Comparison: spark vs google cloud dataflow

Dataflow is unique amongst data parallel systems in that it is built upon a comprehensive model for out-of-order processing, it's designed to meet the challenges of real-time data processing without compromising correctness.
We'll use a mobile gaming scenario as an example to compare dataflow vs spark in detail.
There are four use-cases:

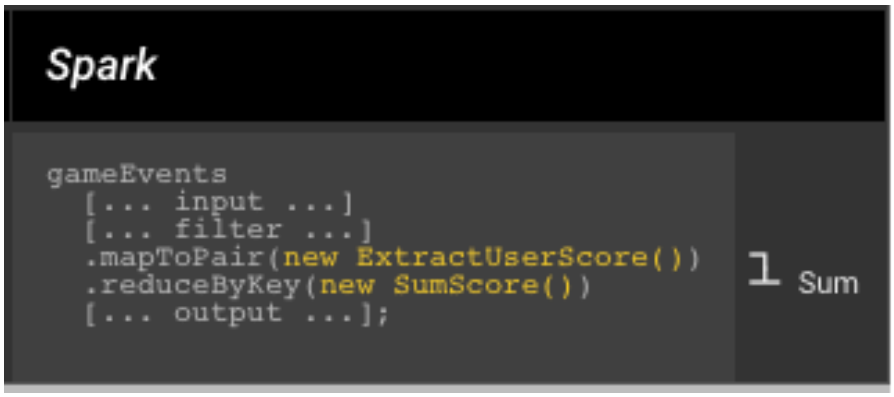
- **User Scores** - A classic batch pipeline calculating per-user scores over a bounded set of input data.
- **Hourly Team Scores** - A batch pipeline calculating per-hour, per-team scores over a bounded set of input data.
- **Leaderboard** - A streaming pipeline continuously calculating two types of scores: per-hour, per-team scores as before, and cumulative per-user score totals over all time.
- **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.

User Scores: Classic batch processing

Before going too farther we need to answer the following question first

What results are calculated?
Answer: Sums of integers, keyed by user

Basically we are going to build a pipeline that could be used to calculate the total cumulative scores racked up by each user of a given game over all time



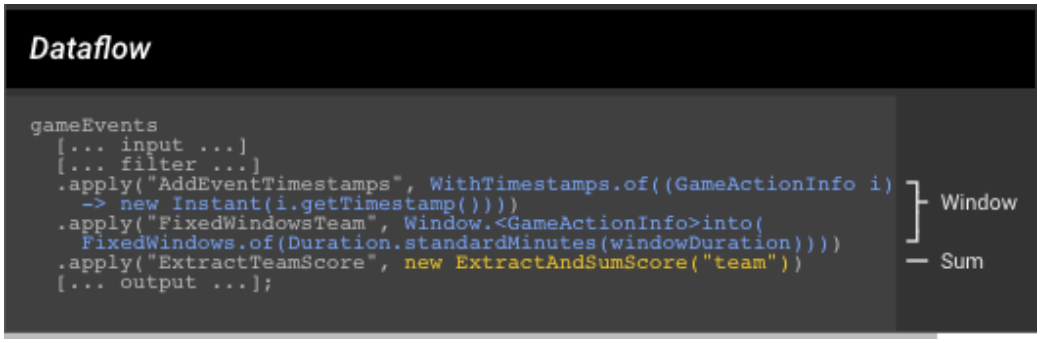
This is a very straight forward pipeline. One thing worth noting up is: ExtractAndSumScore transform does exactly the same thing as the corresponding two lines on the right for Spark; it's just collapsed into a single composite transform to ensure flexibility.

Hourly Team Scores: Windowing

First let's ask ourselves the following question

Where in event time are results calculated?
Within a single, implicit, global event-time window

So in here we are going to build up a batch pipeline with windowing



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 ...];
```

Window & Sum

Now you can see there is an obvious difference between dataflow and spark. In Dataflow, there are two separate statement, one for window, one for sum. However in spark, it's kind of intermingle together. In another way, that means, suppose one day you want to change a classic batch pipeline to use window. You can very easily to do that in dataflow by adding an extra line. while in spark, you probably need to do some sort of refactoring the old code to support new functionality. That's one part of beauty thing in the Dataflow model.

Leaderboard: Robust stream processing

This time we are going to look at Hourly Team Scores

When in processing time are results materialized?
Early: Every 5 mins of processing time
On-time: When the watermark passes the end of the window
Late: Every 10 mins of processing time
Final: When the watermark passes the end of the window + two hours
How do refinements of results relate?
Panes accumulate new values into prior results

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 ...]
```

Window
Watermark Trigger
Early Trigger
Late Trigger
Garbage Collection
Accumulation
Sum

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 if we only look at the number of codes needed, we can say that Dataflow is superior than spark. While after take a deeper look, we can see that dataflow provides a higher modularity and clarity, given the factor that in dataflow each statement supports it's own functionality. While in Spark, the lack of event-time windowing support requires us to emulate as many of these features as possible using the available APIs, resulting in additional code, duplicated logic, and pieces of the four answers scattered across the entire codebase.

Game Stats: Advanced stream processing

This time we are going to add two new advanced features to the pipeline, one spam filtering and another one user behavior analysis. There are going to be two sessions in this pipeline, One per-user session and one global session. For each kind of session:

Per-user session lengths

What? *Per-user sessions lengths in event time.*

Where? *Within session windows with gap duration of five minutes.*

When? *Once, when the watermark passes the end of the window.*

How? *Irrelevant, since only one output is produced per window.*

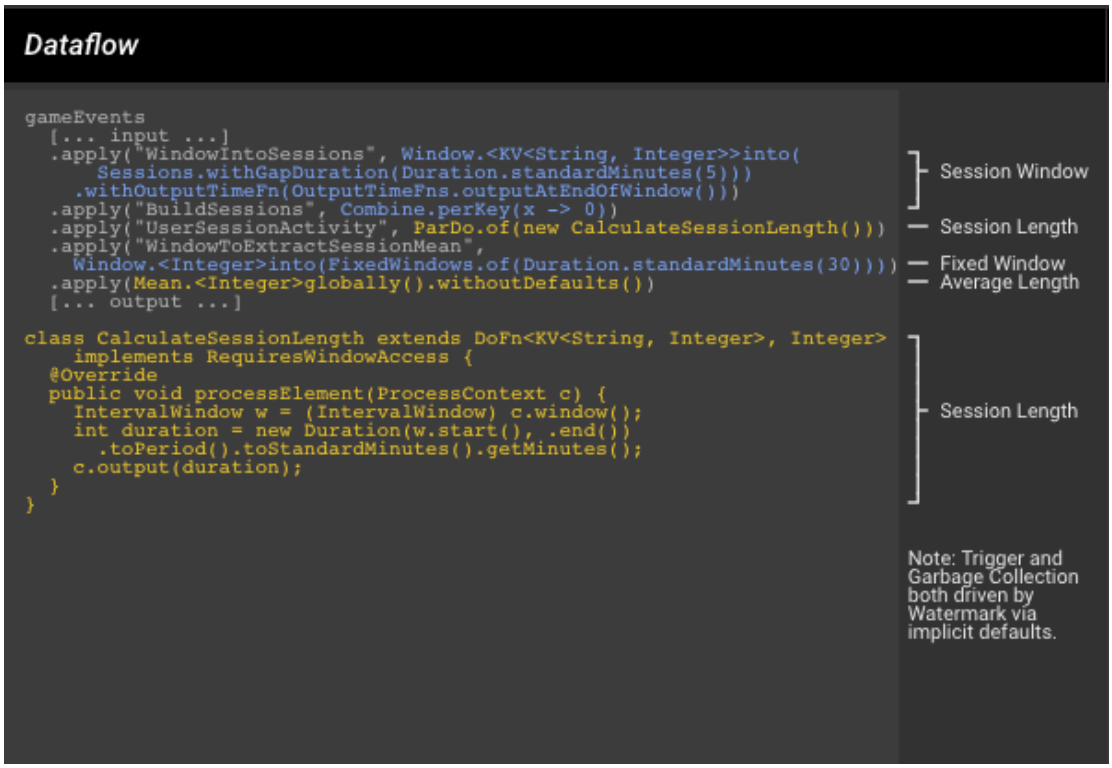
Global session length average

What? *Average session length for all users.*

Where? *Within fixed event-time windows of one hour.*

When? *Once, when the watermark passes the end of the window.*

How? *Irrelevant, since only one output is produced per window.*



Spark

```
final Duration userActivityWindow = Durations.standardMinutes(30);
final Long allowedLatenessMs =
  Durations.standardMinutes(120).milliseconds();

gameEvents
  [... input ...]
  .window(userActivityWindow, userActivityWindow))
  .mapToPair((GameActionInfo gInfo) -> new Tuple2<>(
    gInfo.getUser(), gInfo.getTimestamp()))
  .updateStateByKey(new SessionizeUserActivities()
    .setGap(Durations.minutes(5))
    .setTTL(allowedLatenessMs + userActivityWindow.milliseconds()))
  .mapValues(sessions -> sessions.stream()
    .filter(s -> s.closed)
    .collect(Collectors.toList()));
  .flatMapToPair(new SessionDurationByWindow(
    userActivityWindow.milliseconds()))
  .reduceByKey((a, b) -> new Tuple2<Long, Integer>(
    a._1() + b._1(), a._2() + b._2()))
  .mapValues(v -> v._1().doubleValue() / v._2() / 60.0 / 1000.0);
  [... output ...]

class SessionizeUserActivities implements Function2<
    List, Optional>, Optional>> {
  private Long sessionGap;
  private Long TTL;

  SessionizeUserActivities setGap(Long sessionGap) {
    this.sessionGap = sessionGap;
    return this;
  }

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

  private void closeIfExpired(Session session, long cutoffTime) {
    if (session.end <= cutoffTime)
      session.close();
  }

  @Override
  public Optional> call(
      List timestamps, Optional> state) {
    List sessions = state.or(new ArrayList())
      .stream()
      .filter(session -> !session.closed)
      .collect(Collectors.toList());
    final Long cutoffTime = System.currentTimeMillis() - TTL;

    for (Long ts : timestamps) {
      if (ts > cutoffTime) sessions.add(new Session(ts, ts));
    }

    if (sessions.size() == 0) return Optional.absent();

    List mergedSessions = new ArrayList<>();
    sessions.sort((a, b) -> (a.end.compareTo(b.end)));
    Session current = sessions.get(0);
    for (Session next : sessions.subList(1, sessions.size())) {
      if (next.start < current.end + sessionGap) {
        current.merge(next);
      } else {
        closeIfExpired(current, cutoffTime);
        mergedSessions.add(current);
        current = next;
      }
    }
    closeIfExpired(current, cutoffTime);
    mergedSessions.add(current);

    if (mergedSessions.size() == 0) return Optional.absent();

    return Optional.of(mergedSessions);
  }
}

class SessionDurationByWindow implements PairFlatMapFunction<
    Tuple2<String, List<Session>>, Long, Tuple2<Long, Integer>> {
  private Long windowDuration;

  SessionDurationByWindow(Long windowDuration) {
```

Session Window, Session Length, Fixed Window, Average Length, Trigger, & Garbage Collection (with no Watermark) all mixed together.

As we can see, the Dataflow code is still very nicely partitioned, while the Spark code still remains a tangle of transformation, windowing, and triggering logic mixed together and spread out across the various classes and method invocations. It's also quite a lot of code.

Conclusion

From the above example, we kind of get the expression that dataflow provides the flexibility and power necessary for the next generation of real-time data-processing systems, with a clear, practical, and robust approach to out-of-order processing. It lets us write clean, modular code that evolves beautifully over time as needs change and expand. it looks very promising even through it’s still very young. The model maps directly onto the four questions that are relevant in any out-of-order data processing pipeline:

- What results are calculated? Answered via transformations.
- Where in event time are results calculated? Answered via event-time windowing.
- When in processing time are results materialized? Answered via watermarks, triggers, and allowed lateness.
- How do refinements of results relate? Answered via accumulation modes

One more thing need to be noticed is that google cloud dataflow has better performance than spark.

Reference:

https://cloud.google.com/dataflow/blog/dataflow-beam-and-spark-comparison

http://www.infoworld.com/article/3064728/analytics/google-cloud-dataflow-vs-apache-spark-benchmarks-are-in.html

http://stackoverflow.com/questions/33518104/google-dataflow-vs-apache-spark