# Aggregating Records with Crunch



## Purpose

The Apache Crunch Java library provides a framework for writing, testing, and running applications in MapReduce pipelines.

This tutorial demonstrates how you use Kite's `CrunchDatasets` helper methods to configure source and target datasets for Crunch operations that aggregate individual events into work sessions.

## Prerequisites

- A VM or cluster with CDH installed. See Preparing the Virtual Machine.
- An events dataset populated with sample records.

## Creating the sessions.avsc Schema File

The `session.avsc` schema defines the fields and data types in the `sessions` dataset.

### session.avsc

```
{
  "name": "Session",
  "namespace": "org.kitesdk.examples.demo.event",
  "type": "record",
  "doc": "A session derived from a standard event log.",
  "fields": [
    {
      "name": "user_id",
```

```
      "type": "long",
      "doc": "A unique identifier for the user. Required."
    },
    {
      "name": "session_id",
      "type": "string",
      "doc": "A unique identifier for the session. Required."
    },
    {
      "name": "ip",
      "type": "string",
      "doc": "The IP address of the host where the event originated. Required."
    },
    {
      "name": "start_timestamp",
      "type": "long",
      "doc": "The point in time when the session started, represented as the number
    },
    {
      "name": "duration",
      "type": "long",
      "doc": "The duration of the session in milliseconds. Required."
    },
    {
      "name": "session_event_count",
      "type": "int",
      "doc": "The number of events that occurred in the session. Required."
    }
  ]
}
```

Save the schema definition as `session.avsc`.

# Creating the Sessions Dataset Using the Kite CLI

Create the `sessions` dataset using the default Hive scheme.

To create the sessions dataset:

1. Open a terminal window.
2. Use the `create` command to create the dataset. This example assumes that the schema file is in your home directory. Change the path to `session.avsc` if you stored it in another directory.

```
kite-dataset create sessions --schema ~/session.avsc
```

Use Hue to confirm `sessions` is in your list of Hive tables.

# Aggregating Event Data into Sessions Using Crunch

Run the following command.

```
cd demo-crunch
mvn kite:run-tool
```

The kite:run-tool Maven goal executes the run method of AggregateEvents, which launches a Crunch job on the cluster.

The `Tool` class and cluster settings are configured in the `kite-maven-plugin` element of `pom.xml`.

When the Crunch job is complete, the `sessions` dataset is populated with records representing groups of events with the same session ID. Use the Hive or Impala query editor to see the records in the `sessions` dataset.

# Understanding the AggregateEvents Class

AggregateEvents uses a Crunch `Pipeline` object to apply functions to input data. The general workflow in Crunch is:

1. Read input (for example, a dataset) into a Pipeline object.
2. Perform functions on the input data.
3. Write the result to a destination (for example, another dataset).

The following is the complete code listing for AggregateEvents, with comments inline.

## AggregateEvents.java

```
package org.kitesdk.examples.demo;

import java.io.Serializable;
import java.net.URI;
import java.util.Calendar;
```

```
import java.util.Iterator;
import java.util.TimeZone;
import org.apache.crunch.DoFn;
import org.apache.crunch.Emitter;
import org.apache.crunch.MapFn;
import org.apache.crunch.PCollection;
import org.apache.crunch.Pair;
import org.apache.crunch.Target;
import org.apache.crunch.types.avro.Avros;
import org.apache.crunch.util.CrunchTool;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.util.ToolRunner;
import org.kitesdk.data.Dataset;
import org.kitesdk.data.Datasets;
```

Kite's CrunchDatasets class methods handle the underlying code required to create source and target datasets for use with Crunch.

```
import org.kitesdk.data.crunch.CrunchDatasets;
import org.kitesdk.data.event.StandardEvent;
import org.kitesdk.data.spi.filesystem.FileSystemDatasets;
import org.kitesdk.examples.demo.event.Session;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class AggregateEvents extends CrunchTool implements Serializable {

  private static final Logger LOG = LoggerFactory.getLogger(AggregateEvents.class);

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

    // Turn debug on while in development.
    getPipeline().enableDebug();
    getPipeline().getConfiguration().set("crunch.log.job.progress", "true");
```

Step 1. Load the source dataset. In this example, use the `events` dataset from Hive.

```
    Dataset<StandardEvent> eventsDataset = Datasets.load(
        "dataset:hive:events", StandardEvent.class);
```

If the dataset is empty, stop.

```
    if (eventsDataset.isEmpty()) {
      LOG.info("No records to process.");
      return 0;
    }
```

Still going? Good. Create a *parallel collection* using the `events` dataset as the source. A parallel collection (`PCollection`) is a set of distributed objects that can be processed in parallel.

```
    PCollection<StandardEvent> events = read(
        CrunchDatasets.asSource(eventsDataset));
```

Step 2. Process the data.
a. Create a session key by combining the user ID and session ID.
b. Group together all events with the same session key.
c. For each group, create a Session record as an Avro Specific object.

```
    PCollection<Session> sessions = events
        .by(new GetSessionKey(), Avros.strings())
        .groupByKey()
        .parallelDo(new MakeSession(), Avros.specifics(Session.class));
```

Step 3. Append the derived sessions to the target dataset, defined using the Kite `CrunchDatasets.asTarget` method.

```
    getPipeline().write(sessions,
        CrunchDatasets.asTarget("dataset:hive:sessions"),
        Target.WriteMode.APPEND);
    return run().succeeded() ? 0 : 1;
  }
```

The `GetSessionKey` class combines the session ID with the user ID to create a unique session key.

```
  private static class GetSessionKey extends MapFn<StandardEvent, String> {
    @Override
    public String map(StandardEvent event) {
      // Create a key by combining the session id and user id
      return event.getSessionId() + event.getUserId();
    }
  }
```

The `MakeSession` class extends `DoFn` (Do Function), where you handle transformation of your data

before it goes into the target dataset.

```
private static class MakeSession
    extends DoFn<Pair<String, Iterable<StandardEvent>>, Session> {
```

The `process` method iterates through a group of events that have the same session key.

```
  @Override
  public void process(
      Pair<String, Iterable<StandardEvent>> keyAndEvents,
      Emitter<Session> emitter) {
    final Iterator<StandardEvent> events = keyAndEvents.second().iterator();
    if (!events.hasNext()) {
      return;
    }
```

Initialize the values needed to create a session object for this group.

```
    final StandardEvent firstEvent = events.next();
    long startTime = firstEvent.getTimestamp();
    long endTime = firstEvent.getTimestamp();
    int numEvents = 1;
```

Inspect each event in this session group. Track the earliest timestamp (start time) and latest timestamp (end time). Keep a count of the events in this session group.

```
    while (events.hasNext()) {
      final StandardEvent event = events.next();
```

Reset the start time if the timestamp is earlier than the current start time value.

```
      startTime = Math.min(startTime, event.getTimestamp());
```

Reset the end time if the timestamp is later then the current end time value.

```
      endTime = Math.max(endTime, event.getTimestamp());
```

Keep a count of the events in this session group.

```
        numEvents += 1;
      }
```

Create a session using the Crunch `emitter`. Use values from the first event in the group for fields that don't change. Crunch repeats the `emit` process for each group of events until all sessions are processed.

```
      emitter.emit(Session.newBuilder()
          .setUserId(firstEvent.getUserId())
          .setSessionId(firstEvent.getSessionId())
          .setIp(firstEvent.getIp())
          .setStartTimestamp(startTime)
          .setDuration(endTime - startTime)
          .setSessionEventCount(numEvents)
          .build());
    }
  }

  public static void main(String... args) throws Exception {
    int rc = ToolRunner.run(new AggregateEvents(), args);
    System.exit(rc);
  }

}
```