Apache Crunch

Harsh Asnani, Yan Jianzhi

asnanih@sunyit.edu, yanj@sunyit.edu

CS-518

By: Bruno Andriamanalimanana

State University of New York Polytechnic Institue

Table of Contents

Abstract

Apache crunch is a higher-level API for writing pipelines for but not limited to MapReduce. Main advantages it offers are programmer-friendly Java types and objects. It provides the opportunity of writing reusable code to library functions and use them in any 'Execution Engine' instead of explicitly specifying function details in each job that we want to run. Thus increasing modularity, abstraction and code readability-which are important features to retain in any software development life-cycle.

Apache Crunch

The library uses higher level languages like Java, Scala or Python to write functions but mostly looks like a Java version of Pig. The way it overcomes friction by Pig is replacing language use of Pig Latin. The library is highly composable meaning custom functions can be written into the libraries and embedded in code with ease. This brings in a jointed development experience where the UDFs-User Defined Functions can be written as macros and macro expanded where they are referenced in the code. Although inspired by 'FlumeJava'- a Java library developed at Google for building MapReduce pipelines, Crunch remains untied-it successfully integrated with Spark's execution engine for Cloudera [1]. Crunch doesn't limit our utility to MapReduce thus facilitating the option of choosing the execution engine congenial with our data processing needs. Crunch's high-level abstraction facility allows the user to focus on logical implementation rather than environment implementation details.

Core Elements and terms

- **Pipeline** A Crunch Pipeline is a data stream the type of which depends on the execution engine one wants the Pipeline for. It is a data stream which reads, manipulates and writes data and results to appropriate file formats through robust functions.

  Every Crunch job begins with a Pipeline instance that manages the execution lifecycle of the data pipeline. Pipelines are ***lazily evaluated***. As of now, the three implementations of the Pipeline interface are [2]:

  1. MRPipeline: Executes a pipeline as a series of MapReduce jobs that can run locally or on a Hadoop cluster.

  2. MemPipeline: Executes an in-memory pipeline on client. Typically for testing.

  3. SparkPipeline: Executes a pipeline by running a series of Spark jobs, either locally or on a Hadoop cluster.

- **PCollection<T>** is the core data structure Crunch uses. All objects created inherit PCollection in some form. These objects majorly include the intermediate serializable data types. The output written to file is from PCollection usually. PCollection objects are unordered, distributed and immutable.

- **PType<T>** is associated with every PCollection<T> to encapsulate type information about the elements in the PCollection like the Java class of 'T' and the serialization format used to read and write data from persistent storage. There are two PType families- Writeables and Avro and the selection depends on the file formats being used in the pipeline.

- **Records and Tuples** are data types which come into action when using complex objects with multiple fields like concatenated elements in Values associated with keys in single-maps. Records are accessed using name unlike Tuples which are accessed using indexes. Records are preferred so as to improve code readability.

- **Materialization** is the process of making values in a PCollection available so they can be read in the program. This is similar to 'action' functions in "action and transformations" for RDDs; which act as triggers to indicate that if the pipeline hasn't already been executed to a point where the PCollection object needs materialization, it should now.

  Some functions we discovered that do materialization are:

  1) ***materialize()*** returns a Iterable<T> object when called on a PCollection<T> instance.

  2) ***materializeToMap()*** *returns a Map<K,V>.*

  3) ***PObjects:*** are created from PCollections. They are future objects- meaning the pipeline isn't executed yet. Calling getValue() on a PObject instance will trigger pipeline execution, analogous to materialize(). getValue() returns the elements in the PCollection pointed to by PObject starting from first to last element. Each getValue() call returns one element and successive calls return respective elements until no elements are left.

Primitive Operations

1. ***union()***

PCollection<T> c=a.union(b) returns a PCollection<T> of elements where ‘a’ and ‘b’ are PCollections of T type created from the same pipeline.

2. ***parallelDo()***

- PCollection<T> b=a.parallelDo(DoFn<S,T>, PType<T>). When performed on a PCollection<S> ‘a’ returns another PCollection<T> ‘b’. First Argument specifies what transformation is to be done on ‘a’; second specifies the output type of the resulting collection by inferring the inclusive data type(here T) of PType object.
- PCollection<T> b=a.parallelDo(FilterFn<T>). Uses a filter function argument which returns a boolean based on some conditional statements to include/exclude some elements.
- PTable<K,V> b=a.parallelDo(MapFn<K,V>). Return a multi-map defined as a PTable object in Crunch.

3. ***groupByKey()***

PGroupedTable<K,V> c=b.groupByKey() returns a single-map i.e without duplicate keys by concatenating values separated by commas. Assuming ‘b’ is a PTable object.

4. ***combineValues()***

PTable<K,V> c=b.combineValues(CombineFn<K,V>) uses a CombineFn to specify the arrangement(concatenation details) of value mapping to key. Returns a map object of PTable<K,V> type.

Pipeline Execution

During pipeline construction, Crunch builds an internal execution plan. An execution plan is a DAG-Directed Acyclic Graph of operations on PCollections. Each PCollection holds a reference to the operation that produces it along with PCollections that are arguments to the operation. Each PCollection also has an internal state that records whether it has been materialized.

- ***Running the Pipeline***

  The Pipeline after being instantiated to the desired execution engine's type is run by calling the **run()** method on it. run() is a synchronous function thus blocking further calls until the pipeline has completed before returning. The run() method returns a PipelineResult object which stores statistical information about each stage and whether the pipeline ran successfully.

  There is another method called **runAsync()** different, in that it is an asynchronous function call. It returns a PipelineExecution object which can be used to substantiate motives like-inspecting execution plan, look at status of tasks and pipeline and stop the pipeline, all midway, unlike run(). The execution plan can be viewed as an image which database administrators highly value to further improve scheduling strategies.

  The steps that take place after either run calls are:

  1) Optimization of execution plan as stages in the job. This is variable with execution engine.
  2) Execute each stage in parallel. Materializing each PCollection into intermediate PCollection objects which are serializable for parallelization.
  3) Return a PipelineResult object or PipelineExecution object depending on nature of run call.

4) After the Pipeline is finished, the clean() or done() method is called to free up space occupied by PCollection objects.

Checkpointing Feature

As so often happens in Big Data processing, common data files are read at multiple places in the environment. Pipelines are no different. There will be multiple pipelines which read the same source file and have common intermediate PCollection objects. In anticipation of this and to avoid duplicate computations (foreseen by duplicate PCollection nodes in the DAG), the checkpointing feature was introduced. A pipeline can be checkpointed to persistent storage just by using the write() function of PCollection objects and setting WriteMode=CHECKPOINT in the function call. Any new pipeline that tries to read the file pointed by the checkpointed PCollection, it is referenced to the checkpointed PCollection instead of creating a new PCollection object with same source file in memory.

To maintain validity, Crunch continuously compares the timestamp of the source file to the checkpoint. Anytime the source file is modified-changing the timestamp of the input file to a later timestamp than the checkpoint, the PCollection object is recomputed.

Work and evaluation(until elementary report) :

We have successfully managed to understand and evaluate Crunch's utility and integration to Big Data execution engines, namely Spark and Hadoop MapReduce. Our comprehension is not limited to what we write here, this being just a brief preliminary report. There are complexities involved in writing the library function and embedding them to engine's code which are not all described here; something as trivial as parametrizing types and adhering to them(figuring which types are legal given subclass and superclass constraints). We read API documentation which gave us ideas about function and features that are available to read/write and manipulate data files or various formats.

To further increase conciseness of code, we plan to write our project code using Java 8 lambda expression features since we posses a fair background of functional programming introduced in Java 8's paradigm.

As our project, we try to simulate "Data Processing with Apache Crunch at Spotify" [4]. Spotify neatly used Crunch on top of their already existing Hadoop framework implementation. Spotify also provides open-access to the library functions they wrote for developers to take motivation and embed them in their code. More information on the paper they published is in the references below [4].

Work and evaluation(final) :

We are attaching our project with this report. Our idea which is implemented is as follows: There are two input files to the program. input,txt contains the name of an artist if a song by that artist was played. input2.txt contains the royalty the artist charges for every song that was played. We create a function to pay the artist daily on the basis of the number of songs that were made by the artist, that were played.

As mentioned earlier an execution plan is made by crunch in the from of a .dot file(pipeline.dot in our case) which has the details of a DAG. We render this .dot file using Graphviz's dot tool to a PNG file with the command: "dot -Tpng -O pipeline" which creates a pipeline.png file.

We fell short of implementing our original idea of paying artists based on the currency accepted in their location and a dynamic royalty implementation which changes with the popularity of the artist. Because of time constraints we were not able to debug errors in our additional classes and thus resort to menial version.

Please refer the zip file for project files screenshots and Map of Execution Plan by crunch.

Software and dependencies

- Java 1.8 SE and JRE
- IDE: Eclipse Oxygen 3.A
- Build and repository tool: Apache Maven
- Libraries: Apache Crunch 0.15.0 API [3] , spotify/Crunch-lib [4]

References

[1] Cloudera Documentation. (2018). https://www.cloudera.com/documentation/enterprise/5-5-x/topics/cdh_ig_running_crunch_with_spark.html

[2] Apache Crunch. (2018). *Crunch Pipelines* https://crunch.apache.org/getting-started.html

[3] Apache Crunch. (2018). *Apache Crunch 0.15.0 API* https://crunch.apache.org/apidocs/0.15.0/

[4]David A. Whiting. (2014). *Spotify labs* https://labs.spotify.com/2014/11/27/crunch/

Tom White. (2015). *Chapter 18 Crunch in* Hadoop The Definitive Guide *pages 519-548.*