distributed-dataset

Utku Demir

March 2019, FP Auckland Meetup

distributed-dataset

- A framework to process large amounts of data in a distributed fashion.
- Borrows many ideas from Apache Spark.
  - Resilient Distributed Dataset (RDD)
  - Tasks, executors, driver, stages
  - Differs in some ways:
    - Short-lived executors - executor per task
    - More strongly-typed
    - Composable aggregations
    - Aims to be self-contained
    - Written in Haskell

How?

- A **Dataset** is a multiset of items.
- High level operators to transform **Dataset**'s.
  - map, filter
  - aggregations (count, average etc.)
- A **Dataset** consists of many **Partition**s.
- Each **Partition** can be processed in parallel.

In detail

- A **Dataset** is a collection of rows.
- A Dataset is stored as many **Partitions**.

```
dExternal :: [Partition a]
          -> Dataset a

mkPartition :: Closure (ConduitT () a (ResourceT IO) ())
            -> Partition a
```

In detail

- Lazily transform this Dataset with high level operators
  - map, filter, reduce eg.

```
dMap    :: Closure (a -> b) -> Dataset a -> Dataset b
dFilter :: Closure (a -> Bool) -> Dataset a -> Dataset a
```

Composable Aggregations

```
data Aggr a b =
  forall t. StaticSerialise t =>
  Aggr (Closure (a -> t))                          -- ^ Map
       (Closure (Dict (CommutativeMonoid t)))      -- ^ Reduce
       (Closure (t -> b))                          -- ^ Extract

instance StaticApply (Aggr a)
instance StaticProfunctor Aggr

dAggr :: Aggr a b -> Dataset a -> IO b
dGroupedAggr :: StaticHashable k
             => Aggr a b
             -> Dataset (k, a)
             -> Dataset (k, b)
```

Example

```
ghArchive (fromGregorian 2018 1 1, fromGregorian 2018 12 31)
  -- :: Dataset GHEvent
  & dConcatMap (static (\e ->
      let author = e ^. gheActor . ghaLogin
          commits = e ^.. gheType . _GHPushEvent
                      . ghpepCommits . traverse . ghcMessage
      in  map (author, ) commits
    ))
  -- :: Dataset [(Text, Text)]
  & dFilter (static (\(_, commit) ->
      T.pack "cabal" `T.isInfixOf` T.toLower commit
    ))
  -- :: Dataset [(Text, Text)]
  & dGroupedAggr 50 (static fst) dCount
  -- :: Dataset [(Text, Int)]
  & dToList
  -- :: [(Text, Int)]
  & sortOn (Down . snd) & take 20 & mapM_ (liftIO . print)
```

Behind the Scenes

```
data Dataset a where

  DExternal  :: Typeable a => [Partition a] -> Dataset a

  DPipe      :: (StaticSerialise a, StaticSerialise b)
             => Closure (ConduitT a b (ResourceT IO) ())
             -> Dataset a -> Dataset b

  DPartition :: (StaticHashable k, StaticSerialise a)
             => Int
             -> Closure (a -> k)
             -> Dataset a -> Dataset a
```

Behind the Scenes

In order to execute our transforms, we need two things:

- A **Backend** to run remote functions.
- A **ShuffleStore** to store intermediate outputs.

Both can be supplied externally.

distributed-dataset-aws

- Provides an AWS Lambda Backend, and an S3 ShuffleStore.
  - Scales well
  - No infrastructure necessary

Alternatives in Haskell (as I know of)

Sparkle

A library for writing resilient analytics applications in Haskell that scale to thousands of nodes, using Spark and the rest of the Apache ecosystem under the hood.

Hadron

Construct and run Hadoop MapReduce programs in Haskell

HSpark

A port of Apache Spark to Haskell using distributed process

Cloud Haskell

Erlang-style concurrency in Haskell

Thanks!

Questions?

Extras

An external data source

```haskell
import Network.HTTP.Simple
import Data.Conduit.Zlib (ungzip)
import Data.Conduit.JSON.NewlineDelimited as NDJ

data GHEvent = ... deriving FromJSON

urlToPartition :: String -> Partition GHEvent
urlToPartition url' = mkPartition $ (\url -> do
  req <- parseRequest url
  httpSource req getResponseBody
    .| ungzip
    .| NDJ.eitherParser @_ @GHEvent
    .| C.mapM (either fail return)
  ) `cap` cpure (static Dict) url'
```

How to aggregate

```
input & groupedAggr 3 (static getColor) dSum
```

- Input
  - Partition 1: [3, 5, 2, 1]
  - Partition 2: [3, 7, 2]
  - Partition 3: [1, 2, 8]
- Aggregation Step 1:
  - Partition 1: [5, 5, 1]
  - Partition 2: [5, 7]
  - Partition 3: [1, 2, 8]
- Shuffle!
  - Partition 1: [5, 7, 2]
  - Partition 2: [5, 1, 1]
  - Partition 3: [5, 8]
- Aggregation Step 2:
  - Partition 1: [14]
  - Partition 2: [6, 1]
  - Partition 3: [13]

Composing Aggr's

```
dConstAggr :: (Typeable a, Typeable t)
           => Closure a -> Aggr t a
dSum :: StaticSerialise a
     => Closure (Dict (Num a)) -> Aggr a a

dCount :: Typeable a => Aggr a Integer
dCount =
  static (const 1) `staticLmap` dSum (static Dict)

dAvg :: Aggr Double Double
dAvg =
  dConstAggr (static (/))
    `staticApply` dSum (static Dict)
    `staticApply` staticMap (static realToFrac) dCount
```