Scala / Scio Cheatsheet

Slides and code: `https://github.com/kanterov/lambdaconf-2017-bigdata`.
For more documentation checkout Github: `https://github.com/spotify/scio`.

sbt

```
./sbt

// run code for unit-1 and tests
> unit-1
> test-unit-1

// display input and output
> avro-read in/track_play
> avro-read out/play_count
```

ScioContext

```
/** Distribute a local Scala ‘Iterable‘ to form an SCollection. */
def parallelize[T](elems: Iterable[T]): SCollection[T]

/** Get an SCollection for an Avro file. */
def scalaAvroFile[T](path: String): SCollection[T]

/** Save this SCollection as an Avro file. */
def saveAsScalaAvroFile[T](path: String): SCollection[T]
```

SCollection[A]

```
/** Return a new SCollection by applying a function to
  * all elements of this SCollection. */
def map[B](f: A => B): SCollection[B]

/** Create tuples of the elements in this SCollection by applying ‘f‘. */
def keyBy[B](f: A => B): SCollection[(A, B)]

/** Return a new SCollection by first applying a function to
  * all elements of this SCollection, and then flattening the
  * results. */
def flatMap[B](f: A => Iterable[B]): SCollection[B]

/** Return a new SCollection containing only the elements
  * that satisfy a predicate. */
def filter(f: A => Boolean): SCollection[A]

/** Return a new SCollection containing the distinct elements
  * in this SCollection. */
def distinct: SCollection[A]

/** Count the number of elements in the SCollection. */
def count: SCollection[Long]

/** Reduce the elements of this SCollection using the specified
  * commutative and associative binary operator. */
def reduce(op: (T, T) => T): SCollection[T]

/** Reduce with Semigroup. */
def sum(implicit A: Semigroup[A]): SCollection[A]

/** Return the top k (largest) elements from this SCollection */
def top(num: Int)(implicit ord: Ordering[T]): SCollection[Iterable[T]]
```

SCollection[(K, V)]

```
/** Return an SCollection with the values of each tuple. */
def keys: SCollection[K]

/** Return an SCollection with the keys of each tuple. */
def values: SCollection[V]

/** Merge the values for each key using an associative reduce function. */
def reduceByKey(op: (V, V) => V): SCollection[(K, V)]

/** Group the values for each key in the SCollection into a single sequence. */
def groupByKey: SCollection[(K, Iterable[V])]

/** Count the number of elements for each key.  */
def countByKey: SCollection[(K, Long)]

/** Reduce by key with Semigroup.  */
def sumByKey(num: Int)(implicit V: Semigroup[V]): SCollection[(K, V)]

/** Return the top k (largest) values for each key. */
def topByKey(num: Int)(implicit V: Ordering[V]): SCollection[(K, Iterable[V])]

/** Return an SCollection containing all pairs of elements with matching
  * keys in this and that. */
def join[W](that: SCollection[(K, W)]): SCollection[(K, (V, W))]

/** Return an SCollection containing all pairs (k, (v, Some(w))) for w in ‘that‘,
  * or the pair (k, (v, None)) if no elements in ‘that‘ have key k. */
def leftOuterJoin[W](that: SCollection[(K, W)]): SCollection[(K, (V, Option[W]))]

/** Perform an inner join by replicating that to all workers. */
def hashJoin[W](that: SCollection[(K, W)]): SCollection[(K, (V, W))]
```

WindowOptions

```scala
val default = WindowOptions[IntervalWindow](
  allowedLateness = Duration.ZERO,
  trigger = Repeatedly.forever(AfterWatermark.pastEndOfWindow()),
  accumulationMode = DISCARDING_FIRED_PANES)

val allowLateness = WindowOptions[IntervalWindow](
  allowedLateness = Duration.standardDays(1),
  trigger = Repeatedly.forever(AfterWatermark.pastEndOfWindow()),
  accumulationMode = DISCARDING_FIRED_PANES)

val speculative = WindowOptions[InternalWindow](
  allowedLateness = Duration.standardDays(1),
  trigger = Repeatedly.forever(
    AfterProcessingTime
      .pastFirstElementInPane()
      .plusDelayOf(Duration.standardMinutes(1))),
  accumulationMode = ACCUMULATING_FIRED_PANES)

val sequential = WindowOptions[InternalWindow](
  allowedLateness = ONE_DAY,
  trigger = AfterEach.inOrder(
    Repeatedly.forever(
      AfterProcessingTime
        .pastFirstElementInPane()
        .plusDelayOf(Duration.standardMinutes(1)))
      .orFinally(
        AfterWatermark.pastEndOfWindow()),
    Repeatedly.forever(
      AfterProcessingTime
        .pastFirstElementInPane()
        .plusDelayOf(Duration.standardMinutes(5)))),
  accumulationMode = ACCUMULATING_FIRED_PANES)
```

Accumulators

```scala
// create accumulators to be updated inside the pipeline
val max = sc.maxAccumulator[Int]("max")
val min = sc.minAccumulator[Int]("min")
val sum = sc.sumAccumulator[Int]("sum")
val count = sc.sumAccumulator[Int]("count")

sc.parallelize(1 to 100)
  // accumulators to be used in the next transform
  .withAccumulator(max, min, sum, count)
  // accumulators available via the second argument AccumulatorContext
  .filter { (i, ctx) =>
    // update accumulators via the context
    ctx.addValue(max, i)
       .addValue(min, i)
       .addValue(sum, i)
       .addValue(count, 1)

    i <= 50
  }.toSCollection
```