Introduction

WXSUtils is a wrapper/replacement API for IBM WebSphere eXtreme Scale (WXS). It works with any version of WXS from V7.0 and higher. It requires a Java 5 JVM due to its use of generics and other newer Java language features.
WXSUtils provides the following advantages:

- Client side only API.
  WXSUtils is intended for use by a client, not within the grid itself[1].
- Thread safe APIs
  All objects are thread safe.
- Higher level operations.
  It provides single method call replacements for operations that require multiple lines of code using the native API.
- No complex transactions.
  Each method call in WXSUtils is a single transaction. Transactions are not supported across method calls. Most applications in our experience do not use transactions. Transactions cannot span partitions.
- List and Set support.
  Map entries can be values but can also be advanced data structures such as lists and sets. APIs are provided that work with lists and sets. This reduces the amount of application code significantly. Many WXS applications can benefit from these advanced data structures.
- Property file configuration.
  Applications can simply 'connect' to the grid and all configuration happens in the background using a standard 'wxs.properties' configuration file.
- Bulk operation support.
  Many applications need efficient ways to bulk put or fetch values to and from the grid. WXS provides putAll and getAll APIs making these operations single method calls.
- Partition walking.
  WXS provides frameworks to make visiting partitions one by one a snap.
- All exceptions are runtime exceptions.
  This avoids boilerplate catching of exceptions in applications. This doesn't mean exceptions do not have to be caught however.

Installing WXSUtils

WXSUtils is provided as a sample project on github. All the source code is provided as a maven project. The easiest way to acquire it is to simply download the wxsutils*snapshot.jar from github. Just add this with the normal WXS jars to your classpath and you're ready to go. You can also download the source code and build it

[1] WXSUtils can be used directly in a container JVM against local primary shards. This will be covered in the advanced programming sections.

locally on your workstation. While WXSUtils provides lots of function that's ready to use, it also provides a lot of source code that you can use and modify to customize it for exactly what you need.

Introduction to programming WXSUtils

This section describes the basics of using the WXSUtils APIs from your application. The application typically starts by obtaining a WXSUtils object instance and then uses that for all future operations. The first step is to obtain this object instance.

Connecting to a grid

The first thing most applications need to do is to connect to a remote grid. Typically, this involves capturing the catalog server end points and then doing several API calls to WXS until finally you obtain an ObjectGrid reference. WXSUtils simplifies this considerably. There are three basic approaches to connecting.

Parameter based connection

Here, the application constructs a WXSUtils instance by calling the constructor with the catalog server endpoints and other information. For example:

```
ogclient = WXSUtils.connectClient("host:2809", "Grid",
"/multijob/multijob_objectgrid.xml");
WXSUtils utils = new WXSUtils(ogclient);
```

This shows how to use the WXSUtils.connectClient method to quickly connect to a remote grid. The objectgrid.xml file is found on the local applications classpath.

Wrapping an existing client ObjectGrid instance

If the application already has a client ObjectGrid instance then a WXSUtils instance can be constructed wrapping that existing client connection like this:

```
WXSUtils utils = new WXSUtils(ogclient);
```

Start a development mode test grid instance

WXSUtils provides a way to start a grid within the application JVM. This is convenient when doing development as it allows both the client and the grid to reside within a single JVM for easy debugging and deployment. This is done using the following API call:

```
ogclient = WXSUtils.startTestServer("Grid", "/objectgrid.xml", "/deployment.xml");
```

This starts the grid with the specified name and xml files. It actually starts a catalog instance first in the JVM, followed by a container. It returns a 'client' connection to this development grid for the application to use. Again, the xml files are usually on the application classpath.
This is very commonly used as it allows a developer to start his application in debug mode after placing breakpoints on both the 'client' side application code as well as any grid side code such as agents or loaders. This makes debugging code much easier than trying to work with several JVMs at once.

Configuration based connection

Many customers don't like having hard coded catalog endpoints or xml file paths in the application and many resort to command line arguments or custom property file implementations to avoid this. WXSUtils provides a simple property file to handle this on behalf of the application. The property file is named 'wxsutils.properties'. It should be in the root of your class path or in a folder specified on the application classpath. The configuration file is used to specify the following properties:

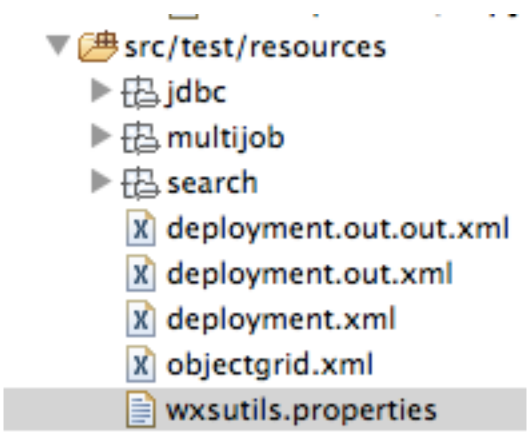
- cep=hostA:2809,hostB:2809
  This is the catalog JVM instances for the grid. If this is not present then a development grid is automatically started similar to startTestServer in the preceding section.
- grid=GridName
  The grid name must be specified. This is usually the grid name that the application requires a remote connection to.
- og_xml_path=/objectgrid.xml
  This must be specified and this should be a file that's present on the applications classpath. If this is not specified then a default value of '/objectgrid.xml' is used.
- dp_xml_path=/deployment.xml
  This is only required when starting a test server. If the cep is not specified then this property must be specified in addition to the og_xml_path property. If this isn't specified then a default value of '/deployment.xml' is used.
- thread=32
  WXSUtils uses a thread pool for its bulk operations. The capacity of this thread pool is specified using this property. The thread pool is shared by all WXSUtils instances in a JVM. The first one started specifies the thread pool size.

Here is an example developer wxsutils.properties:

```
#cep=a11.rchland.ibm.com:2809
grid=Grid
og_xml_path=/objectGrid.xml

# For connecting to a remote grid
#dp_xml_path=/META-INF/objectGridDeployment.xml
```

You can see the cep and dp_xml_path properties are commented out. This is typical in development. The dp_xml_path is defaulted to '/deployment.xml'. Both objectgrid.xml and deployment.xml files are required to start a test server and thus must be provided through explicit naming in the property file or they must use well known paths so that they can be found indirectly. The eclipse project would have resources defined like this:



This shows the objectgrid.xml, deployment.xml and wxsutils.properties in the root path of the test resources for this project.
Once these property files are created then the application uses the following code to obtain a connection to the grid. This will either connect to the remote grid or start a test grid within this application JVM depending on whether the cep property is present or not.

```
WXSUtils utils = WXSUtils.getDefaultUtils();
ObjectGrid ogclient = utils.getObjectGrid();
```

This shows a WXSUtils instance being obtained. It's trivial to obtain a client ObjectGrid instance from the utils instance if required as shown. Typically, the utils instance is stored in a well known place for all threads within the application to use from this point onwards.

Using WXSUtils in Web or EJB Modules

WXSUtils makes extensive use of ThreadLocals and statics. This means that starting and stopping a module is only supported if a servlet listener provided with WXSUtils is specified as a life cycle plugin in the web.xml (com.devwebsphere.wxsutils.WXSUtilsServletListener). This also means then when updating the module, unless this listener is specified then the JVM containing the modules to be updated MUST be RESTARTED to make sure WXSUtils is reinitialized correctly when the new module starts.

Working with Maps

Once the application has obtained a WXSUtils instance then it can start using it to work with Maps/Lists or Sets. We will start with using a Map. The application must have defined an existing map in the grid objectgrid.xml and deployment.xml files. It's normal to use templates so that new maps can be 'created' easily within restarting the grid. However, we will assume there is a 'normal' Map defined in the grid called map. We will also assume it uses a String key and a String value.

```
WXSMap<String, String> map = utils.getCache("map");
String value = "B";
map.put("A", value);
Assert.assertTrue(map.contains("A"));
String v = map.get("A");
Assert.assertEquals(value, v);
```

This shows a simple test code that obtains a WXSMap<String,String>. The first type is the key and the second type is the value. The WXSUtils.getCache method takes the name of the map as a parameter. The application can then work with the Map using the returned WXSMap instance. The WXSMap instance is thread safe and can be used in different threads concurrently. Each method on WXSMap uses it's own transaction. There is no way to group several method calls together as one transaction. This was done for simplicity sake and because it's the most common pattern seen in customer applications. Transactions are rarely used and so WXSUtils is not designed to allow several method calls within a single transaction.
You can see the put method stores the value "B" for the key "A" in the map. The code then checks the map contains A and then checks the value for "A" is indeed "B". The put method will do an insert if the key doesn't exist or an update if the key already exists. The get method simply returns the value for the key or null if the key doesn't exist.

Bulk operations

Many applications want to update large collections of key at once. Typically when preloading the grid with data. The putAll method provides an easy way to achieve this.

```
WXSMap<String,String> map = …;
Map<String, String> batch = new HashMap<String, String>();
batch.put("K1", "V1");
batch.put("K2", "V2");
map.putAll(batch);
```

This shows the putAll method on a WXSMap being used to bulk put two key/value pairs. The application could bulk put thousands of entries if required. WXSUtils implements putAll by sending an RPC per partition. This RPC will store all key/values hashing to that partition with a single operation. These RPCs are executed in parallel using the WXSUtils threadpool. The putAll method blocks until these RPCs are processed.
The putAll method will push the changes through any Loaders that are plugged in to the WXSMaps. This is usual but if the operation is a preload style operation where data is being preloaded in to the grid from a database, for example, then the

putAll_noLoader method is the correct one. This will store the records in the grid but will not write the changes to the Loader/database.

There are also the following bulk operations:

- void removeAll(Collection<K> keys)
  This removes all entries for the keys if they exist
- Map<K,V> getAll(Collection<K> keys)
  This returns the value for every key in the collection Keys. If the key doesn't exist then a null value is returned.
- void invalidateAll(Collection<K> keys)
  This invalidates the entries for all the keys.
- void putAll_noLoader(Map<K,V> batch)
  This is the same as putAll but will not invoke the Loader for the changes. It uses a beginNoWriteThrough transaction to achieve this.

Conditional Put operations

Conditional puts are similar to a normal put operation but they are conditional on the current value (or lack of) for the key. Normally, a put simply overwrites the existing value if it exists or it does an insert if there was no value.
A conditional put will only overwrite the value if the current value equals another value. For example.

```
WXSMap<String,Integer> map = utils.getCache("CounterMap");
Integer origValue = map.get("K");
Map<String, Integer> origValues = new HashMap<String,Integer>();
origValues.put("K", origValue);
Map<String, Integer> newValues = new HashMap<String,Integer>();
newValues.put("K", new Integer(origValue + 1));
Map<String, Boolean> r = map.cond_putAll(origValues, newValues);
If(r.get("K"))
        System.out.println("Value is updated");
else
        System.out.println("Value is unchanged");
```

This shows code that will attempt to atomically increment a counter that is stored in a Map as an integer with a String key. The code first gets the current value for the counter keyed by "K". It then creates a HashMap with the key and current value. It then make a second Map with the key and the desired new value. The new value is simply the counter + 1. The application then calls cond_putAll providing both Maps. This method will then update the values for the entries ONLY if the current values are the same as those in the first Map. This is basically optimistic locking. No lock is kept on the counter. If another client were to update the counter in between the get and the cond_putAll then this client would receive a FALSE for the key in the returned Map.
This code can also be used to only insert items if they don't exist but do nothing if the key already exists. Simply pass a null value for the original value to get this behavior.

Working with Filters

WXSUtils provides a Filter capability similar to the JPA criteria API. This implementation allows an application to construct a filter and then use that filter to test if an object matches that filter or not. Filters can be used several ways with WXSUtils.

1. This approach simply tests if an Object matches the filter or not.
2. This approach obtains records from a WXS index and then filters the records matching the index using the filter.
3. List and Sets use Filters to only return elements that match a filter.

Filter architecture and approach

The Filter code was designed to be independent of the WXS product and WXSUtils library as much as possible. It can be used with just POJOs[2] if desired. Basically, a Filter executes a Boolean expression against an object. The attributes of the object can be extracted and tested against Boolean conditions such as less than, equal to and so on. The Filter framework can be used with any object. The object could be a normal Java object where attributes are obtained using reflection but it could also be a JSON object where attributes are obtained using a json framework instead of reflection. The Filter framework can work with ANY type through the ValuePath plugin mechanism. We will first look at using Filters with POJOs.

ValuePath implementations, extracting attributes from objects

The first thing to learn is how to extract attributes from an object so we can build an expression to test objects. The issue here is that we can create the filter before we even have an instance of the object to test. So, we need a way to describe the attributes on the object that we want to extract for testing. The ValuePath plugins allow us to do this. There are two implementations provided with the current wxsutils (V2.1) library:

```
ValuePath fn = new PojoPropertyPath("FirstName"); // call ‘getFirstName()’
ValuePath sn = new PojoFieldPath("surname"); // access the surname field
```

These two examples show how to construct ValuePaths to extract attributes from a POJO by either calling a property getter method or directly accessing a field on the POJO. Either is fine. Once we have defined some ValuePaths for the attributes that we need then building the filter is not too difficult.

```
FilterBuilder fb = new FilterBuilder();
Filter f = fb.and(fb.eq(fn, "Joe"), fb.eq(sn, "Blogg"));

Assert.assertEquals(f.filter(c), true);
```

This snippet shows a Filter to check if the firstname is equal to “Joe” and the surname is equal to “Blogg”. Any POJO that has a getFirstName() method AND a field called surname can be tested with this filter. It’s not coupled with a specific type. You can see the filter method takes any object and return true if the Filter evaluated

[2] Plain Old Java Objects = POJO

to true. You can implement your own implementations of ValuePath if you want specialized value extractors from your own objects, whether they are XML/JSON or some other store implementation.
The FilterBuilder class has all the normal comparison methods such as:

- and/or
- not
- gt/gte, lt/lte, eq/neq

Filters are serializable and so can easily be stored in Maps, files or serialized across the network to the grid. The only condition is the ValuePath implementations and wxsutils jar must be on the classpath in order to inflate the query.
The filter instance has a filter method that can be called with an object to test if the object meets the filter definition. This can be used to quickly unit test Filters if required.

WXSMap and Filters

WXSMap includes methods to use indices and filters together. Applications can ask WXSMap to use an index to find a subset of the entries and then apply a Filter to that subset. The following example uses an index for the surname to find all Persons in a Map whose surname is less than "M" using the index and that filter that subset to only those with a credit limit of under one million.

```
WXSMap<String, Person> map = utils.getCache(PERSON_MAP);

// keep only people with credit limit under a million;
FilterBuilder fb = new FilterBuilder();
ValuePath creditLimitPath = new PojoPropertyPath("CreditLimit");
Double desiredMaxCreditLimit = 1000000.0;
Filter f = fb.lt(creditLimitPath, desiredMaxCreditLimit);

// find all surnames < "M" with credit limit < 1000000
String value = "M";
GridFilteredIndex<String, Person> q = map.lt("surname", value, f);
Map<String, Person> block = q.getNextResult();

while(block != null)
{
        for(Map.Entry<String, Person> e : block.entrySet())
        {
                Person p = e.getValue();
                System.out.println(p.toString());
        }
        block = q.getNextResult();
}
```

The Person object has a getCreditLimit() methods.

Working with Lists

This section shows how to work with Maps that contain a special list type. Each Map entry has a key like a normal map however the value is a List of Values. The application never works directly with this list however. Instead, it works with a high level set of APIs that can manipulate Lists are a high level. Thus, a WXSMapOfLists does not have typical Map operation methods. Instead, it has list manipulation methods in their place.
The application now works with Lists for each key instead of values for each key. The size of the list is bounded by the available memory for a single JVM however. This trade off improves performance greatly as a complete list can be worked with in a single grid operation involving a single grid container. The list is specially implemented however so that the minimum data is replicated when items are added or removed from the list. The whole list is not replicated each time. This improved performance significantly when working with larger lists.

Configuring Lists in the xml files

There is no need to explicitly create Maps for each list. WXSUtils instead uses WXS templates to automatically create Maps for lists. The following templates MUST be added to the application objectgrid.xml files and the map refs added to the deployment.xml.

```
<!-- You need to define these templates for Lists and Sets -->
<backingMap name="LHEAD.*" template="true" lockStrategy="PESSIMISTIC"
copyMode="COPY_TO_BYTES" lockTimeout="20"/>
<backingMap name="LBUCK.*" template="true" lockStrategy="PESSIMISTIC"
copyMode="COPY_TO_BYTES" lockTimeout="20"/>
<backingMap name="LDIRTY.*" template="true" lockStrategy="PESSIMISTIC"
copyMode="COPY_TO_BYTES" lockTimeout="20"/>
<backingMap name="LEVICT.*" template="true" lockStrategy="PESSIMISTIC"
copyMode="COPY_TO_BYTES" lockTimeout="20"/>
<!-- End of List/Set templates -->
```

This is all that's needed no matter which list maps the application uses. Each application list map will have two Maps created for it always. The LHEAD.NAME and LBUCK.NAME. The LDIRTY.NAME Map will be created if dirty sets are used. The LEVICT.NAME will be created if eviction is used.

It's not currently possible to back a ListMap with a backend through a Loader. Eviction policies must never be specified in the objectgrid.xml file for these templates. The list eviction APIs must be used for eviction.

Introduction to list operations

Basically, a list can be thought of as an ordered collection of elements. There is a lef most element and a right most element. Items can be pushed on to the left or right or the list. For example:

```
WXSMapOfLists<String,String> lmap = utils.getCacheOfLists(“map”);
lmap.lpush(“a”, “3”); // list is now [3]
lmap.lpush(“a”, “2”); // now [2,3]
lmap.lpush(“a”, “1”); // now [1,2,3]
```

This will create a list for the key “A” with the elements [1,2,3]. The items are pushed on the left hand side each time. The same list can be created using rpush also.

```
WXSMapOfLists<String,String> lmap = utils.getCacheOfLists(“map”);
lmap.rpush(“a”, “1”); // now [1]
lmap.rpush(“a”, “2”); // now [1,2]
lmap.rpush(“a”, “3”); // now [1,2,3]
```

Except, the order is changed. This still results in the list [1,2,3]. The elements are added or pushed on the right hand side of the list each time.

The opposite of pushing is popping. Items can be popped from a list from either the left or right hand side also. For example

```
String v1 = lmap.lpop(“a”); // pops “1” leaving [2,3]
String v2 = lmap.lpop(“a”); // pops “2” leaving [3]
String v3 = lmap.lpop(“a”); // pops “3” leaving []
```

Similarly, the rpop operation can be used as follows

```
String v1 = lmap.rpop(“a”); // pops 3 leaving [1,2]
String v2 = lmap.rpop(“a”); // pops 2 leaving [1]
String v3 = lmap.rpop(“a”); // pops 1 leaving []
```

Typically, applications use lists as FIFO queues. Producers push items on one side of the list and consumers pop items from the other side. Applications can also use lists as LIFO queues by popping from the same side the elements are pushed to.

When the last element is popped from a list then the list is removed from memory. If another item is pushed to it later then the list is recreated. Thus, applications don’t usually create or destroy lists. It happens automatically.

Multiple threads can pop elements at the same time with no issues. Each element is guaranteed to be popped to only one thread. That is, no two threads will get the same element returned from pop.

When the list is empty then a null value is returned from lpop or rpop

A consumer can remove every element in the list at once using the popAll method. Again, if multiple threads call popAll at once then only one will get all the current elements. However, if the consuming threads are running in a loop then when items are pushed on to the list, the next consumer thread to call popAll will obtain ALL the current elements pushed since the last popAll call.

The rtrim method can be used to trim a list by removing elements from the right hand side until the list length is no more than the specified size.

The llen method returns the current size of the list but the isEmpty method is typically faster if the application just wants to check if a list is empty.

The lrange method allows an application to retrieve the current contents of a subset of the list. The elements are returned from the left index to the right index inclusive. If the list doesn't have enough elements then the return list will be shorter. Remember, there is no lock for lists. It's entirely possible to check the length of a list and then call lrange only to find that another thread has called popAll in between the length check and lrange calls.

The remove method can be thought of as an optimized popAll. It simply removes the list and does not return the current elements. The remove method MUST not be used when using dirty sets as discussed later. The lpop/rpop or popAll must be used to remove elements.

Conditional push methods

These methods will only push a value if not existing member of the list matches a specified filter. This is useful to prevent duplicate items being added to a list for example. If a user is submitting work that translates in to an item on a list then a cpush method can be used to only do a new push if not existing element already does the same thing.

Summary

Here are the basic methods for WXSMapOfLists:

```
public interface WXSMapOfLists<K,V> {
	public void rtrim(K key, int size);

	public void lpush(K key, V value);
	public void lpush(K key, List<V> values);
	public void lpush(Map<K, List<V>> items);
	public void lcpush(K key, V value, Filter condition);

	public void lpush(K key, V value, K dirtySet);
	public void lpush(K key, List<V> values, K dirtySet);
	public void lpush(Map<K, List<V>> items, K dirtySet);
	public void lcpush(K key, V value, Filter condition, K dirtyKey);

	public void rpush(Map<K, List<V>> items);
	public void rpush(K key, V value);
	public void rpush(K key, List<V> values);
	public void rcpush(K key, V value, Filter condition);

	public void rpush(Map<K, List<V>> items, K dirtySet);
	public void rpush(K key, V value, K dirtySet);
	public void rpush(K key, List<V> values, K dirtySet);
	public void rcpush(K key, V value, Filter condition, K dirtyKey);

	public V lpop(K key);
	public V lpop(K key, K dirtyKey);
	public ArrayList<V> popAll(K key);
	public ArrayList<V> popAll(K key, K dirtyKey);
	public V rpop(K key);
	public V rpop(K key, K dirtyKey);

	public void remove(K key);

	public ArrayList<V> lrange(K key, int low, int high);
	public ArrayList<V> lrange(K key, int low, int high, Filter... filter);

	public int llen(K key);
	public boolean isEmpty(K key);
}
```

Dirty sets will be discussed later as an advanced topic.

Working with sets

WXSUtils also supports Maps whose values are Sets of values. This has also proven to be a common data structure used by customers. Here, each key in a Map has a set of values. Again, like lists, the application doesn't work directly with the map values themselves, i.e. the sets. Instead, WXSUtils provides higher level operations to make working with sets much easier and scalable.
Like lists, the implementation assumes a Set of values will always be small enough to be stored within a single container JVM. The set for a specific key is stored in exactly one partition. This is an implementation choice for higher speed.

Configuring xml files for sets

The application add a single Map named after the set to their objectgrid.xml and deployment.xml files. No eviction policy is supported on this map. No Loader is supported. Typically the map definition should look like this:

```
<backingMap name="SetName" lockStrategy="PESSIMISTIC"  copyMode="COPY_TO_BYTES"
lockTimeout="20"/>
```

Each set key used by the application in a given map actually created many entries in the Map. The set contents are stored in the map as lots of sub-sets. It's possible for each application set to have up to 211 entries[3] in the Set Map.

Introduction to Set operations

An application can work with a Map of sets very easily using the following code to obtain a Map of sets:

```
WXSMapOfSets<String, String> set = utils.getMapOfSets("myset");
// we will work with the set for the key "a"
set.add("a", "0"); // set is now {0}
set.add("a", "1"); // set is now {0,1}
set.add("a", "2"); // set is now {0,1,2}

// check the set contains "1"
if(set.contains("a", Contains.ALL, "1")) System.out.println("YES");

// remove the element "1"
set.remove("a", "1")
```

This obtains a WXSMapOfSets where the key is the first type and the values in the set are the second type. We will use a String key and the set will contain strings also in this example. We simply add three elements to the set. Note that we can add the three elements in any order and end up with the same set. We then check if the set contains a specific value and then remove that value.

Sets are automatically destroyed when the last element is removed from the set. Individual buckets of a single set are also removed if they contain no elements.

[3] The actual number depends on the constant SetAddRemoveAgent.NUM_BUCKETS. This is 211 as of V2.1.0

Bulk set operations

The add method can take either an array of values or a comma separated list of values. All values will be added in one operation. The add method looks like this:

```
	public void add(K key, V... value);
```

Examples of using it would be:

```
set.add(“a”, “V0”, “V1”, “V2”);
// or
String[] values = {“V0”, “V1”, “V2”};
set.add(“a”, values);
```

The contains method can check if ANY or ALL of the passed values are present in the current set. The contains method looks like this:

```
	public boolean contains(K key, Contains op, V... values);
```

Examples of using it would be:

```
// the set “a” currently has the elements {0,1,2,3,4}
boolean rc = set.contains(“a”, Contains.ALL, “1”, “2”); // returns true
String[] values = {“1”, “2”}
rc = set.contains(“a”, Contains.ALL, values); // same thing

rc = set.contains(“a”, Contains.ANY, “1”, “10”, “11”); // returns true
```

Contains is an enum with two values, ALL or ANY. ALL indicates all values must be present in the set. ANY indicates that any of the values must be present.

Advanced set operations

Get with Filter

The get method can return the current elements of a set in no particular order. The elements can be filtered to include only specific elements if required using a Filter. If a filter is specified then only the elements that the filter allows will be returned to the client.

```
	public Set<V> get(K key [, Filter filter]);
```

Add and flush

The addAndFlush method looks like this:

```
	public Set<V> addAndFlush(K key, int maxSize, V... values);
```

This method will first check if the set is currently at least maxSize values. If it is not then the additional value are added to the current set and an empty set is returned. If the current set has at least maxSize values then the set is replaced with just the additional values and the existing set is returned.

Differences between sets and lists

Note that sets do not maintain order. Items can be added or removed randomly in effect but a set cannot contain duplicate values. Lists only allow items to be added and removed from the sides of the list, never the middle. Lists are ordered. Lists can also contain duplicate elements.

Jobs: How to easily visit every partition

Applications some times need a way to visit data in partitions iteratively. An example of such a scenario would be an application running logic on data in the grid that produces a lot of data. The application could run this logic in parallel and have the results pushed all at once to the application but this could easily overwhelm the application. Imagine a grid with 200 or 300GB of data pushing 50GB of results in parallel back to a single client. The client would have major issues.
An alternative would be for the client to run the logic iteratively, partition by partition, and receive the results for the last 'chunk' of data processed. This avoids overwhelming the client as the results are pulled back as a rate it can keep up with.

Looping over grid data

The easiest way to think about this framework is to imagine two nested loops under the control of the client. The first outer loop simply iterates over each partition. If there are 23 partitions then this outer loop will repeat 23 times. Thus, the outer loop is to ensure every partition is visited once. The inner loop iterates over all the data within a single partition. If a single partition has 1m entries then if the application logic works with 100k entries at a time then this inner loop will run ten times. Each time the inner loop runs, the application logic will run on the container for the current partition. It would process the next 100k entries and then return the results. The results are then returned to the application and when the application asks for the next set of results, the business logic is again executed to fetch the next 100k entries results. If no more entries remain in the current partition then the outer loop advances to the next partition and the application business logic starts with the first 100k entries in that partition.
This logic doesn't look like two loops to the application client. Instead, the client just calls a method to get the next set of results. It keeps calling this method until it returns null that means every partition has now been visited.

Job Framework details

The framework was designed as a job framework. An application could create a job that would be executed against the grid in the fashion described above. The job would run sequentially against chunks of data in the grid partition by partition. The first example of using this framework is a simple piece of code to merely visit every partition in the grid. The 'job' will execute once per partition and log a message proving that it visited a specific partition.
The job implementation has two parts. The main class implements the job proper. Its purpose is to create subjobs within partitions and to control when the framework needs to move to the next partition. The main class runs completely within the client JVM.
The sub job runs completely within the container JVM of the partition currently being visited.
The framework uses the job implementation to create a subjob for each visit to the grid for processing another chunk of data. The sub job returns an object to the

framework. This object has the result as well as any information needed by the job implementation to determine whether the current partition is exhausted or the next point to continue processing within that partition. The result is stripped from the returned object and given to the client application every time a sub job completes.

Visit Every Partition Job Example

This first example just creates a sub job for every partition essentially. The main job implementation looks like this.
First, the class implements an interface with two type parameters. The first type is the return type of the sub job. The second is the return type of the application data returned by the sub job.

```
public class PingAllPartitionsJob implements MultipartTask<Boolean, Boolean>
{
	static Logger logger = Logger.getLogger(PingAllPartitionsJob.class.getName());
```

The extractResult method is called by the JobExecutor framework to extract the application data from the sub job return value. They are the same in this simple example.

```
	public Boolean extractResult(Boolean rawRC)
	{
		return rawRC;
	}
```

This is the JobExecutor instance used by this job. The constructor takes an ObjectGrid client reference and then constructs the JobExecutor for this job. The JobExecutor takes the client and a reference to the MultipartTask, i.e. the job as parameters.

```
	JobExecutor<Boolean, Boolean> je;

	public PingAllPartitionsJob(ObjectGrid ogclient)
	{
		je = new JobExecutor<Boolean, Boolean>(ogclient, this);
	}
```

This method is called for each iteration of the inner loop. The JobExecutor moves across partitions and then within each partition, it calls the createTaskForPartition method. It passes a reference to the previous task for this partition of null if this is the first time the subtask will be executed within this partition. The createTaskForPartition method should return a new instance of SinglePartitionTask which will be executed on the container for the current partition. If there is no more work for the current partition then it should return null. A null indicates the JobExecutor needs to move to the next partition. This example simply returns a new PingSinglePartitionTask when it is the first call for a partition and otherwise returns null. This basically results in the PingSinglePartitionTask being executed once per partition.

```
	public SinglePartTask<Boolean, Boolean> createTaskForPartition(
			SinglePartTask<Boolean, Boolean> previousTask) {
		// prevtask is null when called for first time for a partition
		if(previousTask == null)
		{
			PingSinglePartitionTask t = new PingSinglePartitionTask();
			return t;
		}
		else
		{
			return null;
		}
	}
```

This method isn't part of the framework, it's just a helper method to call getNextResult on the JobExecutor.

```
	public Boolean getNextResult()
	{
		return je.getNextResult();
	}
```

Again, this method isn't part of the framework. It just illustrates how to invoke the framework to run a job. You can see that it constructs a Job and then calls the getNextResult method until a null is returned. In this case, getNextResult will return true once for every partition and then null. The getNextResult method has all the logic for the two loops for iterating over data within a partition and across multiple partitions.

```
	static public int visitAllPartitions(ObjectGrid ogClient)
	{
		PingAllPartitionsJob job = new PingAllPartitionsJob(ogClient);
		int count = 0;
		while(job.getNextResult() != null) count++;
		return count;
	}
}
```

Next, we will loop at the worker of the job. The sub job that executes on the container side every iteration of the client driven loops. It looks like this.
The class must implement the SinglePartTask interface. Again, the type parameters are the same as those for MultipartTask. The first type is the result from this class

and the second is the type for the application data extracted each iteration. They are the same in this case.

```
public class PingSinglePartitionTask implements SinglePartTask<Boolean, Boolean>
{
	static Logger logger = Logger.getLogger(PingSinglePartitionTask.class.getName());
	private static final long serialVersionUID = 1722977140374061823L;
```

This class is Serializable and so must have a default constructor.

```
	public PingSinglePartitionTask() {}
```

This method can be called to check if the result of the current sub job is empty and can be skipped. This avoids returning empty results to the application.

```
	public boolean isResultEmpty(Boolean result)
	{
		return false;
	}
```

This is the main method. The process method is executed on the container with the current partition. It is provided with a Session object for this primary shard. It can then execute the business logic and return the result of processing the next chunk of data within this partition.

```
public Boolean process(Session sess)
{
	String aMap = (String)sess.getObjectGrid().getListOfMapNames().iterator().next();
	int partitionId = sess.getObjectGrid().getMap(aMap).getPartitionId();
	logger.log(Level.INFO, "Ping to partition " + partitionId);
	return Boolean.TRUE;
}
```

Run a query over the grid example

This example shows how to execute a simple query over all the data within a grid using the Job framework. There are three classes. The job itself is implemented by the GridQuery class. The subjob that runs on the containers is implemented by GridQueryPartitionTask and the result returned for each sub job is implemented by the class GridQueryChunk.
This job will execute a query against data in a grid and pull the results in blocks of no more than ‘limit’ records at a time. The query is executed against each partition in turn and within a partition, several calls may be made with each call pulling at most ‘limit’ records at a time. Once all results have been pulled from a partition then the Job is moved to the next partition and it then continues until all partitions have been queried.

Lets look at the job implementation first. This implements the MultipartTask interface and specifies the usual type parameters. The first is the subjob return type. The second is the client application visible result type. It contains instance variables for the query and the limit on how many records to pull from the grid at once.

```
public class GridQuery implements MultipartTask<GridQueryChunk, ArrayList<Serializable>>
{
        int limit;
        String queryString;
```

Here is the JobExecutor with the types for the container return value and the application return value.

```
        JobExecutor<GridQueryChunk, ArrayList<Serializable>> je;
```

This flag is updated every time a chunk is returned from the subtask. If the current block of records is the maximum size when it's set to true. This is used to indicate whether there are additional blocks of data in the current partition.

```
        boolean lastExtractWasFull = false;
```

This is called when a sub job completes in order to extract the data for the application from the container sub job result. You can see this is where the lastExtractWasFull variable is updated.

```
        public ArrayList<Serializable> extractResult(GridQueryChunk rawRC)
        {
                ArrayList<Serializable> rc = rawRC.result;
                // check if last block was < limit records and if it was then
                // assume there is no more data in this partition
                lastExtractWasFull = (rc.size() == limit);
                return rc;
        }
```

This constructs a GridQuery job. The desired query and block size are the main arguments. It constructs a JobExecutor also.

```
        public GridQuery(ObjectGrid ogclient, String queryString, int limit)
        {
                this.queryString = queryString;
                this.limit = limit;
                je = new JobExecutor<GridQueryChunk, ArrayList<Serializable>>(ogclient, 
this);
        }
```

This is called to create the next container side sub job. The previous sub job instance is passed as an argument. If previousTask is null then this is the first sub job for the partition, so we run the query starting with the first result in this case. Otherwise, we continue the query on the partition starting with the last record returned. If the last result wasn't full then we stop querying this partition and move to the next by returning a null sub-job.

> *There is a bug here where records in the last chunk are ignored by the query. It should do one more call before returning null.*

```
	public SinglePartTask<GridQueryChunk, ArrayList<Serializable>>
createTaskForPartition(SinglePartTask<GridQueryChunk, ArrayList<Serializable>>
previousTask)
	{
		// prevtask is null when called for first time for a partition
		if(previousTask == null)
		{
			GridQueryPartitionTask qpa = new GridQueryPartitionTask();
			qpa.limit = limit;
			qpa.queryString = queryString;
			qpa.offset = 0;
			return qpa;
		}
		else
		{
			// if last SinglePartTask wasn't the last one then just get the
next
			// block of limit records
			GridQueryPartitionTask qpa = (GridQueryPartitionTask)previousTask;
			if(lastExtractWasFull)
			{
				qpa.offset += qpa.limit;
				return qpa;
			}
			else
				// otherwise we got all the records in this partition
				return null;
		}
	}
```

This is a helper method to return the next block of query results.

```
	public ArrayList<Serializable> getNextResult()
	{
		return je.getNextResult();
	}
}
```

This is the sub job implementation. It executes the query starting and ignores results prior to offset in the result set. It returns the matching records from offset..offset + limit -1.

```
public class GridQueryPartitionTask implements SinglePartTask<GridQueryChunk,
ArrayList<Serializable>>
{
	private static final long serialVersionUID = 2643446835456688514L;

	int offset;
	int limit;
	String queryString;

	public GridQueryChunk process(Session sess)
	{
		ObjectQuery q = sess.createObjectQuery(queryString);
		q.setFirstResult(offset);
		q.setMaxResults(limit);

		Iterator<Serializable> rc = q.getResultIterator();
		ArrayList<Serializable> list = new ArrayList<Serializable>();
		while(rc.hasNext())
		{
			list.add(rc.next());
		}
		GridQueryChunk result = new GridQueryChunk();
		result.result = list;
		return result;
	}

	public boolean isResultEmpty(ArrayList<Serializable> result) {
		return result.isEmpty();
	}
}
```

Lists and dirty Sets

Applications some times need to store lists in a grid and then consume the elements of those lists. If the application uses a static set of keys for the lists then a set of consumers could be written using one thread per key or assigned several keys to a thread to pop elements from those lists. However, it's usually not practical to know all the keys for the lists in advance. WXSUtils provides a way to know ALL the list keys that currently have elements in a simple fashion. This uses the dirty set abstraction. Simply put, the dirty set keeps the keys of ALL lists with elements. There is a single logical dirty set in a grid. The application can specify the name of this dirty set when pushing or popping elements to and from the lists. Consumers can easily retrieve the keys for non empty lists in the grid.

Specifying a dirty set when pushing elements in a list

The lpush and rpush methods take an optional dirty set key as a final parameter. Specifying a dirty set key means WXSUtils will make sure the list key is present in the dirty set. The dirty set key is by convention the same type as the list key.

Specifying a dirty set when popping elements from a list

The dirty set key must also be specified when a consumer does a rpop/lpop or popAll from a list. A consumer can simply use the dirty set to find a key of a non-empty list and then pop elements from it. Note, it's possible that another thread will have consumed the elements first. Consumers are competing for the elements of the lists. WXSUtils guarantees that only one consumer will obtain a specific list element (lpop/rpop) or elements if popAll is used. Consumers must therefore be prepared to obtain a list key from the dirty set that will not actually contain any elements simply because another consumer got them first. List keys appear in the dirty set whenever a lpush/rpush is executed. Any lpop/rpop/popAll that results in the list being empty causes the list key to be removed from the dirty set immediately.

Interrogating dirty sets

A dirty set is not implemented using the normal Set implementation. Normal sets have unique keys in the grid. Normal sets are stored in the partition their key hashes to. Dirty sets are a set with a global name that exists in EVERY partition. This is very different from a normal set. The global name is the dirty key used by the application. **This means the normal Set APIs CANNOT be used from a client with dirty sets**. Instead, a special API is provided to allow client applications to obtain dirty list keys. Each dirty set has a set key and this set is stored in every partition with the same name as noted earlier. The easiest way to obtain the elements of this global dirty set is with a job that visits each partition and returns the current contents of this set. The FetchJobsFromAllDirtyListsJob class implements the MultipartTask pattern to accomplish this. It visits every partition and returns sets of dirty list keys for a given dirty set key.

```
FetchJobsFromAllDirtyListsJob<K, V> job = new FetchJobsFromAllDirtyListsJob<K,
V>(ogClient, listHeadMapName, dirtyKey);

while(true)
{
        // get the next block of dirty list keys
        // this array r is sorted in first dirtied order
        ArrayList<DirtyKey<V>> r = job.getNextResult();
        // when r is null then the whole grid has been checked
        if(r != null)
        {
                // application should work with all lists
                // whose key is in r HERE.
                for(DirtyKey<V> dk : r)
                {
                        V listKey = dk.getValue();
                        // process the items in the list listKey
                }
        }
        else
                break;
}
```

This code snippet should be used in the application to obtain the lists with elements using the specified dirty set key. The list head map name can be obtained using the helper method:

```
String listHeadMapName = WXSMapOfBigListsImpl.getListHeadMapName(listName);
```

This method will likely be changed for the final V2.1 release however to a method on WXSMapOfSets.

Concurrent processing of lists in the dirty set

It is normal for an application to use several threads in several JVMs to process lists with elements in them. These threads will compete for lists using the above code. It's possible that multiple threads will receive the same list keys. However, when the threads attempt to do a pop or popAll operation on those lists, the list elements will be given to only one consumer thread. Thus, it's possible that when a consumer thread attempts to obtain the elements of a list, there may be none. This is not a bug, it just means another consumer has processed that lists items in the time since the list key was obtained.

Multiple dirty keys for advanced applications

Each List Map has an associated automatically created dirty set Map associated with it. The dirty sets are stored under the dirty key in this dirty set map. Simple applications can use a single dirty key. This will group all lists together in a single dirty set across the grid.
However, sometimes it's desirable to split the lists with elements in to different groups. For example, some lists may need to be consumed at a higher priority than others. Lists with the same priority can share a commonly named dirty set, for example, SILVER, GOLD or PLATINUM. Meta-data about each list can determine which dirty key should be used on push and pop operations on the list. The consumer logic can do a weighted round robin visit of the dirty sets. For example something simple like this could be a template for a consumer thread:

```
counter = 0;
while(true)
{
        processAllPlatinumLists();
        if(counter % 2 == 0)
                processAllGoldLists();
        if(counter % 3 == 0)
                processAllBronzeLists();
        counter++;
}
```

This would process platinum every loop. Gold every other iteration and Bronze every third iteration. There are many approaches to implementing this. This is just one.

It's important to remember however that a given list key can be used with a single dirty set key at once. A list is only removed from a dirty set when a pop or popall operation is performed specifying that dirty key AND the pop/popAll operation results in an empty list.

As an example, lets suppose the list key "A" is being used with the dirty key "BRONZE". A push operation is executed using this dirty key so the list "A" is added to the dirty set "BRONZE". Next, the application decides list A is now GOLD. Subsequent pushes and pops occur using the "GOLD" dirty key. The list "A" will never be removed from the BRONZE dirty set in this case as a pop resulting in an empty list never occurred using a BRONZE dirty key. Thus, applications using multiple dirty sets need to be careful if a list can be 'promoted' to a different dirty set[4]. Even if a migrateDirtyKey method was provided, given list pushes can occur from any JVM in the application cluster, application level coordination would be needed to coordinate the transition of a list to another dirty key. Hence, the decision to allow dirty key transitions should not be taken lightly.

[4] There is likely a need for a migrateDirtySet method that moves a list from one dirty set to another here. Something like 'void migrateDirtySet(K listKey, K oldDirtyKey, K newDirtyKey)'.