Transforming Options Market Data with the Dataflow SDK

Salvatore Sferrazza & Sebastian Just, FIS (Salvatore.Sferrazza@fisglobal.com and Sebastian.Just@fisglobal.com)

Preface

When FIS (formerly SunGard) released the whitepaper Scaling to Build the Consolidated Audit Trail: A Financial Services Application of Cloud Bigtable in Q2 2015, a key module of the experiments was utilizing MapReduce for persisting data to Google Cloud Bigtable via the HBase API. At that time, our team had been looking forward to working with Google Cloud Dataflow as a MapReduce alternative, but Google Cloud Dataflow was not yet generally available (GA). In the months since the Bigtable whitepaper was released, the Google Cloud Dataflow managed service has been promoted to GA, and this whitepaper details our experiences applying the Google Cloud Dataflow SDK to the problem of options market data transformation.

Introduction

Much of the software development in the capital markets (and enterprise systems in general) revolves around the transformation and enrichment of data arriving from another system. Traditionally, developers have considered the activities around Extracting, Transforming and Loading (ETL) data a particularly unglamorous dimension of building software products. However, these activities encapsulate functions that are core to every tier of computing. Oftentimes, the activity involves legacy systems integration, and perhaps this is the genesis of ETL's unpopular reputation. Nonetheless, when data-driven enterprises are tasked with harvesting insights from massive data sets, it is quite likely that ETL, in one form or another, is lurking nearby.

The Google Cloud Dataflow service and SDK represent powerful tools for myriad ETL duties. In this paper, we introduce some of the main concepts behind building and running applications that use Dataflow, then get "hands on" with a job to transform and ingest options market symbol data before storing the transformations within a Google BigQuery data set.

Objectives

- Introduce Google Cloud Dataflow's programming and execution model
- Apply Dataflow to transform market data (specifically options prices) and persist the transformations into a BigQuery table for analysis

The specific transformations we'll perform in this use case include the following:

- Ingest options market ticks and extract the contract's expiration date, strike price and reference asset from the symbol associated with each tick
- Load the transformed data into a Google BigQuery schema that supports queries on these individual contract properties. For example, finding all the in-the-money (ITM) contracts traded between 2:00 and 3:00 PM requires comparing the strike price of a contract with the spot price of the underlying asset. Hence, having the strike price encoded within the options symbol string is an obstacle to any analytics that require arithmetic operations involving the strike price

Prerequisites

While Dataflow's SDK support is currently limited to Java, Google has a Dataflow SDK for Python currently in Early Access Program (EAP). As such, all examples herein are in Java using Maven for builds and dependency management. We hope to publish a supplementary implementation of this same project using the Python SDK during public beta.

For the hands-on portion of the paper, the following conditions were required:

- Enabling the **Dataflow API** for a Google Developers Console project
- Enabling the **BigQuery API** for a Google Developers Console project
- Enabling billing on your Google Cloud Platform account

- Installing the **Google Cloud SDK** on your client operating system

- Installing **Maven** for managing the Java project

- For the Dataflow libraries specifically, adding the following dependency to your project's pom.xml as follows:

```
<dependency>
  <groupId>com.google.cloud.dataflow</groupId>
  <artifactId>google-cloud-dataflow-java-sdk-all</artifactId>
  <version>RELEASE</version>
</dependency>
```

Additionally, the source code may be cloned from the Git repository located at https://github.com/SunGard-Labs/dataflow-whitepaper. Although the source code repository is self-contained, parameters may need to be adjusted to reflect the specific Google Cloud project parameters for your environment.

Programming Dataflow

State maintenance and scalability

Because a full description of the role that application state plays in determining scalability is beyond the scope of this paper, please refer to this explanation, which effectively breaks down the trade-offs inherent to various techniques around application state maintenance.

Suffice it to say, state management is a critical factor contributing to a system's scalability. A common solution for applications requiring near-linear scalability amid large workloads is parallelization. However, development for parallel processing can be tricky compared to more traditional serial workloads. As you'll see, Google's Cloud Dataflow SDK removes much of this friction with clear and concise API semantics.

Pipelines, ParDos and DoFns

The bedrock of any Dataflow job is the pipeline, which is responsible for the coordination of an individual Dataflow job. The actual logic of the job stages is performed by ParDo (for **Par**allel **Do**) subclasses that implement their logic within the template of a DoFn (for **Do F**unctio**n**) implementation callback. A ParDo's principal responsibility is the transformation of one (input) PCollection to another (output) PCollection.

At runtime, the pipeline and its component ParDo functions are analyzed by the Dataflow optimizer and factored into its execution plan, possibly merging steps along the way and sequencing them according to the semantics implied by the pipeline class and its component ParDo operations. There is no explicit relationship defining map, reduce or shuffle operations. Rather, the optimizer sequences operations based on the semantics of the input and output PCollections, similar to how a SQL query optimizer builds a query plan based on the selected columns, available indices, query predicates and table schema.

PCollections

Another central concept in the Dataflow model is the PCollection (for **P**arallel **Collection**), which represents the primary group of elements being processed by the Dataflow job at any stage in the pipeline. With the typical MapReduce programming model (e.g., Hadoop), key-value pairs are emitted from collections of elements. In contrast, a PCollection simply represents the collection of active elements within the pipeline and is available as input to ParDos and DoFns. In our experience, we have found that this both flattens the learning curve and simplifies the programming model overall compared to the currently available alternatives.

This design offers great flexibility as the programmer is freed from having to implement a particular Map or Reduce interface imposed by the SDK. In particular, we find this enables greater focus on the business relevance of the associated inputs and outputs, as well as the specific nature of the elements that are processed. For example, the PCollection of one job stage may have a direct lineage from the PCollection of the previous stage, but it is not required to per se. This affords a more reactive/functional approach to programming — the output of one processing stage is simply the input of the subsequent stage, not dissimilar to an idealized callback-driven control flow.

It is also important to remember that each PCollection can be treated as a first-class citizen. A PCollection can be used multiple times in a single pipeline, effectively creating a new logical branch of pipeline execution each time.
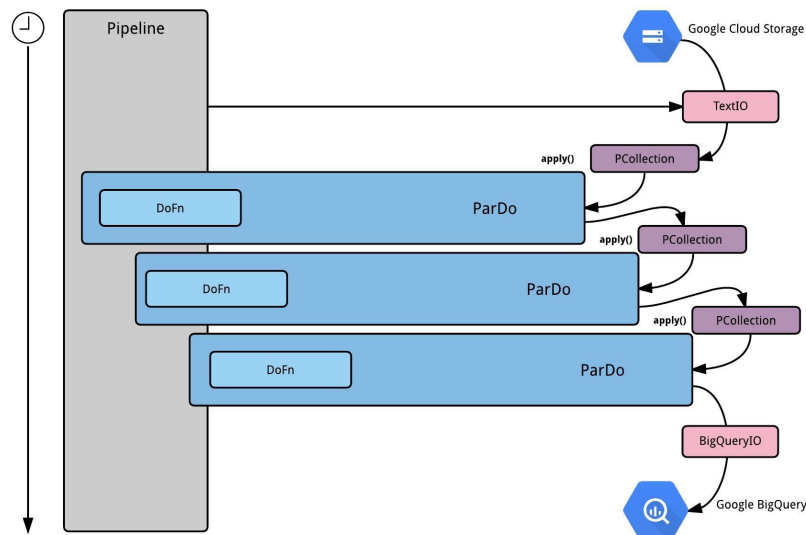
Job inputs

Although Dataflow allows for flexibility with regards to data sources, there is one thing that all jobs share: an input operation such as a file on Google Cloud Storage, the contents of a BigQuery table or query, a message arriving on a Cloud PubSub topic, or even your own customized I/O source — which always bootstraps the initial PCollection for the pipeline.

During the lifecycle of a pipeline, many different (side) inputs may be employed as needed to transform, filter and/or process the currently active PCollection within the pipeline.

Native support for input sources

The stock Dataflow SDK ships with several classes that support input and output operations with resources external to Dataflow's execution context.

For example, for a pipeline to initialize a PCollection from the contents, a file living within Google Cloud Storage requires importing `com.google.cloud.dataflow.sdk.io.TextIO` and, via the PipelineOptions interface, configuring it with the URI of your storage bucket, such as: `gs://<storage-bucket-on-gcs>/DATA1.txt`.

This interface also transparently supports wildcard syntax, so if you have a bucket directory and you would like to ingest all the files within it, simply pass the URI in as:
`gs://<storage-bucket-on-gcs>/mydata/*`

One notable aspect of these I/O classes is that they necessarily terminate a branch of control flow between the pipeline and either an input from or an output to an external system. That is, the Read method within the class TextIO does not take a PCollection as an input parameter; rather, it takes a String representing the URL of the resource containing the text that should be read. On the flipside, the Read method does output PCollections, which are then used as the input for other DoFn's downstream which both accept and return PCollections. At the end of the control flow, one might find an I/O class once more (another TextIO or a BigQueryIO, for example) that takes in a PCollection but does not output one. Rather, the result of their respective I/O operations are lines of text or rows in a BigQuery schema. Since *IO.Write does not output a PCollection, it cannot be applied further down the chain. Hence, they terminate out-of-band vis-a-vis Dataflow's execution context.

Other than that, developers gain a great deal of flexibility by employing relatively simple API semantics. When operating in a distributed mode, all details around workload distribution (i.e., in which worker nodes process which lines of the input file(s) in the case of TextIO) is at the discretion of the Dataflow PipelineRunner chosen to execute the job. Developers are explicitly less distracted from the core features of their application regardless of whether the job input or output is measured in megs or gigs. This is liberating for both the programmer and the programmer's team because it eliminates a great deal of friction often found in the operation of a systematized big data analytics regime.

As of this writing, the I/O APIs supported by the Dataflow SDK include the following:

- TextIO (text file access via URI, including Google Cloud Storage)

- BigQueryIO (Google Cloud Platform BigQuery)

- AvroIO (Apache Avro)

- PubsubIO (Google Cloud Platform Pub/Sub)

- BigTableIO (Google Cloud Bigtable - currently in beta)

By inheriting functionality from more foundational classes, such as TextIO, and overriding the relevant API implementations, the amount of code required to implement a custom CsvIO or XmlIO is reduced. In some cases, you may ultimately inherit from classes that these I/O classes themselves are derived from, or you may inherit directly from an existing I/O class.

The native TextIO Reader method currently returns a PCollection corresponding to each line of text in the specified file(s). But what if the logical records within the file you are processing span multiple lines of text?

In a traditional, non-distributed programming model, one might be inclined to append to some other data structure that is scoped outside the loop processing the individual lines, then concatenate the structure after some arbitrary condition (indicating the end of a logical record) has been reached, but before the next line is read. However, because of the inherent statelessness of Dataflow's programming model, the API prohibits this approach.

However, when presented with a similar challenge, a developer on our team was able to derive a custom MultiLineIO class by overriding methods such as startReading(), readNextRecord() and isAtSplitPoint() from classes found in the com.google.cloud.dataflow.sdk.io package. Thus, a firm grasp of the abstractions used by the Dataflow SDK is quite helpful when planning specific customization vectors for any individual use case, and hence minimizing the amount of custom code that must be maintained.

Side inputs

A key concept behind Dataflow is the journey of a single PCollection of elements through the pipeline. Whereas a single, discrete set of input data elements alone is sufficient for many scenarios, in some cases it may become necessary to have data from multiple disparate sources or, similarly, return two distinct data outputs rather than a single PCollection. To address these situations, Dataflow offers Side Inputs and Outputs.

A common challenge faced by distributed systems such as Dataflow is how to best synchronize input data sets among multiple worker nodes. To be sure, many algorithms (such as write-through, write-behind, read-through and refresh-ahead) exist to ameliorate the problem of distributed locking. However, with decreasing costs in both storage and computing power, replicating data independently to each worker node can have benefits, as opposed to models where those nodes compete for latches and locks against a single shared data set. While this avoids

many concurrency implications inherent to distributed computing, it does impose that the data sets that are distributed among workers remain immutable during their lifetimes.

Therefore, to use PCollections as side inputs, they must be mapped to a separate variable to ensure immutability. This also allows (and is used) to keep the whole side input in the worker's memory for additional performance benefit. Next to this constraint, a side input simply represents a PCollection just as the main PCollection does. Also, like any intermediate main PCollection, a side input PCollection can be reused across various stages of the pipeline.

If the data contains duplicates for the same key, a little "Highlander" logic can be used to resolve only unique map items as required by the Dataflow API:

```
/**
 * This Highlander (there can be only one) function operates
 * on a 'first object takes it all' principle. As it is executed
 * in a distributed environment, 'first' might not be
 * deterministic.
 */
public class HighlanderCombineFn<T>
    extends CombineFn<T, T, T> {

    private static final long serialVersionUID = 1L;

    /**
     * @return <code>null</code>
     * which is fine as accumulator as it is never used */
    @Override
    public T createAccumulator() {
        return null;
    }

    /**
     * Just takes the next element as the new one. No merge
     * with the empty accumulator required
     */
    @Override
    public T addInput(T accumulator, T input) {
        return input;
    }

    /**
     * Just use the first element that comes around
     */
    @Override
    public T mergeAccumulators(Iterable<T> accumulators) {
        return accumulators.iterator().next();
    }

    /**
     * @return The merged accumulator is just returned -
     *         which is the first element itself
     * @see #mergeAccumulators(Iterable)
     */
    @Override
    public T extractOutput(T accumulator) {
        return accumulator;
    }
}
```

Job outputs

Persisting data based on processing within a Dataflow pipeline is always possible. Any stage might be an "outputting" stage that persists data permanently.

Native support for output sinks

The Dataflow SDK provides write access to Google Cloud services that are useful for big data. Namely, local files or those stored within Google Cloud Storage, records in Google Cloud Bigtable or entire tables within Google BigQuery are natively supported by Google's Dataflow SDK. Side outputs

Dataflow also supports the creation of multiple PCollections from a single stage in the pipeline. A distinction is made between the primary (of which only one may exist) and ancillary (0..n) side inputs. However, both primary and side input PCollections may be employed and referenced in a similar fashion. Within Dataflow job code, there is a separate API call that distinguishes between the primary and side-input PCollections.

It is important to note that the input PCollection and the output PCollection of the very same processing stage are two distinct collections of elements. They may be related (i.e., contain similar objects) from a business perspective, but from a software engineering internals standpoint, the input and output PCollections of a particular DoFn represent independent entities.

Custom sinks and sources

If the sinks and sources supplied with the Dataflow SDK do not satisfy the requirements of a specific pipeline, customizing one to suit the particular need takes a straightforward shape. The Dataflow SDK is designed with rich interfaces and abstract classes that already perform many common low-level operations. Thus, before starting off from the ground floor with your own source or sink, it is suggested to peruse the API documents for abstract classes that may provide the framework for customized I/O.

For example, by default, the Dataflow SDK class TextIO interprets input files on a one-record-per-line basis. However, some file formats are incompatible with this convention, instead preferring to define logical records as spanning multiple lines of text. Yet, it is quite simple to author an I/O component that interprets a multi-line source file. Scenarios that require the bundling of multiple lines to a single PCollection element (i.e., a single logical record) would find that extending the SDK class `com.google.cloud.dataflow.sdk.io.FileBasedSource` provides a large percentage of the logical substrate required to appropriately interpret these types of input file formats.

This is true of custom sinks that might interact with other persistence services, keeping with the spirit of Dataflow that enables developers to optimally leverage base functionality shipping with the SDK.

Building a simple pipeline

The following annotated source code illustrates the main entry point into a simple pipeline class:

```java
public static void main(String[] args) {
  // Parses and validates the command line arguments
  Options options = PipelineOptionsFactory.fromArgs(args)
                        .withValidation().as(Options.class);

  // Create the Dataflow pipeline based on the provided arguments
  Pipeline p = Pipeline.create(options);

  // Create the first PCollection from hardcoded data
  PCollection<Integer> inputCollection = p.apply(
     Create.of(3,4,5).withCoder(BigEndianIntegerCoder.of()));

  // First realy processing state - doubling the input
  PCollection<Integer> doubledCollection
    = inputCollection.apply(ParDo.named("Double input").of(
      new DoFn<Integer, Integer>() {

        @Override
        public void processElement(ProcessContext pc)
          throws Exception {
              pc.output(pc.element() * 2);
        }}));

  // Using doubleCollection forms a chain - inputCollection
  // is also still available and can be used (which forms
  // a logical second branch)

  // Start the execution
  p.run()
}
```

It is important to remember that at the time of the pipeline's deployment, this main() method is actually executed, and the execution graph and optimizations are created by the Dataflow PipelineRunner that is being employed.

Thus, access to local resources within DoFns is not possible during actual job execution — a different environment is used for job execution as compared to deployment execution. This can be seen similarly to how static methods within a Java class do not have access to any of its host class's member variables. DoFns share certain characteristics with static methods in this regard. There is a stringent mechanism defining the universe available to both static methods (i.e., their input parameters) and DoFns (i.e., their PCollections) resulting in deterministic input characteristics during their respective execution lifetimes.

Dataflow job execution

Dataflow SDK

As we've seen during our review of Dataflow's programming model, the pipeline abstract class is an environment-independent entry point into the behavior of a particular Dataflow job. In some regards, the Dataflow pipeline implementation represents Dataflow's primary configuration mechanism. This API layer is where the details of the execution environment are specified. Custom ParDo and DoFn subclasses (as discussed in the previous section) are defined to specify the particular logic of your pipeline class implementation. At runtime, the pipeline class and specific DoFn implementations referenced therein are combined with specific environment settings (e.g., local laptop or Google Cloud execution).

Batched and streaming inputs

One of the principal innovations behind the Dataflow SDK is the reconciliation of batch and streaming data ingestion abstractions organized under a single, cohesive umbrella.

The difference between the two actually is not all that dramatic. The fundamental differentiator between these ingestion models is that batched data is, by definition, bounded. That is, at the time of the job's launch, the totality of the input data is knowable and finite. A line is drawn in the proverbial sand upon a batch job's launch that indicates, "this is all the data this job will process."

Alternatively, when input data is expected to arrive from the source for an arbitrary duration (i.e., one does not know at launch-time when, or even whether, the input will end), a streaming job is the proper approach to follow. Dataflow streaming jobs run continuously by applying time-series windowing to the incoming data, effectively creating and processing micro-batches from the incoming data.

Revising application code for streaming jobs

Currently, streaming Dataflow jobs support Google Cloud Pub/Sub topics as their input source natively, using the `com.google.cloud.dataflow.sdk.io.PubSubIO` class of the Dataflow SDK. However, using the Dataflow SDK as a foundation give developers a head start in authoring custom I/O source implementations to provide streaming data semantics for additional input streams.

Execution environments

Several possible execution destinations exist for a particular Dataflow job. These include:

- On a local workstation (i.e., no parallel workload distribution)

- Within the Google Cloud Platform (100% managed cloud infrastructure)

- Using Apache Spark clusters (via the pre-release spark-dataflow provider by Cloudera)

- On top of an Apache Flink cluster

Local environment execution proves generally sufficient for testing and debugging the behavior of individual ParDos while the function behavior is being developed. After initial testing locally, the job can be tested in distributed mode (usually with an abbreviated specimen of input data) to validate the expected behavior. [Note: If you have local text file resources residing on your workstation, those will only be accessible when using the DirectPipelineRunner class for the pipeline since remotely deployed worker nodes (e.g. DataflowPipelineRunner) will not have access to the workstation's file system.]

Introduced in early 2015, Cloudera's Spark Dataflow provider decouples applications written to Google's Dataflow SDK from the specific execution area of Google Cloud Platform. Applications deployed on a private cloud infrastructure and leveraging Apache Spark now can benefit from the improved Dataflow SDK semantics as well as the ability to expand deployment options if necessary (to Flink or Google Cloud Dataflow, for instance).

Job execution

Depending on the environment in which a particular job will run, one may choose to execute it using Maven directly, or by packaging the specific job code into a separate JAR file and following standard JAR execution mechanisms (i.e. java -jar). The main rationale for the latter option is if the job needs to be kicked off from an environment where Maven support is unavailable or undesirable to install. In this case, a single, shaded (i.e., all dependencies are embedded) JAR enables those jobs to be kicked off without requiring Maven (or other build components) as an installation prerequisite (a suitable JRE is still required, of course).

When executing via Maven, which is common when testing during development time, the majority of Dataflow options will be configured within the project's pom.xml. When a job is launched directly from a packaged JAR, the options are passed in as parameters to the Java executable, as illustrated in bold text:

```
java -jar options-transform-0.0.1-SNAPSHOT.jar
--input=gs://bucket/zvzzt.txt
--output=gs://bucket/zvzzt.tsv
--project=my-gcp-project
--runner=DataflowPipelineRunner
--stagingLocation=gs://staging-bucket
--dataflowJobFile=/tmp/dataflow-job-file
--jobName=run-20151206222600
```

Pipeline classes will typically invoke the fromOptions() method of the PipelineRunner base class to populate a PipelineOptions object using the parameters passed to the JVM process (i.e. args[]). This is illustrated in the SymbolTransformationPipeline class within the GitHub project referenced below.

Local workstation execution via Maven, using the pom.xml from the Github repository, resembles the following:

```
mvn -Plocal exec:exec
```

or this instead, to run the job on Google Cloud Platform:

```
mvn exec:gcp
```

In some cases, it can be helpful to create a shell script to manage common runtime parameters around your Dataflow execution pipeline. For an example, refer to the bin/run script within the Github source repository for the project.

Hands-on with Dataflow

```
The source code for the hands-on portion may be found at
https://github.com/SunGard-Labs/dataflow-whitepaper.
```

Selected use case

For the hands-on section, we chose a use case to transform incoming options market data records ("ticks") to extract the relevant options contract terms that are embedded within the symbol.

Please see Appendix C - Guide to OCC Symbology for a more detailed description of OCC-cleared options symbology components.

Analysis of Dataflow's features in support of use case

In using Dataflow as a tool to assist with implementing the required functionality, it's helpful to identify the features within Dataflow that are pertinent.

The input files will be stored within Google Cloud Storage in discrete files. Hence, the input to any particular job will be bounded. In this case, we know that a batch mode Dataflow job is most appropriate. The job can be instructed to process either a single file (`/home/user/data/file.txt`) or a group of files (`gs://bucket/file00*.txt`), but this is not a meaningful distinction for any application logic, as the input set will simply be derived to a PCollection before it reaches any relevant Dataflow code.

One of the core logical functions implied by the requirements is the extraction of standardized equity option contract terms that are embedded directly within the input symbols themselves. For example, the symbol *ZVZZT151120C00024500* represents the following contract terms:

Term	Value
Underlying Equity Symbol	ZVZZT (ticker symbol reserved for testing)
Expiration Date	November 20, 2015
Put or Call? (sell or buy underlying security)	Call (option to buy underlying)
Strike Price	$24.50

This particular logical operation is an excellent candidate to represent inside a DoFn. Since any extraction logic only requires the options symbol itself as input, the operation can be designed as a pure function. One can regard the DoFn interface as the Java class embodiment of a pure function, enabling the logic specified within the specific implementation to scale via parallelization.

To take an inventory of Dataflow features we employed as mapped to our requirements, we derived the following:

- A **TextIO** component to read the source data from a Google Cloud Storage Bucket
- A **Pipeline** class to orchestrate the execution of the Dataflow job
- A **DoFn** to extract and transform the contract terms embedded within the symbol
- A **ParDo** to wrap the above mentioned DoFn
- A **BigQueryIO** to insert the transformed data into Google BigQuery

For the Java portions of the application, Maven's pom.xml can be used to build the compiled application and all of its dependencies into a single, distributable ("shaded") JAR. Alternatively, for launching jobs from environments that have a full Maven build environment deployed, the repository may be cloned and jobs launched using the Maven command line with the appropriate profile (in this case local or gcp) applied.

Project source code

SymbolTransformPipeline main() method

```
public static void main(String args[]) {

  try {

     ExtractContractTerms fn = new ExtractContractTerms();

     SymbolTransformOptions options =
       PipelineOptionsFactory.fromArgs(args).as(SymbolTransformOptions.class);

     Pipeline pipeline = Pipeline.create(options);

     PCollection < OptionsTick > mainCollection =
       pipeline.apply(TextIO.Read.named("Reading input file")
               .from(options.getInputFilePath()))
               .apply(ParDo.named("Extracting contract terms from symbol")
               .of(fn)).setCoder(SerializableCoder.of(OptionsTick.class));

       if (!"".equals(options.getOutputTable())) {

               mainCollection.apply(ParDo.named("Creating BigQuery row from tick")
                       .of(new CreateBigQueryRow()))
                       .apply(BigQueryIO.Write.named("Writing records to BigQuery")
                       .to(options.getOutputTable())
                       .withSchema(SymbolTransformPipeline.generateSchema())
                       .withWriteDisposition(BigQueryIO.Write.WriteDisposition.WRITE_APPEND)
                       .withCreateDisposition(BigQueryIO.Write.CreateDisposition.CREATE_IF_NEEDED));

       } else {

               mainCollection.apply(ParDo.named("Creating String row from tick")
                       .of(new TickToString()))
                       .apply(TextIO.Write.named("Writing output file")
                       .to(options.getOutputFilePath()));

       }

       pipeline.run();
       System.exit(0);

  } catch (Exception ex) {
       System.exit(1);
  }

}
```

OptionsTick model class

```
// Must be serializable to be included within a PCollection

public class OptionsTick implements Serializable {

public OptionsSymbol symbol;
        public long exchangeTimestamp;
        public long insertTimestamp;
        public Integer bidSize;
        public BigDecimal bid;
        public BigDecimal ask;
        public Integer askSize;
        public BigDecimal trade;
        public Integer tradeSize;
        public String exchange;


}
```

OptionsSymbol model class

```
// Must be serializable to be included within a Pcollection

public class OptionsSymbol implements Serializable {

   public String underlying;
   public int expirationYear;
   public int expirationMonth;
   public int expirationDay;
   public String putCall;
   public BigDecimal strikePrice;

   public OptionsSymbol(String occSym) throws Exception {
        char[] chars = occSym.toCharArray();
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < chars.length; i++) {
                if (!Character.isDigit(chars[i])) {
                   sb.append(chars[i]);
                 } else {
                   this.underlying = sb.toString();
                   String sym = occSym.substring(i, occSym.length());
                   this.expirationYear = Integer.parseInt(sym.substring(0, 2));
                   this.expirationMonth = Integer.parseInt(sym.substring(2, 4));
                   this.expirationDay = Integer.parseInt(sym.substring(4, 6));
                   this.putCall = sym.substring(6, 7); // P(ut) or C(all)
                   BigDecimal dollar = new BigDecimal(sym.substring(7, 12));
                   BigDecimal decimal = new BigDecimal(sym.substring(12, 15))
                        .divide(new BigDecimal(100)); // 500 is 50 cents
                   this.strikePrice = dollar.add(decimal);
                   return;
                 }
        }
   }
}
```

ExtractContractTerms processElement() method

```java
public void processElement(ProcessContext c) {

        String line = c.element();
        String[] values = line.split("\t");
        String sym = values[0];
        OptionsTick tick = new OptionsTick();

        tick.exchangeTimestamp = ("".equals(values[1]) ? null :
                                  Long.parseLong(values[1]));

        tick.symbol = new OptionsSymbol(sym);
        tick.exchangeTimestamp = new Long(values[1]).longValue();

        if (values[2] != null) {
                if (values[2].length() > 0) {
                        tick.bidSize = new Integer(values[2]);
                }
        }

        if (values[3] != null) {
                if (values[3].length() > 0) {
                        tick.bid = new BigDecimal(values[3]);
                }
        }

        if (values[4] != null) {
                if (values[4].length() > 0) {
                        tick.ask = new BigDecimal(values[4]);
                }
        }

        if (values[5] != null) {
                if (values[5].length() > 0) {
                tick.askSize = new Integer(values[5]);
                }
        }

        if (values[6] != null) {
                if (values[6].length() > 0) {
                        tick.trade = new BigDecimal(values[6]);
                }
        }

        if (values[7] != null) {
                if (values[7].length() > 0) {
                tick.tradeSize = new Integer(values[7]);
                }
        }

        tick.exchange = values[8];
        tick.insertTimestamp = System.currentTimeMillis() / 1000L;

        c.output(tick); // adds to the output PCollection
        return;
}
```

CreateBigQueryRow toBigQuery() method

```
public static final TableRow toBigQuery(OptionsTick tick) {

        TableRow row = new TableRow();

        row.set("EXCHANGE_TIMESTAMP", tick.exchangeTimestamp);
        row.set("INSERT_TIMESTAMP", tick.insertTimestamp);
        row.set("UNDERLYING", tick.symbol.underlying);
        row.set("EXPIRATION_YEAR", tick.symbol.expirationYear); // Y2.1K bug
        row.set("EXPIRATION_MONTH", tick.symbol.expirationMonth);
        row.set("EXPIRATION_DAY", tick.symbol.expirationDay);
        row.set("PUT_CALL", tick.symbol.putCall);
        row.set("STRIKE_PRICE", tick.symbol.strikePrice.floatValue());
        row.set("BID_SIZE", tick.bidSize);
        row.set("BID", tick.bid);
        row.set("ASK", tick.ask);
        row.set("ASK_SIZE", tick.askSize);
        row.set("TRADE", tick.trade);
        row.set("TRADE_SIZE", tick.tradeSize);

        return row;

}
```

Building the project

You can build the entire project using the following Maven command:

```
mvn clean install
```

Running the job locally

To run the job on your local workstation, build the project as described above, then execute the following command:

```
java -jar target/options-transform-0.0.1-SNAPSHOT.jar <options>
```

Please note that if the project uses any network resources during job execution (e.g., TextIO reading a file located on Google Cloud Storage), then it will require connectivity to complete successfully even if being executed with DirectPipelineRunner on the local system.

Alternatively, from the project root directory, the following commands can be executed:

```
cd bin
./run
```

Running the job under Google Cloud Dataflow

Free trials of Google Cloud Platform are readily available for testing Dataflow job features such as auto-scaling with your specific jobs. To run jobs on Google Cloud Platform, several additional steps need to be taken after signing up for Google Cloud Platform (and if you have a gmail account, most of the work has been done already).

Please be sure to update Maven's pom.xml with your Google Cloud Platform project's parameters before running.

The command to launch the job on Google Cloud Dataflow from the project directory is:

```
mvn -Pgcp exec:exec
```

alternatively, within the bin/run script, simply comment out the line specifying DirectPipelineRunner:

```
# RUNNER=DirectPipelineRunner
```

This will enable the DataflowPipelineRunner that is declared in the immediately preceding line.

Then executing:

```
cd bin && ./run gs://<bucket>/<input-file> gs://<bucket>/<output-file>
```

kicks off the Dataflow job on the Google Cloud Platform infrastructure. You can open up https://console.developers.google.com/dataflow in a browser to see the pipeline in action.

Note: The bin/run script also has environment variables defined that must be set with your Google Cloud Platform account parameters.

What's next?

While we covered a lot of material in this paper, it feels as if we've only scratched the surface on the capabilities the Dataflow SDK offers developers. As the Dataflow ecosystem evolves, some areas that we are particularly excited about are:

• General availability of the Python Dataflow SDK

• Additional gcloud dataflow capabilities for job monitoring and metrics collection

• New PipelineRunners and I/O classes published by the OSS or vendor communities

In the future, we're hoping to author a follow-up paper that delves further into Dataflow's more advanced topics, such as side inputs, stream windowing and custom input sources. Until then, have fun with Dataflow!

Acknowledgements

Appendix A - PipelineRunner cheat sheet

At runtime, every pipeline needs a PipelineRunner implementation specified, which interfaces between the pipeline and the execution environment for the specified job. While PipelineRunner is defined as abstract, and as such cannot be invoked directly, the Dataflow SDK comes with several native PipelineRunner classes to cover most typical scenarios.
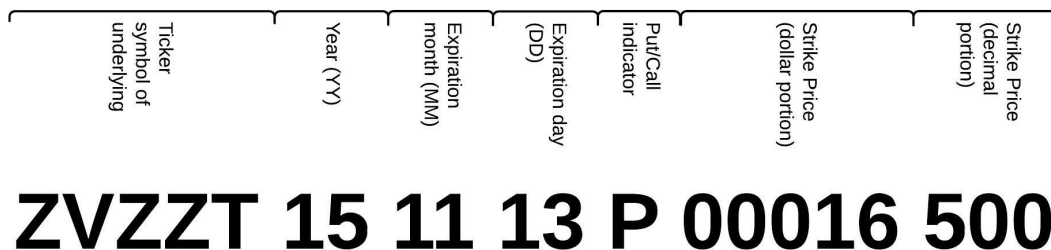
Implementation	Used for
`com.google.cloud.dataflow.sdk.runners.DirectPipelineRunner`	Best suited for small, local executions on very abridged data sets or for functional executions of pipelines that do not need to demonstrate scale.
`com.google.cloud.dataflow.sdk.runners.DataflowPipelineRunner`	Dynamically scaling the execution and distribution of jobs within Google's Cloud Dataflow managed service. The DataflowPipelineRunner class exits the JVM and returns the Job ID of the newly spawned job to the console.
	Dynamically scaling the execution and distribution of jobs within Google's Cloud Dataflow managed service. The BlockingDataflowPipelineRunner class will keep the launching JVM up throughout the job's lifetime, and only return control back to the shell once the job is complete.
`com.google.cloud.dataflow.sdk.runners.BlockingDataflowPipelineRunner`	If you are orchestrating multiple jobs and one particular job's execution is dependent upon another's successful completion, this can be a useful device to employ.
`com.google.cloud.dataflow.sdk.runners.TestDataflowPipelineRunner`	In combination with DataflowAssert and TestPipeline, can be used to execute software tests either locally or using the Cloud Dataflow service.
`cloudera.dataflow.spark.SparkPipelineRunner`	For deployment to Apache Spark clusters.
`com.dataartisans.flink.dataflow.FlinkPipelineRunner`	For execution upon Apache Flink clusters.

Appendix B - Dataflow job options cheatsheet

Configuration Option	Description	Sample Usage
--input	Source of the Dataflow job's input data	`--input=gs://my-gcs-bucket/bigfile.001`
--runner	Source of the Dataflow job's input data	`--runner=DataflowPipelineRunner`
--stagingLocation	Bucket to store temporary libraries used during deployment and worker task distribution	`--stagingLocation=gs://my-temp-bucket`
--numWorkers	Number of workers to employ for job	`--numWorkers=5`
--jobName	Customizes the name of the Dataflow job for display within CLI or web console	`--jobName="MYJOB_`date +%Y-%m-%d-%H-%M-%S`"`
--diskSizeGb	Amount of storage to allocate per individual worker node	`--diskSizeGb=32`
--autoScalingAlgorithm	The algorithm Cloud Dataflow will use to grow and shrink the worker cluster. Mutually exclusive with the --numWorkers option	`--autoScalingAlgorithm=BASIC`
--project	The Google Cloud Platform project under which the Dataflow job should run	`--project=myDataflowProject`

Appendix C - Guide to OCC symbology

The figure below illustrates the different components of a standardized options symbol for contracts centrally cleared by the Options Clearing Corporation (OCC).

("ZVZZT November 13, 2015 $16.50 PUT")