MapReduce on Windows Azure

Daytona - Developer guide

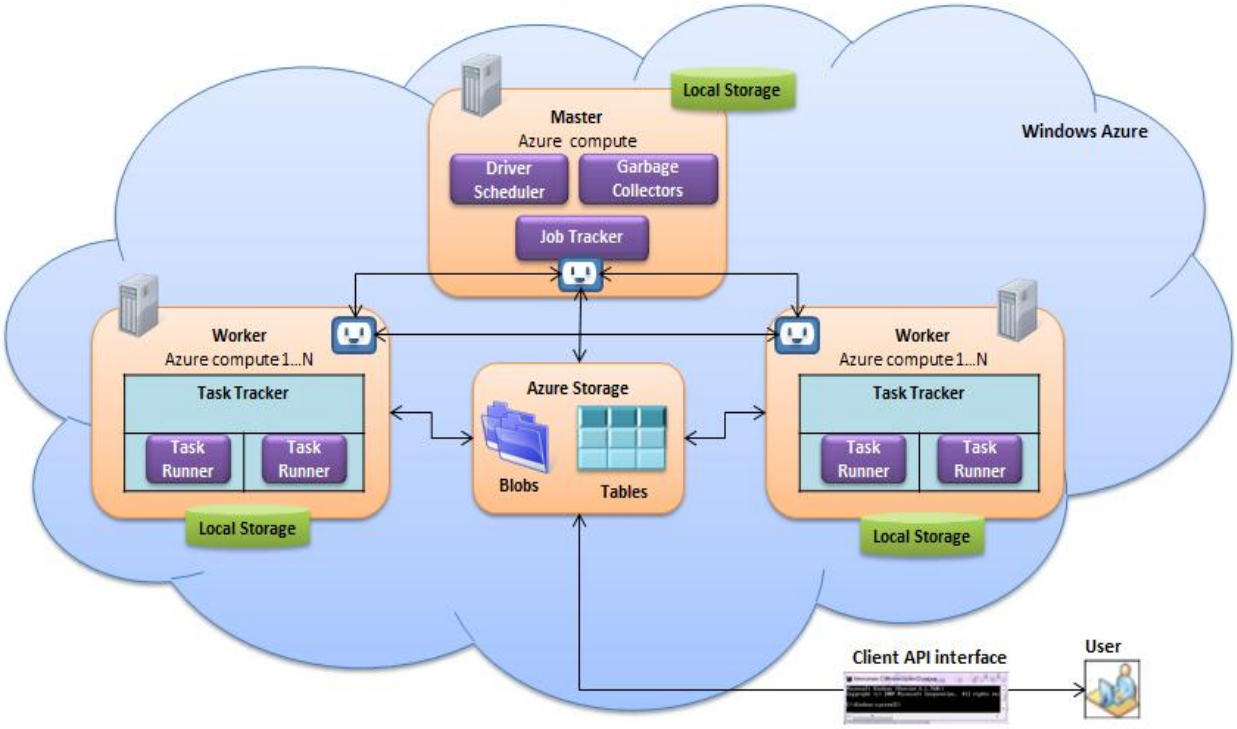
Table of Contents

1 Architectural Overview

The 'Daytona' runtime consists of two worker roles deployed within a single Windows Azure service. One of the worker roles is the 'Master' and the service runs only a single instance of it. Applications submit one or more jobs to the 'Master'. A Job consists of one or more tasks and a task can be one of the following:

- Map: A map task comprises of invoking a user-defined function which processes a key-value pair to generate a set of intermediate key-value pairs.
- Reduce: A reduce task comprises of invoking a user-defined function which processes all values associated with an intermediate key and generates a final set of key-value pairs.

The other role is 'Worker' and the service runs multiple instances of it. The 'Master' assigns one or more tasks to a 'Worker' instance. The Figure below depicts the execution flow of a 'job' submitted by a 'Daytona' application.



1.1 Terminology

- **Application:** This is an abstraction that contains one or more MapReduce computations composed by the user. An Application is composed of a (1) Package, (2) Controller and (3) Controller Arguments.
- **Package:** The collection of assemblies that compose this Application referenced by name.
- **Controller:** Takes arguments passed in from external client during application submittal and configures, creates and controls jobs on behalf of this application.
- **Job:** A single MapReduce computation. An Application can contain multiple MapReduce jobs. Each job is defined by a job configuration and a set of parameters.
- **Task** – Map or Reduce tasks of a Job.
- **Client Interface** – An interface that is used to submit applications to the Daytona runtime.

1.2 Overview

1. The Master picks an application that has been submitted using the client interface. It then creates the user-defined '*Controller*' of the application.
2. The '*Controller*' contains the logic for creating and submitting jobs for execution. Internally the runtime will generate the necessary Map/Reduce tasks per job and schedule them for execution on the workers. Typically, the '*Controller*' will leverage a user-defined '*data partitioner*' to split the input dataset into a number of '*data partitions*'. A map task is assigned to each data partition while the number of reduce tasks is left for the user to define based on the expected size of output generated by the map tasks.
3. Each worker instance is then assigned one or more map tasks by the Master.
4. A worker instance which has been assigned a map task reads the content of the corresponding '*data partition*', parses it into key-value pairs through a user-defined '*reader*'. Each key-value pair is then **sequentially** passed to the user-defined '*map function*' that produces a set of intermediate key-value pairs. The intermediate key-value pairs are partitioned among the reduce tasks specified by the user and serialized in the worker instance's memory or local file storage (Optionally, users can specify a 'combiner' to locally merge intermediate values for each intermediate key. Users can also control which keys go to which reduce task by specifying a '*key partitioner*'). The worker instance then notifies the master about completion of the task.
5. All map and reduce tasks get dispatched after application gets submitted. As soon as the master is notified of the completion of one of the map tasks, it sends signal to workers to start execution of reduce task.
6. A worker which has been assigned a reduce task is provided with the details for downloading its input data (intermediate key-value pairs generated by map tasks). The data exchange is done via inter-role communication. When the worker has downloaded all its input data, it de-serializes and groups it by the intermediate keys as multiple map tasks could have generated the same key. The keys are then sorted and **sequentially** passed one by one along with their values to the user-defined '*reduce function*'. The output of the reduce function is then persisted through a user-defined '*writer*'.
7. The runtime uses a data caching layer and an optimized scheduling mechanism to support iterative MapReduce jobs, i.e. execution of the same job in a loop within an iterative MapReduce computation.

2 Using Daytona

2.1 Overview

A typical user wishing to execute an algorithm on Daytona needs to author a map function and a reduce function. Once those are ready, the user will need to author a controller that will handle the details of job configuration, submission and management on the Master node within the Daytona runtime. The controller can be thought of as the user's embedded application control code that runs within the system. These three components are the key pieces needed to deploy an algorithm within the Daytona runtime. However, for a user to interact with the runtime, they will need to use the MapReduceClient class which is responsible for packaging the various components, uploading them to the runtime and submitting the application to be run on the backend. The following sections describe each component in further detail.

2.2 MapReduceClient

This class facilitates interacting with the Daytona runtime by providing methods for submitting an application, tracking its execution status. Optionally, this class can receive a set of string properties from the Controller of the application at the end of execution as a means of communicating results.

2.2.1 Properties

Property	*Property Type*	*Description*
Succeeded	bool	Indicates whether application execution succeeded or not.
FailReason	string	Details of failure if an error occurred during execution.
Results	Dictionary<string, string>	Name value collection of results optionally written by corresponding controller and passed back to client side.

2.2.2 Constructors

- ```
  MapReduceClient(string packageName,
                  string applicationName,
                  string controllerType
                  Dictionary<string, string> controllerArguments,
                  string assembliesDirectory)
  ```
  - **Description:** Uses the specified controller type for execution and passes the given collection of arguments to the controller. It looks for the application's assemblies in the provided path.
  - **Parameters:**

Parameter Name	*Description*
packageName	The name used to identify the assemblies package associated with this application. This allows an already existing package to be used or a new one to be uploaded as per the caller's choice in the Submit routine below.
applicationName	String representing the name for this application.
controllerType	The assembly qualified type name of the 'Controller' class which is to be used for this application instance.
controllerArguments	Collection of name/value pairs passed as arguments to the controller from the client side.
assembliesDirectory	Location of application's assemblies that should be packaged and uploaded to runtime.

- ```
  MapReduceClient(string packageName,
                  string applicationName,
                  string controllerType
                 Dictionary<string, string> controllerArguments)
  ```
  - **Description:** Uses the specified controller type for execution and passes the given collection of arguments to the controller. It looks for the application's assemblies in the currently executing path and packages the current executable along with the rest.
  - **Parameters:**

Parameter Name	*Description*
packageName	The name used to identify the assemblies package associated with this application. This allows an already existing package to be used or a new one to be uploaded as per the caller's choice in the Submit routine below.
applicationName	String representing the name for this application.
controllerType	The type name of the 'controllerType' class to be used for this application instance.
controllerArguments	Collection of name/value pairs passed as arguments to the controller from the client side.

- ```
  MapReduceClient(string packageName,
                  string applicationName,
                  string controllerType
  ```
  ```

- **Description:** Uses the specified controller type for execution and assumes the application's assemblies are in the current execution directory.
- **Parameters:**

Parameter Name	*Description*
packageName	The name used to identify the assemblies package associated with this application. This allows an already existing package to be used or a new one to be uploaded as per the caller's choice in the Submit routine below.
applicationName	String representing the name for this application.
controllerType	The type name of the 'controllerType' class to be used for this application instance.

2.2.3 Methods

- `void Submit(string masterConnectionString, bool createNewPackage = true)`
  - **Description:** Submits the application for execution by: (1) collecting the application assemblies into a package and (2) creating an application object with that package and finally (3) submitting that application object.
  - **Parameters:**

Parameter Name	*Description*
masterConnectionString	The connection string for the Daytona service Master storage account to which the assemblies package will be uploaded and the application instance submitted.
createNewPackage	True forces an existing package of the same name to be deleted and a new one uploaded in its place. False will either use an existing package if one is found or else create a new one.

  - **Return Value:** void.

- `bool WaitForCompletion(TimeSpan timeout, TimeSpan pollingInterval)`
  - **Description:** Blocking call that waits for application to complete by polling as per user specified interval.
  - **Parameters:**

Parameter Name	*Description*
Timeout	Time to wait for completion before timing out.
pollingInterval	Time to wait (sleep) between successive polls for application state.

  - **Return Value:** True if application concluded before timeout and false if it timed out.

- `void RequestAbort(string masterConnectionString)`
  - **Description:** Requests the runtime to abort an application submitted for execution by setting its state to 'AbortRequested'.
  - **Parameters:**

Parameter Name	*Description*
masterConnectionString	The connection string for the Daytona service Master storage account to which the assemblies package was uploaded and the application instance submitted.

  Return Value: void.

2.3 Controller

This is the core piece of code that gets invoked and executed, per application, within the Daytona runtime to manage job submissions on behalf of its corresponding application. The controller composes the jobs and orchestrates their flow within an application. For example, a controller of a typical MapReduce application may create one or more jobs and executes them one after the other to form a workflow (or a chain of jobs). Similarly, the controller of an iterative application may execute a job in a looping construct. Additionally, it can be used to perform any pre and post processing activities of an application or to evaluate convergence criteria in the case of an iterative application. Any metadata corresponding to the controller can be provided using 'Controller' attributes and this information can later be displayed on UI tools responsible for running the application. Input parameters required for submitting jobs from within the controller can be exposed as properties adorned by the

'ControllerParameter' attribute. The values for these properties can be set during the time of running the application. The base controller class exposes some commonly used parameters during job submission as shown below. All these parameters are defined as 'optional' so as to not impose any design constraints on the actual controller implementation.

2.3.1 Properties

Property	***Property Type***	***Description***
StorageConnectionString	string	Connection string required to connect to the Azure storage used for input/output.
CloudClient	CloudClient	It provides a wrapper to the different clients needed for accessing the cloud storage services using 'StorageConnectionString' if it is set.
InputDataFormat	string	A parameter where the caller can pass information about how the input data is stored.
InputDataLocation	string	A place for the caller to provide the location of the input data. Typically this will be a URI pointing to a blob in Azure storage or a container name for multiple input blobs.
OutputDataFormat	string	A parameter where the caller can pass information about how the output data is to be stored.
OutputDataLocation	string	A place for the caller to provide the location where the output data should be placed. Typically this will be a URI pointing to a blob in Azure storage or a container name for multiple output blobs.
RequestedNumberOfMappers	int	The number of mappers the caller has requested to use for running a job.
RequestedNumberOfReducers	int	The number of Reducers the caller has requested to use for running a job.
Results	Dictionary< string, string>	Used to commuincate a a collection of name/value pairs to the MapReduceClient after the execution of the application.

2.3.2 Methods

2.3.2.1 Run

- `abstract void Run();`
  - **Description:** Entry point of application execution within the Daytona runtime. This is the routine that is invoked by the Daytona runtime to effect the execution of this application instance and it is the place where job(s) are submitted. The developer needs to provide code for this routine.
  - **Parameters:** None.
  - **Return Value:** void.

2.4 JobConfiguration

JobConfiguration object captures the static configuration and parameters of a Job. For example, the Map class type, the Reduce class type and other parameters such as the exception policy are configured using the JobConfiguration. A single JobConfiguration can be shared between multiple jobs. It has the following properties.

Property	***Property Type***	***Description***
JobParameters	Dictionary<string, string>	Collection of key/value string pairs that are passed from the Controller to each task (Map or Reduce) for additional parameterization.
MapperType	Type	The class implementing the IMapper interface which holds the map logic for this application.
KeyPartitioner	Type	This class implements the IKeyPartitioner interface which is responsible for dividing the intermediate keys produced by the Map tasks into partitions each of which corresponds to a subsequent Reduce task.
MapOutputStorage	MapOutputStoreType	Enumerated type with options for specifying the media to use for serializeing the intermediate key/value pairs generated during the map phase. Currently

		only support 'MapOutputStoreType.Local'.
CombinerType	Type	The class implementing the IReducer interface. As an optional optimization, a reduction step referred to as the Combiner can be done on each Mapper after the Map stage to aggregate results before proceeding with the Reduce phase. The interface to the Combiner and Reducers is the same but different logic may be implemented for each phase.
ReducerType	Type	The class implementing the IReducer interface which holds the reduce logic for this application.
ExceptionPolicy	TaskExceptionPolicy	A class inheriting from TaskExceptionPolicy which implements the logic for task exception policy. This revolves around overriding the implementation of the GetAction routine which accepts an exception and determines what action to pursue.
JobTimeout	TimeSpan	The timeout period within which the submitted job must complete.
TaskSchedulingPolicy	*TaskSchedulingPolicy*	A task scheduling policy supported by the runtime. Currently there are three task schdduling policies supported by Daytona. • **Simple Iterative Policy** When a job is marked with simple iterative scheduling policy, the scheduler schedules tasks to the same workers where they ran in previous iterations. Although the dispatcher does not consult the *CacheIndexManger* when dispatching tasks under this policy, the cache will be engaged at the Workers when the records are read multiple times due to iterative execution. For example, under this policy, the static input data for K-Means clustering is read only once from blobs. • **DataLocalityAware Policy** Under this scheduling policy, the dispatcher consults the *CacheIndexManager* before dispatching tasks. The dispatcher schedules tasks to the Workers where the data is available. • **RoundRobin Policy** This is the default scheduling policy used by Daytona. It is also used for tasks that cannot be dispatched using any of the above two policies as well. Note: This is done irrespective of the outcome of the task (whether success or failure). If a task failed to execute, the corresponding data may not be available in the cache at the Worker leading to an inconsistency between the records at the Master and the Worker. However, due to the cache semantics, if a particular data item is not available in a Worker cache, it will retrieve them directly from blobs when a task request that data item in a subsequent execution.

2.5 Job

The Job object captures the dynamic configurations and parameters of a job and provides methods for running and closing a job. A brief description of its methods and properties are given below.

2.5.1 Properties

Property	***Property Type***	***Description***
DataPartitioner	object	An object of the user defined type implementing the IDataPartitioner interface which partitions the input to multiple partitions each of which maps 1-1 with a map task.
NoOfReduceTasks	int	The number of reduce tasks to create for this job.
RecordWriter	object	An object of the user defined type implementing the IRecordWriter interface which writes out the reduce output as per the application's requirements.
UseCacheIfAvailable	bool	Informs the runtime to engage the data cache for the supported input data partitioners and writers used in this job. The defult value for this property is "false". That means, even if the partitioner used for a job supports caching, it will not be engaged unless the user specifically set this property to "true".

2.5.2 Methods

2.5.2.1 Job

- `Job(JobConfiguration jobConfiguration, Controller controller)`
  - **Description:** Constructs a Job object using the provided JobConfiguration for the provided controller. Internally, the job hands over the execution of itself to the controller.
  - **Parameters:**

Parameter Name	*Description*
JobConfiguration	A job configuration capturing the static configurations and parameters.
Controller	A reference to the current controller in whch this job will run.

- `void Run()`
  - **Description:** Starts executing the Job as a MapReduce computation. This method internally performs the following four actions:
    - Validates both Job and JobConfiguration parameters.
    - Creates Map and Reduce tasks.
    - Submits the above tasks to the internal scheduler.
    - Waits until the job is complete. (This is a blocking call)
  - **Parameters:** None.
  - **Return Value:** void.

In an iterative application, the Run method can be called within a loop. Internally, it is optimized to minimize the validations and the task creations when the configurations of a job do not change.

- `void AddOrReplaceMapParameter(string key, object value)`
  - **Description:** Adds or replaces a parameter that will be shared across all the Map tasks. As we have mentioned above, since a job can be run multiple times in an iterative application, an existing parameter will be replaced with its new value when this method is called multiple times.

Parameter Name	*Description*
key	A string key corresponding to the parameter and used subsequently for lookup at the Map task.
value	Any serializable object.

  - **Return Value:** void.

- `void AddOrReplaceReduceParameter(string key, object value)`
  - **Description:** Adds or replaces a parameter that will be shared across all the Reduce tasks. As we have mentioned above, since a job can be run multiple times in an iterative application, an existing parameter will be replaced with its new value when this method is called multiple times.

Parameter Name	*Description*
key	string key corresponding to the parameter and used subsequently for lookup at the Reduce task.
value	Any serializable object.

  - **Return Value:** void.

- `IEnumerable<IReduceResult<ResultKey, ResultValue>> GetReduceOutputs<ResultKey, ResultValue>()`
  - **Description:** This method allows the user to access the reduce outputs at the controller. While it is not institutive to access the entire reduce output at the controller, this method is highly useful when the reduce outputs are small and the controller needs to access them to perform either some post processing or make decisions regarding an iteration condition.

Parameter Name	*Description*
`ResultKey`	A generic type that must match the Reduce output key type.
`ResultValue`	A generic type that must match the Reduce output value type.

  - **Return Value:** A collection of key value pairs for each Reducer. IReduceResult extends from IEnumberable<KeyValuePair>.
- void Close()
  - **Description:** Close and cleanup any job related resources held in the runtime. This explicit close behavior allows users to run a job multiple times before closing. At the end of the Controller's scope any none closed jobs will be closed by the runtime.
  - **Parameters:** None.
  - **Return Value:** void.

Type integrity must be maintained in the configuration across the various components. Below are the signatures of those components that constitute a job and an explanation of how type integrity must be maintained:

- public interface IDataPartitioner<K, V>
- public interface IRecordReader<K, V>
- public interface IMapper<KIN, VIN, KOUT, VOUT> where KOUT : IComparable<KOUT>
- public interface IReducer<KIN, VIN, KOUT, VOUT> where KIN : IComparable<KIN>
- public interface IRecordWriter<K, V>
- IEnumerable<IReduceResult<K, V>>GetReduceOutputs<K, V>()

1. The 'K' & 'V' defined in IDataPartitioner implementation must match those in the IRecordReader implementation and the 'KIN' & 'VIN' in the IMapper implementation.
2. 'KOUT' of IMapper must implement 'IComparable<KOUT>' so that it can be sorted. It must also either override 'Equals' and 'GetHashCode' or implement 'IEqualityComparer<KOUT>' to ensure that similar keys are grouped into one. **Note:** Primitive types and 'String' in .NET already implement 'IComparable' and override 'Equals' and 'GetHashCode', hence they need not be wrapped within new types. However, other reference types like 'System.Uri' don't implement 'IComparable' and hence have to be wrapped inside a new type or converted to 'String'.
3. 'KOUT' & 'VOUT' defined in 'IMapper' implementation must be the same as 'KIN' & 'VIN' defined in 'IReducer' implementation.
4. 'KOUT' and 'VOUT' defined in 'IReducer' implementation must be same as 'K' & 'V' defined in 'IRecordWriter' implementation.
5. 'K' and 'V' specified while calling the method 'GetReduceOutputs <>' must be same as 'K' and 'V' defined in 'IRecordWriter' implementation.

The following code snippet shows the controller of a sample 'WordCount' application.

```
[Controller(Name = "Word Count", Description = "Counts the number of times each unique word is found in
the input data.")]
    public sealed class WordCountController : Controller
    {
        public override void Run()
        {
            string outputContainerName = "word-count-output" + Guid.NewGuid().ToString("N");
            JobConfiguration jobConf = new JobConfiguration
            {
                MapperType = typeof(WordCountMapper),
                CombinerType = typeof(WordCountCombiner),
                ReducerType = typeof(WordCountReducer),
                MapOutputStorage = MapOutputStoreType.Local,
                KeyPartitioner = typeof(HashModuloKeyPartitioner<string, int>),
                ExceptionPolicy = new TerminateOnFirstException(),
                JobTimeout = TimeSpan.FromMinutes(10)
            };

            Job job = new Job(jobConf, this);
```

```
            job.DataPartitioner = new BlobContainerTextPartitioner(this.CloudClient, "word-count-input");
            job.RecordWriter = new BlobTextCsvWriter(this.CloudClient, outputContainerName);
            job.NoOfReduceTasks = 2;

            job.Run();
            this.Results.Add("OutputContainer", outputContainerName);
        }
    }
```

The following two pseudo code snippets show two different approaches to perform iterative computations using the Daytona runtime. In the following snippet, the job is invoked multiple times within a while loop with each iteration passing different parameters to Map tasks. The runtime will utilize the data cache to intelligently schedule the Map and Reduce tasks to the workers where they ran in the previous iteration. Please note the usage of specific task scheduling policy (“SimpleIterative” in this case) and the enabling of cache via the job parameter “UseCacheIfAvailable”.

```
public override void Run()
        {
            string outputContainerName = "simple-iterative -output" + Guid.NewGuid().ToString("N");
            JobConfiguration jobConf = new JobConfiguration
            {
                MapperType = typeof(KMeansClusteringMapper
),
                ReducerType = typeof(KMeansClusteringReducer
),
                TaskSchedulingPolicy = TaskSchedulingPolicy.SimpleIterative,
                ..
            };

            Job job = new Job(jobConf, this);
            job.DataPartitioner = new BlobContainerTextPartitioner(this.CloudClient, "clustering-input");
            job.RecordWriter = new BlobTextCsvWriter(this.CloudClient, outputContainerName);
            job.NoOfReduceTasks = 2;
            job.UseCacheIfAvailable = true;


            while(SomeCondtition())
            {
                job. AddOrReplaceMapParameter(“centroids”,currentCentroidsObject);
                job.Run();
                UpdateCentroids(job.  GetReduceOutputs<int, double>());
                CheckSomeCondition();
            }

            job.Close();
        }
```

Similar to the example above, the job is invoked multiple times within a while loop. However, unlike the previous example the data partitioner of the job is changed on each iteration. This is typically the case in iterative algorithms which consume the output of a previous iteration as the input to the current iteration. Please note the usage of specific task scheduling policy (“DataLocalityAware” in this case) and the enabling of cache via the job parameter “UseCacheIfAvailable”.

```
public override void Run()
        {
            string outputContainerName = "complext-iterative-output" + Guid.NewGuid().ToString("N");
            JobConfiguration jobConf = new JobConfiguration
            {
                MapperType = typeof(ComplexIterativeMapper),
                ReducerType = typeof(ComplexIterativeReducer),
                TaskSchedulingPolicy = TaskSchedulingPolicy.DataLocalityAware
,


                ..
            };

            Job job = new Job(jobConf, this);
            job.RecordWriter = new BlobTextCsvWriter(this.CloudClient, outputContainerName);
            job.NoOfReduceTasks = 2;
            job.UseCacheIfAvailable = true;

            while(SomeCondtition())
            {
                job.DataPartitioner = new BlobContainerTextPartitioner(this.CloudClient, "input-data");
                job. AddOrReplaceMapParameter("centroids",currentCentroidsObject);
                job.Run();
                UpdateInputDataForNextIteration(job.  GetReduceOutputs<string, string>());
                CheckSomeCondition();
            }

             job.Close();
        }
```

2.6 CloudClient

Provides a serializable wrapper for accessing different cloud clients {CloudBlobClient, CloudTableClient, CloudQueueClient } belonging to a particular storage account.

2.6.1 Properties

Property	***Property Type***	***Description***
Account	CloudStorageAccount	The cloud storage account which is going to be used.
BlobClient	CloudBlobClient	Client for accessing the blob service of the storage account being used.
TableClient	CloudTableClient	Client for accessing the table service of the storage account being used.
QueueClient	CloudQueueClient	Client for accessing the queue service of the storage account being used.
CacheContext	CacheContext	A file-ssytem based cache that can be used by this CloudClient for faster access to already downloaded data.
StorageConnectionString	String	The storage connection string.

2.7 Data Partitioning

Data partitioning provides the functionality for splitting the input dataset into partitions as well as parsing the contents of each partition to generate key-value pairs. Data partitioning can be provided by implementing the interface 'IDataPartitioner' and each split generated should implement the empty interface 'IDataPartition'.

2.7.1 IDataPartitioner

- Syntax: `public interface IDataPartitioner<K, V>`

2.7.1.1 Methods

- `IEnumerable<IDataPartition> GetPartitions()`

- **Description:** Contains the logic for partitioning which returns an enumerable collection of IDataPartition objects.
- **Parameters:** None.
- **Return Value:** Enumerable collection of IDataPartition object with one object corresponding to each intended partition.

- `IRecordReader<K, V> GetRecordReader(IDataPartition partition)`
  - **Description:** Returns an instance of an object implementing the IRecordReader interface and corresponding to the passed in partition.
  - **Parameters:**

Parameter Name	*Description*
partition	The partition to which the IRecordReader object is targeted.

  - **Return Value:** An object implementing the IRecordReader interface for the given 'K' & 'V' type values.

2.7.2 IRecordReader

- Syntax: `public interface IRecordReader<K, V>`

2.7.2.1 Methods

- `IEnumerable<KeyValuePair<K, V>> GetRecords()`
  - **Description:** Returns an enumerable collection of key/value pairs.
  - **Parameters:** None.
  - **Return Value:** A collection implementing IEnumerable over KeyValuePair for the given 'K' & 'V' type values.

2.7.3 Sample

The following code snippet shows a data partitioner for splitting an input container that contains blobs having text data in CSV format.

```
[Serializable]
    public class BlobContainerTextPartitioner : IDataPartitioner<int, string>
    {
        public string ContainerName { get; private set; }
        public CloudClient CloudClient { get; private set; }

        public BlobContainerTextPartitioner(CloudClient cloudClient, string containerName)
        {
            if (cloudClient == null)
            {
                throw new ArgumentNullException("cloudClient");
            }
            if (string.IsNullOrEmpty(containerName) || string.IsNullOrWhiteSpace(containerName))
            {
                throw new ArgumentNullException("containerName");
            }

            this.CloudClient = cloudClient;
            this.ContainerName = containerName;
        }

        public IEnumerable<IDataPartition> GetPartitions()
        {
            // Return one BlobPartition per blob in the container.
            CloudBlobContainer container = CloudClient.BlobClient.GetContainerReference(this.
                ContainerName);
            foreach (IListBlobItem blobItem in container.ListBlobs())
```

```
            {
                CloudBlob blob = container.GetBlobReference(blobItem.Uri.AbsoluteUri);
                blob.FetchAttributes();
                yield return new BlobTextPartition(blob, 0, blob.Properties.Length, true);
            }
        }

        public IRecordReader<int, string> GetRecordReader(IDataPartition partition)
        {
            return new BlobTextReader(this.CloudClient, partition as BlobTextPartition);
        }
    }
```

NOTE:

The types implementing 'IDataPartitioner' and 'IDataPartition' should either be marked as 'Serializable' or should implement IBinarySerializable. This is because the Daytona runtime needs to serialize these implementations while transferring them from controller to the workers across machine boundaries.

2.8 Caching Support

Daytona supports data caching for blob based data types. The caching is highly effective for iterative applications that operate on the same input data set and for applications that perform a pipeline or chain of MapReduce computations in which the one computation passes its output to the next computation. To use the caching feature an application needs to satisfy the following properties:

a. It must use a data partitioner that is IContextAware.
b. The data partitioner should produce data partitions that implement ICacheable interface.
c. The caching should be engaged in the job via "UseCacheIfAvailable" property set to true.
d. Use an appropriate task scheduler policy in the JobConfiguration.

A typical example would be the KMeansClustering sample application that uses BlobCsvPartitioner which supports caching. Then it is simply a matter of engaging cache and using the appropriate task scheduling policy to get the benefit of the data cache. For this application, we used "SimpleIterative" tasks scheduling policy so that a given task will be scheduled to the same VM on each iteration maximizing the input data reuse.

2.9 IMapper

The class implementing the IMapper interface provides the functionality for processing the key-value pairs generated from parsing each individual partition. It then generates intermediate key-value pairs as a result of that processing.

- Syntax: `public interface IMapper<KIN, VIN, KOUT, VOUT> where KOUT : IComparable<KOUT>`

2.9.1 Methods

- `void Configure(MapContext<KIN, VIN> context)`
  - **Description:** Invoked once before beginning calls to IMapper.Map for performing any initialization before processing begins. The MapContext is provided to facilitate initialization work.
  - **Parameters:**

Parameter Name	*Description*

context	Provides useful parameterization as well as a CloudClient for remote storage access. For specific details, review the context section later in the document.

- **Return Value:** void.

- `IEnumerable<KeyValuePair<KOUT, VOUT>> Map(KIN key, VIN value, MapContext<KIN, VIN> context)`
  - **Description:** This is the heart of the IMapper interface and is the routine that is invoked for each key/value pair to generate the intermediate set of key/value pairs. This method need not be 'thread-safe' as it will be invoked sequentially for each key/value pair returned by IRecordReader.
  - **Parameters:**

Parameter Name	*Description*
key	An instance of type KIN corresponding to a key for processing.
value	An instance of type VIN corresponding to a value for processing.
context	The same context object passed to the Configure method is passed to each and every invocation of IMapper.Map for parameterization support and optional remote storage access.

  - **Return Value:** An enumerable collection of KeyValuePair for the intermediate key – KOUT - and value – VOUT - types.

Following code snippet shows the mapper for 'WordCount' application.

```
public sealed class WordCountMapper : IMapper<int, string, string, int>
    {
        public IEnumerable<KeyValuePair<string, int>> Map(int key, string value, MapContext<int, string>
context)
        {
            foreach (string word in Regex.Split(value, "[^a-zA-Z0-9]", RegexOptions.Singleline).
                Where(tuple => !string.IsNullOrEmpty(tuple)))
            {
                yield return new KeyValuePair<string, int>(word, 1);
            }
        }

        public void Configure(MapContext<int, string> context)
        {
            throw new System.NotImplementedException();
        }
    }
```

2.10 Key-Value Serialization

The intermediate key-value pairs generated by the 'IMapper' implementation need to be serialized and stored either in memory or on local disk as map output. This output is then transferred as input to the 'IReducer' implementation where it is de-serialized and then processed. To perform efficient data transfer, serialization will always be done in binary format. The 'Daytona' runtime will categorize the Key-Value as one of the following types:

1. Primitive Type
2. String
3. Reference Type

Primitive types and strings will be serialized and de-serialized using a custom formatter so as to minimize the serialized binary data size. By default, reference types that contain the 'Serializable' attribute will be serialized and de-serialized using the in-built '*BinaryFormatter*'. However, the reference types can define their own binary formatting by choosing to implement the interface 'IBinarySerializable'. The reference type must however make sure to provide a parameter-less public constructor.

2.11 Combiner: IReducer

The Combiner performs an optional step immediately after the Map so as to reduce the input payload of Reduce tasks. The Combiner functionality can be provided by implementing the 'IReducer' interface. Typically, the same implementation is shared for reduce and combine. The specifics of the interface will be explained in the IReducer section below.

2.12 Key Partitioning: IKeyPartitioner

Key partitioning provides the functionality for splitting/distributing the intermediate key-value pairs generated after executing Map/Combine among the reduce tasks. The count of reduce tasks is provided to the partition through the Partition method. The method is invoked immediately after the results of Map invocation are available. Based on the value returned from the Partition method, each Map result is categorized into a bucket specific to a reduce task.

- Syntax: `public interface IKeyPartitioner<K, V>`

2.12.1 Methods

- `void Configure(Context context)`
  - **Description:** Invoked once before beginning calls to IKeyPartitioner.Partition for performing any initialization before processing begins. The Context is provided to facilitate initialization work by optionally provided additional parameters and/or access to remote storage.
  - **Parameters:**

Parameter Name	*Description*
context	Provides useful parameterization as well as a CloudClient for remote storage access. For specific details, review the context section later in the document.

  - **Return Value:** void.

- `int Partition(K key, V value, int numPartitions)`
  - **Description:** For the given key-value pair and given the count of available partitions (corresponding to the number of reducers) it returns a partition index based on some logic.
  - **Parameters:**

Parameter Name	*Description*
key	The intermediate key created by mapper/combiner stage.
value	The intermediate value corresponding to the prior key.
numPartitions	The total number of desired partitions (corresponds to the number of reducers requested for the job).

  - **Return Value:** An integer representing the index of the reduce task partition to which the key value pair belongs.

2.13 IReducer

The class implementing the IReducer interface provides the functionality to process all values associated with an intermediate key and generate a final key-value pair as output.

- Syntax: `public interface IReducer<KIN, VIN, KOUT, VOUT> where KIN : IComparable<KIN>`

2.13.1 Methods

- `void Configure(ReduceContext<KIN, VIN> context)`
  - **Description:** Invoked once per reduce task prior to processing (i.e. IReducer.Reduce calls) for performing initialization. Context is provided to facilitate with initialization work.

- **Parameters:**

Parameter Name	*Description*
context	Provides useful parameterization as well as a CloudClient for remote storage access. For specific details, review the context section later in the document.

- **Return Value:** void.

- `IEnumerable<KeyValuePair<KOUT, VOUT>> Reduce(KIN key, IEnumerable<VIN> values, ReduceContext<KIN, VIN> context)`
  - **Description:** This is the core of the IReducer interface and is invoked once per intermediate key with the collection of all the intermediate values corresponding to that key.
  - **Parameters:**

Parameter Name	*Description*
key	An instance of type KIN corresponding to an intermediate key value.
values	An collection of VIN values for all the intermediate values generated by mappers that correspond to the given key value.
context	The same context object passed to the Configure method is passed to each and every invocation of IReducer.Reduce for parameterization support and optional remote storage access.

  - **Return Value:** A collection of key/value pairs with each value representing the reduced intermediate collection of values for the given key.

The following code snippet shows the reducer for the 'WordCount' application:

```
public sealed class WordCountReducer : IReducer<string, int, string, string>
    {
        public IEnumerable<KeyValuePair<string, string>> Reduce(string key, IEnumerable<int> values,
ReduceContext<string, int> context)
        {
            return new KeyValuePair<string, string>[] { new KeyValuePair<string, string>(key, values.
                Sum().ToString()) };
        }

        public void Configure(ReduceContext<string, int> context)
        {
            throw new System.NotImplementedException();
        }
    }
```

2.14 Writing Output: IRecordWriter

The key-value pairs generated by the IReducer implementation need to be output in some fashion. If the output is large and needs to be persisted, then one of the Azure storage services or SQL Azure may be used. However, if the output is small and needs to be sent to the master for a final merge then it can be written to memory. The functionality to write the output may be provided by implementing the 'IRecordWriter' interface.

- Syntax: `public interface IRecordWriter<K, V>`

2.14.1 Methods

- `void Write(string outputPartition, IEnumerable<KeyValuePair<K, V>> records)`
  - **Description:** Using the provided outputPartition, it iterates over and writes out the provided collection of final key/value pairs.
  - **Parameters:**

Parameter Name	*Description*
outputPartition	A string description providing a reference to partition information for this writer.
records	A collection of final key/value pairs produced by one reducer.

- **Return Value:** void.

- `IReduceResult<K, V> GetResult()`
  - **Description:** This returns a collection of key/value pairs that are collected across all reduce tasks and communicated back to the Controller.
  - **Parameters:** None.
  - **Return Value:** A collection of Key/Value pairs over types K and V respectively.

Following code snippet shows the record writer for writing output in CSV format onto an Azure blob. If the output size is less than or equal to 4MB, then it's also stored inside a buffer which is sent back to the controller

```
[Serializable]
    public class BlobTextCsvWriter : IRecordWriter<string, string>
    {
        [NonSerialized]
        private const int BufferSize = 4 * 1024 * 1024; // 4MB
        [NonSerialized]
        private bool bufferFull;
        [NonSerialized]
        protected CloudBlob blob;
        [NonSerialized]
        private byte[] buffer;
        [NonSerialized]
        private int noOfBytesWrittenInBuffer;

        public CloudClient CloudClient { get; private set; }
        public string ContainerName { get; private set; }
        public string DirectoryName { get; private set; }

        public BlobTextCsvWriter(CloudClient cloudClient, string containerName, string directoryName =
null)
        {
            if (cloudClient == null)
            {
                throw new ArgumentNullException("cloudClient");
            }
            if (string.IsNullOrEmpty(containerName) || string.IsNullOrWhiteSpace(containerName))
            {
                throw new ArgumentNullException("containerName");
            }

            this.CloudClient = cloudClient;
            this.ContainerName = containerName;
            this.DirectoryName = directoryName;
        }

        public virtual void Write(string outputPartition, IEnumerable<KeyValuePair<string, string>>
records)
        {
            CloudBlobContainer container = this.CloudClient.BlobClient.GetContainerReference(this.
                ContainerName);
            container.CreateIfNotExist();

            if (!string.IsNullOrEmpty(DirectoryName))
            {
                blob = container.GetDirectoryReference(DirectoryName).GetBlobReference(outputPartition);
            }
            else
            {
                blob = container.GetBlobReference(outputPartition);
            }

            buffer = new byte[BufferSize];
            IEnumerator<KeyValuePair<string, string>> enumerator = records.GetEnumerator();
            WriteToBuffer(enumerator, buffer);
```

```
            using (Stream stream = blob.OpenWrite())
            {
                stream.Write(buffer, 0, noOfBytesWrittenInBuffer);
                if (bufferFull)
                {
                    using (StreamWriter sw = new StreamWriter(stream))
                    {
                        // Re-write the current record as the buffer is full.
                        WriteRecord(sw, enumerator.Current);
                        while (enumerator.MoveNext())
                        {
                            WriteRecord(sw, enumerator.Current);
                        }
                    }
                }
            }
        }

        public IReduceResult<string, string> GetResult()
        {
            if (bufferFull)
            {
                return new BlobTextResult(this.CloudClient, blob.Uri.AbsoluteUri);
            }
            else
            {
                byte[] localBuffer = new byte[noOfBytesWrittenInBuffer];
                Buffer.BlockCopy(buffer, 0, localBuffer, 0, noOfBytesWrittenInBuffer);
                return new BlobTextResult(this.CloudClient, blob.Uri.AbsoluteUri, localBuffer);
            }
        }

        private void WriteToBuffer(IEnumerator<KeyValuePair<string, string>> enumerator, byte[] buffer)
        {
            try
            {
                using (MemoryStream ms = new MemoryStream(buffer))
                {
                    using (StreamWriter sw = new StreamWriter(ms))
                    {
                        while (enumerator.MoveNext())
                        {
                            WriteRecord(sw, enumerator.Current);
                            sw.Flush();
                            noOfBytesWrittenInBuffer = (int)ms.Position;
                        }
                    }
                }
            }
            catch (NotSupportedException)
            {
                bufferFull = true;
            }
        }

        private void WriteRecord(StreamWriter sw, KeyValuePair<string, string> record)
        {
            sw.Write(record.Key);
            sw.Write(",");
            sw.Write(record.Value);
            sw.WriteLine();
        }
    }
```

NOTE:

The type implementing 'IRecordWriter' and 'IReduceResult' should either be marked as 'Serializable' or should implement IBinarySerializable. This is because the Daytona runtime needs to serialize these implementations while transferring them between workers and the controller across machine boundaries.

2.15 Context

The context class provides additional information to various components during the configuration stage. The Context class is specialized through MapContext and Reduce Context classes which inherit from it and provide additional context only relevant in those two components as explained below.

2.15.1 Context Properties

Property	***Property Type***	***Description***
TaskNo	int	A number identifying the associated task (map or reduce) for this context.
CloudClient	CloudClient	A class providing access to Azure storage services including Blob, Queue and Table. This can be leveraged by various components for persistent storage of values or to access such values.
JobParameters	Dictionary<string, string>	A collection of string key-value pairs that represent common input parameters which are to be used throughout the application. These parameters are passed in through the JobConfig parameters object.
TaskParameters	Dictionary<string, object>	A collection of string key-value pairs passed from the controller. At the Map task, the user will be able to access the parameters that she set using the Job object's AddOrReplaceMapParameter() method whereas at the Reduce task she will be able to access the parameters that she set using AddOrReplaceReduceParameter() method.
IterationCount	int	The job iteration count.

2.15.2 MapContext Properties

Property	***Property Type***	***Description***
RecordsEnumerator	IEnumerator<KeyValuePair<K, V>>	This is the enumerator that will be used by the Map task to iterate over all the input key/value pairs.

2.15.3 ReduceContext Class Properties

Property	***Property Type***	***Description***
CreatedForCombining	bool	A boolean indicating if this context instance is created for the local combining purpose. As one Reducer instance can also be applied at the Mapper side in order to aggreagate the local Map output, this property helps the user identify this scenario.
RecordsEnumerator	IEnumerator<KeyVAluePair<K, IEnumerable<V>>>	An enumerator that will be used by the Reduce task to iterate over all the intermediate key/values for this given reducer.

3 Running Applications

Reference to Research.MapReduce.Core must be added to compile Daytona applications. Additionally, reference to Research.MapReduce.Library can also be added to leverage some pre-built data partitioners, record readers etc.

3.1 Upload Input

Daytona Map-Reduce expects the input data as per Reader implementation with default option for Azure storage services. Input can be uploaded to Azure storage service using tool like cerebrata, cloudberry etc. SQL Management studio can be used to upload data to SQL Azure.

3.2 Submit

All the assemblies and its references are bundled in a package. Below code snippet shows how to submit the 'WordCount' application from a console app.

```
string masterConnectionString = ConfigurationManager.AppSettings["MasterConnectionString"];
            if (string.IsNullOrEmpty(masterConnectionString) ||
string.IsNullOrWhiteSpace(masterConnectionString))
            {
                throw new InvalidOperationException("MasterConnectionString configuration
missing.");
            }
MapReduceClient client = new MapReduceClient("word-count-" + Guid.NewGuid(),
                typeof(WordCountController).AssemblyQualifiedName);
 client.Submit(masterConnectionString, false);
Console.WriteLine("Done.");
```

The console app should have the following keys defined in the 'appSettings' of its configuration.

```
<appSettings>
    <add key="MasterConnectionString"
value="DefaultEndpointsProtocol=http;AccountName=<AccountName>;AccountKey=<AccountKey>"/>
<add key="StorageConnectionString"
value="DefaultEndpointsProtocol=http;AccountName=<AccountName>;AccountKey=<AccountKey>"/>
  </appSettings>
```

- **MasterConnectionString:** This should be the same as the storage account used by the Daytona runtime.
- **StorageConnectionString:** This value is automatically set as the value of the controller parameter 'StorageAccountString'. This is done by 'MapReduceClient' which looks for this specific key in the 'appSettings' and sends it as controller argument irrespective of whether the user has specified any controller arguments or not.

3.3 Abort

Once the application has been submitted to the Daytona runtime, the user may need to abort the submitted application run. Below code snippet shows method to abort the already submitted app.

```
MapReduceClient client = new MapReduceClient("word-count-" + Guid.NewGuid(),
                typeof(WordCountController).AssemblyQualifiedName);
client.Submit(masterConnectionString, false);
client.RequestAbort(masterConnectionString);
```

Once the abort is requested, runtime will kill the master job and send abort request to all workers.

3.4 Tracking and Result

Once the application is submitted, one needs to track the progress and get the final result after completion. Following code snippet shows how it is done.

```
Console.WriteLine("Waiting for completion...");
            bool concluded = client.WaitForCompletion(TimeSpan.FromMinutes(10),
TimeSpan.FromSeconds(5));
            if (concluded)
            {
                if (client.Succeeded)
                {
                    Console.WriteLine("Job completed successfully. Following are the results:");
```

```
                    foreach (KeyValuePair<string, string> result in client.Results)
                    {
                        Console.WriteLine("Result Name: '{0}' and Result Value: '{1}'",
result.Key, result.Value);
                    }
                }
                else
                {
                    Console.WriteLine("Job failed due to: \n {0}", client.FailReason);
                }
            }
            else
            {
                Console.WriteLine("Wait timeout elapsed.");
            }

            Console.WriteLine("Press <ENTER> to exit.");
            Console.ReadLine();
```

4 Library

Library (Research.MapReduce.Library) provides a set of commonly used implementation of data partitioning, key partitioning and output writing.

4.1 Data Partitioning

1. BlobContainerTextPartitioner: Partitions each blob (containing text data) in the specified Azure blob container as single partition (BlobTextPartition).
2. BlobTextPartition: Contains the corresponding blob URI and information to determine the amount of data that is to be read.
3. BlobTextPartitioner: Partitions a blob containing textual data into a specified number of partitions of type 'BlobTextPartition'.
4. BlobCsvPartitioner: Partitions a blob containing textual data in CSV format into specified number of partitions of type 'BlobCsvPartition'. (BlobCsvPartition extended from BlobTextPartition. BlobCsvPartition keeps additional information regarding the header of the CSV data).
5. BlobTextReader: For each line (text) in 'BlobTextPartition' it returns a key-value pair where key is line number and value is 'line'.
6. AzureTablePartitioner: Partitions a table into specified number of partitions of type AzureTablePartition.
7. AzureTablePartition: Represents a data partition containing records in a Windows Azure table.
8. KeyValuePartitioner: Partitions each key-value pair in the input list as a single partition of type KeyValuePartition.
9. KeyValuePartition: Represents a data partition containing a key-value pair.
10. SimpleBlobContainerTextPartitioner: Partitions each blob (containing text data) in the specified Windows Azure blob container as a single partition of type BlobTextPartition.

4.2 Key Partitioning

1. HashModuloKeyPartitioner: Partitions a given key based on the following calculation:
   Key's hash value (object.GetHashCode()) % No of reduce tasks

4.3 Output Writing

1. BlobTextCsvWriter: Writes the key and its values in CSV format onto the specified blob.