# Deep Dive on Aggregation and Bulk Operations

There are numerous articles on using the nHydrate code generator for general purpose development using domain driven design (DDD). This article will delve in the specialized realm of aggregation and by extension other bulk operations. The tool can obviously be used to create a strongly-typed, compile-time checked data access layer (DAL); however the DAL also has some really great aggregation features as well.

First let us review. The nHydrate code generator creates collections object entities that you have defined in your model. These collections are containers of specific type of entities as defined by their name. They also have many overloaded, static methods that allow you to easily query, aggregate, update, and delete data with one line of code. For example, the code below can be used to select all Job objects in a database.

|  |
| --- |
| JobCollection jobCollection = JobCollection.RunSelect(); |

Of course that is of limited use unless you want to select all jobs. A more real-world scenario is to define filter criteria. The code below shows how to select a group of job entities with a filter.

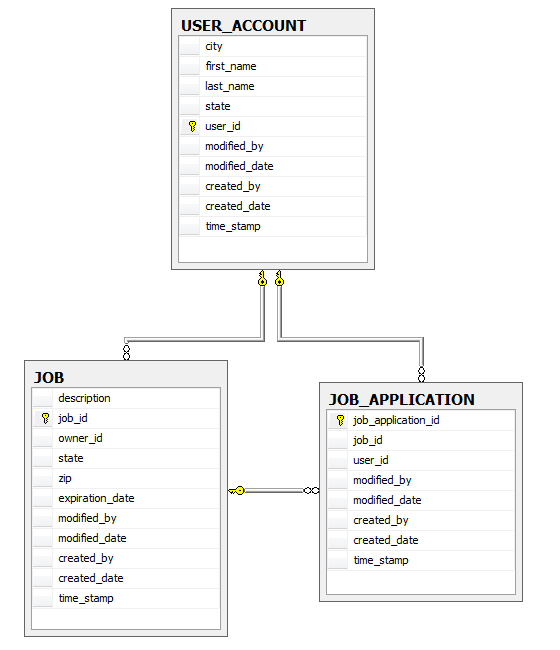
|  |
| --- |
| JobCollection jobCollection = JobCollection.RunSelect(x =>  x.CreatedDate > DateTime.Now.Date &&  x.Description.Contains("lollypop")); |

Now that was the refresher however it is not aggregation. Now following the usage above, we can use various static methods to aggregate data. The code below shows how to call various aggregations.

|  |
| --- |
| int count = JobCollection.GetCount(x =>  x.CreatedDate > DateTime.Now.Date &&  x.Description.Contains("lollypop"));  //The min value is nullable because there maybe no  //matching records, in which case null is returned //the return value is an INT because JobId is an INT  int? min = JobCollection.GetMin(x => x.JobId,  x => x.CreatedDate > DateTime.Now.Date &&  x.Description.Contains("lollypop"));  //Similar to above  int? max = JobCollection.GetMax(x => x.JobId,  x => x.CreatedDate > DateTime.Now.Date &&  x.Description.Contains("lollypop"));  //Get a list of all UserIDs who posted a job that meets the criteria  IEnumerable<int> idList = JobCollection.GetDistinct(x => x.OwnerId,  x => x.CreatedDate > DateTime.Now.Date &&  x.Description.Contains("lollypop")); |

Now these samples show how to get aggregated data from a customer collection. This can be extended across all collections of course. All collections are strongly-typed and contain exactly one type of entity. This is not strictly true if you count inherited objects however the collection will only pull back entity of the specified type. Notice that the aggregates take a where clause that is LINQ syntax. This is compile-time checked against your model. This means that at runtime, there are no surprises, i.e. run-time errors. You do not and should not use magic numbers or literal strings. You also should not define relationships inside of the LINQ. You have already done this in the model. The above examples were simple in that the LINQ was defined for only the customer table. In the real-world our query requirements span multiple entity types of course.

Now we will look at more complex scenario of spanning tables based on dependency walking. This is more complex conceptually, though not much in code. We simply augment our LINQ syntax a bit. In the example we have a three table database. In the database image you can see that the database has a user table and each user can have many jobs posted. Each job can have many job applications. Finally, each user can have many job applications. This is a real-world scenario where you have users, some are posters like recruiters and some are applying for jobs like candidates.



The C# code needed to aggregate this table layout is not that difficult. The first query returns a unique list of user ID's where the user has posted at least one job and one or more of those jobs posted received an application yesterday. The second query returns just the count of users that match the criteria. Notice that these two queries are run against two different collection objects. The first could actually be run against multiple object types since the LINQ query is defining the field to be aggregated. You could have defined that field from any collection object that is related in any way to the table entity containing the field aggregated. This is possible with strongly-typed LINQ syntax because all relationships are defined in the model. The second query can only be run from the UserCollection object since we are counting users. Notice that we never join or define walking criteria in LINQ code. The model is king and the domain designer has already defined the entity interactions pre-generation.

|  |
| --- |
| DateTime startDate = DateTime.Now.Date.AddDays(-1);  DateTime endDate = startDate.AddDays(1);  //Get distinct user ID list for people who posted jobs  //and there was a job application for 1 or more of  //these jobs yesterday. The LINQ knows that Job  //relates to JobApplication so the syntax  //shows a JobApplication object has  //1 related Job entity object  IEnumerable idList =  JobApplicationCollection.GetDistinct(  x => x.JobEntity.OwnerId,  x => startDate <= x.CreatedDate &&  x.CreatedDate < endDate);  //Same criteria as above but just get the number of the  //users with 1 or more jobs that had applications  //yesterday. Notice that we are counting users so  //we must use that collection  int count = UserAccountCollection.GetCount(x =>  startDate <= x.JobApplicationEntity.CreatedDate &&  x.JobApplicationEntity.CreatedDate < endDate); |

## Updates

Other important grouping functionality is updating in bulk. You may need to update the status of a group of users or change the expiration date of a group of job posts. In either case, you need to specify some field to be updated, a filter to select records, and a new value for the specified field. The generated DAL API created and maintained by nHydrate allows you to do this very easily in a strongly-typed fashion. The following code snippet updates a job table’s expiration date field to 30-days out if there were any job applications yesterday. Notice that the filter is complex and spans multiple tables.

|  |
| --- |
| DateTime startDate = DateTime.Now.Date.AddDays(-1);  DateTime endDate = startDate.AddDays(1);  //Update the expiration date of all jobs to 30 days out  //if they had at least one application yesterday  int count = JobCollection.UpdateData(x => x.ExpirationDate,  x => startDate <= x.JobApplicationEntity.CreatedDate &&  x.JobApplicationEntity.CreatedDate < endDate,  DateTime.Now.AddDays(30)); |

You will notice that this query joins the job and job application tables based on the defined model relationship. There is no need to specify the joining criteria in the LINQ filter. Also notice that the value specified field and the new value are the same type. Since the expiration date is a date, the new value can only be of this type too. The new data is not passed in as object. The code will not compile if the data type does not match. This is even true for nullable objects. If the expiration date can accept nulls in the database then the data type in code is a nullable date. The number of rows affected is returned from the UpdateData method.

## Deletes

The final bulk operation is that of delete. There are times when we need to remove a group of records based on a complex filter. It may not be reasonable and certainly not efficient to load all of this data from the database just to mark it for deletion. Again a collection’s static method gives up great power and efficiency. The following code will remove all job application records there the related JobID is less than 100 in value and the user that applied lives in Springfield (in any state) and the posted job is in the state “IL”.

|  |
| --- |
| //Remove all job applications if the related JobID is less then 100  //and the user that applied lives in Springfield  //and the posted job is in the state IL  int count = JobApplicationCollection.DeleteData(x =>  x.JobId < 100 &&  x.UserAccountEntity.City == "Springfield" &&  x.JobEntity.State == "IL"); |

This is a complex filter that spans three different tables. The generated SQL is complex no doubt but your C# code is elegant and simple. Even a non-developer could read this almost in natural language.

## Summary

As you can see, there is great power in the advanced aggregate functionality provided with nHydrate. The greatest thing about it is that this is out of the box functionality. There is no special setup to get it. When you define your model all of this functionality is derived from the entities and relationships you define. I cannot iterate too many times the superior aspect that compile-time checked syntax gives you. You can view this as an extension of the compiler. Once you define a nHydrate model it is validated against various rules. When you generate your code, it is checked by the compiler. When you write your custom code with all the complexity and user-defined fields and values, it too is checked by the compiler in that all interaction with the database is done through LINQ syntax and generated select command. There is no free-form SQL or lazy typing. This gives your application a greatly reduced chance of having runtime errors.