009-08-09T12:04:05","testSmalldt":"2009-08-09T12:0
4:05","testTime":"12:04:05"},
}]}
```

Complex types

Special care must be taken when using DCN to populate MBOs which form the sub-types of a larger complex type. For example a PurchaseOrder type is composed of POHeader and POLineItem MBOs. To successfully populate the rows of the POHeader and POLineItem MBOs with data derived from a PurchaseOrder, the DCN code must properly set the primary key attributes and insert the rows in the correct order so that the relationship between rows from the POHeader and the POLineItem MBOs is successfully defined. See also the "DCN upsert operations and MBO relationships" topic.

DCN and Cache refresh policy

If DCN is the exclusive mechanism for loading and maintaining the data for the MBOs from a cache group, set the cache refresh policy to DCN.

DCN with payload directly updates the CDB by inserting the record into the MBO cache table. Using a DCN cache group policy ensures that MBO data is updated only through DCN and not another refresh mechanism. When designing your MBOs, keep in mind that if you use other methods (for example, a Scheduled cache group policy), DCNs as well as the scheduled refresh update the MBO.

If the cache group relies on a combination of DCN with On-demand or Schedule cache refresh policies, then you cannot use DCN with MBOs that define more than one client defined partition (that is, a load operation mapped to synchronization parameters in Sybase Unwired WorkSpace). The cache refresh policy does not support updating data inserted via DCN when the configured cache policy triggers a refresh of the client defined partitioned data.

If the load operation initializes the MBO, and you use DCN to maintain the MBO, then associate the MBO with a cache group that implements an infinite schedule. Do not send DCN messages until the cache is initialized.

Cache policies and DCN

Do not use a cache policy that invalidates the cache if you use a DCN to populate the MBO.

Data discrepancies and deleted data

If any attribute values differ from the actual EIS values, those values are updated with the actual value when a cache refresh occurs. Any rows inserted into the cache which contain a primary key value which is not present in the EIS are marked as logically deleted when a cache refresh occurs. Once a row has been marked as logically deleted from the cache, attempts to upsert data using that same primary key value fail until the logically deleted row is purged from the cache.

Purging Expired DCN Data From the Unwired Server Cache

Use the **exp** zoneless DATETIME datatype header property in DCN messages to indicate the expiration date, after which the data is purged from the Unwired Server cache based on the defined Synchronization Cache Cleanup schedule.

The **exp** property is useful in cases where the EIS does not efficiently support delete operations. It limits cache growth on the server of unwanted data and also purges data from clients when data has expired.

1. Include the **exp** property in the DCN message.

   For example, this DCN:

   ```
   {"pkg":"Test:1.0","messages":[{"id":"1","mbo":"Department",
   "exp":"2012-09-29 00:20:00",
   "op":":upsert","cols":
   {"dept_id":"1000","dept_name":"QA1000","dept_head_id":"501"}}]}
   ```

   sets the expiration time for the cached data to 2012-09-29 00:20:00 UTC time, which is converted to local time by Unwired Server.

2. Enable the **Synchronization Cache Cleanup** scheduled task in Sybase Control Center and define a purge schedule to remove expired data for a configured DCN cache group.

   - By default, no purge schedule is defined for a specific DCN cache group. The schedule supports logically deleting and/or physically deleting rows (dependent on whether Surrogate Key tracking is enabled) within the context of an expired data purge.
   - Unwired Server automatically purges composite data based on the newest node in the composite data graph. The caller must identify a graph by composite root MBO Name and Business Primary Key.
   - Unwired Server generates log entries for:
     - System log info message when DCN purge executes
     - Server log for any changes to the DCN cache purge schedule

Data Change Notification Results

Each binding in a data change notification (DCN) request is associated with an ID. The result status of the DCN request is returned in JavaScript Object Notation (JSON) format, and includes a list of IDs followed by a Boolean success field and status message, in case of error.

The processing of the individual messages within a DCN request is done as a single transaction. A failure of one message results in the changes from all preceding messages being rolled back and all following messages are skipped. In response to payload and MBO operation DCNs, Unwired Server sends the requester a JSON string containing details about the success and/or failure of the operations. These examples show the JSON-formatted result for a multi-message DCN request, and has been formatted using newlines, and indentations, which are not present in an actual response.

This is an example of a response message for successful processing of a request containing four individual messages, using id values {1, 2, 3, 4}:

```
[{"recordID":"1","success":true,"statusMessage":""},
{"recordID":"2","success":true,"statusMessage":""},
{"recordID":"3","success":true,"statusMessage":""},
{"recordID":"4","success":true,"statusMessage":""}]
```

In this example, the third message in the request contains an error:

```
[{"recordID":"1","success":false,"statusMessage":
"Changes rolled back because dcn message with ID 3 in the DCN request
failed."},

{"recordID":"2","success":false,"statusMessage":
"Changes rolled back because dcn message with ID 3 in the DCN request
failed."},

{"recordID":"3","success":false,"statusMessage":
"VirtualTableName is null. MBO name Departments might be incorrect or
with missing
capitalization in the DCN request"},

{"recordID":"4","success":false,"statusMessage":
"Processing of DCN message skipped because dcn message with ID 3 in
the DCN request failed."}]
```

Data Change Notification Filters

Data change notification (DCN) requests need not always be in the format Unwired Server expects.

You can deploy a DCN filter to Unwired Server and reference it in the DCN request. Unwired Server allows the filter to preprocess the submitted DCN. The filter converts raw data in the DCN request to the required JavaScript Object Notation (JSON) format. The filter can also postprocess the JSON response returned by the Unwired Server into the format preferred by the back end (which is governed by the implementation in the filter class).

The filter interface `DCNFilter` is in the `com.sybase.sup.server.dcn` package in the `sup-server-rt.jar` file. All classes that implement a DCN filter should implement this interface. The functions available in the interface are:

- **String preprocess(String blobDCNRequest, Map<String, String requestHeaders> requestHeaders);** – takes the DCN request as a binary large object (BLOB), converts it into a valid JSON DCN request format, and returns the same.
- **String postprocess(String jsonDCNResult, Map<String, String responseHeaders> responseHeaders);** – takes the DCN result in a valid JSON format, converts it to the EIS-specific format, and returns the same.

Figure 3: DCN filter flow
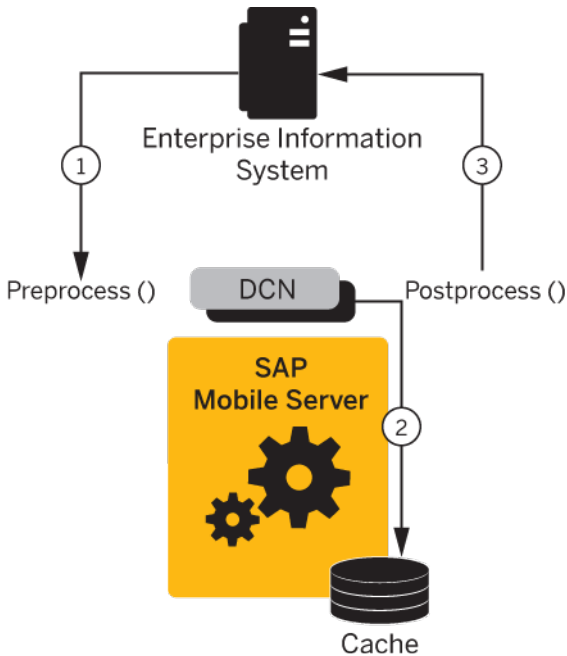
1. Changed data is sent from the EIS to Unwired Server via a DCN request, where any data preprocessing occurs. For example, the EIS data could be sent to Unwired Server as XML where the preprocess filter converts the data to JSON.
2. The DCN executes. For example, apply data changes directly to the Unwired Server cache.
3. Postprocessed DCN response is sent to the originating EIS as an HTTP response to the original DCN request. For example, the JSON response is converted to XML.

Implementing a Data Change Notification Filter

Write and deploy preprocess and postprocess DCN and Hybrid App DCN filters to Unwired Server.

When specifying filters, add a **dcn_filter** parameter to the base URL, and to the parameters specified in the DCN request section. The **dcn_filter** parameter specifies the fully qualified name of the filter class, which must be in a valid CLASSPATH location so Unwired Server can locate it using its fully qualified name.

JSON requires colons to define the object structure, but since colons have a special function in HTTP URLs, use the tilda character "~" instead of colons ":" when implementing the DCN filter, so the JSON dcn_request string can be passed as an HTTP GET or POST parameter:

```
dcn_request={"pkg"~"TestPackage",
    "messages"~[{"id"~"1","mbo"~"Department","op"~"~upsert",
    "cols"~{"DepartmentID"~"3333",
            "DepartmentName"~"My Department",
```

```
           "DepartmentHeadID"~"501"}}]}
```

The dcn_request is in a format that is specific to the back end. The filter class can preprocess to the JSON format expected by Unwired Server.

1. Write the filter. For example:

```
import java.util.Map;
import com.onepage.fw.uwp.shared.uwp.UWPLogger;
import com.sybase.sup.dcn.DCNFilter;

public class CustomDCNFilter implements DCNFilter
{
  String preprocess(String blobDCNRequest, Map<String,String>
headers) {
        String result = blobDCNRequest.replace('~',':');
        return result;
  }

  String postprocess(String jsonDCNResult, Map<String,String>
responseHeaders) {
        String result = jsonDCNResult.replace(':','~');
        return result;
  }

    public static void main( String[] args ) { }
}
```

2. Package your DCN filter class in a JAR file.
3. Deploy the JAR file to Unwired Server by using the Deployment wizard from Sybase Unwired WorkSpace (regular DCN), or manual deployment (Hybrid App DCN):
   - For regular DCN:
     a. Invoke the deployment wizard. For example, right-click in the Mobile Application Diagram and select **Deploy Project**.
     b. Select the JAR file that contains your DCN filter class files to deploy to Unwired Server in the third screen of the wizard (Package User-defined Classes).
     c. Click **Finish** after selecting the target Unwired Server.
   - For Hybrid App DCN:
     a. Place the JAR file in SUP_HOME\Servers\UnwiredServer\lib\ext.
     b. Create an empty file named sup.cff in SUP_HOME\Servers \UnwiredServer\bin\private.
4. Restart Unwired Server.

Custom XSLT Transforms

If you are using data from a SOAP or REST Web service, you may need to use XSLT (Extensible Stylesheet Language Transformations) to modify the structure of the message data generated by the service, so it can be used by an Unwired Platform MBO. Sybase Unwired WorkSpace can create XSLT transforms automatically, however sometimes these generated transforms are not sufficient and do not yield the results you require.

MBOs typically require a flat and tabular message structure from a Web service. This tabular structure corresponds to the rows and columns that eventually materialize the MBO's instances and attributes, respectively. Therefore the message structure used by a Web service must align correctly. Transformation must be precise to avoid unexpected results in an MBO.

Therefore, always validate the transform before deploying it to a production environment.

Custom XSLT Use Cases

In most cases, the XSLT that is generated by Sybase Unwired WorkSpace is sufficient. However, in some cases, you may need to modify the generated XSLT file, or to create a new one manually.

Some of these cases include:

* Web service response messages do not precisely conform to the schema.
  For example, the schema indicated that an integer field is not nullable, but the Web service response message failed to return a valid integer value. This omission triggers an error on the device application. even though the root issue is the data from the Web service, not Unwired Platform.
  In this scenario, it is simpler to modify the generated XSLT slightly, by changing the single op_nullable field from `false` to `true`.

Implementing Custom Transforms

When the generated transform does not yield expected results in the MBO, you need to either modify the generated transform or create a custom transform outside of Sybase Unwired WorkSpace.

1. Make changes to an existing transform or write a new one.
2. Save the changes and overwrite the file that already exists. This ensures that the binding remains intact for the MBO. See *Binding Mobile Business Objects to Data Sources* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

3. Redeploy the MBO, and include the transform in the deployment package.

See *Packaging and Deploying Mobile Business Objects* in *Sybase Unwired WorkSpace - Mobile Business Object Development*.

Note: If you are redeploying to a production environment, ensure the administrator redeploys the MBO with the modified transform.

XSLT Stylesheet Syntax

XSLT stylesheet must follow Unwired Platform stylesheet syntax requirements so that the Web service response message is formatted correctly for MBOs bound to this datasource.

The stylesheet is applied to different parts of the Web service response message, depending on the type:

- For SOAP web service response messages, the stylesheet is applied to the contents of the SOAP body.
- For REST web service messages, the stylesheet is applied to the contents of the HTTP response body.

Table 14. Stylesheet elements

Element	Description	Contains
Data	The root element of the stylesheet.	One or more Record elements.
Record	The element that corresponds to a row in the tabular MBO data structure. The first Record element resulting from the transform describes the column using metadata (that is, names, data types, nullability, and so on). The Record element has no attributes, except when it is a metadata element. The contents of the Field elements should match the corresponding op_label values. The Record or Field values from this first Record element will not appear in the resulting tabular data structure.	One or more Field elements.

Element	Description	Contains
Field	The element that corresponds to the column value. The Field element has a number of attributes that can be used.	One or more attributes. See the *Attributes* table.

Table 15. Attributes

Attribute	Applicability	Description
op_label	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	The column name.
op_position	Required by all.	The attribute's position in the tabular structure. The first attribute is at position 1
op_nullable	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	Whether (TRUE) or not (FALSE) the attribute is nullable.
op_datatype	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	The data type. Supported values inlclude STRING, INT, LONG, BOOLEAN, DECIMAL, BINARY, FLOAT, DOUBLE, DATE, TIME, DATETIME, CHAR, BYTE, SHORT, INTEGER. See *Datatype Support*.
op_xsdtype	Required by the stylesheet and the resulting transformed structure's metadata. Ignored by the data field elements.	The XML schema primitive type name corresponding to this attribute.

XSLT Stylesheet Example

Use the example XSLT stylesheet to understand the structure required by Unwired Platform.

The bolded elements are required. The <xsl:stylesheet> needs a <xsl:template> element. The first child element of <xsl:template> must be the <data> that also requires the metadata <Record> element.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform"; xmlns:ns1="urn:Sample_Enrollments" exclude-result-
```

```
prefixes="ns1">
        <xsl:template match="//ns1:OpGetListResponse">
                <data>
                        <Record>
                                <Field op_label="Class_Cost"
op_position="1" op_datatype="DECIMAL"
op_nullable="false">Class_Cost</Field>
                                <Field op_label="Class_ID"
op_position="2" op_datatype="STRING" op_nullable="false">Class_ID</
Field>
                                <Field op_label="Class_Start_Date___Time"
op_position="3" op_datatype="DATETIME"
op_nullable="false">Class_Start_Date___Time</Field>
                                <Field op_label="Class_Title"
op_position="4" op_datatype="STRING"
op_nullable="false">Class_Title</Field>
                                <Field op_label="Enrollee_Login"
op_position="5" op_datatype="STRING"
op_nullable="false">Enrollee_Login</Field>
                                <Field op_label="Temp_Number"
op_position="6" op_datatype="INT" op_nullable="true">Temp_Number</
Field>
                        </Record>
                        <xsl:for-each select="ns1:getListValues">
                                <Record>
                                        <Field>
                                                <xsl:attribute
name="op_position">1</xsl:attribute>
                                                <xsl:value-of
select="ns1:Class_Cost"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">2</xsl:attribute>
                                                <xsl:value-of
select="ns1:Class_ID"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">3</xsl:attribute>
                                                <xsl:value-of
select="ns1:Class_Start_Date___Time"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">4</xsl:attribute>
                                                <xsl:value-of
select="ns1:Class_Title"/>
                                        </Field>
                                        <Field>
                                                <xsl:attribute
name="op_position">5</xsl:attribute>
                                                <xsl:value-of
select="ns1:Enrollee_Login"/>
                                        </Field>
                                        <Field>
```

```
                                                        <xsl:attribute
name="op_position">6</xsl:attribute>
                                                        <xsl:value-of
select="ns1:Temp_Number"/>
                                        </Field>
                                </Record>
                        </xsl:for-each>
                </data>
        </xsl:template>
</xsl:stylesheet>
```

If you use this style sheet, the output generated by this transform would be:

```
<data>
        <Record>
                <Field op_label="Class_Cost" op_position="1"
op_datatype="DECIMAL" op_nullable="false">Class_Cost</Field>
                <Field op_label="Class_ID" op_position="2"
op_datatype="STRING" op_nullable="false">Class_ID</Field>
                <Field op_label="Class_Start_Date___Time"
op_position="3" op_datatype="DATETIME"
op_nullable="false">Class_Start_Date___Time</Field>
                <Field op_label="Class_Title" op_position="4"
op_datatype="STRING" op_nullable="false">Class_Title</Field>
                <Field op_label="Enrollee_Login" op_position="5"
op_datatype="STRING" op_nullable="false">Enrollee_Login</Field>
                <Field op_label="Temp_Number" op_position="6"
op_datatype="INT" op_nullable="true">Temp_Number</Field>
        </Record>
        <Record>
                <Field op_position="1">100.00</Field>
                <Field op_position="2">00001</Field>
                <Field op_position="3">2010-07-02T10:27:35-07:00</
Field>
              <Field op_position="4">Managing Within the Law</Field>
                <Field op_position="5">Demo</Field>
                <Field op_position="6"/>
        </Record>
        <Record>
                <Field op_position="1">150.00</Field>
                <Field op_position="2">00005</Field>
                <Field op_position="3">2005-11-17T08:00:00-08:00</
Field>
               <Field op_position="4">Microsoft Word for Beginners</
Field>
                <Field op_position="5">Demo</Field>
                <Field op_position="6"/>
        </Record>
        <Record>
                <Field op_position="1">299.00</Field>
                <Field op_position="2">00006</Field>
                <Field op_position="3">2005-11-15T08:00:00-08:00</
Field>
                <Field op_position="4">Meeting Planning and
Facilitation</Field>
                <Field op_position="5">Demo</Field>
```

```
                <Field op_position="6"/>
        </Record>
</data>
```

Index