Experimenting with Distributed Data Processing in Haskell

Utku Demir

YOW! Lambda Jam 2019

Distributed Data Processing

- Lots of data ($>$ TB)
- Complex, flexible queries

Resilient Distributed Datasets (Apache Spark)

- **Dataset** is a multiset of rows.
- A Dataset is stored as many **Partition**'s.
- Partitions can be processed in parallel on many **Executor**'s.
- High-level combinators for transforming a Dataset
  - `map`, `filter`, `groupBy`, `reduce` eg.

distributed-dataset

- Written in Haskell.
- Borrows ideas from Apache Spark.
- Composes nicely with the existing Haskell ecosystem.
- Can be used with “Function as a Service” / “Serverless” offerings.

-XStaticPointers

- `GHC.StaticPtr`

  The language extension -XStaticPointers adds a new syntactic form 'static e', which stands for a reference to the closed expression 'e'. This reference remains valid across different processes on possibly different machines. [1]

- distributed-closure

  This package implements a serializable closure abstraction on top of static pointers. [2]

[1] GHC User Guide
[2] https://github.com/tweag/distributed-closure

In detail

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

data Dataset a where
  DExternal  :: StaticSerialise 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
```

Aggregations

```
-- Aggregates many a's to a single b.
data Aggr a b = ...

instance StaticApply (Aggr a)
instance StaticProfunctor Aggr

aggrFromMonoid :: StaticSerialise a
               => Closure (Dict (Monoid a))
               -> Aggr a a

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

Backend

- LocalProcessBackend
- distributed-dataset-aws
  - Uses AWS Lambda to run executors and S3 to exchange information.
  - Scales well, cost-effective
  - No infrastructure necessary

Example

```
ghArchive (fromGregorian 2018 1 1, fromGregorian 2018 12 31)
  & dConcatMap (static (\e ->
      let author = e ^. gheActor . ghaLogin
          commits = e ^.. gheType . _GHPushEvent
                        . ghpepCommits . traverse . ghcMessage
      in  map (author, ) commits
    ))
  & dFilter (static (\(_, commit) ->
      "cabal" `T.isInfixOf` T.toLower commit
    ))
  & dGroupedAggr 50 (static fst) dCount
  & dAggr (dTopK (static Dict) 20 (static snd))
  >>= mapM_ (liftIO . print)
```

- 126 GB compressed, 909 GB uncompressed JSON
- 2190 executors
- $\sim$ 2 minutes, including the time to deploy infrastructure

Future

- Performance optimisations
- More backends
  - static machines, Apache OpenWhisk, YARN
- Documentation, tutorials
- ML algorithms
  - clustering, recommendation

Alternatives in Haskell (as far as I know)

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*

Thanks!

- https://github.com/utdemir/distributed-dataset
- me@utdemir.com

Questions?

Extras

An external data source

```
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 (static Dict))
```

- 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
```

Results

```
("peti",899)
("haskell-pushbot",535)
("bgamari",418)
("phadej",307)
("23Skidoo",208)
("alanz",174)
("edolstra",141)
("quasicomputational",136)
("jneira",135)
("hvr",133)
("Ericson2314",133)
("felixonmars-bot",130)
("DanielG",129)
("philderbeast",120)
("coreyoconnor",115)
("rcaballeromx",109)
```