**Apache Crunch User Guide**

1. [Introduction to Crunch](http://crunch.apache.org/user-guide.html#intro)
   1. [Motivation](http://crunch.apache.org/user-guide.html#motivation)
   2. [Data Model and Operators](http://crunch.apache.org/user-guide.html#datamodel)
2. [Data Processing with DoFns](http://crunch.apache.org/user-guide.html#dataproc)
   1. [DoFns vs. Mapper and Reducer Classes](http://crunch.apache.org/user-guide.html#dovsmap)
   2. [Runtime Processing Steps](http://crunch.apache.org/user-guide.html#runproc)
   3. [Accessing Runtime MapReduce APIs](http://crunch.apache.org/user-guide.html#mrapis)
   4. [Configuring the Crunch Planner and MapReduce Jobs with DoFns](http://crunch.apache.org/user-guide.html#doplan)
   5. [Common DoFn Patterns](http://crunch.apache.org/user-guide.html#mapfn)
3. [Serializing Data with PTypes](http://crunch.apache.org/user-guide.html#serde)
   1. [Core PTypes](http://crunch.apache.org/user-guide.html#corept)
   2. [Extending PTypes](http://crunch.apache.org/user-guide.html#extendpt)
4. [Reading and Writing Data](http://crunch.apache.org/user-guide.html#rwdata)
   1. [A Note on Sources, Targets, and Hadoop APIs](http://crunch.apache.org/user-guide.html#notethis)
   2. [Sources](http://crunch.apache.org/user-guide.html#sources)
   3. [Targets](http://crunch.apache.org/user-guide.html#targets)
   4. [SourceTargets and Write Modes](http://crunch.apache.org/user-guide.html#srctargets)
   5. [Materializing Data Into the Client](http://crunch.apache.org/user-guide.html#materialize)
5. [Data Processing Patterns in Crunch](http://crunch.apache.org/user-guide.html#patterns)
   1. [groupByKey](http://crunch.apache.org/user-guide.html#gbk)
   2. [combineValues](http://crunch.apache.org/user-guide.html#aggregators)
   3. [Simple Aggregations](http://crunch.apache.org/user-guide.html#simpleagg)
   4. [Joining Data](http://crunch.apache.org/user-guide.html#joins)
      1. [Reduce-side Joins](http://crunch.apache.org/user-guide.html#reducejoin)
      2. [Map-side Joins](http://crunch.apache.org/user-guide.html#mapjoin)
      3. [Sharded Joins](http://crunch.apache.org/user-guide.html#shardedjoin)
      4. [Bloom Filter Joins](http://crunch.apache.org/user-guide.html#bloomjoin)
      5. [Cogroups](http://crunch.apache.org/user-guide.html#cogroups)
   5. [Sorting](http://crunch.apache.org/user-guide.html#sorting)
      1. [Standard and Reverse Sorting](http://crunch.apache.org/user-guide.html#stdsort)
      2. [Secondary Sorts](http://crunch.apache.org/user-guide.html#secsort)
   6. [Other Operations](http://crunch.apache.org/user-guide.html#otheropts)
      1. [Cartesian Products](http://crunch.apache.org/user-guide.html#cartesian)
      2. [Coalescing](http://crunch.apache.org/user-guide.html#shard)
      3. [Distinct](http://crunch.apache.org/user-guide.html#distinct)
      4. [Sampling](http://crunch.apache.org/user-guide.html#sampling)
      5. [Set Operations](http://crunch.apache.org/user-guide.html#sets)
      6. [Splits](http://crunch.apache.org/user-guide.html#splits)
   7. [Retaining objects within DoFns](http://crunch.apache.org/user-guide.html#objectreuse)
6. [Crunch for HBase](http://crunch.apache.org/user-guide.html#hbase)
7. [Managing Pipeline Execution](http://crunch.apache.org/user-guide.html#exec)
8. [The Different Pipeline Implementations (Properties and Configuration options)](http://crunch.apache.org/user-guide.html#pipelines)
   1. [MRPipeline](http://crunch.apache.org/user-guide.html#mrpipeline)
   2. [SparkPipeline](http://crunch.apache.org/user-guide.html#sparkpipeline)
   3. [MemPipeline](http://crunch.apache.org/user-guide.html#mempipeline)
9. [Unit Testing Pipelines](http://crunch.apache.org/user-guide.html#testing)

**Introduction to Crunch**

**Motivation**

Let's start with a basic question: why should you use *any* high-level tool for writing data pipelines, as opposed to developing against the MapReduce, Spark, or Tez APIs directly? Doesn't adding another layer of abstraction just increase the number of moving pieces you need to worry about, ala the [Law of Leaky Abstractions](http://www.joelonsoftware.com/articles/LeakyAbstractions.html)?

As with any decision like this, the answer is "it depends." For a long time, the primary payoff of using a high-level tool was being able to take advantage of the work done by other developers to support common MapReduce patterns, such as joins and aggregations, without having to learn and rewrite them yourself. If you were going to need to take advantage of these patterns often in your work, it was worth the investment to learn about how to use the tool and deal with the inevitable leaks in the tool's abstractions.

With Hadoop 2.0, we're beginning to see the emergence of new engines for executing data pipelines on top of data stored in HDFS. In addition to MapReduce, there are new projects like [Apache Spark](http://spark.incubator.apache.org) and [Apache Tez](http://tez.incubator.apache.org). Developers now have more choices for how to implement and execute their pipelines, and it can be difficult to know in advance which engine is best for your problem, especially since pipelines tend to evolve over time to process more data sources and larger data volumes. This choice means that there is a new reason to use a high-level tool for expressing your data pipeline: as the tools add support for new execution frameworks, you can test the performance of your pipeline on the new framework without having to rewrite your logic against new APIs.

There are many high-level tools available for creating data pipelines on top of Apache Hadoop, and they each have pros and cons depending on the developer and the use case. Apache Hive and Apache Pig define domain-specific languages (DSLs) that are intended to make it easy for data analysts to work with data stored in Hadoop, while Cascading and Apache Crunch develop Java libraries that are aimed at developers who are building pipelines and applications with a focus on performance and testability.

So which tool is right for your problem? If most of your pipeline work involves relational data and operations, than Hive, Pig, or Cascading provide lots of high-level functionality and tools that will make your life easier. If your problem involves working with non-relational data (complex records, HBase tables, vectors, geospatial data, etc.) or requires that you write lots of custom logic via user-defined functions (UDFs), then Crunch is most likely the right choice.

Although all of these projects have their own development philosophies and communities, they are all implemented using roughly the same set of patterns. The following table illustrates the relationship between these patterns across the various data pipeline projects that run on top of Apache Hadoop:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Concept | [Apache Hadoop MapReduce](http://hadoop.apache.org) | Apache Crunch | [Apache Pig](http://pig.apache.org) | [Apache Spark](http://spark.incubator.apache.org) | [Cascading](http://cascading.org) | [Apache Hive](http://hive.apache.org) | [Apache Tez](http://tez.incubator.apache.org) |
| Input Data | InputFormat | Source | LoadFunc | InputFormat | Tap (Source) | SerDe | Tez Input |
| Output Data | OutputFormat | Target | StoreFunc | OutputFormat | Tap (Sink) | SerDe | Tez Output |
| Data Container Abstraction | N/A | PCollection | Relation | RDD | Pipe | Table | Vertex |
| Data Format and Serialization | Writables | POJOs and PTypes | Pig Tuples and Schemas | POJOs and Java/Kryo Serialization | Cascading Tuples and Schemes | List<Object> and ObjectInspectors | Events |
| Data Transformation | Mapper, Reducer, and Combiner | DoFn | PigLatin and UDFs | Functions (Java API) | Operations | HiveQL and UDFs | Processor |

In the next section, we'll give a quick overview of Crunch's version of these abstractions and how they relate to each other before going into more detail about their usage in the rest of the guide.

**Data Model and Operators**

Crunch's Java API is centered around three interfaces that represent distributed datasets: [PCollection](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PCollection.html), [PTable](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PTable.html), and [PGroupedTable](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PGroupedTable.html).

A PCollection<T> represents a distributed, immutable collection of elements of type T. For example, we represent a text file as a PCollection<String> object. PCollection<T> provides a method, *parallelDo*, that applies a [DoFn](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/DoFn.html) to each element in the PCollection<T> in parallel, and returns a new PCollection<U> as its result.

A PTable<K, V> is a sub-interface of PCollection<Pair<K, V>> that represents a distributed, unordered multimap of its key type K to its value type V. In addition to the parallelDo operation, PTable provides a *groupByKey* operation that aggregates all of the values in the PTable that have the same key into a single record. It is the groupByKey operation that triggers the sort phase of a MapReduce job. Developers can exercise fine-grained control over the number of reducers and the partitioning, grouping, and sorting strategies used during the shuffle by providing an instance of the [GroupingOptions](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/GroupingOptions.html) class to the groupByKey function.

The result of a groupByKey operation is a PGroupedTable<K, V> object, which is a distributed, sorted map of keys of type K to an Iterable that may be iterated over exactly once. In addition to parallelDo processing via DoFns, PGroupedTable provides a *combineValues* operation that allows a commutative and associative [Aggregator](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Aggregator.html) to be applied to the values of the PGroupedTable instance on both the map and reduce sides of the shuffle. A number of common Aggregator<V> implementations are provided in the [Aggregators](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/fn/Aggregators.html) class.

Finally, PCollection, PTable, and PGroupedTable all support a *union* operation, which takes a series of distinct PCollections that all have the same data type and treats them as a single virtual PCollection.

All of the other data transformation operations supported by the Crunch APIs (aggregations, joins, sorts, secondary sorts, and cogrouping) are implemented in terms of these four primitives. The patterns themselves are defined in the [org.apache.crunch.lib](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/package-summary.html) package and its children, and a few of of the most common patterns have convenience functions defined on the PCollection and PTable interfaces.

Every Crunch data pipeline is coordinated by an instance of the [Pipeline](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Pipeline.html) interface, which defines methods for reading data into a pipeline via [Source](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Source.html) instances and writing data out from a pipeline to [Target](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Target.html) instances. There are currently three implementations of the Pipeline interface that are available for developers to use:

1. [MRPipeline](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/impl/mr/MRPipeline.html): Executes the pipeline as a series of MapReduce jobs.
2. [MemPipeline](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/impl/mem/MemPipeline.html): Executes the pipeline in-memory on the client.
3. [SparkPipeline](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/impl/spark/SparkPipeline.html): Executes the pipeline by converting it to a series of Spark pipelines.

**Data Processing with DoFns**

DoFns represent the logical computations of your Crunch pipelines. They are designed to be easy to write, easy to test, and easy to deploy within the context of a MapReduce job. Much of your work with the Crunch APIs will be writing DoFns, and so having a good understanding of how to use them effectively is critical to crafting elegant and efficient pipelines.

**DoFns vs. Mapper and Reducer Classes**

Let's see how DoFns compare to the Mapper and Reducer classes that you're used to writing when working with Hadoop's MapReduce API. When you're creating a MapReduce job, you start by declaring an instance of the Job class and using its methods to declare the implementations of the Mapper and Reducer classes that you want to use:

Configuration conf = ...;

Job job = new Job(conf);

job.setMapperClass(MyMapper.class);

job.setReducerClass(MyReducer.class);

The key thing to note here is that you're specifying the *classes* to use, not *instances* of those classes. The Job class does not have methods like this:

// Note: these functions don't exist on the Job API, it's just to illustrate a point.

job.setMapper(new MyMapper());

job.setReducer(new MyReducer());

The reason for this has to do with how the MapReduce framework communicates information about the Job from the client (that is, the machine where you run the hadoop jar command to kick off a job) to the Hadoop cluster. All of the information that you configure on the Job object is serialized to an XML file called job.xml that gets passed to the cluster, where it is used to configure the execution of your MapReduce job. This XML file is made up of key-value pairs, including key-value pairs that indicate which class to use for the Mapper and which to use for the Reducer. The MapReduce framework uses Java reflection to create instances of those classes to execute the logic of the job. This means that your Mapper and Reducer classes must have no-arg constructors, and that any parameters that the Mapper or Reducer classes need to perform their data processing must be communicated to them via the Configuration object associated with the Job.

Although the MapReduce framework's approach makes it easy for the framework to serialize data from the client to the cluster, it imposes some costs on MapReduce developers. First, you need to verify that any parameters your Mapper needs can be properly serialized into the Configuration object and parsed by the Mapper's initialize method. This is relatively simple to do for primitive types like integers and strings, but becomes more painful when you need to serialize collections and other complex types.

In contrast to MapReduce, Crunch uses Java serialization to serialize the contents of all of the DoFns in a pipeline definition into the job.xml file used to configure the MapReduce jobs that it executes. The abstract DoFn base class implements the java.io.Serializable interface, which means that all of the member variables of a DoFn must be either serializable or marked as transient. There is an [excellent overview of Java serializability](http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html) that is worth reviewing if you aren't familiar with it already.

If your DoFn needs to work with a class that does not implement Serializable and cannot be modified (for example, because it is defined in a third-party library), you should use the transient keyword on that member variable so that serializing the DoFn won't fail if that object happens to be defined. You can create an instance of the object during runtime using the initialize method described in the following section.

One place where the serializable DoFns can trip up new Crunch developers is when they specify in-line DoFns inside of methods of non-serializable outer classes. Although their pipelines compile fine and will execute locally with Crunch's MemPipeline, the MRPipeline or SparkPipeline versions will fail with Java's NotSerializableException. Let's look at the following example:

public class MyPipeline extends Configured implements Tool {

public static void main(String[] args) throws Exception {

ToolRunner.run(new Configuration(), new MyPipeline(), args);

}

public int run(String[] args) throws Exception {

Pipeline pipeline = new MRPipeline(MyPipeline.class, getConf());

PCollection<String> lines = pipeline.readTextFile(args[0]);

// This will throw a NotSerializableException!

PCollection<String> words = lines.parallelDo(new DoFn<String, String>() {

@Override

public void process(String line, Emitter<String> emitter) {

for (String word : line.split("\\s+")) {

emitter.emit(word);

}

}

}, Writables.strings());

words.count().write(To.textFile(args[1]));

pipeline.done();

}

}

Here, the inline DoFn that splits a line up into words is an inner class of MyPipeline. Inner classes contain references to their parent outer classes, so unless MyPipeline implements the Serializable interface, the NotSerializableException will be thrown when Crunch tries to serialize the inner DoFn. The other way to solve this problem while still using inner classes is to define the operations they perform inside of static methods on the class, e.g.:

public static PCollection<String> splitLines(PCollection<String> lines) {

// This will work fine because the DoFn is defined inside of a static method.

return lines.parallelDo(new DoFn<String, String>() {

@Override

public void process(String line, Emitter<String> emitter) {

for (String word : line.split("\\s+")) {

emitter.emit(word);

}

}

}, Writables.strings());

}

DoFn instances that are defined inline inside of static methods do not contain references to their outer classes, and so can be serialized regardless of whether the outer class is serializable or not. Using static methods to define your business logic in terms of a series of DoFns can also make your code easier to test by using in-memory PCollection implementations in your unit tests.

**Runtime Processing Steps**

After the Crunch runtime loads the serialized DoFns into its map and reduce tasks, the DoFns are executed on the input data via the following sequence:

1. First, the DoFn is given access to the TaskInputOutputContext implementation for the current task. This allows the DoFn to access any necessary configuration and runtime information needed before or during processing. Remember, DoFns can execute in either the map or reduce phase of a pipeline; your logic should not generally assume knowledge of which phase a particular DoFn will be executed in.
2. Next, the DoFn's initialize method is called. The initialize method is similar to the setup method used in the Mapper and Reducer classes; it is called before processing begins in order to enable any necessary initialization or configuration of the DoFn to be performed. For example, if we were making use of a non-serializable third-party library, we would create an instance of it here.
3. At this point, data processing begins. The map or reduce task will begin passing records in to the DoFn's process method, and capturing the output of the process method into an Emitter<T> that can either pass the data along to another DoFn for processing or serialize it as the output of the current processing stage.
4. Finally, after all of the records have been processed, the void cleanup(Emitter<T> emitter) method is called on each DoFn. The cleanup method has a dual purpose: it can be used to emit any state information that the DoFn wants to pass along to the next stage (for example, cleanup could be used to emit the sum of a list of numbers that was passed in to the DoFn's process method), as well as to release any resources or perform any other cleanup task that is appropriate once the task has finished executing.

**Accessing Runtime MapReduce APIs**

DoFns provide direct access to the TaskInputOutputContext object that is used within a given Map or Reduce task via the getContext method. There are also a number of helper methods for working with the objects associated with the TaskInputOutputContext, including:

* getConfiguration() for accessing the Configuration object that contains much of the detail about system and user-specific parameters for a given MapReduce job,
* progress() for indicating that a slow-running or computationally intensive MapReduce job is still making progress so that the MapReduce framework won't kill it,
* setStatus(String status) and getStatus for setting and retrieving task status information, and
* getTaskAttemptID() for accessing the current TaskAttemptID information.

DoFns also have a number of helper methods for working with [Hadoop Counters](http://codingwiththomas.blogspot.com/2011/04/controlling-hadoop-job-recursion.html), all named increment. Counters are an incredibly useful way of keeping track of the state of long-running data pipelines and detecting any exceptional conditions that occur during processing, and they are supported in both the MapReduce-based and in-memory Crunch pipeline contexts. You can retrieve the value of the Counters in your client code at the end of a MapReduce pipeline by getting them from the [StageResult](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PipelineResult.StageResult.html) objects returned by Crunch at the end of a run.

* increment(String groupName, String counterName) increments the value of the given counter by 1.
* increment(String groupName, String counterName, long value) increments the value of the given counter by the given value.
* increment(Enum<?> counterName) increments the value of the given counter by 1.
* increment(Enum<?> counterName, long value) increments the value of the given counter by the given value.

(Note that there was a change in the Counters API from Hadoop 1.0 to Hadoop 2.0, and thus we do not recommend that you work with the Counter classes directly in your Crunch pipelines (the two getCounter methods that were defined in DoFn are both deprecated) so that you will not be required to recompile your job jars when you move from a Hadoop 1.0 cluster to a Hadoop 2.0 cluster.)

**Configuring the Crunch Planner and MapReduce Jobs with DoFns**

Although most of the DoFn methods are focused on runtime execution, there are a handful of methods that are used during the planning phase before a pipeline is converted into MapReduce jobs. The first of these functions is float scaleFactor(), which should return a floating point value greater than 0.0f. You can override the scaleFactor method in your custom DoFns in order to provide a hint to the Crunch planner about how much larger (or smaller) an input data set will become after passing through the process method. If the groupByKey method is called without an explicit number of reducers provided, the planner will try to guess how many reduce tasks should be used for the job based on the size of the input data, which is determined in part by using the result of calling the scaleFactor method on the DoFns in the processing path.

Sometimes, you may know that one of your DoFns has some unusual parameter settings that need to be specified on any job that includes that DoFn as part of its processing. A DoFn can modify the Hadoop Configuration object that is associated with the MapReduce job it is assigned to on the client before processing begins by overriding the void configure(Configuration conf) method. For example, you might know that the DoFn will require extra memory settings to run, and so you could make sure that the value of the mapred.child.java.opts argument had a large enough memory setting for the DoFn's needs before the job was launched on the cluster.

**Common DoFn Patterns**

The Crunch APIs contain a number of useful subclasses of DoFn that handle common data processing scenarios and are easier to write and test. The top-level [org.apache.crunch](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/package-summary.html) package contains three of the most important specializations, which we will discuss now. Each of these specialized DoFn implementations has associated methods on the PCollection, PTable, and PGroupedTable interfaces to support common data processing steps.

The simplest extension is the [FilterFn](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/FilterFn.html) class, which defines a single abstract method, boolean accept(T input). The FilterFn can be applied to a PCollection<T> by calling the filter(FilterFn<T> fn) method, and will return a new PCollection<T> that only contains the elements of the input PCollection for which the accept method returned true. Note that the filter function does not include a PType argument in its signature, because there is no change in the data type of the PCollection when the FilterFn is applied. It is possible to compose new FilterFn instances by combining multiple FilterFns together using the and, or, and not factory methods defined in the [FilterFns](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/fn/FilterFns.html) helper class.

The second extension is the [MapFn](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/MapFn.html) class, which defines a single abstract method, T map(S input). For simple transform tasks in which every input record will have exactly one output, it's easy to test a MapFn by verifying that a given input returns a given output.

MapFns are also used in specialized methods on the PCollection and PTable interfaces. PCollection<V> defines the method PTable<K,V> by(MapFn<V, K> mapFn, PType<K> keyType) that can be used to create a PTable from a PCollection by writing a function that extracts the key (of type K) from the value (of type V) contained in the PCollection. The by function only requires that the PType of the key be given and constructs a PTableType<K, V> from the given key type and the PCollection's existing value type. PTable<K, V>, in turn, has methods PTable<K1, V> mapKeys(MapFn<K, K1> mapFn) and PTable<K, V2> mapValues(MapFn<V, V2>) that handle the common case of converting just one of the paired values in a PTable instance from one type to another while leaving the other type the same.

The final top-level extension to DoFn is the [CombineFn](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/CombineFn.html) class, which is used in conjunction with the combineValues method defined on the PGroupedTable interface. CombineFns are used to represent the associative operations that can be applied using the MapReduce Combiner concept in order to reduce the amount data that is shipped over the network during a shuffle.

The CombineFn extension is different from the FilterFn and MapFn classes in that it does not define an abstract method for handling data beyond the default process method that any other DoFn would use; rather, extending the CombineFn class signals to the Crunch planner that the logic contained in this class satisfies the conditions required for use with the MapReduce combiner.

Crunch supports many types of these associative patterns, such as sums, counts, and set unions, via the [Aggregator](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Aggregator.html) interface, which is defined right alongside the CombineFn class in the top-level org.apache.crunch package. There are a number of implementations of the Aggregator interface defined via static factory methods in the [Aggregators](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/fn/Aggregators.html) class. We will discuss Aggregators more in the section on [common MapReduce patterns](http://crunch.apache.org/user-guide.html#aggregators).

**Serializing Data with PTypes**

Every PCollection<T> has an associated PType<T> that encapsulates the information on how to serialize and deserialize the contents of that PCollection. PTypes are necessary because of [type erasure](http://docs.oracle.com/javase/tutorial/java/generics/erasure.html); at runtime, when the Crunch planner is mapping from PCollections to a series of MapReduce jobs, the type of a PCollection (that is, the T in PCollection<T>) is no longer available to us, and must be provided by the associated PType instance. When you're creating a new PCollection by using parallelDo against an existing PCollection, **the return type of your DoFn must match the given PType**:

public void runPipeline() {

PCollection<String> lines = ...;

// Valid

lines.parallelDo(new DoFn<String, Integer>() { ... }, Writables.ints());

// Compile error because types mismatch!

lines.parallelDo(new DoFn<String, Integer>() { ... }, Writables.longs());

}

Crunch supports two different *type families*, which each implement the [PTypeFamily](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/types/PTypeFamily.html) interface: one for Hadoop's [Writable interface](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/types/writable/WritableTypeFamily.html) and another based on [Apache Avro](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/types/avro/AvroTypeFamily.html). There are also classes that contain static factory methods for each PTypeFamily to allow for easy import and usage: one for [Writables](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/types/writable/Writables.html) and one for [Avros](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/types/avro/Avros.html).

The two different type families exist for historical reasons: Writables have long been the standard form for representing serializable data in Hadoop, but the Avro based serialization scheme is very compact, fast, and allows for complex record schemas to evolve over time. It's fine (and even encouraged) to mix-and-match PCollections that use different PTypes in the same Crunch pipeline (e.g., you could read in Writable data, do a shuffle using Avro, and then write the output data as Writables), but each PCollection's PType must belong to a single type family; for example, you cannot have a PTable whose key is serialized as a Writable and whose value is serialized as an Avro record.

**Core PTypes**

Both type families support a common set of primitive types (strings, longs, ints, floats, doubles, booleans, and bytes) as well as more complex PTypes that can be constructed out of other PTypes:

1. Tuples of other PTypes (pairs, trips, quads, and tuples for arbitrary N),
2. Collections of other PTypes (collections to create a Collection<T> and maps to return a Map<String, T>),
3. and tableOf to construct a PTableType<K, V>, the PType used to distinguish a PTable<K, V> from a PCollection<Pair<K, V>>.

The tableOf type is especially important to be familiar with, because it determines whether the return type of a parallelDo call on a PCollection will be a PTable instead of a PCollection, and only the PTable interface has the groupByKey method that can be used to kick off a shuffle on the cluster.

public static class IndicatorFn<T> extends MapFn<T, Pair<T, Boolean>> {

public Pair<T, Boolean> map(T input) { ... }

}

public void runPipeline() {

PCollection<String> lines = ...;

// Return a new PCollection<Pair<String, Boolean>> by using a PType<Pair<String, Boolean>>

PCollection<Pair<String, Boolean>> pcol = lines.parallelDo(new IndicatorFn<String>(),

Avros.pairs(Avros.strings(), Avros.booleans()));

// Return a new PTable<String, Boolean> by using a PTableType<String, Boolean>

PTable<String, Boolean> ptab = lines.parallelDo(new IndicatorFn<String>(),

Avros.tableOf(Avros.strings(), Avros.booleans()));

}

If you find yourself in a situation where you have a PCollection<Pair<K, V>> and you need a PTable<K, V>, the [PTables](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/PTables.html) library class has methods that will do the conversion for you.

Let's look at some more example PTypes created using the common primitive and collection types. For most of your pipelines, you will use one type family exclusively, and so you can cut down on some of the boilerplate in your classes by importing all of the methods from the Writables or Avros classes into your class:

// Import all of the PType factory methods from Avros

import static org.apache.crunch.types.avro.Avros.\*;

import org.apache.crunch.Pair;

import org.apache.crunch.Tuple3;

import org.apache.crunch.TupleN;

import java.nio.ByteBuffer;

import java.util.Collection;

import java.util.Map;

public class MyPipeline {

// Common primitive types

PType<Integer> intType = ints();

PType<Long> longType = longs();

PType<Double> doubleType = doubles();

// Bytes are represented by java.nio.ByteBuffer

PType<ByteBuffer> bytesType = bytes();

// A PTableType: using tableOf will return a PTable instead of a

// PCollection from a parallelDo call.

PTableType<String, Boolean> tableType = tableOf(strings(), booleans());

// Pair types:

PType<Pair<String, Boolean>> pairType = pairs(strings(), booleans());

PType<Pair<String, Pair<Long, Long>> nestedPairType = pairs(strings(), pairs(longs(), longs()));

// A triple

PType<Tuple3<Long, Float, Float>> tripType = trips(longs(), floats(), floats());

// An arbitrary length tuple-- note that we lose the generic type information

PType<TupleN> tupleType = tupleN(ints(), ints(), floats(), strings(), strings(), ints());

// A Collection type

PType<Collection<Long>> longsType = collections(longs());

// A Map Type-- note that the keys are always strings, we only specify the value.

PType<Map<String, Boolean>> mapType = maps(booleans());

// A Pair of collections

PType<Pair<Collection<String>, Collection<Long>>> pairColType = pairs(

collections(strings()),

collections(longs()));

}

Both type families also have a method named PType<T> records(Class<T> clazz) that can be used to create PTypes that support the common record format for each type family. For the WritableTypeFamily, the records method supports PTypes for implementations of the Writable interface, and for the AvroTypeFamily, the records method supports PTypes for implementations of Avro's IndexedRecord interface, which includes both Avro generic and specific records:

PType<FooWritable> fwType1 = Writables.records(FooWritable.class);

// The more obvious "writables" method also works.

PType<FooWritable> fwType = Writables.writables(FooWritable.class);

// For a generated Avro class, this works:

PType<Person> personType1 = Avros.records(Person.class);

// As does this:

PType<Person> personType2 = Avros.containers(Person.class);

// If you only have a schema, you can create a generic type, like this:

org.apache.avro.Schema schema = ...;

PType<Record> avroGenericType = Avros.generics(schema);

The [Avros](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/types/avro/Avros.html) class also has a reflects method for creating PTypes for POJOs using Avro's reflection-based serialization mechanism. There are a couple of restrictions on the structure of the POJO. First, it must have a default, no-arg constructor. Second, all of its fields must be Avro primitive types or collection types that have Avro equivalents, like ArrayList and HAshMap<String, T>. You may also have arrays of Avro primitive types.

// Declare an inline data type and use it for Crunch serialization

public static class UrlData {

// The fields don't have to be public, just doing this for the example.

double curPageRank;

String[] outboundUrls;

// Remember: you must have a no-arg constructor.

public UrlData() { this(0.0, new String[0]); }

// The regular constructor

public UrlData(double pageRank, String[] outboundUrls) {

this.curPageRank = pageRank;

this.outboundUrls = outboundUrls;

}

}

PType<UrlData> urlDataType = Avros.reflects(UrlData.class);

PTableType<String, UrlData> pageRankType = Avros.tableOf(Avros.strings(), urlDataType);

Avro reflection is a great way to define intermediate types for your Crunch pipelines; not only is your logic clear and easy to test, but the fact that the data is written out as Avro records means that you can use tools like Hive and Pig to query intermediate results to aid in debugging pipeline failures.

**Extending PTypes**

The simplest way to create a new PType<T> for a data object is to create a *derived* PType from one of the built-in PTypes from the Avro and Writable type families. If we have a base PType<S>, we can create a derived PType<T> by implementing an input MapFn<S, T> and an output MapFn<T, S> and then calling PTypeFamily.derived(Class<T>, MapFn<S, T> in, MapFn<T, S> out, PType<S> base), which will return a new PType<T>. There are examples of derived PTypes in the [PTypes](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/types/PTypes.html) class, including serialization support for protocol buffers, Thrift records, Java Enums, BigInteger, and UUIDs. The [crunch module](https://github.com/kevinweil/elephant-bird/tree/master/crunch) of [Twitter's ElephantBird](https://github.com/kevinweil/elephant-bird/) project also defines PTypes for working with protocol buffers and Thrift records that are serialized using ElephantBird's BinaryWritable<T>.

A common pattern in MapReduce programs is to define a Writable type that wraps a regular Java POJO. You can use derived PTypes to make it easy to work with the POJO directly in your DoFns, while still taking advantage of the custom serialization of your Writable implementation:

PType<Foo> fooType = Writables.derived(

Foo.class,

new MapFn<FooWritable, Foo>() {

public Foo map(FooWritable fw) { return fw.get(); }

},

new MapFn<Foo, FooWritable>() {

public FooWritable map(Foo foo) { return new FooWritable(foo); }

},

Writables.writable(FooWritable.class));

**Reading and Writing Data**

In the introduction to this user guide, we noted that all of the major tools for working with data pipelines on Hadoop include some sort of abstraction for working with the InputFormat<K, V> and OutputFormat<K, V> classes defined in the MapReduce APIs. For example, Hive includes SerDes, and Pig requires LoadFuncs and StoreFuncs. Let's take a moment to explain what functionality these abstractions provide for developers.

Most developers get started using one of the pipeline tools for Hadoop because they have a problem that requires joining data from multiple input files together. Although Hadoop provides support for reading from multiple InputFormat instances at different paths via the [MultipleInputs class](http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/lib/MultipleInputs.html), MultipleInputs does not solve some of the common problems that developers encounter when reading multiple inputs.

Most InputFormats and OutputFormats have static methods that are used to provide additional configuration information to a pipeline task. For example, the [FileInputFormat](http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapreduce/lib/input/FileInputFormat.html) contains a static void setMaxInputSplitSize(Job job, long size) method that can be used to control the split strategy of a given MapReduce job. Other InputFormats, like Elephant Bird's [MultiInputFormat](https://github.com/kevinweil/elephant-bird/blob/master/core/src/main/java/com/twitter/elephantbird/mapreduce/input/MultiInputFormat.java), contain configuration parameters that specify what type of protocol buffer or Thrift record the format should expect to read from the given paths.

All of these methods work by setting key-value pairs in the Configuration object associated with a data pipeline run. This is a simple and standard approach to including configuration information, but we run into a problem when we want to read from multiple paths that require different values to be associated with the same key, such as when we are joining protocol buffers, Thrift records, or Avro records that have different schemas. In order to properly read the contents of each input path, we need to be able to specify that certain key-value pairs in the Configuration object should only be applied to certain paths.

To handle the different Configuration settings needed by each of our different inputs from each other, Crunch uses the Source abstraction to wrap an InputFormat and its associated key-value pairs in a way that can be isolated from any other Sources that are used in the same job, even if those Sources have the same InputFormat. On the output side, the Target interface can be used in the same way to wrap a Hadoop OutputFormat and its associated key-value pairs in a way that can be isolated from any other outputs of a pipeline stage.

**A Note on Sources, Targets, and Hadoop APIs**

Crunch, like Hive and Pig, is developed against the [org.apache.hadoop.mapreduce](http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapreduce/package-summary.html) API, not the older [org.apache.hadoop.mapred](http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapred/package-summary.html) API. This means that Crunch Sources and Targets expect subclasses of the new [InputFormat](http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapreduce/InputFormat.html) and [OutputFormat](http://hadoop.apache.org/docs/r2.2.0/api/org/apache/hadoop/mapreduce/OutputFormat.html) classes. These new classes are not 1:1 compatible with the InputFormat and OutputFormat classes associated with the org.apache.hadoop.mapred APIs, so please be aware of this difference when considering using existing InputFormats and OutputFormats with Crunch's Sources and Targets.

**Sources**

Crunch defines both Source<T> and TableSource<K, V> interfaces that allow us to read an input as a PCollection<T> or a PTable<K, V>. You use a Source in conjunction with one of the read methods on the Pipeline interface:

Pipeline pipeline = ...;

PCollection<String> lines = pipeline.read(new TextFileSource(new Path("/user/crunch/text"), Writables.strings()));

PTable<Long, ByteBuffer> binaryTable = pipeline.read(new SeqFileTableSource(new Path("/user/crunch/binary"),

Writables.tableOf(Writables.longs(), Writables.bytes())));

Note that Sources usually require a PType to be specified when they are created. The [From](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/From.html) class provides a number of factory methods for literate Source creation:

// Note that we are passing a String "/user/crunch/text", not a Path.

PCollection<String> fromLines = pipeline.read(From.textFile("/user/crunch/text"));

PTable<Long, ByteBuffer> fromBinaryTable = pipeline.read(From.sequenceFile("/user/crunch/binary",

Writables.longs(),

Writables.bytes()));

The From class also provides a method named formattedFile for creating a TableSource from an existing InputFormat. Let's say that we wanted to create a Source for Hadoop's [NLineInputFormat](https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/lib/input/NLineInputFormat.html):

TableSource<Long, String> nLineSource = From.formattedFile("/user/crunch/nline", NLineInputFormat.class,

Writables.longs(), Writables.strings());

The NLineInputFormat class defines a configuration parameter named LINES\_PER\_MAP that controls how the input file is split. We can set the value of this parameter via the Source interface's inputConf(String key, String value) method:

nLineSource.inputConf(NLineInputFormat.LINES\_PER\_MAP, "5");

Because we specified this parameter on the Source instance and not the Configuration object directly, we can process multiple different files using the NLineInputFormat and not have their different settings conflict with one another.

Here is a table of commonly used Sources and their associated usage information:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Type | Source | Returns | From method (if available) | Notes |
| Text | [org.apache.crunch.io.text.TextFileSource](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/text/TextFileSource.html) | PCollection<String> | textFile | Works for both TextInputFormat and AvroUtf8InputFormat |
| Sequence | [org.apache.crunch.io.seq.SeqFileTableSource](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/seq/SeqFileTableSource.html) | PTable<K, V> | sequenceFile | Also has a [SeqFileSource](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/seq/SeqFileSource.html) which reads the value and ignores the key. |
| Avro | [org.apache.crunch.io.avro.AvroFileSource](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/avro/AvroFileSource.html) | PCollection<V> | avroFile | No PTable analogue for Avro records. |
| Parquet | [org.apache.crunch.io.parquet.AvroParquetFileSource](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/parquet/AvroParquetFileSource.html) | PCollection<V> | N/A | Reads Avro records from a parquet-formatted file; expects an Avro PType. |

**Targets**

Crunch's Target interface is the analogue of Source<T> for OutputFormats. You create Targets for use with the write method defined on the Pipeline interface:

Pipeline pipeline = ...;

PCollection<String> lines = ...;

pipeline.write(lines, new TextFileTarget(new Path("/user/crunch/textout")));

The PCollection interface also declares a PCollection<T> write(Target t) method that can be used to write out a PCollection:

// Equivalent to the above

lines.write(new TextFileTarget(new Path("/user/crunch/textout")));

Just as the Source interface has the From class of factory methods, Target factory methods are defined in a class named [To](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/To.html) to enable literate programming:

lines.write(To.textFile("/user/crunch/textout"));

You can create a Target for a custom OutputFormat using the formattedFile methods on the To class, just as you did for InputFormats with the From class. Let's create a Target for the SequenceFileAsBinaryOutputFormat:

Target binaryTarget = To.formattedFile("/user/crunch/binary", SequenceFileAsBinaryOutputFormat.class);

We can use the outputConf(String key, String value) method on the Target interface to set the additional configuration parameters that this Target needs:

// We can chain these methods together because outputConf() returns its Target.

binaryTarget

.outputConf(SequenceFileAsBinaryOutputFormat.KEY\_CLASS, LongWritable.class.toString())

.outputConf(SequenceFileAsBinaryOutputFormat.VALUE\_CLASS, Text.class.toString());

Here is a table of commonly used Targets:

|  |  |  |  |
| --- | --- | --- | --- |
| Output Type | Target | To method (if available) | Notes |
| Text | [org.apache.crunch.io.text.TextFileTarget](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/text/TextFileTarget.html) | textFile | Will write out the string version of whatever it's given, which should be text. See also: Pipeline.writeTextFile. |
| Sequence | [org.apache.crunch.io.seq.SeqFileTarget](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/seq/SeqFileTarget.html) | sequenceFile | Works on both PCollection and PTable. |
| Avro | [org.apache.crunch.io.avro.AvroFileTarget](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/avro/AvroFileTarget.html) | avroFile | Treats PTables as PCollections of Pairs. |
| Parquet | [org.apache.crunch.io.parquet.AvroParquetFileTarget](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/parquet/AvroParquetFileTarget.html) | N/A | Writes Avro records to parquet-formatted files; expects an Avro PType. |

**SourceTargets and Write Modes**

The SourceTarget<T> interface extends both the Source<T> and Target interfaces and allows a Path to act as both a Target for some PCollections as well as a Source for others. SourceTargets are convenient for any intermediate outputs within your pipeline. Just as we have the factory methods in the From and To classes for Sources and Targets, factory methods for SourceTargets are declared in the [At](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/At.html) class.

In many pipeline applications, we want to control how any existing files in our target paths are handled by Crunch. For example, we might want the pipeline to fail quickly if an output path already exists, or we might want to delete the existing files and overwrite them with our new outputs. We might also want to use an output path as a *checkpoint* for our data pipeline. Checkpoints allow us to specify that a Path should be used as the starting location for our pipeline execution if the data it contains is newer than the data in the paths associated with any upstream inputs to that output location.

Crunch supports these different output options via the [WriteMode](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Target.WriteMode.html) enum, which can be passed along with a Target to the write method on either PCollection or Pipeline. Here are the supported WriteModes for Crunch:

PCollection<String> lines = ...;

// The default option is to fail if the output path already exists.

lines.write(At.textFile("/user/crunch/out"), WriteMode.DEFAULT);

// Delete the output path if it already exists.

lines.write(At.textFile("/user/crunch/out"), WriteMode.OVERWRITE);

// Add the output of the given PCollection to the data in the path

// if it already exists.

lines.write(At.textFile("/user/crunch/out"), WriteMode.APPEND);

// Use this directory as a checkpoint location, which requires that this

// be a SourceTarget, not just a Target:

lines.write(At.textFile("/user/crunch/out"), WriteMode.CHECKPOINT);

**Materializing Data Into the Client**

In many analytical applications, we need to use the output of one phase of a data pipeline in order to configure subsequent pipeline stages. For example, many machine learning applications require that we iterate over a dataset until some convergence criteria is met. Crunch provides API methods that make it possible to materialize the data from a PCollection and stream the resulting data into the client.

The key method here is PCollection's appropriately named Iterable<T> materialize() method. The Iterable that is returned by this method contains logic for reading the data referred to by the PCollection into the client using the PType for that PCollection. Data is read in from the filesystem in a streaming fashion, so there is no requirement for the contents of the PCollection to fit in memory for it to be read into the client using materialization.

If the data in the PCollection is the result of a series of Crunch operations (parallelDo, groupByKey, etc.), then we will need to run a pipeline in order to write the data in that PCollection to the filesystem so that it may be read into the client. This is done in a lazy fashion, so that we will not actually kick off the pipeline tasks required to create the data until the iterator() method on the Iterable returned by Iterable<T> materialize() is called. You can also cause a materialized data set to be created by calling one of the run methods on the Pipeline interface that are used to manage overall pipeline execution. This means that you can instruct Crunch to materialize multiple PCollections and have them all created within a single Pipeline run.

If you ask Crunch to materialize a PCollection that is returned from Pipeline's PCollection<T> read(Source<T> source) method, then no MapReduce job will be executed if the given Source implements the [ReadableSource](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/ReadableSource.html) interface. If the Source is not readable, then a map-only job will be executed to map the data to a format that Crunch knows how to read from disk.

Sometimes, the output of a Crunch pipeline will be a single value, such as the number of elements in a PCollection. In other instances, you may want to perform some additional client-side computations on the materialized contents of a PCollection in a way that is transparent to users of your libraries. For these situations, Crunch defines a [PObject](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PObject.html) interface that has an associated V getValue() method. PCollection's PObject<Long> length() method returns a reference to the number of elements contained in that PCollection, but the pipeline tasks required to compute this value will not run until the Long getValue() method of the returned PObject is called.

**Data Processing Patterns in Crunch**

This section describes the various data processing patterns implemented in Crunch's library APIs, which are in the [org.apache.crunch.lib](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/package-summary.html) package.

**groupByKey**

Most of the data processing patterns described in this section rely on PTable's groupByKey method, which controls how data is shuffled and aggregated by the underlying execution engine. The groupByKey method has three flavors on the PTable interface:

1. groupByKey(): A simple shuffle operation, where the number of partitions of the data will be determined by the Crunch planner based on the estimated size of the input data,
2. groupByKey(int numPartitions): A shuffle operation where the number of partitions is explicitly provided by the developer based on some knowledge of the data and the operation performed.
3. groupByKey(GroupingOptions options): Complex shuffle operations that require custom partitions and comparators.

The [GroupingOptions](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/GroupingOptions.html) class allows developers to exercise precise control over how data is partitioned, sorted, and grouped by the underlying execution engine. Crunch was originally developed on top of MapReduce, and so the GroupingOptions APIs expect instances of Hadoop's [Partitioner](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Partitioner.html) and [RawComparator](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/io/RawComparator.html) classes in order to support partitions and sorts. That said, Crunch has adapters in place so that these same classes may also be used with other execution engines, like Apache Spark, without a rewrite.

The GroupingOptions class is immutable; to create a new one, take advantage of the [GroupingOptions.Builder](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/GroupingOptions.Builder.html) implementation.

GroupingOptions opts = GroupingOptions.builder()

.groupingComparatorClass(MyGroupingComparator.class)

.sortComparatorClass(MySortingComparator.class)

.partitionerClass(MyPartitioner.class)

.numReducers(N)

.conf("key", "value")

.conf("other key", "other value")

.build();

PTable<String, Long> kv = ...;

PGroupedTable<String, Long> kv.groupByKey(opts);

GroupingOptions, just like Sources and Targets, may have additional Configuration options specified that will only be applied to the job that actually executes that phase of the data pipeline.

**combineValues**

Calling one of the groupByKey methods on PTable returns an instance of the PGroupedTable interface. PGroupedTable provides a combineValues that can be used to signal to the planner that we want to perform associative aggregations on our data both before and after the shuffle.

There are two ways to use combineValues: you can create an extension of the [CombineFn](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/CombineFn.html) abstract base class, or you can use an instance of the [Aggregator](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Aggregator.html) interface. Of the two, an Aggregator is probably the way you want to go; Crunch provides a number of [Aggregators](http://crunch.apache.org/0.10.0/org/apache/crunch/fn/Aggregators.html), and they are a bit easier to write and compose together. Let's walk through a few example aggregations:

PTable<String, Double> data = ...;

// Sum the values of the doubles for each key.

PTable<String, Double> sums = data.groupByKey().combineValues(Aggregators.SUM\_DOUBLES());

// Find the ten largest values for each key.

PTable<String, Double> maxes = data.groupByKey().combineValues(Aggregators.MAX\_DOUBLES(10));

PTable<String, String> text = ...;

// Get a random sample of 100 unique elements for each key.

PTable<String, String> samp = text.groupByKey().combineValues(Aggregators.SAMPLE\_UNIQUE\_ELEMENTS(100));

We can also use Aggregators together in combination to build more complex aggregations, like to compute the average of a set of values:

PTable<String, Double> data = ...;

// Create an auxillary long that is used to count the number of times each key

// appears in the data set.

PTable<String, Pair<Double, Long>> c = data.mapValues(

new MapFn<Double, Pair<Double, Long>>() { Pair<Double, Long> map(Double d) { return Pair.of(d, 1L); } },

pairs(doubles(), longs()));

// Aggregate the data, using a pair of aggregators: one to sum the doubles, and the other

// to sum the auxillary longs that are the counts.

PTable<String, Pair<Double, Long>> agg = c.groupByKey().combineValues(

Aggregators.pairAggregator(Aggregators.SUM\_DOUBLES(), Aggregators.SUM\_LONGS()));

// Compute the average by dividing the sum of the doubles by the sum of the longs.

PTable<String, Double> avg = agg.mapValues(new MapFn<Pair<Double, Long>, Double>() {

Double map(Pair<Double, Long> p) { return p.first() / p.second(); }

}, doubles());

**Simple Aggregations**

Many of the most common aggregation patterns in Crunch are provided as methods on the PCollection interface, including count, max, min, and length. The implementations of these methods, however, are in the [Aggregate](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Aggregate.html) library class. The methods in the Aggregate class expose some additional options that you can use for performing aggregations, such as controlling the level of parallelism for count operations:

PCollection<String> data = ...;

PTable<String, Long> cnt1 = data.count();

PTable<String, Long> cnt2 = Aggregate.count(data, 10); // use 10 reducers

PTable has additional aggregation methods, top and bottom, that can be used to get the most (or least) frequently occuring key-value pairs in the PTable based on the value, which must implement Comparable. To count up all of the elements in a set and then get the 20 most frequently occuring elements, you would run:

PCollection<String> data = ...;

PTable<String, Long> top = data.count().top(20);

**Joining Data**

Joins in Crunch are based on equal-valued keys in different PTables. Joins have also evolved a great deal in Crunch over the lifetime of the project. The [Join](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Join.html) API provides simple methods for performing equijoins, left joins, right joins, and full joins, but modern Crunch joins are usually performed using an explicit implementation of the [JoinStrategy](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/join/JoinStrategy.html) interface, which has support for the same rich set of joins that you can use in tools like Apache Hive and Apache Pig.

All of the algorithms discussed below implement the JoinStrategy interface, which defines a single join method:

PTable<K, V1> one = ...;

PTable<K, V2> two = ...;

JoinStrategy<K, V1, V2> strategy = ...;

PTable<K, Pair<V1, V2>> joined = strategy.join(one, two, JoinType);

The [JoinType](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/join/JoinType.html) enum determines which kind of join is applied: inner, outer, left, right, or full. In general, the smaller of the two inputs should be the left-most argument to the join method.

Note that the values of the PTables you join should be non-null. The join algorithms in Crunch use null as a placeholder to represent that there are no values for a given key in a PCollection, so joining PTables that contain null values may have surprising results. Using a non-null dummy value in your PCollections is a good idea in general.

**Reduce-side Joins**

Reduce-side joins are handled by the [DefaultJoinStrategy](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/join/DefaultJoinStrategy.html). Reduce-side joins are the simplest and most robust kind of joins in Hadoop; the keys from the two inputs are shuffled together to the reducers, where the values from the smaller of the two collections are collected and then streamed over the values from the larger of the two collections. You can control the number of reducers that is used to perform the join by passing an integer argument to the DefaultJoinStrategy constructor.

**Map-side Joins**

Map-side joins are handled by the [MapsideJoinStrategy](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/join/MapsideJoinStrategy.html). Map-side joins require that the smaller of the two input tables is loaded into memory on the tasks on the cluster, so there is a requirement that at least one of the tables be relatively small so that it can comfortably fit into memory within each task.

For a long time, the MapsideJoinStrategy differed from the rest of the JoinStrategy implementations in that the left-most argument was intended to be larger than the right-side one, since the right-side PTable was loaded into memory. Since Crunch 0.10.0/0.8.3, we have deprecated the old MapsideJoinStrategy constructor which had the sizes reversed and recommend that you use the MapsideJoinStrategy.create() factory method, which returns an implementation of the MapsideJoinStrategy in which the left-side PTable is loaded into memory instead of the right-side PTable.

**Sharded Joins**

Many distributed joins have skewed data that can cause regular reduce-side joins to fail due to out-of-memory issues on the partitions that happen to contain the keys with highest cardinality. To handle these skew issues, Crunch has the [ShardedJoinStrategy](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/join/ShardedJoinStrategy.html) that allows developers to shard each key to multiple reducers, which prevents a few reducers from getting overloaded with the values from the skewed keys in exchange for sending more data over the wire. For problems with significant skew issues, the ShardedJoinStrategy can significantly improve performance.

**Bloom Filter Joins**

Last but not least, the [BloomFilterJoinStrategy](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/join/BloomFilterJoinStrategy.html) builds a [bloom filter](http://en.wikipedia.org/wiki/Bloom_filter) on the left-hand side table that is used to filter the contents of the right-hand side table to eliminate entries from the (larger) right-hand side table that have no hope of being joined to values in the left-hand side table. This is useful in situations in which the left-hand side table is too large to fit into memory on the tasks of the job, but is still significantly smaller than the right-hand side table, and we know that the vast majority of the keys in the right-hand side table will not match the keys in the left-hand side of the table.

**Cogroups**

Some kinds of joins are richer and more complex then the typical kind of relational join that are handled by JoinStrategy. For example, we might want to join two datasets together and only emit a record if each of the sets had at least two distinct values associated with each key. For arbitrary complex join logic, we can always fall back to the [Cogroup](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Cogroup.html) API, which takes in an arbitrary number of PTable instances that all have the same key type and combines them together into a single PTable whose values are made up of Collections of the values from each of the input PTables.

PTable<String, Long> one = ...;

PTable<String, String> two = ...;

PTable<String, Boolean> three = ...;

// cogroup is also a built-in method on the PTable interface.

PTable<String, Pair<Collection<Long>, Collection<String>>> cg = one.cogroup(two);

// For more than two cogrouped tables, we have a helper interface called Tuple.Collect to cut down on

// the typing.

PTable<String, Tuple3.Collect<Long, String, Boolean>> cgAll = Cogroup.cogroup(one, two, three);

Crunch's cogroup operations work just like the cogroup operation in Apache Pig; for more details on how they work, you can consult the [section on cogroups](http://chimera.labs.oreilly.com/books/1234000001811/ch06.html) in the Apache Pig book.

**Sorting**

After joins and cogroups, sorting data is the most common distributed computing pattern. The Crunch APIs have a number of utilities for performing fully distributed sorts as well as more advanced patterns like secondary sorts.

**Standard and Reverse Sorting**

The [Sort](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Sort.html) API methods contain utility functions for sorting the contents of PCollections and PTables whose contents implement the Comparable interface. By default, MapReduce does not perform total sorts on its keys during a shuffle; instead a sort is done locally on each of the partitions of the data that are sent to each reducer. Doing a total sort requires identifying a set of partition keys in the input data and then sending the keys to a different partition based on the keys assigned to each partition. This has the potential to lead to highly imbalanced shuffles that can take a long time to run, but it does make total sorts at scale possible.

By default, Crunch will prefer to handle sorts with a single reducer. The Sort API provides a number of methods that expose the option for a larger number of partitions to be used, at which point the total order partitioner and sorting controls will be enabled.

PCollection<Long> data = ...;

// Sorted in descending order, by default.

PCollection<Long> sorted = Sort.sort(data);

// Sorting in ascending order, with 10 partitions.

PCollection<Long> sorted2 = Sort.sort(data, 10, Sort.Order.ASCENDING);

For more complex PCollections or PTables that are made up of Tuples (Pairs, Tuple3, etc.), we can specify which columns of the Tuple should be used for sorting the contents, and in which order, using the [ColumnOrder](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Sort.ColumnOrder.html) class:

PTable<String, Long> table = ...;

// Sorted by value, instead of key -- remember, a PTable is a PCollection of Pairs.

PCollection<Pair<String, Long>> sortedByValue = Sort.sortPairs(table, ColumnOrder.by(1, Sort.Order.DESCENDING));

**Secondary Sorts**

Another pattern that occurs frequently in distributed processing is *secondary sorts*, where we want to group a set of records by one key and sort the records within each group by a second key. The [SecondarySort](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/SecondarySort.html) API provides a set of sortAndApply methods that can be used on input PTables of the form PTable<K, Pair<K2, V>>, where K is the primary grouping key and K2 is the secondary grouping key. The sortAndApply method will perform the grouping and sorting and will then apply a given DoFn to process the grouped and sorted values.

**Other Operations**

Crunch provides implementations of a number of other common distributed processing patterns and techniques throughout its library APIs.

**Cartesian Products**

Cartesian products between PCollections are a bit tricky in distributed processing; we usually want one of the datasets to be small enough to fit into memory, and then do a pass over the larger data set where we emit an element of the smaller data set along with each element from the larger set.

When this pattern isn't possible but we still need to take the cartesian product, we have some options, but they're fairly expensive. Crunch's [Cartesian](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Cartesian.html) API provides methods for a reduce-side full cross product between two PCollections (or PTables.) Note that this is a pretty expensive operation, and you should go out of your way to avoid these kinds of processing steps in your pipelines.

**Coalescing**

Many MapReduce jobs have the potential to generate a large number of small files that could be used more effectively by clients if they were all merged together into a small number of large files. The [Shard](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Shard.html) API provides a single method, shard, that allows you to coalesce a given PCollection into a fixed number of partitions:

PCollection<Long> data = ...;

PCollection<Long> sharded = Shard.shard(data, 10);

This has the effect of running a no-op MapReduce job that shuffles the data into the given number of partitions. This is often a useful step at the end of a long pipeline run.

**Distinct**

Crunch's [Distinct](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Distinct.html) API has a method, distinct, that returns one copy of each unique element in a given PCollection:

PCollection<Long> data = ...;

PCollection<Long> distinct = Distinct.distinct(data);

The distinct method operates by maintaining a Set in each task that stores the elements it has seen thus far from each of the partitions, and then periodically flushing the contents of this Set to disk. You can control how often this flushing occurs in order to optimize runtime performance or control memory use with another method in Distinct:

PCollection<Long> data = ...;

int flushEvery = 20000;

PCollection<Long> distinct = Distinct.distinct(data, flushEvery);

The default value of flushEvery is 50000, but you can test out the performance of different settings of this value for your own pipelines. The optimal value will depend on some combination of the size of the objects (and thus the amount of memory they consume) and the number of unique elements in the data.

**Sampling**

The [Sample](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Sample.html) API provides methods for two sorts of PCollection sampling: random and reservoir.

Random sampling is where you include each record in the same with a fixed probability, and is probably what you're used to when you think of sampling from a collection:

PCollection<Double> data = ...;

PCollection<Double> sample = Sample.sample(data, 0.05);

Here, we create the same by generating a uniform random number between 0 and 1 for each input record, and including the record in the sample if the random value is less than 0.05. We expect that we will get roughly 5% of the input data included in the sample, but we cannot know precisely how many elements will be in the sample.

In reservoir sampling, we use an algorithm to select an exact number of elements from the input data in a way that each input has an equal probability of being selected- even if we don't know how many elements are in the input collection! You can read more about how this works [here](http://en.wikipedia.org/wiki/Reservoir_sampling).

PCollection<Double> data = ...;

// Choose 500 elements from this input randomly.

PCollection<Double> sample = Sample.reservoirSample(data, 500);

There are also methods on the Sample API that work on PTables and allow you to control the seed used by the random number generators. Note that all of the sampling algorithms Crunch provides, both random and reservoir, only require a single pass over the data.

**Set Operations**

The [Set](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Set.html) API methods complement Crunch's built-in union methods and provide support for finding the intersection, the difference, or the [comm](http://en.wikipedia.org/wiki/Comm) of two PCollections.

**Splits**

Sometimes, you want to write two different outputs from the same DoFn into different PCollections. An example of this would be a pipeline in which you wanted to write good records to one file and bad or corrupted records to a different file for further examination. The [Channels](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/lib/Channels.html) class provides a method that allows you to split an input PCollection of Pairs into a Pair of PCollections:

PCollection<Pair<L, R>> in = ...;

Pair<PCollection<L>, PCollection<R>> split = Channels.split(in);

split.first().write(goodOutputs);

split.second().write(badOutputs);

**Retaining objects within DoFns**

For reasons of efficiency, Hadoop MapReduce repeatedly passes the [same references as keys and values to Mappers and Reducers](https://issues.apache.org/jira/browse/HADOOP-2399) instead of passing in new objects for each call. The state of the singleton key and value objects is updated between each call to Mapper.map() and Reducer.reduce(), as well as updating it between each call to Iterator.next while iterating over the Iterable within a Reducer.

The result of this optimization in MapReduce is that a reference to an object received within a map or reduce call cannot be held on to past the scope of that single method call invocation, as its value will change between invocations of the method call. In some (but not all) situations, the consequences of this optimization affect DoFns as well, meaning that you can't simply retain a reference that is passed in to DoFn.process past the lifetime of a method call.

A convenience method called getDetachedValue is specified in the PType interface to get around this limitation. Implementations of this method perform a deep copy of values of their configured type if needed, and return the value that has been "detached" from the ownership of the MapReduce framework.

In order to make use of the getDetachedValue method in a PType, you need to have an initialized instance of the PType within the DoFn. Note that the initialization of the PType should be performed in the initialize() method of the DoFn.

An example of a DoFn that would make use of getDetachedValue to correctly emit the maximum value encountered would be implemented as follows:

public class FindMax<T extends Comparable> extends DoFn<T, T> {

private PType<T> ptype;

private T maxValue;

public FindMax(PType<T> ptype) {

this.ptype = ptype;

}

public void initialize() {

this.ptype.initialize(getConfiguration());

}

public void process(T input, Emitter<T> emitter) {

if (maxValue == null || maxValue.compareTo(input) > 0) {

// We need to call getDetachedValue here, otherwise the internal

// state of maxValue might change with each call to process()

// and we won't hold on to the max value

maxValue = ptype.getDetachedValue(input);

}

}

public void cleanup(Emitter<T> emitter) {

if (maxValue != null) {

emitter.emit(maxValue);

}

}

}

**Crunch for HBase**

Crunch is an excellent platform for creating pipelines that involve processing data from HBase tables. Because of Crunch's flexible schemas for PCollections and PTables, you can write pipelines that operate directly on HBase API classes like Put, KeyValue, and Result.

Be sure that the version of Crunch that you're using is compatible with the version of HBase that you are running. The 0.8.x Crunch versions and earlier ones are developed against HBase 0.94.x, while version 0.10.0 and after are developed against HBase 0.96. There were a small number of backwards-incompatible changes made between HBase 0.94 and 0.96 that are reflected in the Crunch APIs for working with HBase. The most important of these is that in HBase 0.96, HBase's Put, KeyValue, and Result classes no longer implement the Writable interface. To support working with these types in Crunch 0.10.0, we added the [HBaseTypes](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/hbase/HBaseTypes.html) class that has factory methods for creating PTypes that serialize the HBase client classes to bytes so that they can still be used as part of MapReduce pipelines.

Crunch supports working with HBase data in two ways. The [HBaseSourceTarget](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/hbase/HBaseSourceTarget.html) and [HBaseTarget](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/hbase/HBaseTarget.html) classes support reading and writing data to HBase tables directly. The [HFileSource](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/hbase/HFileSource.html) and [HFileTarget](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/hbase/HFileTarget.html) classes support reading and writing data to hfiles, which are the underlying file format for HBase. HFileSource and HFileTarget can be used to read and write data to hfiles directly, which is much faster than going through the HBase APIs and can be used to perform efficient bulk loading of data into HBase tables. See the utility methods in the [HFileUtils](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/io/hbase/HFileUtils.html) class for more details on how to work with PCollections against hfiles.

**Managing Pipeline Execution**

Crunch uses a lazy execution model. No jobs are run or outputs created until the user explicitly invokes one of the methods on the Pipeline interface that controls job planning and execution. The simplest of these methods is the PipelineResult run() method, which analyzes the current graph of PCollections and Target outputs and comes up with a plan to ensure that each of the outputs is created and then executes it, returning only when the jobs are completed. The [PipelineResult](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PipelineResult.html) returned by the run method contains information about what was run, including the number of jobs that were executed during the pipeline run and the values of the Hadoop Counters for each of those stages via the [StageResult](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PipelineResult.StageResult.html) component classes.

The last method that should be called in *any* Crunch pipeline run is the Pipeline interface's PipelineResult done() method. The done method will ensure that any remaining outputs that have not yet been created are executed via the run, and it will clean up the temporary directories that Crunch creates during runs to hold serialized job information and intermediate outputs.

Crunch also allows developers to execute finer-grained control over pipeline execution via Pipeline's PipelineExecution runAsync() method. The runAsync method is a non-blocking version of the run method that returns a [PipelineExecution](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PipelineExecution.html) instance that can be used to monitor the currently running Crunch pipeline. The PipelineExecution object is also useful for debugging Crunch pipelines by visualizing the Crunch execution plan in DOT format via its String getPlanDotFile() method. PipelineExection implements Guava's [ListenableFuture](https://code.google.com/p/guava-libraries/wiki/ListenableFutureExplained), so you can attach handlers that will be called when your pipeline finishes executing.

Most of the job of the Crunch planner involves deciding where and when to cache intermediate outputs between different pipeline stages. If you find that the Crunch planner isn't optimally deciding where to split two dependent jobs, you can control which PCollections are used as split points in a pipeline via the Iterable<T> materialize() and PCollection<T> cache() methods available on the PCollection interface. If the planner detects a materialized or cached PCollection along the path between two jobs, the planner will prefer the already cached PCollection to its own choice. The implementation of materialize and cache vary slightly between the MapReduce-based and Spark-based execution pipelines in a way that is explained in the subsequent section of the guide.

**The Different Pipeline Implementations (Properties and Configuration options)**

This section adds some additional details about the implementation and configuration options available for each of the different execution engines.

**MRPipeline**

The [MRPipeline](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/impl/mr/MRPipeline.html) is the oldest implementation of the Pipeline interface and compiles and executes the DAG of PCollections into a series of MapReduce jobs. MRPipeline has three constructors that are commonly used:

1. MRPipeline(Class<?> jarClass) takes a class reference that is used to identify the jar file that contains the DoFns and any associated classes that should be passed to the cluster to execute the job.
2. MRPipeline(Class<?> jarClass, Configuration conf) is like the class constructor, but allows you to declare a Configuration instance to use as the basis for the job. This is a good constructor to use in general, especially when you are using Hadoop's Tool interface and Configured base class to declare the main methods for running your pipeline.
3. MRPipeline(Class<?> jarClass, String appName, Configuration conf) allows you to declare a common prefix (given by the appName argument) for all of the MapReduce jobs that will be executed as part of this data pipeline. That can make it easy to identify your jobs on the JobTracker or ApplicationMaster.

There are a number of handy configuration parameters that can be used to adjust the behavior of MRPipeline that you should be aware of:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Usage Notes** |
| crunch.tmp.dir | string | The base directory for Crunch to use when it writes temporary outputs for a job. Default is "/tmp". |
| crunch.debug | boolean | Enables debug mode, which traps and logs any runtime exceptions and input data. Can also be enabled via enableDebug() on the Pipeline interface. False by default, because it introduces a fair amount of overhead. |
| crunch.job.name.max.stack.length | integer | Controls the length of the name of the job that Crunch generates for each phase of the pipeline. Default is 60 chars. |
| crunch.log.job.progress | boolean | If true, Crunch will print the "Map %P Reduce %P" data to stdout as the jobs run. False by default. |
| crunch.disable.combine.file | boolean | By default, Crunch will use CombineFileInputFormat for subclasses of FileInputFormat. This can be disabled on a per-source basis or globally. |
| crunch.combine.file.block.size | integer | The block size to use for the CombineFileInputFormat. Default is the dfs.block.size for the cluster. |
| crunch.max.running.jobs | integer | Controls the maximum number of MapReduce jobs that will be executed simultaneously. Default is 5. |

**SparkPipeline**

The SparkPipeline is the newest implementation of the Pipeline interface, and was added in Crunch 0.10.0. It has two default constructors:

1. SparkPipeline(String sparkConnection, String appName) which takes a Spark connection string, which is of the form local[numThreads] for local mode or master:port for a Spark cluster. This constructor will create its own JavaSparkContext instance to control the Spark pipeline that it runs.
2. SparkPipeline(JavaSparkContext context, String appName) will use a given JavaSparkContext directly, instead of creating its own.

Note that the JavaSparkContext creates its own Configuration instance itself, and that there currently isn't a way to set it during pipeline startup. We hope and expect that this will be corrected in a future version of Spark.

Crunch delegates much of the pipeline execution to Spark and does relatively little of the pipeline planning tasks, with a few crucial exceptions:

1. *Multiple inputs*: Crunch does the work of abstracting the combination of input format and configuration parameters in a way that makes it easy to work with multiple inputs in a pipeline, even if they're of the same type and would have conflicting conf parameters (e.g., if you were trying to join two avro files w/different schemas, the schema conf parameters would typically conflict with each other.)
2. *Multiple outputs*: Spark doesn't have a concept of multiple outputs; when you write a data set to disk, the pipeline that creates that data set runs immediately. This means that you need to be a little bit clever about caching intermediate stages so you don't end up re-running a big long pipeline multiple times in order to write a couple of outputs. Crunch does that for you, along with the same output format and parameter wrapping you get for multiple inputs.
3. *Data serialization*: Spark uses Java serialization or Kryo, with some specialized handling for Writables. Kryo doesn't handle Avro records well, so Crunch's serialization turns those records into byte arrays for you so they don't break your pipelines.
4. *Checkpointing*: Target.WriteMode allows for checkpointing data across Spark pipeline runs. This is useful during active pipeline development, since most of the failures that occur when creating data pipelines are in user-defined functions that come across unexpected input.

Note that the cache(CacheOptions options) method on the PCollection interface exposes the same level of control over RDD caching that the Spark API provides, in terms of memory vs. disk and serialized vs. deserialized data. Although these same methods exist for the MRPipleine implementation, the only caching strategy that is applied is MapReduce's disk and serialization caching, the other options are ignored.

It is important that you call the done() method on the SparkPipeline at the end of your job, which will cleanup the JavaSparkContext. You may get strange and unpredictable failures if you do not do this. As the newest Pipeline implementation, you should expect that SparkPipeline will be a little rough around the edges and may not handle all of the use cases that MRPipeline can handle, although the Crunch community is actively working to ensure complete compatibility between the two implementations.

**MemPipeline**

The [MemPipeline](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/impl/mem/MemPipeline.html) implementation of Pipeline has a few interesting properties. First, unlike MRPipeline, MemPipeline is a singleton; you don't create a MemPipeline, you just get a reference to it via the static MemPipeline.getInstance() method. Second, all of the operations in the MemPipeline are executed completely in-memory, there is no serialization of data to disk by default, and PType usage is fairly minimal. This has both benefits and drawbacks; on the upside, MemPipeline runs are extremely fast and are a good way to test the internal logic of your DoFns and pipeline operations. On the downside, MemPipeline will not exercise serialization code, so it's possible for a MemPipeline run to work fine while a real cluster run using MRPipeline or SparkPipeline fails due to some data serialization issue. As a rule, you should always have integration tests that run either MapReduce or Spark in local mode so that you can test for these issues.

You can add data to the PCollections support by MemPipeline in two primary ways. First, you can use Pipeline's read(Source<T> src) methods, just as you would for MRPipeline or SparkPipeline. MemPipeline requires that any input sources implement the ReadableSource interface so that the data they contain can be read into memory. You can also take advantage of a couple of handy factory methods on MemPipeline that can be used to create PCollections from Java Iterables:

PCollection<String> data = MemPipeline.collectionOf("a", "b", "c");

PCollection<String> typedData = MemPipeline.typedCollectionOf(Avros.strings(), "a", "b", "c");

List<Pair<String, Long>> list = ImmutableList.of(Pair.of("a", 1L), Pair.of("b", 2L));

PTable<String, Long> table = MemPipeline.tableOf(list);

PTable<String, Long> typedTable = MemPipeline.typedTableOf(

Writables.tableOf(Writables.strings(), Writables.longs()), list);

As you can see, you can create either typed or untyped collections, depending on whether or not you provide a PType to be used with the PCollection you create. In general, providing a PType is a good idea, primarily because so many of the Crunch API methods assume that PCollections have a valid and non-null PType available to work with.

On the output side, there is some limited support for writing the contents of an in-memory PCollection or PTable into an Avro file, a Sequence file, or to a text file, but the support here isn't nearly as robust as the support on the read side because Crunch does not an equivalent WritableTarget interface that matches the ReadableSource<T> interface on the read side. Often the best way to verify that the contents of your pipeline are correct is by using the materialize() method to get a reference to the contents of the in-memory collection and then verify them directly, without writing them out to disk.

**Unit Testing Pipelines**

For production data pipelines, unit tests are an absolute must. The [MemPipeline](http://crunch.apache.org/user-guide.html#mempipeline) implementation of the Pipeline interface has several tools to help developers create effective unit tests, which will be detailed in this section.

**Unit Testing DoFns**

Many of the DoFn implementations, such as MapFn and FilterFn, are very easy to test, since they accept a single input and return a single output. For general purpose DoFns, we need an instance of the [Emitter](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/Emitter.html) interface that we can pass to the DoFn's process method and then read in the values that are written by the function. Support for this pattern is provided by the [InMemoryEmitter](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/impl/mem/emit/InMemoryEmitter.html) class, which has a List<T> getOutput() method that can be used to read the values that were passed to the Emitter instance by a DoFn instance:

@Test

public void testToUpperCaseFn() {

InMemoryEmitter<String> emitter = new InMemoryEmitter<String>();

new ToUpperCaseFn().process("input", emitter);

assertEquals(ImmutableList.of("INPUT"), emitter.getOutput());

}

**Testing Complex DoFns and Pipelines**

Many of the DoFns we write involve more complex processing that require that our DoFn be initialized and cleaned up, or that define Counters that we use to track the inputs that we receive. In order to ensure that our DoFns are working properly across their entire lifecycle, it's best to use the [MemPipeline](http://crunch.apache.org/user-guide.html#mempipeline) implementation to create in-memory instances of PCollections and PTables that contain a small amount of test data and apply our DoFns to those PCollections to test their functionality. We can easily retrieve the contents of any in-memory PCollection by calling its Iterable<T> materialize() method, which will return immediately. We can also track the values of any Counters that were called as the DoFns were executed against the test data by calling the static getCounters() method on the MemPipeline instance, and reset those Counters between test runs by calling the static clearCounters() method:

public static class UpperCaseWithCounterFn extends DoFn<String, String> {

@Override

public void process(String input, Emitter<T> emitter) {

String upper = input.toUpperCase();

if (!upper.equals(input)) {

increment("UpperCase", "modified");

}

emitter.emit(upper);

}

}

@Before

public void setUp() throws Exception {

MemPipeline.clearCounters();

}

@Test

public void testToUpperCase\_WithPipeline() {

PCollection<String> inputStrings = MemPipeline.collectionOf("a", "B", "c");

PCollection<String> upperCaseStrings = inputStrings.parallelDo(new UpperCaseWithCounterFn(), Writables.strings());

assertEquals(ImmutableList.of("A", "B", "C"), Lists.newArrayList(upperCaseStrings.materialize()));

assertEquals(2L, MemPipeline.getCounters().findCounter("UpperCase", "modified").getValue());

}

**Designing Testable Data Pipelines**

In the same way that we try to [write testable code](http://misko.hevery.com/code-reviewers-guide/), we want to ensure that our data pipelines are written in a way that makes them easy to test. In general, you should try to break up complex pipelines into a number of function calls that perform a small set of operations on input PCollections and return one or more PCollections as a result. This makes it easy to swap in different PCollection implementations for testing and production runs.

Let's look at an example that computes one iteration of the [PageRank](http://en.wikipedia.org/wiki/PageRank) algorithm that is taken from one of Crunch's integration tests:

// Each entry in the PTable represents a URL and its associated data for PageRank computations.

public static PTable<String, PageRankData> pageRank(PTable<String, PageRankData> input, final float d) {

PTypeFamily ptf = input.getTypeFamily();

// Compute the outbound page rank from each of the input pages.

PTable<String, Float> outbound = input.parallelDo(new DoFn<Pair<String, PageRankData>, Pair<String, Float>>() {

@Override

public void process(Pair<String, PageRankData> input, Emitter<Pair<String, Float>> emitter) {

PageRankData prd = input.second();

for (String link : prd.urls) {

emitter.emit(Pair.of(link, prd.propagatedScore()));

}

}

}, ptf.tableOf(ptf.strings(), ptf.floats()));

// Update the PageRank for each URL.

return input.cogroup(outbound).mapValues(

new MapFn<Pair<Collection<PageRankData>, Collection<Float>>, PageRankData>() {

@Override

public PageRankData map(Pair<Collection<PageRankData>, Collection<Float>> input) {

PageRankData prd = Iterables.getOnlyElement(input.first());

Collection<Float> propagatedScores = input.second();

float sum = 0.0f;

for (Float s : propagatedScores) {

sum += s;

}

return prd.next(d + (1.0f - d) \* sum);

}

}, input.getValueType());

}

By embedding our business logic inside of a static method that operates on PTables, we can easily unit test our PageRank computations that combine custom DoFns with Crunch's built-in cogroup operation by using the [MemPipeline](http://crunch.apache.org/user-guide.html#mempipeline) implementation to create test data sets that we can easily verify by hand, and then this same logic can be executed on a distributed data set using either the [MRPipeline](http://crunch.apache.org/user-guide.html#mrpipeline) or [SparkPipeline](http://crunch.apache.org/user-guide.html#sparkpipeline) implementations.

**Pipeline execution plan visualizations**

Crunch provides tools to visualize the pipeline execution plan. The [PipelineExecution](http://crunch.apache.org/apidocs/0.10.0/org/apache/crunch/PipelineExecution.html) String getPlanDotFile() method returns a DOT format visualization of the exaction plan. Furthermore if the output folder is set then Crunch will save the dotfile diagram on each pipeline execution:

Configuration conf =...;

String dotfileDir =...;

// Set DOT files out put folder path

DotfileUtills.setPipelineDotfileOutputDir(conf, dotfileDir);

Additional details of the Crunch execution plan can be exposed by enabling the dotfile debug mode like this:

// Requires the output folder to be set.

DotfileUtills.enableDebugDotfiles(conf);

This will produce (and save) 4 additional diagrams that visualize the internal stages of the pipeline execution plan. Such diagrams are the PCollection pineage, the pipeline base and split graphs and the run-time node (RTNode) representation.

(Note: The debug mode requires the output folder to be set. )