

Аналізуємо події за допомогою Riak, Pipes та Foldl

Kostiantyn Rybnikov

November 26, 2016

Огляд проблеми, яку вирішуємо

- Порахувати від 1 до 1000 не так-то й просто
- Система записує складні структуровані дані у великій кількості
- Їх потрібно аналізувати, генеруючи деякий результат, візуалізацію

Мотивація, більшість існуючих систем

- Слабка або відсутність типізації
- Не на Хаскелі
- Важко писати та підтримувати map/reduce
- Важко тестувати
- Відсутність композиції

“Beautiful folds” (Красиві згортачі?)

- Beautiful folding (Max Rabkin, 2008)
- Composable streaming folds (Gabriel Gonzalez, 2013)
- foldl-1.0.0: Composable, streaming, and efficient left folds (Gabriel Gonzalez, 2013)
- Популяризовано Scala-бібліотекою `algebird`, див MuniHac 2016: Beautiful folds are practical, too

Beautiful folds – проблема

```
:{  
Prelude Data.List Data.List| sum :: (Num a) => [a] -> a  
Prelude Data.List Data.List| sum = foldl' (+) 0  
Prelude Data.List Data.List| :}  
>>> genericLength [1..100000000]  
100000000  
>>> sum [1..100000000]  
50000000050000000  
>>> let average xs = sum xs / genericLength xs  
>>> average [1..100000000]  
<Huge space leak>
```

Beautiful folds – наївне вирішення

```
mean :: [Double] -> Double
mean = go 0 0
  where
    go s l []      = s / fromIntegral l
    go s l (x:xs) = s `seq` l `seq`
                      go (s+x) (l+1) xs
```

Еволюція

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
```

```
data Fold b c = forall a. F (a -> b -> a) a (a -> c)
```

```
data Fold b a = F (a -> b -> a) a
```

```
data Fold i o = forall m . Monoid m => Fold (i -> m) (m -> o)
```

Beautiful folds в один слайд

Імплементація в 14 строк Хаскеля:

```
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE RankNTypes                #-}

import Control.Lens (Getting, foldMapOf)

data Fold i o = forall m . Monoid m => Fold (i -> m) (m -> o)

instance Functor (Fold i) where
  fmap k (Fold tally summarize) = Fold tally (k . summarize)

instance Applicative (Fold i) where
  pure o = Fold (\_ -> ()) (\_ -> o)

  Fold tallyF summarizeF <*> Fold tallyX summarizeX = Fold tally summarize
    where
      tally i = (tallyF i, tallyX i)
      summarize (mF, mX) = summarizeF mF (summarizeX mX)

focus :: (forall m . Monoid m => Getting m b a) -> Fold a o -> Fold b o
focus lens (Fold tally summarize) = Fold (foldMapOf lens tally) summarize
```


Простий приклад

```
{-# LANGUAGE ExistentialQuantification #-}

import Data.Monoid
import Prelude hiding (sum)

import qualified Data.Foldable

data Fold i o = forall m . Monoid m => Fold (i -> m) (m -> o)

fold :: Fold i o -> [i] -> o
fold (Fold tally summarize) is = summarize (reduce (map tally is))
  where
    reduce = Data.Foldable.foldl' (<>) mempty

sum :: Num n => Fold n n
sum = Fold Sum getSum
```

Простий приклад

```
>>> fold sum [1..10]  
55
```

```
main :: IO ()  
main = print (fold sum [(1::Int)..1000000000])
```

```
$ time ./example # 0.3 ns / elem  
5000000005000000000
```

```
real    0m0.322s  
user    0m0.316s  
sys     0m0.003s
```

Як це працює?

```
print (fold sum [1, 2, 3, 4])

-- sum = Fold Sum getSum
= print (fold (Fold Sum getSum) [1, 2, 3, 4])

-- fold (Fold tally summarize) is = summarize (reduce (map tally is))
= print (getSum (reduce (map Sum [1, 2, 3, 4])))

-- reduce = foldl' (<>) mempty
= print (getSum (foldl' (<>) mempty (map Sum [1, 2, 3, 4])))

-- Definition of `map` (skipping a few steps)
= print (getSum (foldl' (<>) mempty [Sum 1, Sum 2, Sum 3, Sum 4]))

-- `foldl' (<>) mempty` (skipping a few steps)
= print (getSum (mempty <> Sum 1 <> Sum 2 <> Sum 3 <> Sum 4))

-- mempty = Sum 0
= print (getSum (Sum 0 <> Sum 1 <> Sum 2 <> Sum 3 <> Sum 4))

-- Sum x <> Sum y = Sum (x + y)
= print (getSum (Sum 10))

-- getSum (Sum x) = x
= print 10
```

Більш цікавий приклад

```
{-# LANGUAGE BangPatterns #-}

data Average a = Average { numerator :: !a, denominator :: !Int }

instance Num a => Monoid (Average a) where
    mempty = Average 0 0
    mappend (Average xL nL) (Average xR nR) = Average (xL + xR) (nL + nR)

-- Not a numerically stable average, but humor me
average :: Fractional a => Fold a a
average = Fold tally summarize
  where
    tally x = Average x 1

    summarize (Average numerator denominator) =
      numerator / fromIntegral denominator
```

Приклад

```
>>> fold average [1..10]  
5.5
```

```
main :: IO ()  
main = print (fold average (map fromIntegral [(1::Int)..1000000000]))
```

```
$ time ./example # 1.3 ns / elem  
5.00000000067109e8
```

```
real    0m1.251s  
user    0m1.237s  
sys     0m0.005s
```

Немає витоку простору!

Наша average працює за константну пам'ять:

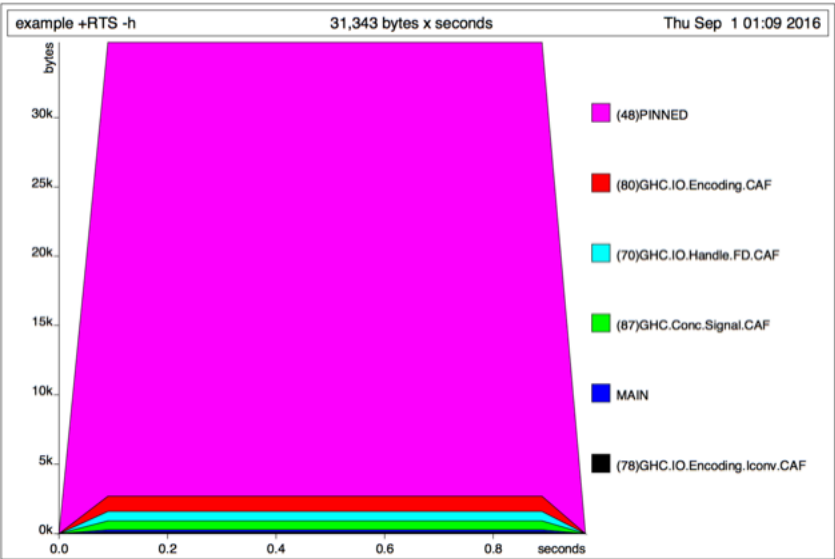


Figure 1:

Як це працює?

```
print (fold average [1, 2, 3])

-- average = Fold tally summarize
= print (fold (Fold tally summarize ) [1, 2, 3])

-- fold (Fold tally summarize) is = summarize (reduce (map tally is))
= print (summarize (reduce (map tally [1, 2, 3])))

-- reduce = foldl' (<>) mempty
= print (summarize (foldl' (<>) mempty (map tally [1, 2, 3])))

-- Definition of `map` (skipping a few steps)
= print (summarize (foldl' (<>) mempty [tally 1, tally 2, tally 3]))

-- tally x = Average x 1
= print (summarize (mconcat [Average 1 1, Average 2 1, Average 3 1]))

-- `foldl' (<>) mempty` (skipping a few steps)
= print (summarize (mempty <> Average 1 1 <> Average 2 1 <> Average 3 1))

-- mempty = Average 0 0
= print (summarize (Average 0 0 <> Average 1 1 <> Average 2 1 <> Average 3 1))

-- Average xL nL <> Average xR nR = Average (xL + xR) (nL + nR)
= print (summarize (Average 6 3))

-- summarize (Average numerator denominator) = numerator / fromIntegral denominator
= print (6 / fromIntegral 3)
```

Прості Foldи

Все в `Data.Monoid` можна загорнути в `Fold`

```
import Prelude hiding (head, last, all, any, sum, product, length)
```

```
head :: Fold a (Maybe a)
head = Fold (First . Just) getFirst
```

```
last :: Fold a (Maybe a)
last = Fold (Last . Just) getLast
```

```
all :: (a -> Bool) -> Fold a Bool
all predicate = Fold (All . predicate) getAll
```

```
any :: (a -> Bool) -> Fold a Bool
any predicate = Fold (Any . predicate) getAny
```

```
sum :: Num n => Fold n n
sum = Fold Sum getSum
```

```
product :: Num n => Fold n n
product = Fold Product getProduct
```

```
length :: Num n => Fold i n
length = Fold (\_ -> Sum 1) getSum
```


Приклади

```
>>> fold head [1..10]
Just 1
>>> fold last [1..10]
Just 10
>>> fold (all even) [1..10]
False
>>> fold (any even) [1..10]
True
>>> fold sum [1..10]
55
>>> fold product [1..10]
3628800
>>> fold length [1..10]
10
```

Експоненційне ковзне середнє

Експоненційне ковзне середнє у вигляді Foldy:

```
data EMA a = EMA { samples :: !Int, value :: !a }

instance Fractional a => Monoid (EMA a) where
    mempty = EMA 0 0

    mappend (EMA nL xL) (EMA 1 xR) = EMA n x  -- Optimize common case
        where
            n = nL + 1

            x = xL * 0.7 + xR

    mappend (EMA nL xL) (EMA nR xR) = EMA n x
        where
            n = nL + nR

            x = xL * (0.7 ^ nR) + xR

ema :: Fractional a => Fold a a
ema = Fold tally summarize
    where
        tally x = EMA 1 x

        summarize (EMA _ x) = x * 0.3
```

Приклад

```
>>> fold ema [1..10]  
7.732577558099999
```

```
main :: IO ()  
main = print (fold ema (map fromIntegral [(1::Int)..1000000000]))
```

```
$ time ./example # 2.6 ns / elem  
9.999999976666665e8
```

```
real    0m2.577s  
user    0m2.562s  
sys     0m0.009s
```

Оцінка кардинальності

Типове питання, що виникає — “оцінити кількість унікальних відвідувачів”

Наївне рішення:

```
import Data.Set (Set)
```

```
import qualified Data.Set
```

```
uniques :: Ord i => Fold i Int
```

```
uniques = Fold Data.Set.singleton Data.Set.size
```

... потребує багато пам'яті

... погано для великих даних

Приблизна оцінка кардинальності

Алгоритм HyperLogLog дає приблизну оцінку кардинальності

Спрощене пояснення на Хаскелі:

```
import Data.Word (Word64)

import qualified Data.Bits

newtype Max a = Max { getMax :: a }

instance (Bounded a, Ord a) => Monoid (Max a) where
    mempty = Max minBound

    mappend (Max x) (Max y) = Max (max x y)

uniques :: (i -> Word64) -> Fold i Int
uniques hash = Fold tally summarize
    where
        tally x = Max (fromIntegral (Data.Bits.countLeadingZeros (hash x)) :: Word64)

        summarize (Max n) = fromIntegral (2 ^ n)
```

Справжня версія набагато більш “дужа” (Див hyperloglog від E. Kmett)

Приклад

```
main :: IO ()
main = print (fold (uniques id) (take 1000000000 (cycle randomWord64s)))

randomWord64s :: [Word64]
randomWord64s = [11244654998801660968,16946641599420530603,652086428930367189,5128055280221172986,16587432539185930121,22285709

$ time ./example # 5.5 ns / elem
16

real    0m5.543s
user    0m5.526s
sys     0m0.007s
```

Код, що варто вкрасти

Деякі ідеї, що вкрадено зі Скала-бібліотеки `algebird`

- Quantile digests (for medians, percentiles, histograms)
 - `algebird` calls these `QTrees`
- Count-min sketch (for top N most frequently occurring items)
 - `algebird` generalizes this as `SketchMaps`
- Stochastic gradient descent (for linear regression)
 - `algebird` calls this `SGD`
- Bloom filters (for approximate membership testing)
 - `algebird` calls this `BF`

`algebird`'s version of Fold is called `Aggregator`

Комбінуємо Foldи

Уявімо, що ми хочемо зкомбінувати два Foldи в один

Ми би зробили щось типу:

```
combine :: Fold i a -> Fold i b -> Fold i (a, b)
combine (Fold tallyL summarizeL) (Fold tallyR summarizeR) = Fold tally summarize
  where
    tally x = (tallyL x, tallyR x)

    summarize (sL, sR) = (summarizeL sL, summarizeR sR)
```


Приклад

```
>>> fold (combine sum product) [1..10]  
(55,3628800)
```

Applicative

Можемо узагальнити combine написавши інстанс Fold для Applicative

```
instance Functor (Fold i) where
    fmap k (Fold tally summarize) = Fold tally (k . summarize)

instance Applicative (Fold i) where
    pure o = Fold (\_ -> ()) (\_ -> o)

    Fold tallyF summarizeF <*> Fold tallyX summarizeX = Fold tally summarize
    where
        tally i = (tallyF i, tallyX i)

        summarize (mF, mX) = summarizeF mF (summarizeX mX)
```

Строгість

Ми хочемо, аби внутрішній Monoid був строгим Pair:

```
data Pair a b = P !a !b

instance (Monoid a, Monoid b) => Monoid (Pair a b) where
    mempty = P mempty mempty

    mappend (P aL bL) (P aR bR) = P (mappend aL aR) (mappend bL bR)

data Fold i o = forall m . Monoid m => Fold (i -> m) (m -> o)

instance Functor (Fold i) where
    fmap k (Fold tally summarize) = Fold tally (k . summarize)

instance Applicative (Fold i) where
    pure o = Fold (\_ -> ()) (\_ -> o)

    Fold tallyF summarizeF <*> Fold tallyX summarizeX = Fold tally summarize
    where
        tally i = P (tallyF i) (tallyX i)

        summarize (P mF mX) = summarizeF mF (summarizeX mX)
```

Це дозволить використовувати seq замість deepseq

Pairing values

Тепер ми можемо написати:

```
combine :: Fold i a -> Fold i b -> Fold i (a, b)
combine = liftA2 (,)
```

... що має більш узагальнений тип:

```
combine :: Applicative f => f a -> f b -> f (a, b)
```

Альтернативно, можемо використовувати Applicative напямую:

```
>>> fold ((,) <$> sum <*> product) [1..10]
(55,3628800)
```

Анти-паттерн

Порівняйте дві функції:

```
bad :: [Double] -> (Double, Double)
bad xs = (Prelude.sum xs, Prelude.product xs)
```

```
good :: [Double] -> (Double, Double)
good xs = fold ((,) <$> sum <*> product) xs
```

Яка проблема в першій з них?

Застосовуємо Applicative

Можемо використовувати Applicative-інстанси не тільки для пар:

```
sum :: Num n => Fold n n
sum = Fold Sum getSum
```

```
length :: Num n => Fold i n
length = Fold (\_ -> Sum 1) getSum
```

```
average :: Fractional n => Fold n n
average = (/) <$> sum <*> length
```

Генерує код, еквівалентний average, написаному "вручну":

```
main :: IO ()
main = print (fold average (map fromIntegral [(1::Int)..1000000000]))

$ time ./example # 1.3 ns / elem
5.00000000067109e8
```

```
real    0m1.281s
user    0m1.266s
sys     0m0.006s
```

Num

Можемо надати інстанси Fold класам Num, Fractional та Floating!

```
instance Num b => Num (Fold a b) where
    fromInteger n = pure (fromInteger n)

    negate = fmap negate
    abs     = fmap abs
    signum  = fmap signum

    (+) = liftA2 (+)
    (*) = liftA2 (*)
    (-) = liftA2 (-)

instance Fractional b => Fractional (Fold a b) where
    fromRational n = pure (fromRational n)

    recip = fmap recip

    (/) = liftA2 (/)

instance Floating b => Floating (Fold a b) where
    pi = pure pi

    exp    = fmap exp
    sqrt   = fmap sqrt
    log    = fmap log
    sin    = fmap sin
    tan    = fmap tan
    cos    = fmap cos
    asin   = fmap sin
    atan   = fmap atan
    acos   = fmap acos
    sinh   = fmap sinh
    tanh   = fmap tanh
    cosh   = fmap cosh
    asinh  = fmap asinh
    atanh  = fmap atanh
    acosh  = fmap acosh
```

```
(**)      = liftA2 (**)
logBase = liftA2 logBase
```


Числові Folds

Можемо робити круті штуки:

```
>>> fold (length - 1) [1..10]
9
>>> let average = sum / length
>>> fold average [1..10]
5.5
>>> fold (sin average ^ 2 + cos average ^ 2) [1..10]
1.0
>>> fold 99 [1..10]
99
```

Стандартне відхилення

Формула стандартного відхилення:

$$\sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2} = \sqrt{\frac{1}{N} \left(\sum_{i=1}^N x_i^2 \right) - \bar{x}^2} = \sqrt{\left(\frac{1}{N} \sum_{i=1}^N x_i^2 \right) - \left(\frac{1}{N} \sum_{i=1}^N x_i \right)^2}$$

Еквівалент через Fold читається майже так само просто, як остання формула:

```
standardDeviation :: Floating n => Fold n n
standardDeviation = sqrt ((sumOfSquares / length) - (sum / length) ^ 2)
  where
    sumOfSquares = Fold (Sum . (^2)) getSum

>>> fold standardDeviation [1..100]
28.86607004772212
```

Фолдимо ListT

ListT із пакету list-transformers визначено так:

```
newtype ListT m a = ListT { next :: m (Step m a) }
```

```
data Step m a = Cons a (ListT m a) | Nil
```

Можемо його згорнути!

```
{-# LANGUAGE BangPatterns #-}
```

```
import List.Transformer (ListT(..), Step(..))
```

```
import qualified System.IO
```

```
foldListT :: Monad m => Fold i o -> ListT m i -> m o
```

```
foldListT (Fold tally summarize) = go mempty
```

```
  where
```

```
    go !m l = do
```

```
      s <- next l
```

```
      case s of
```

```
        Nil      -> return (summarize m)
```

```
        Cons x l' -> go (mappend m (tally x)) l'
```

Приклад

Можемо таким чином згорнути “effectful streams”:

```
stdin :: ListT IO String
stdin = ListT (do
  eof <- System.IO.isEOF
  if eof
    then return Nil
    else do
      line <- getLine
      return (Cons line stdin) )
```

```
main :: IO ()
main = do
  n <- foldListT length stdin
  print n
```

```
$ yes | head -10000000 | ./example
10000000
```

Згортаємо потокові бібліотеки

Можемо таким самим чином згорнути:

- `conduit`
- `io-streams`
- `list-t`
- `logict`
- `machines`
- `pipes`
- `turtle`

Кожен Fold може бути перевикористаним в будь-якій із цих систем

Лінзи

```
{-# LANGUAGE RankNTypes #-}

import Control.Lens (Getting, foldMapOf)

focus :: (forall m . Monoid m => Getting m b a) -> Fold a o -> Fold b o
focus lens (Fold tally summarize) = Fold tally' summarize
  where
    tally' = foldMapOf lens tally

focus _1 :: Fold i o -> Fold (i, x) o

focus _Just :: Fold i o -> Fold (Maybe i) o
```

Приклад

```
items1 :: [Either Int String]
items1 = [Left 1, Right "Hey", Right "Foo", Left 4, Left 10]

>>> fold (focus _Left sum) items1
15
>>> fold (focus _Right length) items1
2

items2 :: [Maybe (Int, String)]
items2 = [Nothing, Just (1, "Foo"), Just (2, "Bar"), Nothing, Just (5, "Baz")]

>>> fold (focus (_Just . _1) product) items2
10
>>> fold (focus _Nothing length) items2
2
```

На (нашій) практиці

- Великі структури даних івентів
- Декілька джерел
- Великі структури даних звітів
- Більшість результатів — “в часі”
- Багато обв’язки для збереження та завантаження

Ивенти

```
data DsAdBid a = DsAdBid
  { _abImpressionId :: ImpressionId
  , _abDelay         :: Maybe PageLoadDelay
  , _abDsAccountId   :: DsAccountId
  , _abAdregionId    :: AdregionId
  , _abBid           :: Bid
  , _abCreativeCore  :: Maybe Cr.CreativeCore
  }
deriving (Show, Eq, Generic)

data CreativeCore
= CreativeCore
  { _coreTitle       :: Maybe Title
  , _coreDescription :: Maybe Description
  , _coreSponsor     :: Maybe Sponsor
  , _coreImgURL      :: Maybe (Valid RawURL)
  }
deriving (Eq, Generic, Show)
```

Folds.hs

```
-- / Count events
countFold :: (Hashable k, Eq k)
           => (ev -> Maybe k)
           -> L.Fold ev (MonoidalHashMap k (Sum Integer))
countFold keyF = L.Fold s mempty id incr
  where
    incr m ev = m (\k -> M.modify (+1) k m) (keyF ev)
```

Простий випадок

```
-- | SELECT GROUPKEY(round_one_hour(timestamp)),
--      count()
--      FROM ds_ad_load_events
processCountLoadsPerHour :: Mode -> IO ()
processCountLoadsPerHour mode = void . runConsumerInMode mode $ do
    let kf ev = Just (rewindToHour (ev ^. timestamp))
    let (EvProducer prod) = evProducerW start end
    res <- purelyFold (countFold kf) (prod :: EventProducer (Event DsAdLoad))
    liftIO (print res)
    -- :: MonoidalHashMap KeyDsAdregion (Sum Integer)
```

Складніше

- помітьте, як ми передаємо мапу в пам'яті замість джойнів

```
-- | SELECT GROUPKEY(ds_id, adregion_id, round5min(timestamp)),
--      count(ds_ad_load_events)
--      earnings(ds_ad_load_events)
--      count(ds_click_events)
-- FROM ds_ad_load_events, ds_click_events
processCountLoadsAndClicksAndEarningsConcurrent :: Mode -> IO ()
processCountLoadsAndClicksAndEarningsConcurrent mode = void . runConsumerInMode mode $ do
    (KeyFunction kf) <- getKeyFunctionDsAdregion
    let (EvProducer prod) = evProducerW start end
    ((lc,le),cc) <- concurrently2
        (purelyFold ((,) <$> countFold kf <*> earningsFold kf)
          (prod :: EventProducer (Event DsAdLoad)))
        (purelyFold (countFold kf)
          (prod :: EventProducer (Event DsClick)))
    liftIO (print (mhmZip3 lc le cc))
    -- :: MonoidalHashMap KeyDsAdregion (Sum Integer, Sum MoneyAmount,
    --                                     Sum Integer)

type KeyAdregionDs = (AdregionId, DsProviderId, UTCTime)
getKeyFunctionDsAdregion :: Consumer (KeyFunction KeyAdregionDs)

keyFunctionAdregionDs :: KeyConstraints KeyAdregionDs ev
                     => HashMap DsAccountId DsProviderId
                     -> (ev -> Maybe KeyAdregionDs)

keyFunctionAdregionDs dsMap event = do
    let accountID = event ^. dsAccountId
    dsID          <- H.lookup accountID dsMap
    let region    = event ^. adregionId
    let timekey   = rewindTo5Min (event ^. timestamp)
    return (region, dsID, timekey)
```

Складніші репорти

```
data Report label =  
    Report { scope      :: label  
          , timeseries :: MonoidalHashMap UTCTime ReportPayload  
          } deriving (Show, Eq)  
  
data ReportPayload  
    = ReportPayload { loads          :: !(Sum Integer)  
                    , impressions    :: !(Sum Integer)  
                    , clicks         :: !(Sum Integer)  
                    , ctr            :: !(Avg Double)  
                    , earnings       :: !(Sum MoneyAmount)  
                    , avgEarnings    :: !(Avg MoneyAmount)  
                    , avgEarningsCpc :: !(Avg MoneyAmount)}  
  
deriving (Show, Eq, Generic)
```

Foldи для складніших репортів

```
processStats = do
  aggs <- aggregateEventStreams dsMap (evProducer  startTime endTime
                                       :: EventProducer (Event DsAdLoad))
    (evProducer  startTime endTime
     :: EventProducer (Event DsImpression))
    (evProducer  startTime endTime
     :: EventProducer (Event DsClick))
    (evProducer  startTime endTime
     :: EventProducer (Event DsAvailabilityCheck))
    (evProducer  startTime endTime
     :: EventProducer (Event DsAdAvail))
    (evProducer  startTime endTime
     :: EventProducer (Event DsAdUnavail))
    (evProducer  startTime endTime
     :: EventProducer (Event DsAdError))
    (evProducer  startTime endTime
     :: EventProducer (Event DsCookieSync))

  where
    ...
```

... продолжения ...

```
impFold :: PayloadFold (Event DsImpression)
impFold = mkBaseFold impressionsL countFold <>
         mkBaseFold ctrL ctrFold
```

```
clickFold :: PayloadFold (Event DsClick)
clickFold = mkBaseFold clicksL countFold <>
           mkBaseFold ctrL ctrFold
```

Візуалізація

- GHCJS + Reflex

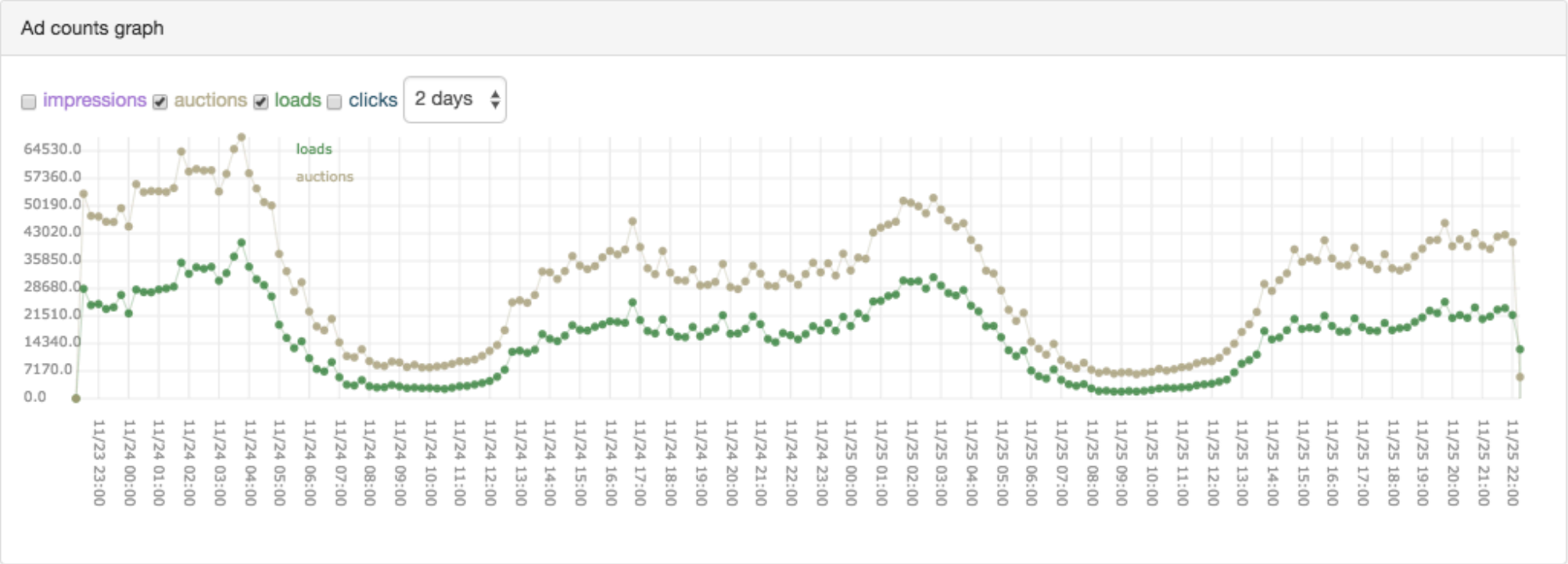


Figure 2: dashboard

Чого не вистачає

- Швидкість (data locality, нерівномірність бакетів)
- Компактне збереження результатів (абстракція розділення по інтервалах та збереження)
- REPL або блокнотик
- GUI для репроцессингу
- Real-time in-memory analytics