Community Technology Preview

ComponentOne LiveLinq

By ComponentOne LLC

This manual was produced using *ComponentOne Doc-To-Help.*™

Contents

Getting Started 1

LiveLinq Overview....1
Quick Start: How to Use LiveLinq....2
How to query collections with LiveLinq....2
How to create indexes....5
How to use indexes programmatically....6
How to create a live view....6
How to bind GUI controls to a live view....7
How to use live views in non-GUI code....7

Samples 9

Quick Start Samples....9
Indexing samples....9
Live view samples....9
Sample Applications....10
Query Performance sample application (LiveLinqQueries)....10
Live views sample application (LiveLinqIssueTracker)....10

Programming Guide 13

Query Operators Supported in Live Views....13
Query Expressions Supported in Live Views....14
View Maintenance Mode....16
Live View Performance....16
LiveLinq query performance: logical optimization....17
LiveLinq Query Performance: Tuning Indexing Performance....17
Live Views How To: Create Views Based on Other Views and Create Indexes on Views....21
Live Views How To: Use Live Views to Create Non-GUI Applications as a Set of Declarative Rules Instead of Procedural Code....22

Getting Started

LiveLinq Overview

LiveLinq is a class library that augments the functionality of LINQ in two related directions:

- **It makes LINQ faster**

LiveLinq uses indexing and other optimizations to speed up LINQ queries in memory. Speed gains can be hundreds and even thousands of times on queries with high selectivity conditions. Typical/average gains are lower but still significant, a 10-50 times speedup can be considered typical.

- **It adds support for live views to LINQ**

Live view is a LINQ query result that is kept constantly up-to-date without re-populating it every time its base data changes. This makes LiveLinq extremely useful in common data-binding scenarios where objects are edited and may be filtered in or out of views, have their associated subtotals updated and so on. Using an old jargon, one could say that LINQ queries correspond to snapshots, while LiveLinq view correspond to dynasets. Since live views automatically react to changes, they greatly widen the sphere of declarative programming, not only in data binding and GUI but in many other programming scenarios as well.

The scope of LiveLinq

LiveLinq implements three LINQ varieties:

- LINQ to Objects
- LINQ to XML
- LINQ to DataSet

In other words, its scope is LINQ in memory. The current LiveLinq version does not offer an alternative to LINQ to databases. However, this does not prevent you from using LiveLinq with data retrieved to memory from a database. For example, you can retrieve data from a database to a dataset in memory using ADO.NET and then use LiveLinq to DataSet, or you can retrieve objects from a database using LINQ to SQL or Entity Framework, then operate on that data using LiveLinq and then send changes to the database using again Entity Framework or another framework of your choice.

LiveLinq and LINQ

LiveLinq has the same syntax as the standard LINQ. The only change necessary is to wrap your data source by applying to it an extension method AsIndexed() (for indexing) or AsLive() (for live views).

How the two parts of LiveLinq are related to each other

The two areas of LiveLinq functionality, indexing and live views can interoperate but are independent. It is not necessary to define indexes if you need to create a live view. A view, is populated initially by executing a query, so, if the query can be optimized using indexing, the view will be populated faster. But after its initial population, changes to the view are made using special optimization techniques (known as Incremental View Maintenance) that don't require the user to explicitly define indexes.

Faster LINQ with Indexing

LiveLinq contains an indexing framework that is uses for optimizing query performance. For example, by defining an index by ProductID, we can dramatically speed up queries like

```
from p in products where p.ProductID == 100
```

because it will use the index to access the requested product directly instead of going through the entire large collection in search of a single product.

Same indexes can also be used programmatically, in code, even without LINQ, for various kinds of searches, including range search and others.

Declarative programming with Live Views

Live (real) data binding

LiveLinq adds the concept of ***view*** to LINQ. A view is a query with a resultset that remains live, dynamic after it is initially populated by executing the query. A standard LINQ query is a snapshot, in the sense that its result list does not change when you change the underlying (base) data. So it can't be used for full-featured data binding. A live view is automatically kept in sync with base data, so it enables full data binding to LINQ queries.

Moreover, many views are updatable themselves, you can modify properties in their objects and add and delete objects directly in the view. So, a view can be modifiable in both directions: changes in base data propagate to the view and changes in the view propagate to base data (the latter pursuant to usual conditions of updatability).

This is full-featured two-way data binding.

Reactive = "View-Oriented" = Declarative Programming

Data binding, which is mostly used in GUI, is a very important part of live views functionality, but not the only one. More generally, live views enable a declarative style of programming which we, for the lack of a better term, tentatively call "view-oriented programming". Non-GUI, batch processing code can also be made declarative using live views.

Enabling technology: Incrementality

When a change occurs in base data, live views update themselves in a smart, fast way, not simply re-populate themselves from scratch. The changes are made locally, ***incrementally***, calculating the delta in the view from the delta in the base data. In most cases, it allows to propagate the change from base data to the view very fast. This is a key component of LiveLinq, its enabling technology, based on an area of Computer Science known as Incremental View Maintenance. But the user does not need to do anything to enable this optimization, it is done entirely beneath the hood, transparently to the user.

Quick Start: How to Use LiveLinq

How to query collections with LiveLinq

Using LiveLinq in Visual Studio starts with adding it to your project's References. LiveLinq consists of two assemblies:

C1.LiveLinq.dll
C1.LiveLinq.LiveViews.dll

The first, C1.LiveLinq.dll, is required in all cases, the second is necessary for live view functionality and also for LiveLinq to XML that is based on live views.

So, we can start from adding a reference to C1.LiveLinq.dll. Then let's add the 'using' directives to the source file that will enable us to use LiveLinq in code:

```
using C1.LiveLinq;
using C1.LiveLinq.Indexing;
using C1.LiveLinq.Collections;
```

(In fact, not all of them are always necessary, but let's put all of them there at once for simplicity)

In order to query a collection in LiveLinq, we need to wrap it in an interface IIndexedSource<T> that will tell LiveLinq to take over. Otherwise, standard LINQ will be used. This wrapping is done with a call to the AsIndexed extension method, for example:

```
from c in source.AsIndexed() where c.p == 1 select source
```

However, not every collection can be wrapped this way. For instance, if we try this with a **List<T>** source, we'll get a compilation error. To be usable in LiveLinq, a collection must support change notifications, it must notify LiveLinq when changes are made to its objects and when objects are added to or deleted from the collection. This requirement is satisfied for collections used for data binding, that is, implementing either **IBindingList** (WinForms data binding) or **INotifyCollectionChanged/INotifyPropertyChanged** (WPF data binding)

In particular, AsIndexed() is applicable to ADO.NET collections (**DataTable**, **DataView**) and to LINQ to XML collections, as we will soon see.

Using the built-in collection class IndexedCollection<T> (LiveLinq to Objects)

Suppose first that we don't care what collection we use for our objects. We have a Customer class, something like

```
public class Customer
{
    public string Name { get; set; }
    public string City { get; set; }
}
```

and we rely on LiveLinq to supply the collection class we need.

Then we need look no further than the C1.LiveLinq.Collections.IndexedCollection class that is supplied by LiveLinq and is specifically optimized for LiveLinq use:

```
var customers = new IndexedCollection<Customer>();
customers.Add(cust1);
customers.Add(cust2);
...
var query =
  from c in customers where c.City == "London" select c;
```

Note that we can use simply **customers** instead of **customers.AsIndexed()**. It is because the IndexedCollection<T> class already implements the IIndexedSource<T> interface that LiveLinq needs, there is no need to use the AsIndexed() extension method to wrap it into that interface.

There is an important consideration to be kept in mind using your own classes such as **Customer** above for collection elements. The **Customer** class above is so basic that it does not support property notifications. If you set one of its properties in code, nobody will notice it, including any live views you may create over that collection. So, it is highly recommended (and for live views, necessary) to provide property change notifications in such classes. Property change notifications is a standard .NET feature that is recommended to support for a variety of reasons, LiveLinq just adds another reason to do that. You can support property change notifications in your class by implementing the **INotifyPropertyChanging** and **INotifyPropertyChanged** interfaces, or you can use a LiveLinq facility for that, by deriving your class from IndexableObject and calling OnPropertyChanging/OnPropertyChanged like this:

```
public class Customer
{
    private string _name;
    public string Name
    {
        get {return _name} ;
        set {OnPropertyChanging("Name")"; _name = value; OnPropertyChanged("Name")";}
    {
    private string _city;
    public string City
    {
        get {return _city} ;
        set {OnPropertyChanging("City")"; _city = value; OnPropertyChanged("City")";}
    {
}
```

Using ADO.NET data collections (LiveLinq to DataSet)

ADO.NET **DataTable** can also be used in LiveLinq, for example:

```
CustomersDataTable customers = ...
var query =
  from c in customers.AsIndexed() where c.City == "London" select c;
```

For that, you'll need to add

```
using C1.LiveLinq.AdoNet;
```

to your source file. Then the **AsIndexed**() used will be C1.LiveLinq.AdoNet.AdoNetExtensions.AsIndexed, so you will be using LiveLinq to DataSet, which is specifically optimized for data residing in an ADO.NET DataSet.

Using XML data (LiveLinq to XML)

LiveLinq can query data directly from XML in memory (stored in a LINQ to XML **XDocument** class). It dramatically speeds up LINQ to XML performance by supporting XML indexing (not supported by the regular LINQ to XML).

To use LiveLinq to XML, you need to have a reference to the C1.LiveLinq.LiveViews.dll assembly in your project, and add

```
using C1.LiveLinq.LiveViews.Xml;
```

to your source file.

Then you need to create some live views over your XML data. Unlike LiveLinq to Objects and LiveLinq to DataSet, where you can query data without live views, in LiveLinq to XML queries and live views are always used together. You are using LiveLinq to XML versions of LINQ query operators if you apply them to a live view. Otherwise, they will be the standard, not LiveLinq query operators. To start creating live views, simply apply the extension method AsLive() to an **XDocument**:

```
XDocument doc = …
View<XDocument> docView = doc.AsLive();
```

Once you have a live view, you can use the familiar (from LINQ to XML) query operators **Elements, Descendants** and others (see XmlExtensions Class) as well as all Standard Query Operators Supported in Live Views to define a live view. For example,

```
View<XElement> orders = docView.Descendants("Orders");
```

defines the collection of orders in the document. This collection is live, it is automatically kept up-to-date with the data in the **XDocument**, that can be changed by your program. So you can work with data in this collection in the same way as you would work with any dynamic collection, including ADO.NET DataTable or DataView. In particular, you can create indexes over it and use them for speeding up queries and for fast programmatic searches as shown in other sections of this documentation. It means that LiveLinq to XML makes it possible to work with XML data in memory without losing performance, so it is no longer necessary to create custom collection classes or use ADO.NET or other framework to write a performant application working with XML data, all you need is LINQ to XML with addition of LiveLinq.

Once you have some live views defined over XML data, you can query them. For example,

```
var query = from Order in orders.AsIndexed()
where (string)Order.IndexedAttribute("CustomerID") == "ALFKI"
```

gives you a query for orders of a particular customer.

Note. It is important to distinguish between live views and queries in LiveLinq to XML. Since you always start with a live view, your query will be a live view by default, unless you specify that you don't want a live view. In the example above, we used AsIndexed() to specify that we only need a query, don't need that query to define a live view. Live views extend query functionality, so they can be used instead of queries, you will get the same results. However, live views have performance overhead, so you should avoid using a live view where a simple query would suffice. As a general rule, avoid creating live views and throwing them away after using them just once to get the results. Live views are designed to remain active (hence they are called *live*) and show up-to-date data.

Using bindable collection classes (LiveLinq to Objects)

As was already said above, any collection class with change notifications (in other words, a bindable class, that can serve as a data source in data binding) can be used in LiveLinq queries through the adapter classes IndexedBindingList and IndexedObservableCollection.

Using these out-of-the box classes, you can apply LiveLinq to various preexisting data collection. For example, since the **EntityCollection** and **ObjectResult** classes of the ADO.NET Entity Framework are bindable, you can use ADO.NET Entity Framework data in LiveLinq queries and views.

LiveLinq to Objects: IndexedCollection<T> and other collection classes

Querying general collection classes in LiveLinq is called LiveLinq to Objects, to distinguish it from LiveLinq to DataSet and LiveLinq to XML that query such specific (and quite popular) data sources as DataSet and XML. So, use LiveLinq to DataSet if you need to query data in a DataSet, and LiveLinq to XML if your data is in XML. For other cases, we already saw two options in LiveLinq to Objects:

a. Use IndexedCollection<T> if you don't have preexisting collection classes and rely on LiveLinq to supply them.

b. Use IndexedBindingList or **IndexedObservableCollection** adapter to query preexisting collection classes that support data binding.

And there is also a more advanced option that is not frequently needed:

c. Define your own collection class, usually derived from IndexedCollection<T>, if you want some non-standard functionality.

And finally, an option that can also be considered advanced, but, actually, is not particularly difficult, and allows you to bring virtually any collection into the LiveLinq orbit:

d. You can make any existing collection class **C** usable in LiveLinq provided that it somehow lets you know when changes occur. Then you can create an adapter class implementing the C1.LiveLinq.IObservableSource<T> interface and delegating most of its functionality to that preexisting collection class. Having an IObservableSource<T> implementation, you can apply the AsIndexed() extension method to it, it will return IIndexedSource<T>. LiveLinq extension methods implementing query operators take IIndexedSource<T> as their argument.

How to create indexes

Now that we can query our collections in LiveLinq, we need to create some indexes for them, otherwise LiveLinq won't query them faster than the standard LINQ does.

Suppose we have a collection implementing **IIndexedSource<Customer>**, for example, like this:

```
var customers = new IndexedCollection<Customer>();
```

or like this:

```
var customers = CustomersDataTable.AsIndexed();
```

The IIndexedSource<T> interface has an Indexes collection (which is actually the only thing it has), so we use that collection to create an index like this:

```
var indexByCity = customers.Indexes.Add(x => x.City);
```

That creates an index of customers by city. Once the index is created, it will be automatically maintained on every change made to the **customers** collection. This maintenance comes with a performance cost. The cost is not high, index maintenance is fast and you can usually ignore it as long as you don't have too many indexes, but still, it may become a concern if you modify the collection very intensively.

To avoid heavy index maintenance cost, you can use the **BeginUpdate**/**EndUpdate** methods while making massive changes to a collection that has indexes attached to it, for example, while populating such a collection.

This index will make queries such as

```
from c in customers where c.City == "London"
```

execute very fast because LiveLinq will use the index to go directly to the required items instead of searching for them by traversing the entire collection. Indexes can speed up many queries, not just this simple one, they can also speedup range conditions (with inequalities), joins, grouping, and other LINQ operators. See LiveLinq Query Performance: Tuning Indexing Performance for the full description of what classes of queries benefit from indexes.

Explicitly creating indexes by adding them to the collection as shown above is only needed if you want direct control over their lifetimes and/or direct access to the indexes (Index<T> objects) to use them programmatically (see How to use indexes programmatically). If all you need is to optimize a few LiveLinq queries, you can use the alternative, implicit method of creating indexes, using so called *hints* in LiveLinq queries. A hint .Indexed() is an extension method that can be applied to a property in a query. It does not change the value of that property, it only tells LiveLinq to create an index on that property (if possible). So, instead of creating an index by city explicitly as shown above, you could write the query like this:

```
from c in customers where c.City.Indexed() == "London"
```

That would tell LiveLinq to create an index by city if it has not been already created before. This hint does not affect the value of the property, that is, the value of **c.City.Indexed()** is the same as the value of **c.City**.

How to use indexes programmatically

Indexes are used by LiveLinq to optimize query execution, but they are also accessible for programmatic use. You can use indexes directly in code, calling methods of the Index<T> class to perform a variety of fast searches. This makes LiveLinq indexes useful even outside the framework of LINQ, outside queries.

For example, searching for a particular value can look like this:

```
indexByCity.Find("London")
```

or even like this:

```
indexByCity.FindStartingWith("L")
```

The Index<T> class also has other methods for performing fast searches and fast joins and groupings that can be useful in any code, not just in queries: FindGreater, FindBetween, Join, GroupJoin, and a few others.

Examples of programmatic searches (without LINQ) can be found in the LiveLinq indexing demo, see Query Performance sample application (LiveLinqQueries). In fact, every query that is shown in that demo has an alternative implementation via direct programmatic search in code without LINQ.

How to create a live view

Consider a simple query

```
from a in A where a.p == 1 select a
```

In standard LINQ, the result of this query is a snapshot. The result collection is formed at the time when the query is executed and it does not change after that. If one of its objects changes so it no longer satisfies the condition, that object will not be removed from the result collection. And if an object in **A** changes so it now satisfies the condition, that object will not be added to the result collection. The result of even such a simple query is not live, not dynamic, does not change automatically when the base collection **A** changes, not kept in sync automatically with the base data.

In LiveLinq, if we create a view based on that query, it will be live, dynamic, will change automatically when the base data changes, will be automatically kept in sync with the base data.

All we need to do to make a view out of this query is to use the extension method .AsLive():

```
var view = from a in A.AsLive() where a.p == 1 select a;
```

The AsLive() extension method is the analog of AsIndexed(), it can be used everywhere where the AsIndexed() extension method can be used. So, live views are supported in all cases in LiveLinq to Objects, LiveLinq to XML and LiveLinq to DataSet (with some restrictions on the supported query operators, see Query Operators Supported in Live Views). The difference between AsIndexed() and AsLive() is that AsLive() creates a view, thus enabling LiveLinq both to query to populate the view and to maintain the view after it has been initially populated. The AsIndexed() extension method does only the first part, enables LiveLinq querying.

The example above is a very simple one, it's just one simple Where condition. There are other tools that can do the same, for example, **DataView** in ADO.NET or **CollectionView** in WPF. The power of LiveLinq is that it supports most of the LINQ operators, including joins and others. So you can create a live view based not just on a simple condition, but on virtually any query you need.

How to bind GUI controls to a live view

LiveLinq views can be used as data sources for GUI controls, because they implement the data binding interfaces. So you can simply bind any control to it as to any other data source. For example, in WinForms:

```
View<T> view = from a in A.AsLive() join b in B.AsLive() on a.k == b.k
    where a.p == 5 select new {a, b};
dataGridView1.DataSource = view;
```

Data binding has long been the tool of choice for most developers creating GUI, but its scope was limited to simple cases only. You could bind to your base data, like your base collections, tables of a data set, etc, but not to your derived data, not to data shaped by some query. Yes, some shaping is supported by some tools, but it is very limited, mostly just filtering and sorting. If you have data derived/shaped in any more complex way, for example, with joins (a very common case), you could not use data binding (more exactly, you could only use it for one-time snapshot binding, show data once, no changes allowed).

LiveLinq makes data binding extremely powerful by removing these limitations. Now you have the full power of LINQ to bind to!

With live views, you can create entire GUI, sophisticated applications, virtually without procedural code, using declarative data binding alone. An example of such application is the LiveLinqIssueTracker demo.

How to use live views in non-GUI code

Using live views is not limited to GUI. In a more general way, live views enable a declarative style of programming which we tentatively call ***view-oriented programming***. Non-GUI, batch processing code can also be made declarative using live views.

Generally speaking, any application operates on some data, and any data can be seen as either based or derived data. Base data contains the main objects your application is dealing with, like Customers, Orders, Employees, etc. Those objects (collections containing all objects of a certain class, sometimes called *extent* of a class) are the objects that your application modifies when it needs to make some change in the base data. But applications rarely operate on this base data directly, rarely directly operate on the entire extent of a class. They filter and shape those extents and combine them together to get to a slice of data they need for a particular operation. This shaping/filtering/joining/slicing of data is the

process of getting ***derived*** data (as opposed to ***base,*** underlying data). Before the emergence of LINQ, it was done by purely procedural code, with all the ensuing complexity. LINQ made it declarative, which is a huge leap forward. But, although declarative, it is not live, not dynamic, does not react to changes in base data automatically. So, its reaction to change is not declarative, the programmer needs to react to changes in procedural, imperative code. Complexity is diminished by LINQ, but reacting to change remains complex. And reacting to change is pervasive, because change is everywhere.

LiveLinq live views make reacting to change declarative too, thus closing the circle of declarativeness. Now entire applications, not just GUI ones, can be made virtually entirely declarative!

To give just a small example, we can consider a sample in the LiveLinqIssueTracker demo. It has two operations performed on some issue data:

(1) Assign as many unassigned issues to employees as possible, using information on pending issues, product features (every issue belongs to a feature) and assignments.

(2) Collect information on open issues for given employees.

Each of these two operations (and in a real application there is, of course, many more operations like this than two) works on data shaped by a query with joins (see the actual queries in Live Views How To: Use Live Views to Create Non-GUI Applications as a Set of Declarative Rules Instead of Procedural Code). Both operations can be performed more than once during program execution. Data is changing while these and other operations on data are performed. Live views used to implement these operations change automatically with that changing data, so the operations can be kept simple and completely declarative, and, as a result of that, robust, reliable and flexible, easily modifiable when business requirements change.

There is nothing else needed for this style of programming in addition to what we already know about how to create live views.

Samples

Quick Start Samples

Indexing samples

Quick Start Indexing samples demonstrate basic use of the first of the two areas of LiveLinq functionality: using indexes to make LINQ queries faster. They don't use live views, except for LiveLinqToXml, where live views are necessary to define base data collections.

Every sample uses two queries: one the most basic and the other a little more complex, with a join. The queries are identical in functionality in all three samples, differing only in the nature of the underlying data: user object collections (LiveLinqToObjects), ADO.NET DataSet (LiveLinqDataSet) or XML (LiveLinqToXml).

Every sample shows two alternative ways of creating indexes: explicitly, by adding to the Indexes collection, or implicitly, by using hints in queries.

LiveLinqToObjects

This sample shows how to use the IndexedCollection class to create and query data collections using indexes to speed up query performance.

LiveLinqToDataSet

This sample shows how to use LiveLinq to query data residing in an ADO.NET DataSet, how to create indexes over that data that make querying faster than regular LINQ to DataSet queries. Both strongly typed and untyped datasets are demonstrated in the sample.

LiveLinqToXml

This sample shows how to use LiveLinq to query XML data. It creates indexes on 'customers' and 'orders' live views defined directly over XML data. This easy XML indexing allows to dramatically speed up LINQ queries over XML data, often make them hundreds of times faster than regular LINQ to XML, see Query Performance sample application (LiveLinqQueries) for performance metrics.

Live view samples

Quick Start LiveViews samples show the second and main part of LiveLinq functionality: LINQ queries as live views, automatically reflecting changes in the data on which they are based, so they can be used to build applications declaratively, minimizing procedural, manual coding.

Every sample uses two views over the same data: one the most basic (filtered view) and the other a little more complex, with a join. The functionality is identical in all three samples, differing only in the nature of the underlying data: user object collections (LiveLinqToObjects), ADO.NET DataSet (LiveLinqDataSet) or XML (LiveLinqToXml).

To see live views in action, you can try, for example:

- Change the ShipCity value from "London" to "Colchester" or vice versa in any of the two grids and leave the current row (to commit the change). Observe that the value in the other grid changes correspondingly.
- Change the ShipCity value to any string other than "London" and "Colchester" in any of the two grids. Observe that the row disappears from both grids.

See also the Live views sample application (LiveLinqIssueTracker) for more advanced functionality.

LiveViewsObjects

This sample demonstrates basic live view functionality for views based on user-defined object collections (LiveLinq to Objects).

LiveViewsDataSet

This sample demonstrates basic live view functionality for views based on ADO.NET DataSet (LiveLinq to DataSet). This sample uses a typed data set, but untyped data sets can be used as well, as shown in the LiveLinqToDataSet sample.

LiveViewsXml

This sample demonstrates basic live view functionality for views based on XML (LiveLinq to XML).

Sample Applications

Query Performance sample application (LiveLinqQueries)

The querying demo shows all three LiveLinq varieties: LiveLinq to Objects, LiveLinq to DataSet and LiveLinq to XML, side by side, performing the same queries on the same data. The data are the good old Northwind **Customers** and **Orders** tables, in the three incarnations: custom object collections, a data set and XML, respectively.

The demo contains various queries, some with a parameter. For each query it shows the code, how that query is formulated in each of the three LiveLinq varieties. Every query is implemented in two different forms:

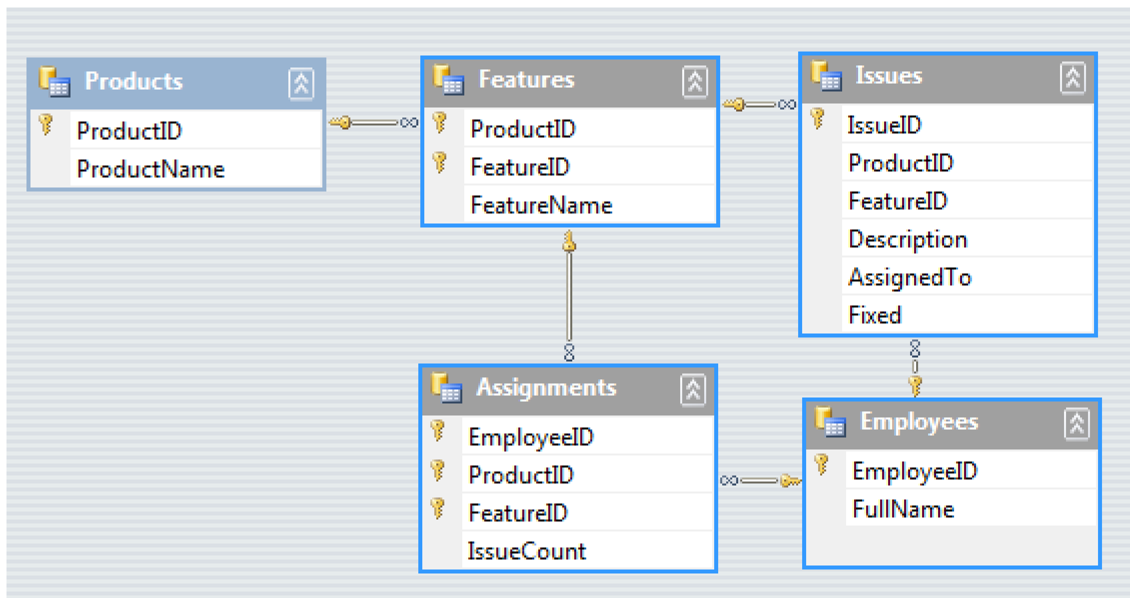
- Programmatic index search, that is, code without LINQ, directly using indexes for search, and
- LINQ query syntax.

These two LiveLinq implementations are compared with a standard LINQ implementation shown along with them. For each query, the demo shows the speedup factor, how faster the LiveLinq implementation is compared with the standard LINQ one.

Live views sample application (LiveLinqIssueTracker)

This demo shows how live views can be used to construct entire GUI applications based virtually exclusively on declarative queries/views and data binding, with little procedural code. It also contains an example of how live views can be used in non-GUI, batch, perhaps server-side processing.

It is a mockup bug/issue tracking application in a fictitious company that produces Shakespeare's plays. The demo comes in three variations, each built on top of a different data store: Collections (uses LiveLinq to Objects), ADO.NET (uses LiveLinq to DataSet) and XML (uses LiveLinq to XML). It uses WinForms data binding, but WPF data binding could be used instead just as well, the live views would be exactly the same.



The data schema contains **Products** (each product is a Shakespeare's play). Each product has features (which, incidentally, are roles in the play). And there are **Issues**, in other words, bugs (such as bad acting in a certain scene, for example). Also, there are **Employees** who are assigned those issues (incidentally, those employees are actors playing those roles), and, an important part of this schema, there are **Assignments**: every employee is assigned a certain number of features to take care of. These assignments can overlap, meaning that more than one employee can be assigned to a feature.

When we start the application and open the **Assigned Issues** form, we can see issues assigned to any given employee. The view representing it is (we are giving LiveLinq to DataSet versions of the views here, Objects and XML versions are similar):

```
from i in _dataSet.Issues.AsLive()
join p in _dataSet.Products.AsLive()
    on i.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { i.ProductID, i.FeatureID }
    equals new { f.ProductID, f.FeatureID }
join e in _dataSet.Employees.AsLive()
    on i.AssignedTo equals e.EmployeeID
where i.AssignedTo == employeeID
select new Issue
{
    IssueID = i.IssueID,
    ProductName = p.ProductName,
    FeatureName = f.FeatureName,
    Description = i.Description,
    AssignedTo = e.FullName
};
```

It is a non-trivial LINQ query, with joins and filtering, and the demo shows that it is live, dynamic, automatically reacts to changes in data, and that GUI controls such as **DataGridView** can be bound to it.

For example, if we add an issue in the **Add Issue** form and assign it to the same employee that is shown in the **Assigned Issues** form, the list of issues for that employee is automatically updated to include the newly created issue.

The **Assigned Issues** form also demonstrates the difference between **Immediate** and **Deferred** maintenance mode, allowing switching between them with two radio buttons. If we switch to the **Deferred** mode, meaning update on demand, and add an issue, that change is not immediately reflected in the view, it is reflected only when we request data from that view, for example, by changing the employee and then changing it back.

There is also a **(Re)assign Issue** form that uses basically the same view, so, without writing code, using only live views and data binding, we can do things like assigning issues and reassigning them.

Assigned Issues and **(Re)assign Issue** forms can be opened in any number of instances side-by-side. You can experiment with opening several forms like that, re-assigning issues in one of them and observe the changes automatically reflected in all others.

The demo also shows more complicated actions than that, shows how live views can change more than just one row at a time. There is a form called **All Issues Concerning an Employee**. It shows all issues for all features assigned to a particular employee, using the following view:

```
from a in _dataSet.Assignments.AsLive()
join p in _dataSet.Products.AsLive()
    on a.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { a.ProductID, a.FeatureID }
        equals new { f.ProductID, f.FeatureID }
join i in _dataSet.Issues.AsLive()
    on new { a.ProductID, a.FeatureID }
        equals new { i.ProductID, i.FeatureID }
join e in _dataSet.Employees.AsLive()
    on i.AssignedTo equals e.EmployeeID
where a.EmployeeID == employeeID
select new Issue
{
    IssueID = i.IssueID,
    ProductName = p.ProductName,
    FeatureName = f.FeatureName,
    Description = i.Description,
    AssignedTo = e.FullName
};
```

Assignments of features to an employee can be changed in a **(Re)assign Features** form. You can experiment with changing the set of features assigned to an employee and see how the changes, that can be quite massive, automatically occur in the **All Issues Concerning an Employee** form. And, although it's not easy to see with such a small data set (unless you add thousands of issues to it, which likely would be the case in a real-life application), the changes to the views shown in the grids are fast, without delays for re-querying the views and refreshing the entire grid, thanks to the incremental view maintenance algorithms implemented in LiveLinq.

The demo also shows how to create views based on other views and create indexes on views, see Live Views How To: Create Views Based on Other Views and Create Indexes on Views, and how to use live views to create non-GUI applications as a set of declarative rules instead of procedural code, see Live Views How To: Use Live Views to Create Non-GUI Applications as a Set of Declarative Rules Instead of Procedural Code.

Programming Guide

Query Operators Supported in Live Views

LiveLinq supports all LINQ query operators in its queries. Not all operators have LiveLinq-specific implementation benefiting from indexes and other query execution optimization techniques, but such operations simply use the standard LINQ to Objects (or LINQ to XML) implementations, so it is transparent to the user.

However, not all query operations can be used in live views. This is because not all query operations have incremental view maintenance algorithms, some would require re-populating from scratch (requerying) every time they are maintained. If your query contains such operations, you won't be able to use that query to create a live view. An attempt to do that will cause a compilation error.

Here is the list of query operators allowed in live views:

Operator	Notes
Select	Overload with **selector** depending on the index is not allowed.
Where	Overload with **predicate** depending on the index is not allowed.
Join	Overload with **comparer** is not allowed.
GroupJoin	Overload with **comparer** is not allowed.
OrderBy	Overload with **comparer** is not allowed.
OrderByDescending	Overload with **comparer** is not allowed.
GroupBy	Overloads with **comparer** and with **resultSelector** are not allowed.
SelectMany	Overloads with **selector** and **collectionSelector** depending on the index are not allowed.
Union	
Concat	
Aggregate	Use **LiveAggregate** if you want it live, otherwise it is the standard LINQ operator, can be used in live views but does not automatically reflect data changes.
Count	Use **LiveCount** if you want it live, otherwise it is the standard LINQ operator, can be used in live views but does not automatically reflect data changes.
Min/Max	Use **LiveMin/LiveMax** if you want it live, otherwise it is the standard LINQ operator, can be used in live views but does not automatically reflect data changes.
Sum	Use **LiveSum** if you want it live, otherwise it is the standard LINQ operator, can be used

	in live views but does not automatically reflect data changes.
Average	Use **LiveAverage** if you want it live, otherwise it is the standard LINQ operator, can be used in live views but does not automatically reflect data changes.

Query Expressions Supported in Live Views

Apart from the limitations on query operators, a query must satisfy an additional condition in order to be used as a live view. Fortunately, this condition is satisfied in most cases, so you should care about it only if your query contains some special expressions, which is relatively rare (except that all classes used in LiveLinq to Objects must satisfy the property notification condition, but that was already mentioned in Using the built-in collection class IndexedCollection<T> (LiveLinq to Objects)). Unfortunately, this condition is not verified by LiveLinq automatically, so it is your responsibility to make sure it is satisfied. If this condition is not satisfied, your view will not react to changes in its base data. We give two descriptions of this condition, one short, for basic understanding and another detailed, for advanced usage.

Short Description

The short answer is, basically, that in some cases your classes need to provide property notifications. There are two such cases where you need to care about that:

(1) **LiveLinq to Objects.**

There your views' arguments are collections of your own classes, so you must make sure that your classes provide property notifications, otherwise your views will not react to changes of those properties, see Using the built-in collection class IndexedCollection<T> (LiveLinq to Objects).

(2) **Views over views and indexes over views** (applies to LiveLinq to DataSet and LiveLinq to XML as well as to LiveLinq to Objects).

If you have a view

```
View<T> v;
```

and want to create another view over view **v** (use **v** as its argument) or create an index over **v**, then make sure that the **T** class has property notifications, otherwise the view (or index) you define over **v** will not react to changes of those properties.

Also, it should be mentioned here that you don't need to care about these conditions at all if you don't use classes (reference types) for elements of your collections, if you only use structs (value types). This is simply because value types are immutable, you can't change the elements of such collection, so there is no need in notifications.

Detailed Description

The condition limits the choice of functions (such as result selector, key selector, etc) that you can use in your query expression depending on whether or not your classes support property notifications, see Using the built-in collection class IndexedCollection<T> (LiveLinq to Objects).

Observable and non-observable functions

We distinguish two types of functions:

1. Observable function.

A function (expression) is observable if the only kinds of elementary expressions it contains are property or method calls whose values are observable, in the sense that any change to that value is always accompanied by the change notification events.

Examples of observable functions:

```
x => x.P1
x => x.P1 + x.P2 + x.M(c1)
(x, y) => new {x.P1, x.P2, y.M(), y.P3}
```

Here **P1**,**P2**,**P3**,**M**() are properties/methods with change notifications, and **c1** is a value that never changes.

2. Non-observable function.

Those are functions that have something else in their expressions except observable properties/methods.

Examples of non-observable functions:

```
x => x
(x, y) => (x, y)
(x, y) => (x, y.P)
(x, y) => x == y ? c1 : c2
x => x == c1 ? c2 : c3
```

Note that a function on an **Order** class (that has a **Customer** property) defined as

```
o => o.Customer.City
```

may be observable if you took care to trigger the property change notification events for the **Orders** collection every time the **City** property in the **Customer** class changes, but this is usually not the case unless you specifically do it in code, so such functions are usually non-observable.

If the function is observable, the view is OK

If all your functions in a view are observable, then there is no problem and you can skip the rest of this section, the view will work under any conditions. But note that this condition of observability depends not only on the function but also on the class of the argument of that function: that class must have property change notifications for everything that is used in the function. So, even the most simple functions, such as **x => x.P** can be non-observable if **P** is non-observable (that is, the class does not issue change notifications for **P**).

Common non-observable functions are also OK

Some special cases of non-observable functions, those that are common and important, are also supported. These are some common cases of object initializers, that is, creating objects without calling a constructor, such as, for example

```
(x, y) => new C { P1 = x.Q1, P2 = y.Q2 }
```

(class **C** defined somewhere in your code), or

```
x => new { P1 = x.Q1, P2 = x.Q2 }
```

(anonymous class)

You may notice that these examples are already observable according to our definition, no need for a special treatment, if the **Q1** and **Q2** properties are observable. But if we use a simple parameter reference in this initializer, our definition above is no longer satisfied:

```
(x, y) => new C { P1 = x, P2 = y }
```

However, such function is still supported, as a special case, because such cases are common. More exactly, as a special case, we allow an object initializer (regardless of whether the class is regular or anonymous) where each initialization is either observable function or a simple reference to one of the parameters.

Examples of allowed object initializers (all properties are assumed observable):

```
(x, y) => new { x, y }
(x, y, z) => new C { x, y.P, y, y.P + z.Q }
x => new { x, y.P }
x => x
```

(the last one is not, strictly speaking, an object initializer, but it is also allowed)

Examples of object initializers that are not allowed:

```
x => new { P = f(x) }, if f(x) is not observable
x => new { P = x == c1 ? c2 : c3 }
(x, y) => new { P1 = x.P1 + y.P2 }
```

Note: Simple reference to a parameter is understood here to include not only this function's parameters, but also, recursively, the parameters of functions preceding this function in the query, if they can be reached through (possibly a chain of) simple references in object initializers. For example,

```
....Select((x, y) => new { P1 = x, P2 = y }).Select(z => new { z.P2, z.P1 })
```

is allowed. Here **z.P1, z.P2**, although not simple parameters of their own function, are simple parameters of a previous function in the same query.

Special case - value types

As mentioned in Short Description, everything goes if you use structs (value types) instead of classes (reference types), because value types are immutable, so there is no need in change notifications because there is no change.

View Maintenance Mode

Live views can be used in different kinds of applications. They can be used in GUI, interactive applications and in non-GUI, batch processing-style applications, see How to use live views in non-GUI code. Live views are optimized for both modes, GUI (interactive) and non-GUI (batch). They distinguish between these modes and operate accordingly. In GUI, live views react to a change immediately when the change happens. They do it fast, using incremental algorithms without re-populating themselves, see Maintaining the view: Incremental View Maintenance, but still it is not always suitable for batch, non-interactive processing. In GUI, interactive programs, with data binding, immediate reaction to change is what is usually needed because the user needs to see the change on the screen. In batch processing, on the other hand, a view may be accessed long after the change occurred or even not at all. So updating that view before it is actually accessed by the program may be an unnecessary drain on resources. By default, live views distinguish between these two modes automatically, but the programmers has an option to control that using the MaintenanceMode. In **Immediate** mode, which is the default for GUI, the view is maintained immediately on every change. In **Deferred** mode, which is the default in a non-GUI case (when there are no listeners attached to the view), the view is maintained on demand. It is not kept in sync with base data until it is needed, until there is a request for data from that view. When such request arrives, the view sees that it is in a "dirty" unsynchronized state, so it automatically updates (maintains) itself to come in sync with the changed base data.

Live View Performance

First performance consideration with live views is that, naturally, live view functionality has a price. Maintaining the view in sync with base data is fast, optimized, but still it consumes some resources when base data changes and the view is maintained. And it consumes some additional resources when it is first populated as well. That additional cost is not high, but it exists, manifesting itself mostly in additional memory consumption: when a live view is populated, it creates some internal data in memory to help it maintain itself faster on base data changes.

This additional memory consumption is moderate, roughly equivalent to the amount of memory needed for the resulting list itself. So, although additional resources needed for live view functionality are light to moderate, the obvious suggestion is to use live views only where you are really interested in keeping the result of a query live, or, in other words, where you need that result more than once and the base data is changing so that the result is changing too.

Other than this, there are two different aspects to live view performance:

Populating the view: query performance

Query performance is how long it takes to populate the view for the first time by executing the query (or re-populate by re-executing the query, see View.Rebuild). This is covered in LiveLinq query performance: logical optimization and LiveLinq Query Performance: Tuning Indexing Performance.

Maintaining the view: Incremental View Maintenance

Maintaining live views on base data changes is optimized using techniques known as Incremental View Maintenance. It means that a view does not simply re-populate itself, it adjusts to accommodate base data changes in a more smart way. The changes to the view are made locally, incrementally, calculating the delta in the view from the delta in the base data. In most cases, it is very fast and does not require any special performance tuning from the user.

As a guideline to estimating the time of view maintenance, we can say that it is roughly proportional to the number of changed items (including added and deleted ones) in the view resulting collection. So, if only a few items change as a result of a change in base data, the view maintenance time is very small and probably negligible, but it can be considerable if a considerable part of the view result changes. It can even become more costly to maintain a view than to re-query it from scratch, if, for example, half of the resulting list changes.

Finally, it is worth noting that maintenance performance for a view does not depend on its query performance. It does not matter how long it takes to populate the view initially, maintenance time is roughly the same, it depends on the size of the change in (delta), not on the size of the overall data.

LiveLinq query performance: logical optimization

Standard LINQ to Objects does not perform any logical optimization, it executes queries exactly as they are written. And, of course, standard LINQ to Object does not use indexes for optimization. In comparison, LiveLinq performs physical optimization (using indexes, if they are present) and logical optimization (re-writing the query in a more efficient form before processing it).

However, LiveLinq does not contain a full-scale query optimizer like one in a relational database such as SQL Server or Oracle, so it can still matter how a query is written. But it is largely limited to join ordering. LiveLinq does not try to re-order your joins. Joins are executed always in the same order as specified in the query. So, you should avoid writing queries with obviously inefficient join order. But it must be noted that it is a relatively rare problem (and the same consideration, of course, applies to the standard LINQ to Objects anyway). It can be an issue in a query like

```
from a in A
join b in B on a.k equals b.k                        (1)
where b.p == 1
```

if there is, say, only 10 elements in **B** with **b.p == 1** and only 100 elements in **A** that satisfy **a.k == b.k** and **b.p == 1**, but the overall number of elements in **A** is many times higher, say, 10000. Then the query above will require 10000 "cycles", whereas rewritten with the right join order,

```
from b in B
join a in A on b.k equals a.k                         (2)
where b.p == 1
```

this query will require only 100 cycles to complete. Note that the position of **where** is not important, the above query has the same performance as

```
from b in B
where b.p == 1                                        (3)
join a in A on b.k equals a.k
```

That's because of LiveLinq query optimization: LiveLinq re-writes (2) internally to (3) when it executes it, because it is preferable to check conditions before performing other operations, to exclude elements that don't contribute to the result. This is one of the logical optimizations performed by LiveLinq internally.

LiveLinq Query Performance: Tuning Indexing Performance

Query speedup achieved on any particular query by LiveLinq using indexes for optimization depends on how that query is written. This dependence is usually not dramatic. LiveLinq does a fair job of recognizing opportunities to use indexes

for optimization regardless of the way the query is written. For example, it will still execute a query efficiently using indexes even if the condition consists of multiple terms connected with logical operators.

If you want to ensure that your indexes are used effectively by LiveLinq, consider the following guidelines:

Elementary predicates that can benefit from indexing

LiveLinq can use an index by a property **P** in an elementary predicate (condition without boolean operations) that has one of forms (patterns) (1) – (6) listed below. The simplest and perhaps the most common of such elementary predicates is a **where** condition with an equality involving a property, as in

```
from x in X where x.P == 1 select x
```

It will use the index by x.P, provided that such index exists, which, as described in How to create indexes, can be ensured either by creating this index explicitly

```
X.Indexes.Add(x => x.P);
```

or by using the Indexed() hint:

```
from x in X where x.P.Indexed() == 1 select x
```

In fact, LiveLinq supports a more general indexing. It does not necessarily have to be a property of **x**, it can be any expression depending on **x** (and only on **x**). This can come in handy, for example, if we are querying an untyped ADO.NET DataTable, so, because it is untyped, it does not have properties that we can index and query by their names and we need to use a query like this:

```
from c in customersTable.Rows where c.Field<string>("CustomerID") == "ALFKI" select x
```

Then we can use an index by the following expression:

```
X.Indexes.Add(c => c.Field<string>("CustomerID"));
```

We will quote only the most common case of a simple property, **x.P** in the list below, but keep in mind that it can be replaced everywhere with any expression depending on **x** (and only on **x**).

Following is the list of patterns recognized by LiveLinq as allowing optimization with indexes:

(1) Equality:

x.P == Const (**Const** is a value constant in the given query operator, which means that it can be an expression depending on external parameters and variables, but it must have the same value for all elements that are tested by this **where** condition).

Example:

```
from o in Orders.AsIndexed() where o.OrderID.Indexed() == 10011
```

(2) Inequality:

x.P op Const, where **op** is one of the comparison operators: >, >=, <, <=

Example:

```
from o in Orders.AsIndexed() where o.OrderID.Indexed() > 10011
```

(3) Left and right parts of an equality or inequality are interchangeable (commutative).

Example: The following query will use indexing exactly as the one in (1):

```
from o in Orders.AsIndexed() where 10011 == o.OrderID.Indexed()
```

(4) StartsWith:

x.P.StartsWith(Const), if **x.P** is of type **string**.

Example:

```
from o in Orders.AsIndexed() where o.CustomerID.StartsWith("A")
```

(5) Belongs to (in, is an element of):

ConstColl.Contains(x.P), if **ConstColl** implements **IEnumerable<T>** where **T** is the type of **x.P**.

Example:

```
from o in Orders.AsIndexed()
where (new int[]{"ALFKI", "ANATR", "ANTON"}).Contains(o.CustomerID)
```

(6) Year comparison:

x.P.Year op Const, where **x.P** is of type **DateTime** and **op** is any of the comparison operators ==, >, >=, <, <=

Example:

```
from o in Orders.AsIndexed()
where o.Date.Indexed().Year == 2008
```

Other elementary predicates ***don't*** use indexes. Examples where indexing will not be used:

```
from o in Orders.AsIndexed() where o.Freight > 10
```

(if there is no index defined for the **Freight** property)

```
from o in Orders.AsIndexed() where o.OrderID.Indexed() < o.Freight
```

(comparison must be with a constant, not with a variable value)

```
from o in Orders.AsIndexed() where o.OrderID.Indexed() != 10011
```

(only certain comparisons are used, ! (**not equal**) is not one of them)

Boolean operators

Conjunction (&&) and disjunction (||) are handled by LiveLinq optimizer, including their arbitrary combinations and including possible parentheses. Other boolean operators, such as negation (!) are not handled by the optimizer, they block its use of indexing.

Conjunction:

Conjunction does not prevent the use of indexes. For example,

```
from o in Orders.AsIndexed() where o.Freight > 10 && o.Lines.Count > 5
```

will use the index by **Freight** property (supposing such index exists) even though the second condition cannot use indexes. LiveLinq will simply check the other condition for each item that is found using the index from the first condition.

Conjunction of conditions using the same property

Moreover, conjunctions of conditions with the same property will be optimized to render the best execution plan. For example, if the **Freight** property is indexed,

```
from o in Orders.AsIndexed() where o.Freight > 10 && o.Freight < 20
```

will not go through all orders with **Freight > 10** and check if it is less than 20 for each of them, but will use the index to go directly to the orders between 10 and 20 and won't check any other orders.

Conjunction of conditions with different properties: where subindexes come into play

Conjunctions using different properties, like for example

```
from o in Orders.AsIndexed() where o.OrderID > 10011 && o.Freight < 20
```

can utilize subindexes, see Subindex(T, TKey) Class. If the index by **OrderID** has a subindex by **Freight**, LiveLinq will not check all orders with **OrderID** greater than 10000 (which can be quite numerous). It will go directly to the subindex corresponding to the value 10011 (only one such subindex exists, and it can be accessed directly, without any search) and enumerate the items in that subindex whose **Freight** is less than 20. Both operations (finding a subindex and enumerating a range of items in it) are direct access operators, without search, so the query is executed in the fastest way possible, without spending any time on checking items that don't contribute to the result.

Disjunction:

(a) For an indexed property **x.P**,

```
x.P == Const1 || x.P == Const2
```

is handled by re-writing the condition as

```
(new ...[]{Const1, Const2}).Contains(x.P)
```

(b) If two different properties **x.P** and **x.Q** are indexed, then

```
x.P op Const1 || x.Q op Const2 (op is ==, <, >, <=, etc)
```

is handled by re-writing the query as a union of two separate queries with corresponding elementary conditions.

Join

If either of the two key selectors in a join, that is, either **x.K** or **y.K** in

```
from x in X
join y in Y on x.K equals y.K
```

is indexed, LiveLinq will use that index to perform the join.

Ideally, both **x.K** and **y.K** are indexed, and then LiveLinq will execute the join in the fastest way possible (using so called *merge join* algorithm, which is very fast, basically, it takes the same time to build as to traverse the result of such join, there is no time lost on anything).

If there are no indexes on **x.K** or **y.K**, LiveLinq will still try to find the best algorithm for the join, which is sometimes merge join (if both parts of the join are ordered by the merge key) and sometimes the *hash join* algorithm used by the standard LINQ to Objects.

As a general guideline, it should be noted that defining indexes only for optimizing **Join** operations in your query will rarely lead to dramatic speedups. That's because the hash join algorithm (the one used in the standard LINQ, it does need indexes) is already quite fast. It does make sense to define indexes for join keys sometimes, but consider it when you are fine-tuning your query, not when you need the first quick and dramatic speedup. For dramatic speedups, first consider optimizing your **Where** conditions, that can speed up your query often hundreds and even thousands times (depending on the query, of course).

Live Views How To: Create Views Based on Other Views and Create Indexes on Views

Multiple queries with parameters can often be replaced with a single "big" live view. That can help making code clear and simple and can often make it faster. This technique is based on two features of live views:

- A view can be based on other views in the same way as it can be based on base data.
- Indexes can be created for a view in the same way as indexes are created for base data.

So, instead of issuing many queries by varying a parameter value, you can create a single live view, create an index for it, and, when you need the list of items corresponding to a particular value of the parameter, you simply get those items from the index for that particular value.

It is illustrated in the LiveLinqIssueTracker demo:

The **Assigned Issues** form uses multiple views, a separate view for every employee: views depending on a parameter **employeeID** (we are giving LiveLinq to DataSet versions of the views here, Objects and XML versions are similar):

```
from i in _dataSet.Issues.AsLive()
join p in _dataSet.Products.AsLive()
    on i.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { i.ProductID, i.FeatureID }
        equals new { f.ProductID, f.FeatureID }
join e in _dataSet.Employees.AsLive()
    on i.AssignedTo equals e.EmployeeID
where i.AssignedTo == employeeID
select new Issue
{
    IssueID = i.IssueID,
    ProductName = p.ProductName,
    FeatureName = f.FeatureName,
    Description = i.Description,
    AssignedTo = e.FullName
};
```

The demo application also contains an alternative implementation of the same functionality, in the form **Assigned Issues 2**. That form uses a single view containing data for all employees, instead of multiple views depending on a parameter. This single view does not have parameters:

```
_bigView =
    from i in _dataSet.Issues.AsLive()
    join p in _dataSet.Products.AsLive()
        on i.ProductID equals p.ProductID
    join f in _dataSet.Features.AsLive()
        on new { i.ProductID, i.FeatureID }
            equals new { f.ProductID, f.FeatureID }
    join e in _dataSet.Employees.AsLive()
        on i.AssignedTo equals e.EmployeeID
    select new Issue
    {
        IssueID = i.IssueID,
        ProductName = p.ProductName,
        FeatureName = f.FeatureName,
        Description = i.Description,
        AssignedToID = e.EmployeeID,
        AssignedToName = e.FullName
    };
```

That view is indexed by the employee id field:

```
_bigView.Indexes.Add(x => x.AssignedToID);
```

so we can retrieve the items for a particular employee id at any time very fast.

Moreover, we can create a live view of that data given an employee id value (which is **comboAssignedTo.SelectedIndex** in this demo), simply by creating a view over the big view:

```
from i in _bigView where i.AssignedToID == comboAssignedTo.SelectedIndex select i;
```

Live Views How To: Use Live Views to Create Non-GUI Applications as a Set of Declarative Rules Instead of Procedural Code

In addition to making GUI applications declarative, LiveLinq also enables a programming style (which we tentatively call ***view-oriented programming***) that makes non-GUI, batch processing declarative too. See How to use live views in non-GUI code for the introductory description of this technique.

The LiveLinqIssueTracker demo application contains a **Batch Processing** form where this technique is demonstrated by implementing two actions:

(1) Assign as many unassigned issues to employees as possible. Looking at the feature to which a particular issue belongs, find an employee assigned to that feature, and, if that employee does not have other issues for that feature, assign that issue to that employee.

(2) Collect information on all open (not fixed) issues for a given employee (perhaps for the purpose of emailing that list to that employee).

We perform action (1) by defining the following view (we are giving LiveLinq to DataSet versions of the views here, Objects and XML versions are similar):

```
_issuesToAssign =
    from i in issues
    where i.AssignedTo == 0
    join a in _dataSet.Assignments.AsLive()
        on new { i.ProductID, i.FeatureID }
            equals new { a.ProductID, a.FeatureID }
    join i1 in issues
        on new { i.ProductID, i.FeatureID, a.EmployeeID }
            equals new { i1.ProductID, i1.FeatureID, EmployeeID = i1.AssignedTo }
        into g
    where g.Count() == 0
    join e in _dataSet.Employees.AsLive()
        on a.EmployeeID equals e.EmployeeID
    select new IssueAssignment
    {
        IssueID = i.IssueID,
        EmployeeID = a.EmployeeID,
        EmployeeName = e.FullName
    };
```

and performing the operation with simple code based on that view:

```
foreach (IssueAssignment ia in _issuesToAssign)
    _dataSet.Issues.FindByIssueID(ia.IssueID).AssignedTo = ia.EmployeeID;
```

Action (2) is performed by defining the following view:

```
from i in _dataSet.Issues.AsLive()
join p in _dataSet.Products.AsLive()
    on i.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { i.ProductID, i.FeatureID }
        equals new { f.ProductID, f.FeatureID }
join em in _dataSet.Employees.AsLive()
    on i.AssignedTo equals em.EmployeeID
    where i.AssignedTo == employeeID && !i.Fixed
select i.IssueID;
```

If we need to perform the operations (1) and (2) just once, then there is no point in using live views. But suppose we a writing a program that needs to perform those actions (1) and (2) many times as steps of the overall algorithm it is executing. This is a common case, especially in server-side programming.

Without live views, we would either need to requery each time, which is very costly, or we would need to create and maintain some collections throughout our processing algorithm. Those collections would need to be maintained by manual code, often complicated, and written by different programmers and often containing bugs. Different people write different functions (actions (1) and (2) are, of course, just two small examples of such functions), they need to know what others are doing if they want to keep it consistent. All this is hard to maintain. When a new action or function is added a year after that, the logic that connects everything is by that time forgotten, the new action breaks that logic, and so the vicious cycle of two complicated programming goes on.

Compare this with how it can be done with live views:

Actions (1) and (2) are not actually procedures, they are declarative rules. Rule (1) states that this view contains the assignments to be made. Whatever changes are made to our data, this view will always contain the needed assignments, regardless of anything we add a year from now or at any other time. The logic is guaranteed to be correct because it is a declarative rule, not procedural logic.

And that rule (1) works fast. We can perform that action a thousand times over data that is always changing, and every time it will recompute only the required amount of data, only the amount that is affected by changes. It will perform only the needed ***incremental*** recomputations.

Same with rule (2): it is a declarative rule, and it executes fast without repeating calculations, recalculating only those parts that are affected by changes.

If a substantial part of our algorithm is programmed in this ***view-oriented*** way, as a set of rules like (1) and (2), then they all work together, their consistency is automatically maintained. Ideally, we can represent our entire algorithm with views/rules like that. If not, a part of it can be implemented like that, and the rest can still be done with the usual rocedural code, that is also possible.