## DAG Processing Design notes/Readme

This is an open source project and contributions are welcome!

This document serves as a Readme for the DAGBuilder and related projects. It is **divided into 2 major sections**

1. Scenarios Addressed and Features Provided
   1. the project is addressing and a high level features that it provides.
2. Project Organization
   1. Details about the code/project organization and details about each of the project.

## Scenario addressed and features provided.

This project addresses large scale distributed graph execution scenario. The work to be done can use any arbitrary code that can be launched on the command line. It also gives users the ability to run the distributed work on different sized (CPU/Memory) nodes.

[Azure Batch](https://azure.microsoft.com/en-us/services/batch/) is used to run the samples, but any other cluster can be hooked in to run jobs on that cluster.

#### High level concepts and features provided

The diagram below shows how a job graph is defined. The left shows how the graph is defined as stages, and each stage is a collection of Tasks. Together they form a single job, that can be run in a distributed environment or all on a single machine.

**NOTE: Although the below diagram shows a map-reduce scenario, the system can run any arbitrary DAG.**

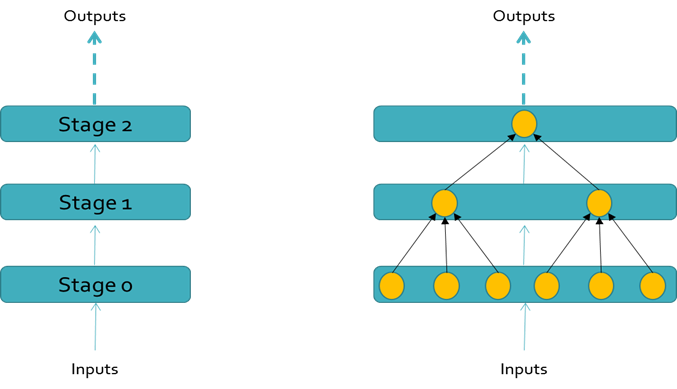


Figure 1- Job Definition. Each Stage is a collection of Tasks

**NOTE:**

* A Stage is a concept that is used for defining the Job.
* It makes job definition easy by acting as a container/template for specifications. Stage level settings can be set that get applied to all tasks of the stage. Thus no need to apply the same settings to each individual task separately.
  + Settings like environment variables, resource files, Node sizes etc. can be applied at Stage level
* Although a Job is defined in Stages, the **execution of the graph is not Stage based**. The graph execution progresses based on the Task dependencies. Thus Tasks from multiple stages could be running at the same time and are not blocked by Tasks of previous stages to finish. This is an important concept to note as this concept allows the graph to progress as tasks finish and therefore allow the graph to finish faster.

#### Basic terminology

This section outlines the basic terminology that is used by the system

The below diagram shows the various terminology. There is a detailed explanation after the diagram

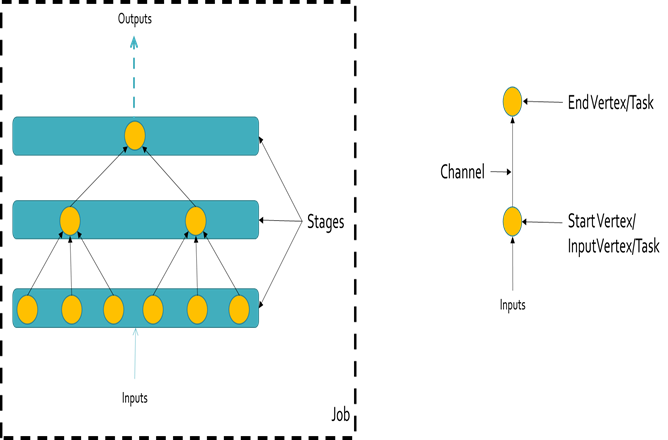


Figure 2- The Terminology used

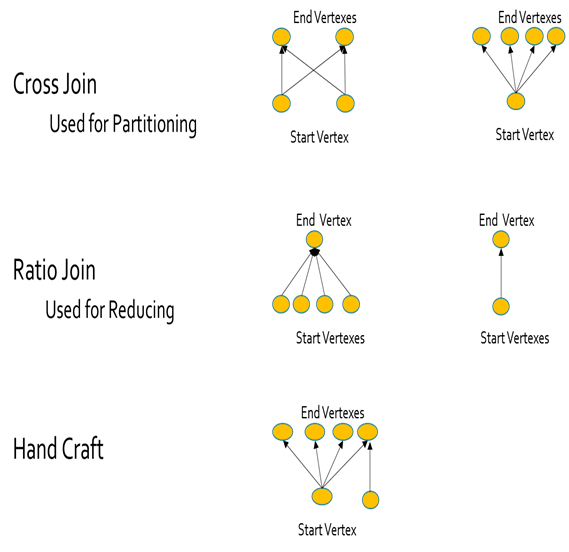
* **Job** – A scenario that requires a set of work to be completed.
  + E.g. Convert a million pdfs to text.
  + Render an animation movie using open source software packages
* **Graph** – The collection of distributed work that needs to be done to finish a Job. A Graph comprises of Stages.
* **Stage –** A container/template to define common properties for the vertexes (Tasks) of the Stage. A Stage is a collection of Tasks (1 or Many). Each Stage can have an associated Cluster.
* **Task/Vertex –** The smallest execution unit of work. Each Task at a minimum needs a command line. This is the command line that is used to execute the work for Task on a compute node. Optionally a Task/Vertex can have multiple input and multiple output channels.
* **Channel –** This is the data channel. When a vertex produces data, it is expressed as an output channel. The vertex consumes data via its input channel. A vertex can have multiple input channels to consume data from and multiple output channels to produce its data into various buckets/Partitions.
* **Join –** Are used to express dependencies. By joining an output channel of a given vertex as the input channel of another vertex a dependency between the two vertexes is created. More details are explained later
* **Cluster –** The execution environment for Tasks. The cluster provides the compute nodes on which the Tasks run
* **Compute Node –** The physical machine/VM where the task code executes.

#### High level features of the system.

Following are the high level features offered by the system

1. Ability to run any arbitrary code.
   * Each Task can run any arbitrary code. There is no need to implement any interface to be part of graph execution
2. Ability to define Job Stages
   * Job Stages makes it easy to express a graph of work
   * Job Stages act as container/Templates.
   * User can specify stage wide settings that get applied to every vertex of the stage.
   * Ability to associate a cluster with a stage.
3. Ability to run work on different sizes of compute nodes
   * By associating a cluster with a Stage, the user is able to run portions of the graph on different sized nodes.
   * Thus following types of scenarios get covered. Where for movie rendering, each scene might require smaller sized machine to render, while the final merge of rendered scenes into a movie might require a bigger machine (more CPU, memory, disk etc.)
4. Ability to run job on multiple clusters (Advanced scenario).
   * The user can implement the ComputeCluster interface to hook any cluster into the system and the tasks of the job can then run on that cluster.
5. Built in Joins – Cross Join, Ratio Join
   * Joins can be expressed between stages and the system creates the proper joins instead of user needing to hand author
   * NOTE: For special cases the system also supports hand authored joins
6. Built in Clusters
   * The system provides built in **Azure Batch Cluster**
     + Azure Batch is a system which allows running of tasks at massive scale and reliability. It is easy to use and there is very minimal compute cluster management. See [here](https://azure.microsoft.com/en-us/services/batch/) for more information on Azure Batch.
   * The system also provides built in **Fake Cluster**
     + The Fake Cluster can be used to test the graph on a single box
7. Built in Data Providers. Data Providers are used to Read and Write data used by the Tasks.
   * The system provides built in **Azure Blob DataProvider**
     + The user can easily create a Blob Channel using the built in Azure Blob Data Provider. The channel can be used for both reading and writing data to Azure Blobs
   * The system provides built in **Azure Batch Task File Provider**
     + The Azure Batch Task File Provider lets a task read data that was produced locally (stored on compute nodes disk) by another task running on a remote compute node

#### Types of Joins supported by the system

The diagram below outlines the various types of Joins supported by the system

## Project Organization

The system is composed of various projects. The various projects, their brief overview and how they are organized is given below. Detailed documentation on each of the classes and their methods/properties is available here(TODO: Add a link to the chm file here).

### DAGBuilderInterface.csproj (DAGBuilderInterface.dll)

This project defines the various interfaces. The DAGBuilder (explained in sections below) drives the graph execution by calling methods/properties of interfaces listed in this project. The interfaces themselves are documented in the code.

Among the other interfaces defined the major ones are the

* ComputeCluster Interface.
* ComputeTask Interface

These two interfaces are the main interfaces that represent a compute cluster. The interfaces themselves define the minimum set of methods that are required to be implemented. The Interface writer is free to add as many other methods and properties that are required that are specific to the cluster.

For e.g. in the implementation for **AzureBatchCluster,** thereare a bunch of methods related to initialization of the cluster, where the cluster allows the user to hook in pre-created Azure Batch Pools. Also the cluster initialization takes configuration settings, where AzureBatch jobs can be auto initialized etc.

These interfaces provide a facility for users to implement their own cluster. As an example the user can implement the interfaces to hook up an HDInsight cluster and run part of the job (tasks in a stage) as pig/hive queries.

### DAGBuilder.csproj (DAGBuilder.dll)

This project has all the classes and methods that the user would use to define and execute a graph of work.

**NOTE**: There are samples included in the project that can be used to look at sample code for executing a graph of work. Each sample directory also has readme.txt

The class names match closely with the terminology explained earlier. The main class names that are of interest are:

* JobGraph
* JMVertex
* JobStageManager
* JMVertex
* Channel

A general outline for a user to express a JobGraph and execute it would be:

1. Create a JobGraph object
2. Add one or more Stages to the Graph. Using the JobGraph.AddStage() API
3. Add Vertexes the Stages
4. Configure Stage wide resources, environment setting etc.
5. Override any specific settings that need to be over-ridden at task level
6. Define dependencies between tasks by Joining Stages (Either by using inbuilt join primitives, or handcrafting the joins)
7. Call JobGraph.Execute()

A sample code is given below: (Full samples are part of the project)

Machine generated alternative text:
try 
int numVertexesStgø 
int numVertexesStgI 
/ /Create Cluster 
4; 
cluster. "MyApp. config"); 
JobGraph graph new cluster); 
// Add Stage 
var Stageø graph (uint)numVertexesStgø); 
int counter a; 
foreach (var Vertex in Stageø.StageVertexes . Values) 
//AIternate vertexes in this stage sleep for 3øsec and Isec respectively 
string cmdLine 
"cmd /C 
if (counter numVertexesStgø / numVertexesStgI) 
cmdLine +2 
"ping -n 3B 127.ø.ø.1øø"; 
else 
cmdLine +2 
"ping -n 1 127.ø.ø.1øø"; 
Vertex. SetCowna nd Line ( cmd Line ) ; 
Vertex. Addoutputs (typeof (Channel), true, 
counteN*; 
//Adding the next stage 
I); 
//Each vertex gets I output (null channel) 
var Stagel graph 
(u int) n umVertexesStgI) ; 
foreach (var Vertex in StageI.StageVertexes . Values) 
/C dir" 
// Ratio join the vertexes from Stage 
// In this case 2 Stageø vertexes join 
graph . RatioJoinStages (Stageø, Stagel, 
//Execute the graph 
graph . Execute(); 
to Stage I. 
to I Stagel 
n umVertexes Stgø / n umVertexes Stg I , 
Channel Type. NULL); 
work done"); 
catch (Exception e) 
Console. WriteLine (e. ToString()) ; 

Figure 1Sample code to create and run a graph of work.

**What is the above code doing?**

The sample above is demonstrating how the vertex execution progresses independent of Stages (Thus showing that Stages are useful while expressing a graph, but execution does not depend on it).

In the above sample, a job with 2 stages is created. Stage0 and Stage1. Both the stages run on a single Cluster (The Cluster DataType is the AzureBatchCluster, which is not shown in the sample).

Stage0 has 4 vertexes/tasks and Stage2 has 2. The 1st two vertexes in Stage0 sleep for 30 sec, while the other 2 sleep for 1 sec each.

Stage1 has 2 vertexes. The 1st vertex depends on the two tasks from Stage 0 that sleep for 30 sec, while the other one depends on the other two tasks of Stage 0 that sleep for 1 sec.

The graph.RatioJoinStages is used to express the dependency of Stage1 vertexes on Stage0.

### AzureBatchCluster.csproj (AzureBatchCluster.dll)

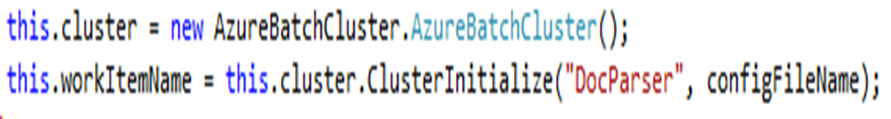
This project implements the Azure Batch Cluster.

As stated earlier, Azure Batch is a PaaS service provided by Microsoft that is built for highly parallel workload execution. It is massively scalable service that can schedule millions of tasks under a single job. It also provides resource management (compute nodes for execution of the work). The Azure batch service can manage 10 of thousands of nodes without the user having to maintain the nodes.

All the examples in the current project use either the AzureBatchCluster (to show real samples) or the FakeCluster (to show concepts by running them on a single machine).

**What azure batch cluster provides:**

* An azure batch cluster, that hides all the complexities from the users of keeping track of all the tasks, scheduling them, monitoring the tasks for completion/failures, reporting etc.
* Massively scalable, users can create and run jobs with 10’s of 1000’s of tasks on Azure Batch Cluster.
* Easy to initialize.



* Initialization either through setting in configuration file or via parameters.
* The configuration file, parameters schema is simple as shown below

/// <summary>

/// [Required]The Azure Batch Account Name to use

/// </summary>

public string AccountName { get; set; }

/// <summary>

/// [Required]The Key associated with the Azure Batch account.

/// </summary>

public string Key { get; set; }

/// <summary>

/// [Optional]The Name of the Job associated with the Azure Batch Cluster.

/// </summary>

public string JobName { get; set; }

/// <summary>

/// [Required]The name of the Pool(compute nodes) associated with the Cluster.

/// </summary>

public string PoolName { get; set; }

/// <summary>

/// [Required]The Azure Batch Cluster Url

/// </summary>

public string TenantUrl { get; set; }

/// <summary>

/// [Optional]Choice on how to initialize the Job associated with the cluster.

/// </summary>

private JobInitializationChoices JobInitChoice { get; set; }

/// <summary>

/// [Optional]The Storage Account to be used. If resources handling required.

/// The cluster provides helpers to automatically take care of resources

/// </summary>

public string StorageAccountName { get; set; }

/// <summary>

/// [Optional]The Key associated with the storage account

/// </summary>

public string StorageAccountKey { get; set; }

/// <summary>

/// [optional]Blob URL endpoint

/// </summary>

public string BlobUrl { get; set; }

### FakeCluster.csproj (FakeCluster.dll)

The FakeCluster provides a one box cluster for user. This is useful to test the code on a single box, without having to deploy code on to a real cluster. This is extremely useful during development process and debugging process. The FakeCluster implements all the interfaces specified in the DAGBuilderInterface project.

Since cluster hook up to the Graph is as simple as passing a parameter in the constructor, the user can start off working with FakeCluster and then move to using real cluster.

### DataProviders.csproj (DataProviders.dll)

This project implements the inbuilt data providers. Thus users can use the already written data providers to hook in with the channel code to read/write data that tasks produce.

This makes the task code very simple. The data providers provide simple read/write methods (both sync and async), that the task code can use.

All the data providers implement the IXCDataPartition interface. A IXCRangePartition is also provided. But not all implement it. It is not necessary for a data provider to implement those interfaces. But it is recommended that DataProviders implement those interfaces, so that users have a consistent way of using providers.

Following DataProviders are already implemented (Both sync and async methods exist).

1. AzureBatchFileDataProvider
   1. This provider allows the user to be able to read remote task data and write current task data to its local disk storage.
   2. Using this provider, users can read data that was produced by a task that ran on a different compute node.
   3. It also provides the users to be able to not only read a single file, but all the files in a given directory. Thus a task can read a remote tasks entire directory on content that it produced.
2. BlobDataProvider
   1. This provider allows users to write task outputs to a blob in Azure Storage.
   2. It also allows users to read blob data as well

### AzureBatchClientUtils.csproj (AzureBatchClientUtils.dll)

This project provides various utility functions. These will be enhanced more.

## Going Forward

The project allows users to run graph of work. The way the Channels and DataProviders are laid out, there are many possible things that can be enhanced here.

The intent is to keep doing regular updates. Also counting on contributions from the community.