Data Virtualization in WPF and beyond

Introduction

How do you show a 100,000-item list in WPF? Anyone who tried to deal with such a volume of information in a WPF client knows that it takes some careful development in order to make it work well.

Getting the data from where it is (a remote service, a database) to where it needs to be (your client) is one part of the problem. Getting WPF controls to display it efficiently is another part. This is especially true for controls deriving from `ItemsControl` like `ListView` and the newly released `DataGrid`, since these controls are likely to be served large data sets.

One can question the usefulness of displaying hundreds of thousands of rows in a `ListView`. There is, however, always one good reason: *the customer requests it*. And the customer is king, even if the reasoning behind the request is slightly flawed. So, faced with this challenge, what can we do as WPF developers to make both the coding and user experience as painless as possible?

As of .NET 3.5SP1, this is what you can do today to improve performance in `ItemsControl` and derivatives:

- Make the number of UI elements to be created proportional to what is visible on screen using `VirtualizingStackPanel.IsVirtualizing="True"`.
- Have the framework recycle item containers instead of (re)creating them each time, by setting `VirtualizingStackPanel.VirtualizationMode="Recycling"`.
- Defer scrolling while the scrollbar is in action by using `ScrollViewer.IsDeferredScrollingEnabled="True"`. Note that this only improves *perceived* performance, by waiting until the user releases the scrollbar thumb to update the content. However, we will see that it also improves actual performance in the scenarios described below.

All these things take care of the user interface side of the equation. Sadly, nothing in WPF takes care of the data side. *Data virtualization* is on the roadmap for a future release of WPF, but will not be available in the upcoming .NET 4.0, according to Samantha MSFT (http://www.codeplex.com/wpf/Thread/View.aspx?ThreadId=40531).

All is not lost, however. I will show you various ways to have your favorite `ItemsControl` scroll through hundreds of thousands, even millions of items with little effort. Of course, every solution has a price tag, but for most situations it will be acceptable. Promised!

My “solutions” for data virtualization in WPF relies on two key insights and two usage assumptions. The two key insights are:

1. It is possible to automatically construct for an instance of any type T an equivalent lightweight object which, at least for WPF's binding engine, is indistinguishable from T in most binding scenarios involving binding to properties of T.
2. `ItemsControl`'s access patterns for its item source are highly predictable and need at any time only a fraction of the entire data set. The size of this data set is proportional to the number of visible rows, not to the total number of rows in the data set.

Two approaches are derived from these two key insights: the *item virtualization* approach, where individual objects are loaded on demand, and the *collection virtualization* approach, where the entire data set is virtualized. These two approaches virtually (pun intended) split this article in 2 parts.

The usage assumptions are:

1. In the presence of a large number of items, the users will not look at each and every one of them *at the same time*.
2. Scenarios involving a large number of items are predominantly read-only. If there's any editing to be done, it will not take place in the `ItemsControl` holding the large data set.

If usage assumption 1 is valid, we only need to load what the user needs to see. This assumption is already exploited by `VirtualizingStackPanel`'s `IsVirtualizing` and `VirtualizationMode` modes, but it's valid for the data side of the equation as well. Therefore, we can concentrate on techniques that load small amounts of data efficiently.

If usage assumption 2 is valid, we can ignore scenarios where users start editing large data sets in-place. In-place editing with all the bells and whistles (cancellable, transaction safe) has its own set of problems and solutions that is outside the scope of this article.

Scenario 0: Setting the stage

To set the stage for what is to come, consider this simple type:

```
class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(int id)
    {
        Id = id;
        FirstName = String.Format("FirstName{0}", id);
        LastName = String.Format("LastName{0}", id);
    }
}
```

Suppose millions of instances of that type exist somewhere, and are obtained through some loading mechanism. This mechanism will be given some unique id, and returns the corresponding instance of `Person`.

The accompanying code (http://home.scarlet.be/thehive/DataVirtualizationArticleCode.zip) has a demo program which constructs lists of `Person` instances using different scenarios, and shows them in an ordinary `ListView`.

The relevant markup for the `ListView` in the demo program is defined as follows:

```
<ListView Name="TheView" VirtualizingStackPanel.IsVirtualizing="True"
VirtualizingStackPanel.VirtualizationMode="Recycling"
ScrollViewer.IsDeferredScrollingEnabled="True">
    <ListView.View>
        <GridView>
            <GridViewColumn Width="60" DisplayMemberBinding="{Binding
Id}" Header="Id" />
            <GridViewColumn Width="120" DisplayMemberBinding="{Binding
FirstName}" Header="First Name" />
            <GridViewColumn Width="120" DisplayMemberBinding="{Binding
LastName}" Header="Last Name" />
        </GridView>
    </ListView.View>
</ListView>
```

Note that the `ListView` already implements all the optimizations outlined in the introduction. For the remainder of this article, this markup will not change. I.e. all the solutions presented here will work with the existing markup transparently.

Now we want to simulate the fact that instances of `Person` may be expensive to construct. For example, they need to be obtained through some remote service (slow), or they may already be present in some local cache (fast). Therefore, accessing an instance has a significant but unpredictable overhead. Any data virtualization solution should take this into account.

To simulate this unpredictable overhead, the demo program introduces a property called `m_CreationOverhead`, and implements the loading mechanism for `Person` objects is as follows:

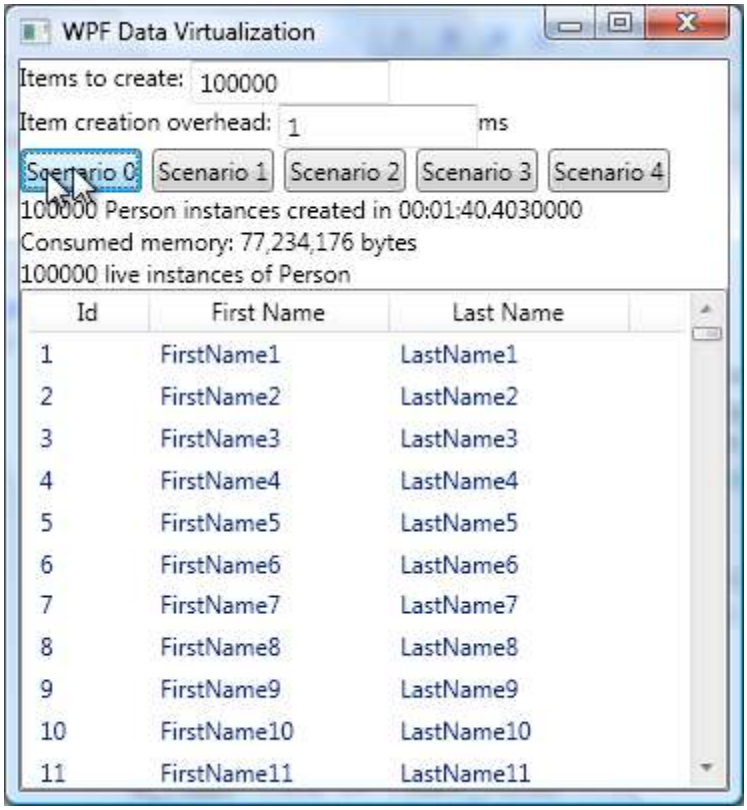
```
private Person LoadPerson(int id)
{
    Thread.Sleep(Math.Max(1, m_Random.Next(m_CreationOverhead)));     //
simulate expensive operation
    return new Person(id);
}
```

Note the `Thread.Sleep`, which will introduce a random delay between 1 and `m_CreationOverhead` milliseconds before a new instance is returned, making calls to create new instances expensive at will, but unpredictable.

Suppose we want to populate the view with 100,000 instances of Person. The obvious way is to call the following function with `itemCount` set to 100,000:

```
private void Scenario0(int itemCount)
{
    var list = new List<Person>(itemCount);
    for (int i = 0; i < itemCount; ++i)
        list.Add(LoadPerson(i));
    TheView.ItemsSource = list;
}
```

You can try this if you want, by running the demo program and clicking on the "Scenario 0" button. Please be patient. The demo program will appear to hang for a bit less than 2 minutes before you will this result:



Since each object needs at least 1 millisecond to instantiate, by the time all 100,000 instances are created, at least 1 minute and 40 seconds will have elapsed. And of course, all your live instances will be in memory. Let's see if we can improve on this horrible performance.

Introducing DataRefBase<T>

Instead of loading each and every one of those 100,000 instances in memory, we should find a way to delay the creation of `Person` instances until the time the WPF binding engine really needs them. This should make the list construction faster, since only about 11 out of the 100,000 `Person` instances are

actually visible in the previous screenshot. Therefore only about that much instancing is needed. All the rest is overhead.

It's easy enough to develop a custom type, say `PersonRef`, with exactly the same properties as `Person`, but which delays loading the corresponding `Person` instance until a property value is really needed. As far as WPF data binding is concerned, different types are OK since matching property names are sufficient to make the existing markup work.

But this solution is fragile (because `PersonRef` needs to evolve together with `Person`) and not reusable (because it applies to the `Person` type only). We can solve the fragility problem by "programming to interfaces", but this may not be feasible for pre-existing types. And it doesn't make the solution more reusable.

Meet our new best friend, `ICustomTypeDescriptor`. This is an interface that supplies dynamic custom type information for an object. The WPF binding engine checks if an object implements this interface and if so uses it to get at the property definitions and bind to them. This is the source of our key insight #1: we can use this interface to implement a type that mimics the properties of another type. Instead of providing *dynamic* custom type information, we'll provide *static* custom type information for an existing type T. Using the same interface, we can intercept property getters and setters and load the "real" instance as soon as these are called.

More specifically, we define a generic base type as follows:

```
public abstract class DataRefBase<T> :
      ICustomTypeDescriptor,
      INotifyPropertyChanged where T : class
{
      …
      protected abstract T Data { get; }
      …
      // implementation of ICustomTypeDescriptor
      …
      // implementation of INotifyPropertyChanged
      …
}
```

The generic parameter `T` must be a reference type, because we need null references to express the fact that an instance is not (yet) available. We can't do this with value types, hence the `T : class` constraint.

The type is abstract, because the exact way an instance of `T` is obtained is of no concern to `DataRefBase<T>` to do its job: the abstract `Data` property will be called when an instance is required, and it's up to the derived classes to implement it. Various implementations will be discussed in the scenarios below.

Apart from the implementation of `ICustomTypeDescriptor` , the type also implements `INotifyPropertyChanged`: as we shall see, not only is this extremely useful but the implementation

is a no-brainer. As a nice side effect, our substitute type will always implement `INotifyPropertyChanged`, even if our generic parameter `T` does not.

`ICustomTypeDescriptor` defines 12 methods. This sounds like a lot of work to implement, but fortunately, 10 of these methods are either trivial to write, or can be delegated to the corresponding method of the static `TypeDescriptor` helper class. The following table will help you to nail down the implementation of these 10 methods. Refer to the code accompanying this article for the specifics.

Method	Implementation
`AttributeCollection GetAttributes()`	`return TypeDescriptor.GetAttributes(typeof(T));`
`string GetClassName()`	`return TypeDescriptor.GetClassName(typeof(T));`
`string GetComponentName()`	`return TypeDescriptor.GetComponentName(this);`
`TypeConverter GetConverter()`	`return TypeDescriptor.GetConverter(typeof(T));`
`EventDescriptor GetDefaultEvent()`	`return TypeDescriptor.GetDefaultEvent(typeof(T));`
`PropertyDescriptor GetDefaultProperty()`	`return TypeDescriptor.GetDefaultProperty(typeof(T));`
`object GetEditor(Type editorBaseType)`	`return TypeDescriptor.GetEditor(typeof(T), editorBaseType);`
`EventDescriptorCollection GetEvents(Attribute[] attributes)`	`return TypeDescriptor.GetEvents(typeof(T), attributes);`
`EventDescriptorCollection GetEvents()`	`return TypeDescriptor.GetEvents(typeof(T));`
`object GetPropertyOwner(PropertyDescriptor pd)`	`return this;`

What's left to implement are 2 overloaded variants of `GetProperties`:

```
PropertyDescriptorCollection GetProperties();
PropertyDescriptorCollection GetProperties(Attribute[] attributes);
```

This is where it gets interesting. Both these methods return a collection of `PropertyDescriptor`s. The latter is an abstract class, which is perfect since we need to implement our own anyway. The story is much the same as with `ICustomTypeDescriptor`: most of `PropertyDescriptor` 's implementation is trivial and won't be shown here. The 2 interesting methods are those called when a property value is obtained or set. These methods just delegate to private `DataRefBase<T>` methods:

```
private class DataRefPropertyDescriptor : PropertyDescriptor
{
    …

    public override object GetValue(object component)
    {
```

```
            return
((DataRefBase<T>)component).GetValue(m_PropertyDescriptor);
        }


        public override void SetValue(object component, object value)
        {
            ((DataRefBase<T>)component).SetValue(this, value);
        }

        …

    }
```

And the corresponding methods of `DataRefBase<T>` are:

```
private void SetValue(DataRefPropertyDescriptor propertyDescriptor,
object value)
{
    var data = Data;
    if (data != null)
    {
        propertyDescriptor.SetValue(data, value);
        NotifyPropertyChanged(propertyDescriptor);
    }
}

private object GetValue(PropertyDescriptor propertyDescriptor)
{
    var data = Data;
    if (data != null)
        return propertyDescriptor.GetValue(data);
    else
        return null;
}
```

Both methods call our abstract `Data` property to get an instance. Both check if the instance they got back is not `null`. Do you know why this is needed? (Hint: it has to do with asynchronous calls) If you don't, wait until scenario 3 is discussed below.

Because all property changes go through `SetValue`, calling `NotifyPropertyChanged` is the only thing we need to do for the implementation of the `INotifyPropertyChanged` interface.

Note that the class has no instance variables and all method implementations are very simple. This matches our goal to have a "lightweight" object. The construction and storage of property descriptor collections is done in the static constructor and shared among all instances:

```
private static readonly IDictionary<PropertyDescriptor,
PropertyDescriptor> m_PropertyMap;
internal static readonly PropertyDescriptorCollection
PropertyDescriptorCollection;

static DataRefBase()
```

```
{
    PropertyDescriptorCollection = new
PropertyDescriptorCollection(null);
    var propertyDescriptorCollection =
TypeDescriptor.GetProperties(typeof(T));
    m_PropertyMap = new Dictionary<PropertyDescriptor,
PropertyDescriptor>(propertyDescriptorCollection.Count);
    foreach (PropertyDescriptor propertyDescriptor in
propertyDescriptorCollection)
    {
        var mappedPropertyDescriptor = new
DataRefPropertyDescriptor(propertyDescriptor);
        m_PropertyMap.Add(propertyDescriptor,
mappedPropertyDescriptor);
        PropertyDescriptorCollection.Add(mappedPropertyDescriptor);
    }
}
```

The abstract type is now complete: we now have a useful base type from which we can derive concrete types start using them in various scenarios.

Scenario 1: Loading on demand with DataRef<TId, T>

Our first concrete type loads an instance of `T` on demand, and keeps a strong reference to it. The implementation is shown here:

```
public class DataRef<TId, T> : DataRefBase<T> where T : class
{
    private readonly TId m_Id;
    private T m_Data;

    private readonly Func<TId, T> Load;

    public DataRef(TId id, Func<TId, T> load)
    {
        m_Id = id;
        Load = load;
    }

    protected override T Data
    {
        get
        {
            if (m_Data == null)
                m_Data = Load(m_Id);
            return m_Data;
        }
    }
}
```

Instances are constructed by giving an `id`, and a `load` function which knows how to load the type, given the `id`.

Note the use of the `Func<TId, T>` delegate. We use delegates instead of another abstract method for two reasons. First, delegates enable better decoupling: we don't have to force the class users to create a separate type just to specify the way it is loaded. Second, delegates have a built-in capability for asynchronous invocation, which will be useful later.

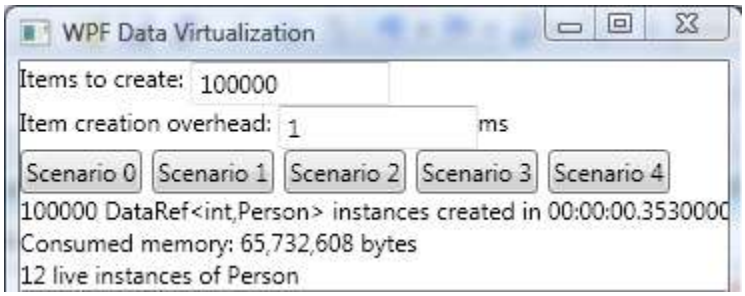
We don't make any assumptions on `T` other than the fact that it must be a reference type. One thing you might do is check if `T` implements `INotifyPropertyChanged`. If it does, link it to the corresponding event of `DataRefBase<T>`. This would enable changes made directly to T to be propagated to our substitute object. Because I didn't need it, this isn't done here.

For our example, populating the view becomes:

```
private void Scenario1(int itemCount)
{
    var list = new List<DataRefBase<Person>>(itemCount);

    for (int i = 1; i <= itemCount; ++i)
        list.Add(new DataRef<int, Person>(i, LoadPerson));
    TheView.ItemsSource = list;
}
```

Let's check how that code performs by clicking the "Scenario 1" button of the demo application. Go ahead: try 100,000 items with 1ms creation overhead. Here is what you should see:



Less than half a second to create 100,000 items and only 12 live instances of Person in memory, without modifying the `ListView` markup at all. Not bad.

DataRef<T>: the price to pay

What we have done in scenario 1 is pure and simple type substitution. True, our `ListView` continues to work, but that's because the WPF binding engine binds property names to those advertised by `ICustomTypeDescriptor.GetProperties()` in our substitute type. As long as only properties are involved in the binding process, this will work.

But consider the following data template (assuming `local` is the namespace where our `Person` type definition lives):

```
<DataTemplate DataType="{x:Type local:Person}">
    <TextBlock Text="{Binding FirstName}" />
```

```
</DataTemplate>
```

… and suppose we had the following column definition in our `ListView` markup:

```
<GridViewColumn Width="100" Header="DataTemplate test">
    <GridViewColumn.CellTemplate>
        <DataTemplate>
            <ContentControl Content="{Binding}" />
        </DataTemplate>
    </GridViewColumn.CellTemplate>
</GridViewColumn>
```

In scenario 0, this column would show the instance's first name. In scenario 1, this would not work.

Data templates with a specific `DataType` will not be considered because we're dealing with a different type here. For looking up data templates, WPF's data binding engine is not smart enough to detect that the bound type implements `ICustomTypeDescriptor` and that the "real" type can be obtained by calling `ICustomTypeDescriptor.GetClassName`. As a result, the match fails and the content shown will be return value of `DataRef<TId, T>.ToString()`, which is ugly.

There is no solution that takes care of the problem transparently, but we can work around the problem by providing a property that returns the actual instance. Note that `DataRefBase<T>` already has such a property: Data. However, we can't use it because (1) it is protected, and (2) you will confuse WPF's binding engine by trying to bind to an actual property on an object implementing `ICustomTypeDescriptor`. If you make the Data property public and try to bind to it, WPF will throw an "ambiguous match" exception.

What we can do, is expose the property through `ICustomTypeDescriptor`. We can do this by adding the following lines in `DataRefBase<T>`'s static constructor:

```
// create an artificial read-only property for the referenced instance
var instancePropertyDescriptor =
TypeDescriptor.CreateProperty(typeof(DataRefBase<T>), "__DATA__",
typeof(T));
var mappedInstancePropertyDescriptor = new
InstancePropertyDescriptor(instancePropertyDescriptor);
m_PropertyMap.Add(instancePropertyDescriptor,
mappedInstancePropertyDescriptor);
PropertyDescriptorCollection.Add(mappedInstancePropertyDescriptor);
```

We need to be careful, however, not to have a naming conflict between the property we're exposing and the existing properties of `T`. Sadly, there's no way to guarantee this. Here, I've named the property `__DATA__`, which violates enough naming conventions to make it an unlikely naming conflict.

The `InstancePropertyDescriptor` is just a specialization of `DataRefPropertyDescriptor`, exposing the instance as read-only:

```
private class InstancePropertyDescriptor : DataRefPropertyDescriptor
{
    public InstancePropertyDescriptor(PropertyDescriptor
propertyDescriptor)
        : base(propertyDescriptor)
    {
    }

    …

    public override object GetValue(object component)
    {
        return ((DataRefBase<T>)component).Data;
    }

    public override Type PropertyType
    {
        get { return typeof(T); }
    }

    …
}
```

The cell template markup becomes:

```
<GridViewColumn.CellTemplate>
    <DataTemplate>
        <ContentControl Content="{Binding __DATA__}" />
    </DataTemplate>
</GridViewColumn.CellTemplate>
```

Now your data template will be invoked correctly. Not a transparent solution, and not even a pretty one, but it works.

Another shortcoming of our solution in scenario 1 becomes apparent when you start to scroll or click on a header to sort the `ListView`. If you scroll down to items that haven't been loaded yet, you will see the "live instances" count increase. Scroll through the entire list, and all 100,000 instances will be loaded.

Sorting is even more dramatic. Sorting is a process that needs all the instances. So by a single click on a header, all 100,000 items will be loaded.

After sorting is complete or the entire list has been paged through, all the 100,000 instances remain in memory, even though the same number of rows is visible. Let's see what we can do about that.

Scenario 2: Loading weak references WeakDataRef<TId, T>

Instead of holding a strong reference to an instance, we can use a weak reference. The code is as follows:

```
public class WeakDataRef<TId,T>: DataRefBase<T> where T: class
{
    private readonly TId m_Id;
    private readonly WeakReference m_Data = new WeakReference(null);
    private readonly Func<TId, T> Load;

    public WeakDataRef(TId id, Func<TId,T> load)
    {
        m_Id = id;
        Load = load;
    }


    protected override T Data
    {
        get
        {
            var data = (T)m_Data.Target;
            if (data == null)
                m_Data.Target = data = Load(m_Id);
            return data;
        }
    }
}
```

The idea is that the garbage collector will clean up the live references when they are not needed anymore.

You can try this in the demo program by clicking on the “scenario 2” button. Performance will be roughly the same, but after garbage collection (which is periodically forced by the demo) the live instances will remain at 0. This is an interesting observation: once an `ItemsControl` has shown a row, the item data is completely discarded. This is fortunate for us, since this behavior enables our solutions to work!

So what’s the price to pay in this scenario? Let’s see: we’re using weak references. We should not confuse this with caching: caching implies policy, and there is no controllable policy here. Exactly which weak references will get garbage collected is the decision of the GC (garbage collector). We should not make any assumptions about the factors the garbage collector takes into account to make that selection.

This needs to be mentioned explicitly, because I’ve seen many projects using weak references as the cornerstone for some kind of caching, based on the mistaken assumption that the garbage collector will use LRU (Least Recently Used) or LFU (Least Frequently Used) or some other cache-friendly strategy to select candidates for collecting. This is not true.

And this brings us back to our price to pay: to sort 100,000 weak-referenced elements will take a very, very long time. If you’re working on battery power, make sure you have a full charge. If, during the sorting process, you place a breakpoint on:

```
m_Data.Target = data = Load(m_Id);
```

… you will notice it will be hit much more than 100,000 times, even if you have ample memory. This is a sign that the GC collects items on its own terms, completely oblivious of the fact that the sorting algorithm will need O(nlogn) comparison operations and that it would be better for all instances to remain in memory for that time. Any half-decent caching policy would handle this correctly.

There's little known about the GC other than the fact that it is generational. My educated guess would be that most of the weakly-referenced instances are short-lived enough to remain in Gen0, and never promoted to older generations. When Gen0 is full, these instances are collected even though the sorting algorithm will need them again soon. But that's just a guess.

We'll see how to "improve" the sorting process in scenario 4. But there's another issue we can handle right now. Both our solutions in scenarios 1 and 2 suffer from the same problem: they block the entire thread while waiting for their instance to be available. This can be illustrated in the demo application by trying a creation overhead of 500ms, instead of 1ms. The creation time won't go up, but you will need to wait about 5 seconds before the `ListView` is completely shown. Scrolling performance will be sluggish. Fortunately, we have deferred scrolling enabled on the `ListView`. If we didn't it would be almost impossible to scroll to the end of the `ListView` using the scrollbar thumb. This is one of those cases where deferred scrolling not only improves perceived performance, but also improves actual performance.

Time to introduce asynchronous loading!

Scenario 3: Loading asynchronously with AsyncDataRef<TId, T>

We can use the fact that `Func<TId, T>` denotes a delegate, and we can call delegates asynchronously. The code is as follows:

```
public class AsyncDataRef<TId, T> : DataRefBase<T> where T : class
{
    private readonly TId m_Id;
    private int m_Loading;
    private readonly Func<TId, T> Load;
    private volatile T m_Data;

    public AsyncDataRef(TId id, Func<TId, T> load)
    {
        m_Id = id;
        Load = load;
    }

    protected override T Data
    {
        get
        {
            if (m_Data != null)
                return m_Data;
            if (Interlocked.Increment(ref m_Loading) == 1)
                if (m_Data == null)
```

```
                    Load.BeginInvoke(m_Id, AsyncLoadCallback, null);
                else
                    Interlocked.Decrement(ref m_Loading);
            else
                Interlocked.Decrement(ref m_Loading);
            return m_Data;
        }
    }


    private void AsyncLoadCallback(IAsyncResult ar)
    {
        m_Data = Load.EndInvoke(ar);
        Interlocked.Decrement(ref m_Loading);
        // when the object is loaded, signal that all the properties
have changed
        NotifyAllPropertiesChanged();
    }
}
```

Two things are worth mentioning about this implementation.

First, asynchronous loading means that we cannot guarantee that the instance will be immediately available. In that case, we return `null`. Now these strange tests for `null` in our base class should make sense: while the instance is loading, the property value returned is always `null`. Once the instance is available, we need to signal to the binding engine it needs to reevaluate the properties. Fortunately, our base type implements `INotifyPropertyChanged`, and contains a method I haven't mentioned until now:

```
protected void NotifyAllPropertiesChanged()
{
    foreach (DataRefPropertyDescriptor propertyDescriptor in
PropertyDescriptorCollection)
        NotifyPropertyChanged(propertyDescriptor);
}
```

This does nothing more than call `NotifyPropertyChanged` for all the properties, and will therefore trigger a reevaluation of all the instances' bindings.

The second thing worth mentioning is that we need to check if a call to `Data` comes in while the instance is still loading. If we called `BeginInvoke` on each occasion, we would load an instance multiple times. This hurts performance. The code explicitly checks if a call to `Load` is in progress and if so, does not call the delegate again. The simplest solution I could come up with (remembering the need for a lightweight object) is using a reference count and Interlocked methods to change it atomically.

You can try the demo program and click on the "scenario 3" button with 100,000 items and a creation overhead of 500ms. It's quite fun to watch the `ListView` responding immediately to your scrolling command, but the content itself will appear incrementally (and in random order).

Once the excitement subsides, we need to pay up. What are the shortcomings of this solution?

One issue is the fact that when a property value is not available, we return `null`. That's OK for the WPF binding engine when binding simple properties, but may not be the correct behavior in more complex scenarios. There's no simple solution. One workaround would be to define a special static instance of T, (whose properties are properly initialized) which is returned while the real instance is loading.

Another issue is sorting: sorting will never work reliably. Sorting needs correct property values for comparing items. However, because we are loading asynchronously, some of these values may be `null` when the algorithm executes the comparison. Although we do implement `INotifyPropertyChanged`, the sorting algorithm used inside the .NET framework doesn't recompare items when their property values change. In fact, it would be quite a challenge to design a sort algorithm that takes property value changes into account while sorting (with minimal comparisons, of course). This is left as an exercise for the reader.

If you're feeling adventurous, you may combine the previous 2 loading techniques to create a AsyncWeakDataRef<TId,T> (or a WeakAsyncDataRef<TId, T>), which loads the instance asynchronously and keeps a weak reference to it. The combination is simple and will not be discussed here.

If you followed me so far and looked at the demo application, you know I still owe you one scenario. But before clicking on that "Scenario 4" button, we have some sleuthing to do. Read on.

Virtualizing item collections

Up till now, all our solutions have one thing in common: *they virtualize the item, not the item collection*. All the scenarios described thus far still create a list containing 100,000 objects. The improvements rely on the fact that we created lightweight objects instead of "real" ones, and delayed loading the "real" objects until they were was actually needed.

But this also meant that for 12 items that were actually in use, the other 99,988 `DataRefBase<T>` items just sit there. They are only needed when their row scrolls into view, or during sorting. Is this really necessary?

Before being capable of answering that question, I needed a little refresher about collection binding, which I would like to share here. Remember that WPF never binds directly to a collection, but to a view. In our case, the view can be obtained by calling:

```
CollectionViewSource.GetDefaultView(TheView.ItemsSource)
```

The return value is a type implementing `ICollectionView`. In all the scenarios above, the actual type is `ListCollectionView`. Don't bother looking it up in the online help: the type is internal and therefore undocumented. It is chosen by `GetDefaultView` because our `ItemsSource` is bound to a `List<T>`, and `List<T>` (despite being a modern generic class) implements the untyped collection interface `System.Collections.IList`. This tells `CollectionViewSource` to use `ListCollectionView`.

Tracing through all the mechanics of item sources and views yielded key insight #2: `ItemsControl`'s access patterns for its item source (actually, its view) is highly predictable. All it needs is a count of the total number of items and "slice" of the actual items. This slice roughly starts at the index of the first visible item, and ends at the last visible item.

Armed with this knowledge, my first attempt at list virtualization was simply to implement a "virtual" variant of `List<T>`. I reasoned that, because I was also implementing `IList`, a `ListCollectionView` instance would be created automatically and I wouldn't have to do anything else. I was right… unfortunately that didn't help performance a bit.

You see, one of the purposes in the life of a view is to leave the original source collection alone. As soon as you're trying to sort, `ListCollectionView` creates a copy of your `ItemsSource` and sorts that internal list instead, thereby defeating the purpose of a virtual list: we still end up with a 100,000 object collection in memory.

This means that any "true" collection virtualization should implement both a virtual list as an associated (virtual) view, and somehow convince to `CollectionViewSource` to pick the latter. It also means that both our list and our view will share much of the same functionality since they will both be virtualized. In fact, we can define a common base type that will be used by both. Say hi to `VirtualListBase<T>`.

Hello VirtualListBase<T>

The base type for both our virtual list and matching view is defined as follows:

```
public abstract class VirtualListBase<T> :
    IList<DataRefBase<T>>,
    IList,
    IItemProperties,
    INotifyPropertyChanged,
    INotifyCollectionChanged where T : class
{
      …
      protected abstract int InternalLoad(T[] data, int startIndex);
      …
}
```

The complete implementation can be found in the article's code. We discuss only the highlights here.

To make our base type usable in both typed and untyped scenarios, it implements both variants of `IList`. Furthermore:

- `IItemProperties` is implemented to provide information about the properties available on the items of the collection. Since these are in fact properties of `T`, we can obtain this information easily from `DataRefBase<T>.PropertyDescriptorCollection`. and repackage it as a read-only collection of `ItemPropertyInfo`s.

- `INotifyPropertyChanged` and `INotifyCollectionChanged` are implemented to track collection changes. This may seem strange for a read-only scenario, but as we shall see shortly, it makes very much sense in many cases.

When deriving from `VirtualListBase<T>`, the only thing you need to provide is a method:

```
int InternalLoad(T[] data, int startIndex)
```

Remember that we defined a `Data` property for `DataRefBase<T>`, to load a single object. Since we are now dealing with a collection, we need a method to load a "slice" (or "page") of that collection. The slice starts at index `startIndex` in the "real" collection, and continues till `startIndex + data.Length - 1` or until the end of the collection, whichever comes first. The method returns the total number of items in the "real" collection.

While it's impossible to make an exhaustive study of all possible ways of obtaining a list of items, most business layers or search services have a way of returning the total number of results, and a "slice" of these results. Web application developers have been using this method for list pagination. We just reuse the same functionality, but expose it to the outside world as a continuous list of items.

Of course, the content of this slice depends on the order of the results and (possibly) a filter, which are usually specified as parameters. This is of no concern to `VirtualListBase<T>` and derived types will have to handle this in some way or another. We'll see how in a moment.

Why do we always return the total number of items in the "real" collection? Isn't once enough? Suppose our search service is stateless and performs the search every time a slice of the results is requested (for some search engines like Lucene.net, this is actually the preferred mode of operation). If other users are modifying the underlying data, the resulting slice may change. Ignoring those changes is bad. Tracking them exhaustively is either very expensive or very complex. We can compromise and check for differences in list size: if the return value of `InternalLoad` is different from the last one, we assume the underlying "real" collection has changed and signal the event to interested parties (via `INotifyCollectionChanged` and `INotifyCollectionChanged`). If there's no difference in count, the list is assumed to be unchanged. This assumption can be wrong, but the risk is somewhat mitigated because we don't keep items in memory for too long (they're virtual!), and their correct value will be obtained when they are retrieved again.

As can be derived from the interfaces, the actual objects the collection holds derive from `DataRefBase<T>`. This is the concrete implementation:

```
private class CachedDataRef : DataRefBase<T>
{
    public readonly VirtualListBase<T> List;
    public readonly int Index;

    public CachedDataRef(int index, VirtualListBase<T> list)
    {
        Index = index;
        List = list;
```

```
    }

    protected override T Data
    {
        get { return List.LoadData(Index); }
    }

    public override int GetHashCode()
    {
        return Index ^ List.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        CachedDataRef other = obj as CachedDataRef;
        return other != null && Index == other.Index && List ==
other.List;
    }
}
```

The actual instance is loaded by calling `VirtualListBase<T>.LoadData(Index)`. This method either finds the instance in a local cache, or ends up calling `InternalLoad` to bring the slice it resides on in cache memory. The local cache replacement policy is LRU, simulated by a move-to-front method. You will find the implementation details in the article code.

Equality and hash code are redefined to make sure we're able to compare items without forcing their instances to load. This proved helpful when using virtual collections in algorithms that compare items at some point.

Implementing VirtualList<T>

With `VirtualListBase<T>` as base type doing most of the heavy lifting, our actual `VirtualList<T>` implementation is very simple. Here is the complete code:

```
public class VirtualList<T> : VirtualListBase<T>,
      ICollectionViewFactory where T : class
{
    internal readonly Func<SortDescriptionCollection,
Predicate<object>, T[], int, int> Load;

    public VirtualList(Func<SortDescriptionCollection,
Predicate<object>, T[], int, int> load, int numCacheBlocks, int
cacheBlockLength)
        : base(numCacheBlocks, cacheBlockLength)
    {
        Load = load;
    }

    public VirtualList(Func<SortDescriptionCollection,
Predicate<object>, T[], int, int> load)
        : base(5, 200)
    {
```

```
            Load = load;
        }

        protected override int InternalLoad(T[] data, int startIndex)
        {
            return Load(null, null, data, startIndex);
        }

        #region ICollectionViewFactory Members

        public ICollectionView CreateView()
        {
            return new VirtualListCollectionView<T>(this);
        }

        #endregion
    }
```

The first thing you'll notice is the implementation of `ICollectionViewFactory`. This is a signal to `CollectionViewSource.GetDefaultView` that we will be providing our own collection view.

The other thing you'll notice is that, just like in the `DataRef<TId, T>` case, I've defined a delegate to load the items. The definition needed to be flexible enough to support sorting and filter predicates. Rather than defining my own parameter structure, I've reused the existing `SortDescriptionCollection` and `Predicate<object>` definitions as parameters, which will be directly usable by the associated view. This is a design decision motivated by the fact that I avoid adding needless layers of complexity for article code. Feel free to make other choices, as long as you remember that the view's sort descriptions and filter predicates should somehow be passed to your method that loads the "real" items, if they are needed at all.

The demo application contains a sample implementation, which takes into account the sort descriptor, but not the filter:

```
    private int Load(SortDescriptionCollection sortDescriptions,
    Predicate<object> filter, Person[] person, int startIndex)
    {
        Thread.Sleep(Math.Max(1, m_Random.Next(m_CreationOverhead)));     //
    simulate expensive operation
        bool isDescending = sortDescriptions != null &&
    sortDescriptions.Count > 0 && sortDescriptions[0].Direction ==
    ListSortDirection.Descending;
        for (int i = 0; startIndex < m_ItemCount && i < person.Length; ++i,
    ++startIndex)
            person[i] = new Person(isDescending ? m_ItemCount - startIndex
    - 1 : startIndex);
        return m_ItemCount;
    }
```

Note that we sort "by id", which is different from the simple property sort used in the default ListCollectionView.

I will not insult the intelligence of the reader by pointing out that this is just an example of sorting. Feel free to plug in your own data shaping operations.

Scenario 4: VirtualListView<T> in action

The implementation of the virtual view associated with `VirtualList<T>` is more involved because of the complexity of the `ICollectionView` interface. I only show selected parts of the constructor and the implementation of `InternalLoad` here, you can pick up the other sordid details from the article code:

```
class VirtualListCollectionView<T> :
      VirtualListBase<T>, ICollectionView where T : class
{
      private readonly VirtualList<T> m_SourceCollection;
      private readonly Func<SortDescriptionCollection,
      Predicate<object>, T[], int, int> Load;

    …
    public VirtualListCollectionView(VirtualList<T> list)
        : base(…)
    {
        Load = list.Load;
        m_SourceCollection = list;
        …
    }

    …
    protected override int InternalLoad(T[] data, int startIndex)
    {
        return Load(m_SortDescriptionCollection, m_Filter, data,
startIndex);
    }

    …
}
```

The important thing to note here are that our view just uses the original list's Load delegate. The list itself is never touched.

It should come as no surprise that the view supports both sorting and filtering.

What about grouping? As far as performance is concerned, grouping is a party pooper, even if you can do it efficiently in your virtual view. If `ItemsControl` detects a GroupStyle, it will silently change the items panel to a `StackPanel`, even if you explicitly set `VirtualizingStackPanel.IsVirtualizing="True"`, effectively turning off UI virtualization behind your back. This causes all your item containers to be realized, and performance suffers accordingly. Data virtualization will not help here. The problem was present on the September 2006 CTP of WPF, and persists until today. The related connect bug 123998, opened on April 4, 2006 (see https://connect.microsoft.com/feedback/ViewFeedback.aspx?FeedbackID=123998&SiteID=212&wa=wsi

gnin1.0) was marked closed (postponed). The only workaround I know of is to avoid using GroupStyle and use your own styles to simulate grouping. However, making this work will when nesting group is difficult. At this time, it looks like that the problem will not be solved in WPF 4.0 either. Worse: WPF 4.0 will not contain any solution for UI virtualization with grouping.

So, no, the virtual view doesn't do grouping. Add this to the price to pay.

But now you can click on that scenario 4 button. Go ahead and create 1,000,000 items. Scroll, sort, do whatever you like. Be amazed at the response time. Just keep in mind the limitations I've discussed!

Virtualization beyond WPF

The previous virtualization techniques have been discussed in the context of WPF. However, the actual techniques are independent of WPF and can be used anywhere you like. After all, the only thing you have to do is to deal with `ICustomTypeDescriptor` directly. The code will be tedious to write, but it will work.

If you hate tediousness (who doesn't?) and want to use the above virtualization tricks outside WPF, you should keep an eye on the upcoming C# 4.0 and its support for dynamic objects (`IDynamicObject`). A `DataRefBase<T>` object can be wrapped in a dynamic object, which will expose the properties in an easy "object.property" format you can write directly, and let the compiler figure out the details. If you have the Visual Studio 2010 CTP, make sure to add the implementation of the `System.Dynamic.DynamicObject` type (which can be found at http://www.gotnet.biz/Blog/post/The-Missing-SystemDynamicDynamicObject-Class.aspx). The implementation of such a wrapper is very simple:

```
public class DynamicDataRef<T> : DynamicObject
{
    private readonly PropertyDescriptorCollection
m_PropertyDescriptorCollection;
    private readonly DataRefBase<T> m_DataRef;

    public DynamicDataRef(DataRefBase<T> dataRef)
    {
        m_DataRef = dataRef;
        m_PropertyDescriptorCollection = dataRef.GetProperties();
    }

    private PropertyDescriptor GetPropDescriptor(string name)
    {
        var propDescriptor = m_PropertyDescriptorCollection[name];
        if (propDescriptor == null)
            throw new ArgumentException(String.Format("Property {0} not
found", name));
        return propDescriptor;
    }

    public override object GetMember(GetMemberAction action)
    {
        return GetPropDescriptor(action.Name).GetValue(m_DataRef);
```

```
        }

        public override void SetMember(SetMemberAction action, object
    value)
        {
            GetPropDescriptor(action.Name).SetValue(m_DataRef, value);
        }
    }
```

If you don't like wrappers and don't mind your code being directly dependent on .NET 4.0, you can simply make `DataRefBase<T>` inherit from `DynamicObject` directly.

Keep in mind, however, that dynamic objects are no substitute for `ICustomTypeDescriptor`: unless things change between the CTP and the final release of .NET 4.0, you still need the latter interface for all existing data binding scenarios.