Final Project

Google Cloud DataFlow real-time service for batch and stream processing

Shan Zhou, Jay Upadhyay,Wendy Jiang

CSCI E-63 Big Data Analytics

Harvard University Extension School

Prof. Zoran B. Djordjević

Introduction

- Dataflow is a fully managed, no-ops service from Google that attempts to make data processing and analytics easy and accessible to everyone. Cloud Dataflow complements the rest of the Google cloud platform and works very well with BigQuery
- In this project, We looked at google cloud dataflow deeply from three perspectives.

1. Cloud dataflow concepts and properties
2. Example in Java
3. Comparison between Spark and Google Cloud DataFlow

What is GOOGLE CLOUD DATAFLOW?

- Google Cloud Dataflow is a cloud-based data processing service for both batch and real-time data streaming applications.
- The Cloud Dataflow software expands on earlier Google parallel processing projects, including MapReduce, which originated at the company.
- Google Cloud Dataflow overlaps with competitive software frameworks and services such as Amazon Kinesis, Apache Storm, Apache Spark and Facebook Flux.
- It agnostically handles data of varying sizes and structures using a format called PCollections, which is short for "parallel collections."
- The Google Cloud Dataflow service also includes a library of parallel transforms, or PTransforms, which allow high-level programming of often-repeated tasks using basic templates

DATAFLOW FEATURES

- Reliable execution for large-scale data processing
- Resource Management
- On Demand
- Intelligent Work Scheduling
- Auto Scaling
- Unified Programming Model
- Open Source
- Monitoring
- Integrated
- Reliable and Consistent Processing

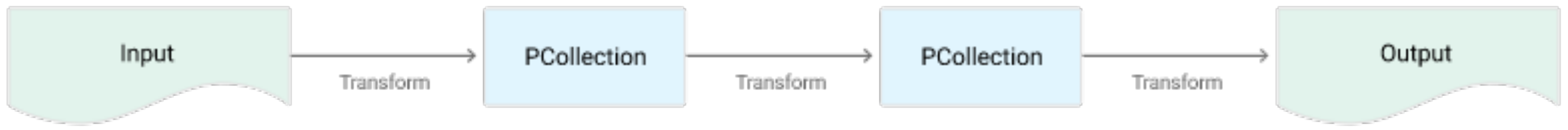
Pipeline Design Principles

- **The Dataflow SDK 2.x for Java and the Dataflow SDK for Python are based on Apache Beam.**

Points to Consider:

  - **Where is your input data stored?**
  - **What does your data look like?**
  - **What do you want to do with your data?**
  - **What does your output data look like, and where should it go?**

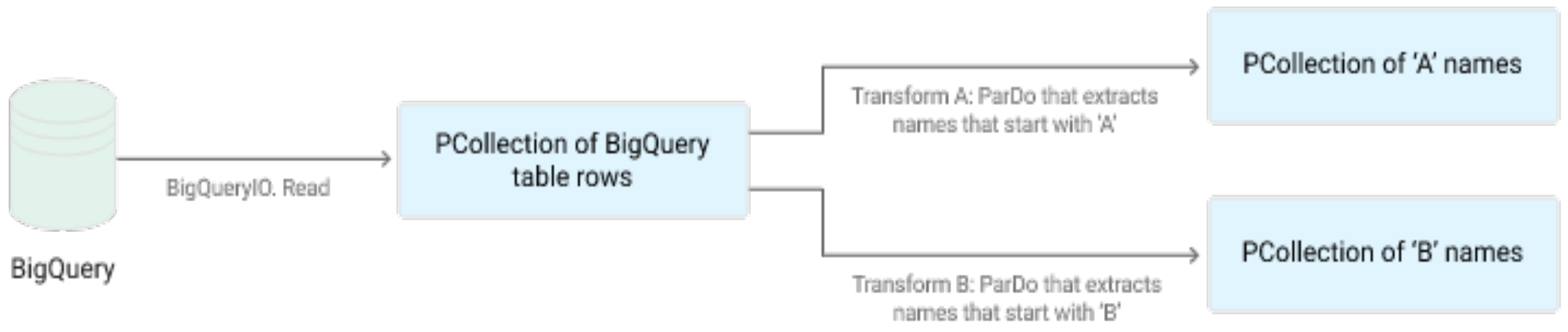
A basic pipeline



A pipeline represents a Directed Acyclic Graph of steps. It can have multiple input sources, multiple output sinks, and its operations (transforms) can output multiple PCollections

Different pipeline shapes

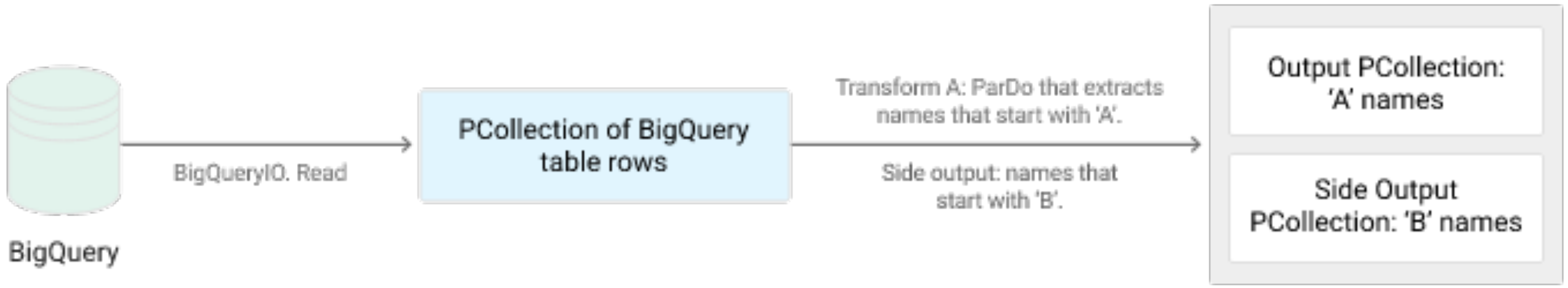
- Branching Pcollections:
  - Multiple transforms process the same Pcollection:


BigQuery
BigQueryIO. Read
PCollection of BigQuery table rows
Transform A: ParDo that extracts names that start with 'A'
Transform B: ParDo that extracts names that start with 'B'
PCollection of 'A' names
PCollection of 'B' names

```
if (starts with 'A') { outputToPCollectionA }
if (starts with 'B') { outputToPCollectionB }
```

Different pipeline shapes

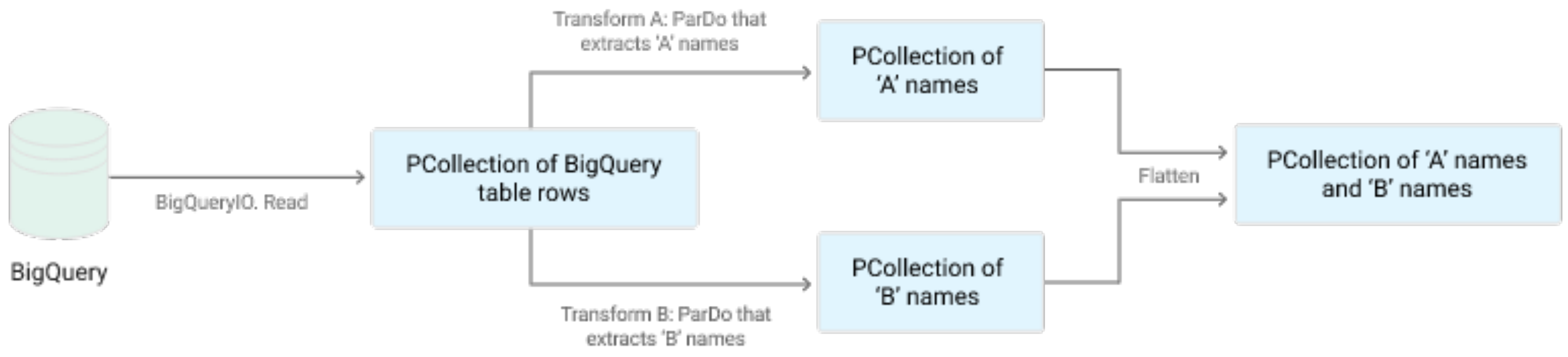
- Branching Pcollections:
  - A single transform that uses side outputs:


BigQuery
BigQueryIO. Read
PCollection of BigQuery table rows
Transform A: ParDo that extracts names that start with 'A'.
Side output: names that start with 'B'.
Output PCollection: 'A' names
Side Output PCollection: 'B' names

```
if (starts with 'A') { outputToPCollectionA }
else if (starts with 'B') { outputToPCollectionB }
```

Different pipeline shapes

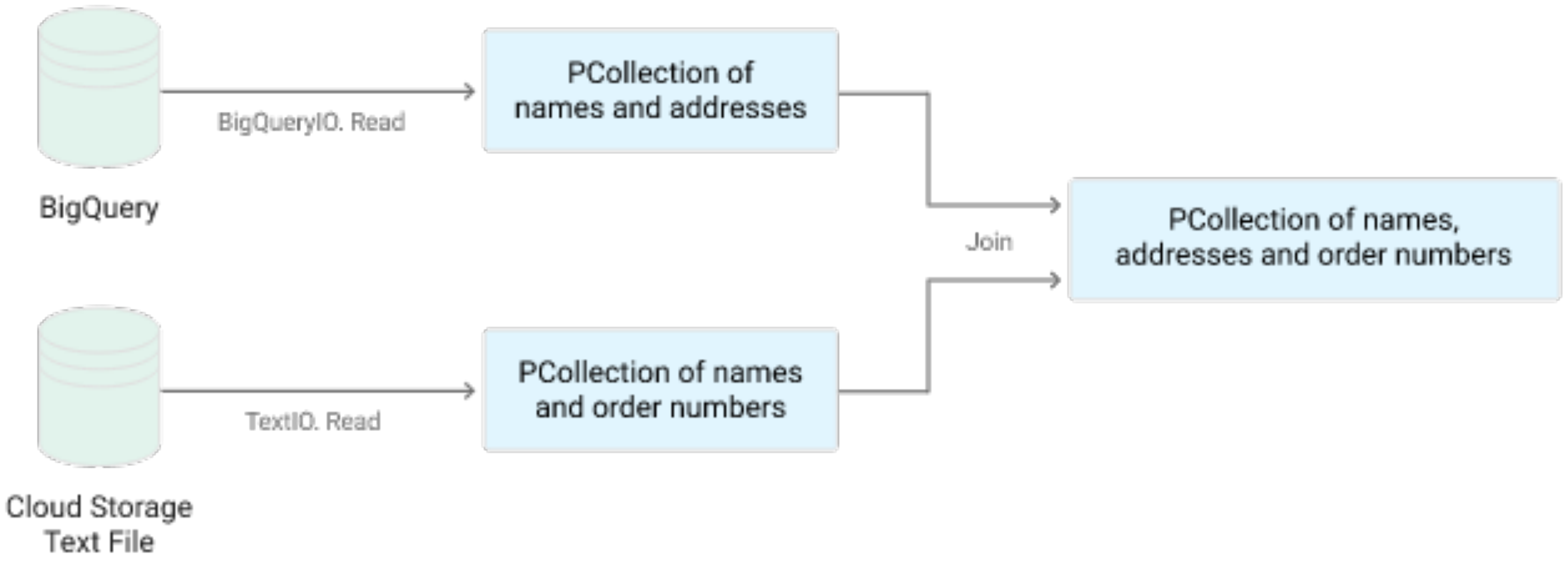
- Merging Pcollections:


Transform A: ParDo that extracts 'A' names
PCollection of 'A' names
BigQuery
BigQueryIO. Read
PCollection of BigQuery table rows
Flatten
PCollection of 'A' names and 'B' names
PCollection of 'B' names
Transform B: ParDo that extracts 'B' names

- **Flatten**
- **Join**

Different pipeline shapes

- Multiple sources:


BigQuery
BigQueryIO. Read
PCollection of names and addresses
Cloud Storage Text File
TextIO. Read
PCollection of names and order numbers
Join
PCollection of names, addresses and order numbers

Pipeline Creation

- Create a Pipeline object.
- Use a **Read** or **Create** transform to create one or more PCollections for your pipeline data.
- Apply **transforms** to each PCollection. Transforms can change, filter, group, analyze, or otherwise process the elements in a PCollection. Each transform creates a new output PCollection, to which you can apply additional transforms until processing is complete.
- **Write** or otherwise output the final, transformed PCollections.
- **Run** the pipeline.

Core Transforms

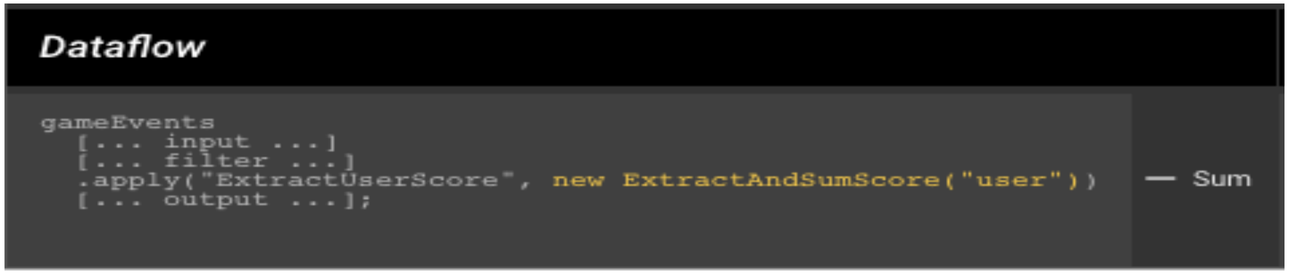
- ParDo
- GroupByKey
- Combine
- Flatten and Partition

Spark vs Cloud Dataflow

- we used a mobile gaming scenario as an example to compare dataflow vs spark four use-cases/three types of pipeline

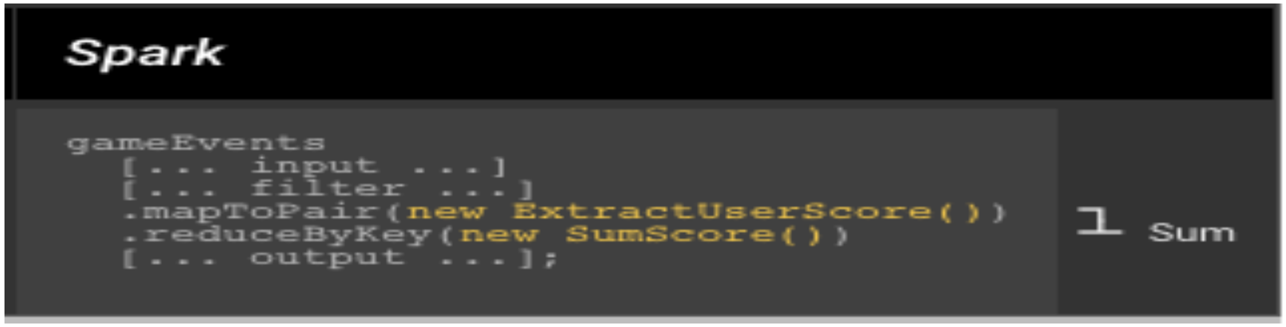
1. **User Scores** - A classic batch pipeline calculating per-user scores over a bounded set of input data.
2. **Hourly Team Scores** - A batch pipeline calculating per-hour, per-team scores over a bounded set of input data.
3. **Leaderboard** - A streaming pipeline continuously calculating two types of scores: per-hour, per-team scores as before, and cumulative per-user score totals over all time.
4. **Game Stats** - A streaming pipeline computing spam-filtered, per-hour, per-team scores, as well as a more complex hourly analysis of average per-user engagement time for the game.

Classic batch pipeline



Dataflow

```
gameEvents
  [... input ...]
  [... filter ...]
  .apply("ExtractUserScore", new ExtractAndSumScore("user"))
  [... output ...];
```

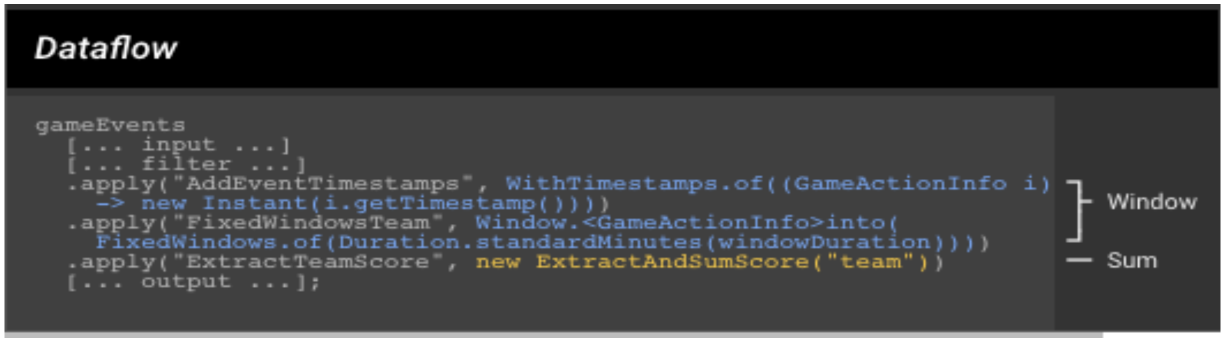


Spark

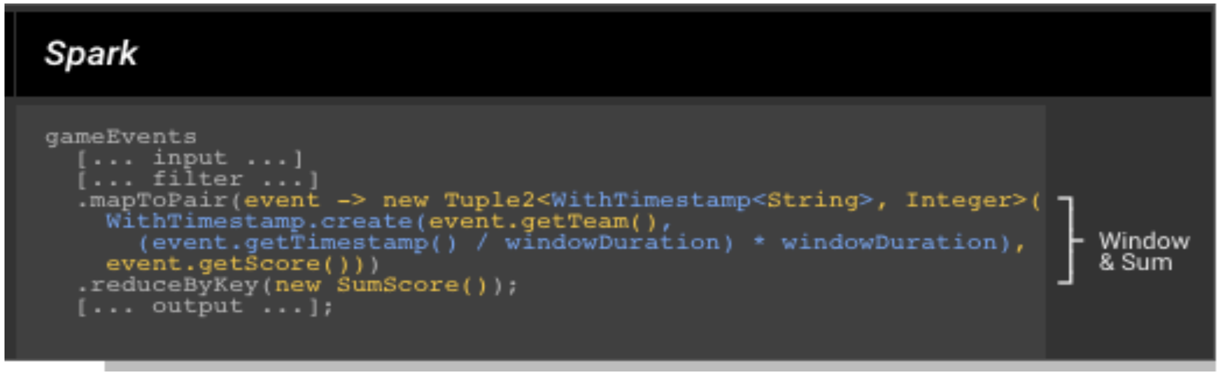
```
gameEvents
  [... input ...]
  [... filter ...]
  .mapToPair(new ExtractUserScore())
  .reduceByKey(new SumScore())
  [... output ...];
```

In dataflow, ExtractAndSumScore transform does exactly the same thing as the corresponding two lines on the right for Spark. However they are collapsed into a single composite transform in dataflow to provide more flexibility and modularity

Batch pipeline with windowing


Dataflow

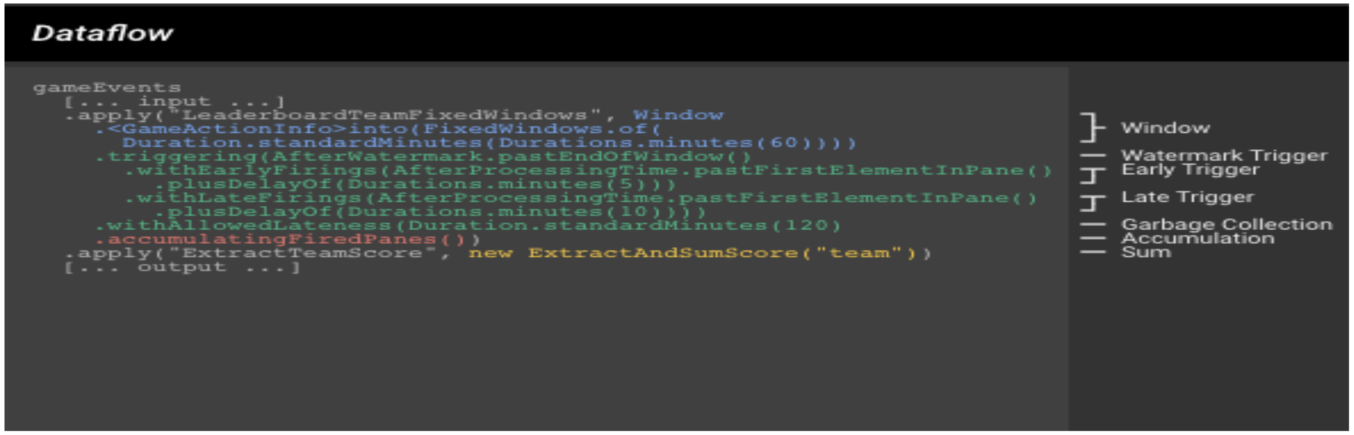
```
gameEvents
  [... input ...]
  [... filter ...]
  .apply("AddEventTimestamps", WithTimestamps.of((GameActionInfo i)
    -> new Instant(i.getTimestamp())))
  .apply("FixedWindowsTeam", Window.<GameActionInfo>into(
    FixedWindows.of(Duration.standardMinutes(windowDuration))))
  .apply("ExtractTeamScore", new ExtractAndSumScore("team"))
  [... output ...];
```


Spark

```
gameEvents
  [... input ...]
  [... filter ...]
  .mapToPair(event -> new Tuple2<WithTimestamp<String>, Integer>(
    WithTimestamp.create(event.getTeam(),
      (event.getTimestamp() / windowDuration) * windowDuration),
    event.getScore()))
  .reduceByKey(new SumScore());
  [... output ...];
```

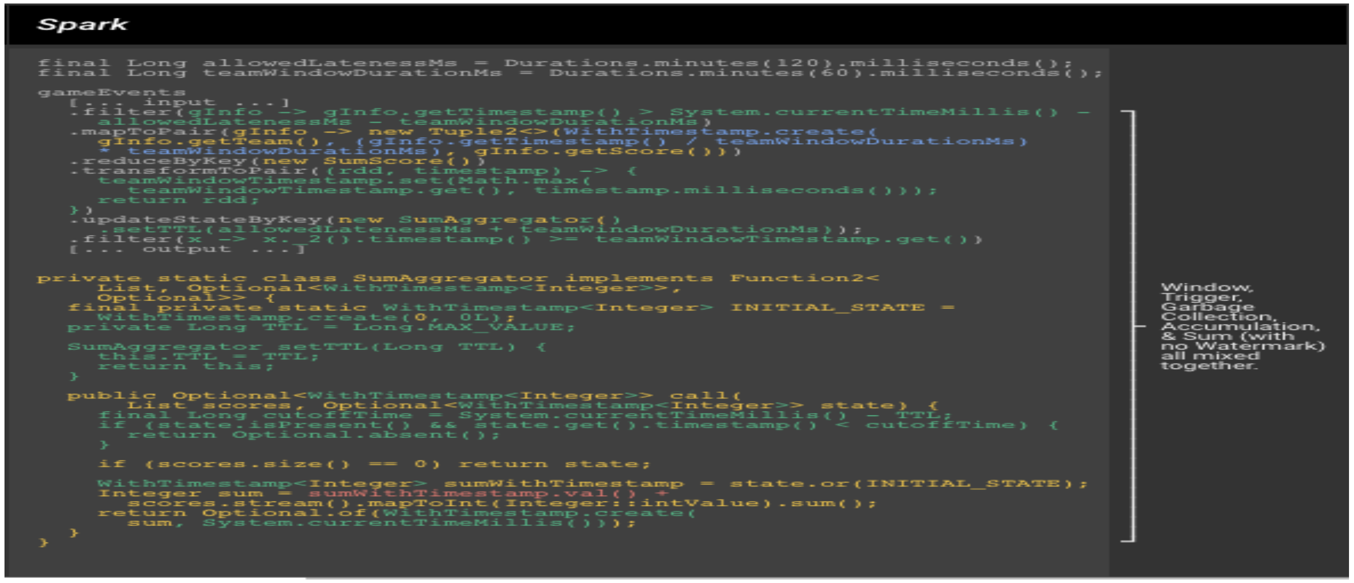
In this pipeline two new statements are added: one to assign timestamps to records, and another to assign a windowing strategy.
In the Spark model, due to lacking a formal notion of event-time windowing, it requires us to intermingle two statements, by introducing a representation of the current window into the data themselves as a secondary portion of the key.

Stream pipeline



Dataflow

```
gameEvents
  [... input ...]
  .apply("LeaderboardTeamFixedWindows", Window
    .<GameActionInfo>into(FixedWindows.of(
      Duration.standardMinutes(Durations.minutes(60))))
    .triggering(AfterWatermark.pastEndOfWindow()
      .withEarlyFirings(AfterProcessingTime.pastFirstElementInPane()
        .plusDelayOf(Durations.minutes(5)))
      .withLateFirings(AfterProcessingTime.pastFirstElementInPane()
        .plusDelayOf(Durations.minutes(10))))
    .withAllowedLateness(Duration.standardMinutes(120)
    .accumulatingFiredPanes())
  .apply("ExtractTeamScore", new ExtractAndSumScore("team"))
  [... output ...]
```



Spark

```
final Long allowedLatenessMs = Durations.minutes(120).milliseconds();
final Long teamWindowDurationMs = Durations.minutes(60).milliseconds();

gameEvents
  [... input ...]
  .filter(gInfo -> gInfo.getTimestamp() > System.currentTimeMillis() -
    allowedLatenessMs - teamWindowDurationMs)
  .mapToPair(gInfo -> new Tuple2<>(WithTimestamp.create(
    gInfo.getTeam(), (gInfo.getTimestamp() / teamWindowDurationMs)
      * teamWindowDurationMs), gInfo.getScore()))
  .reduceByKey(new SumScore())
  .transformToPair((rdd, timestamp) -> {
    teamWindowTimestamp.set(Math.max(
      teamWindowTimestamp.get(), timestamp.milliseconds()));
    return rdd;
  })
  .updateStateByKey(new SumAggregator()
    .setTTL(allowedLatenessMs + teamWindowDurationMs));
  .filter(x -> x._2().timestamp() >= teamWindowTimestamp.get())
  [... output ...]

private static class SumAggregator implements Function2<
    List, Optional<WithTimestamp<Integer>>,
    Optional>> {
  final private static WithTimestamp<Integer> INITIAL_STATE =
    WithTimestamp.create(0, 0L);
  private Long TTL = Long.MAX_VALUE;

  SumAggregator setTTL(Long TTL) {
    this.TTL = TTL;
    return this;
  }

  public Optional<WithTimestamp<Integer>> call(
      List scores, Optional<WithTimestamp<Integer>> state) {
    final Long cutoffTime = System.currentTimeMillis() - TTL;
    if (state.isPresent() && state.get().timestamp() < cutoffTime) {
      return Optional.absent();
    }

    if (scores.size() == 0) return state;

    WithTimestamp<Integer> sumWithTimestamp = state.or(INITIAL_STATE);
    Integer sum = sumWithTimestamp.val() +
      scores.stream().mapToInt(Integer::intValue).sum();
    return Optional.of(WithTimestamp.create(
      sum, System.currentTimeMillis()));
  }
}
```

Window, Trigger, Garbage Collection, Accumulation, & Sum (with no Watermark) all mixed together.

Now it's more obvious that dataflow version provides more clarity and modularity. While in spark, it has a lot of additional code, duplicated logic, and pieces of the four answers scattered across the entire codebase.

Installation and Quick Start

1. Open an account for Google Cloud Platform, install Google Cloud SDK, and create credentials for API
2. In google console, create a new project named first game, then go to the API manager to enable the DataFlow API. In google storage, create a new bucket for first-game project. Go to BigQuery, create a dataset, which is like a schema in relational database. The output data will be stored as tables in this BigQuery.
3. If project is written in Java:

   Make sure Java 1.8 and Maven are installed in your machine. Set up JAVA_HOME environment and add Maven bin directory to the path.
4. If it is written in Python:

   Create and open an environment for project with name glouddata and python 2.7
   Install google cloud data flow : pip install google-cloud-dataflow
5. Run word-count example in python:

   python -m apache_beam.examples.wordcount --output OUTPUT_FILE

Installation and Quick Start

6. Run example in Java:
   - create a maven project containing cloud dataflow

```
mvn archetype:generate -DarchetypeArtifactId=google-cloud-dataflow-java-
archetypes-examples -DarchetypeGroupId=com.google.cloud.dataflow -
DarchetypeVersion=1.9.0 -DgroupId=com.example -DartifactId=first-dataflow -
Dversion="0.1" -DinteractiveMode=false -Dpackage=com.example
```

   - build and run example wordcount pipeline using maven.

```
    mvn compile exec:java -Dexec.mainClass=com.example.WordCount -
Dexec.args="--output=./output/"
```

Conclusion

- Given the clear, practical, and robust approach that dataflow provides us, and with the flexibility and modularity, google cloud dataflow acts a very promising role in the next generation of real-time data-processing systems.
- It lets us write clean, modular code that evolves beautifully over time as needs change and expand.
- It has another bunches of advantages like providing better performance than spark; can be easily integrated with Google Platform and its different services; open-source; etc
- More detail could be found in “comparison dataflow vs spark.pdf”

YouTube URLs, Last Page

- Two minute (short):https://youtu.be/l2eHgQAWdio
- 15 minutes (long):https://youtu.be/-2sF5Q0TplA

References

- https://cloud.google.com/dataflow/
- https://cloud.google.com/dataflow/pipelines/constructing-your-pipeline
- http://searchdatamanagement.techtarget.com/definition/Google-Cloud-Dataflow
- https://beam.apache.org/documentation/programming-guide