channable

# Parallel Streaming

Yorick Sijsling & Joris Burgers

June 5, 2023

# Channable

- Channable provides tools to process *product data*
- For example, customers can connect their webshop, and configure the tool to generate Google ads based on their product data
- Team of  20 mostly-Haskell developers (we’re hiring[1])

# Motivation

- Users can specify rules to process their data
- Rule processing engine handles 1.7 million items per second
- Jobs were initially single threaded:
  - Simple
  - Parallelism from running multiple jobs
  - Easier to optimize
  - Efficient use of resources (no overhead from multithreading)
  - Good for high overall throughput
- Some jobs are just too big, taking minutes or even hours
- Task: add in-job parallelism!
  - We also wrote a blog[2] about this

# Overview

- Rationale for streaming
- Quick introduction to `conduit`
- Adding parallelism
- Parallel aggregations
- Running conduits with parallelism
- Zipping conduits with parallelism

# Rule processing

# What’s in a job

The rule processing engine runs different types of jobs.
We’ll focus on one type that:

- Starts with a data set in memory (akin to `[Item]`)
- Runs a bunch of actions
- Ends with a data set in memory

# Storing items

We don't actually use `[Item]` as an in-memory data set:

- Can't do efficient indexing or slicing
- Bad for garbage collection (GC)

Assume that `StoredItems` is some efficient, GC-friendly way to store items, for instance using:

- Unboxed Vectors
- Vectors in compact regions (see our blog post[3])
- Or serialized data on disk, as a memory-mapped file

```
evalActionsJob :: [Action] -> StoredItems -> _ StoredItems
```

# Evaluating actions

Actions come in the form of an AST:

```
data Action
  = Map Expression
  | Filter Expression
  | SortOn Expresssion
  | DeduplicateOn Expresssion
  ...
evalAction :: Action -> [Item] -> _ [Item]
```

# Evaluating actions

Creating a `StoredItems` after each action is too expensive.

When possible, we want to use *streaming*

- Each `Item` is sent through multiple actions, before starting on the next `Item`
- This limits the amount of live data outside of `StoredItems`
- Good fit for chains of map and filter

# Evaluating actions

Laziness allows `[Item]` to work as a stream.

Each `Item` is only produced when the consumer needs it.

We use a dedicated library `conduit`[4] instead, to get several benefits:

- Interleaving `IO` between items. (yield an item, do some IO, yield next item, …)
- A nice streaming-specific interface
- (Hopefully) more reliable stream fusion

```
evalAction :: Action -> ConduitT Item Item IO ()
```

# Conduits

# Conduits

The `ConduitT` type is the core of the `conduit` library.

A `ConduitT i o m r` is a stream processor:

- It *consumes* a stream of values of type `i`
- It *produces* a stream of values of type `o`
- It can run effects in monad `m`
- At the end, it produces a single `r`

# Conduits

```
> runConduit (C.yieldMany [1,2,3] .| C.map (*2) .| C.sum)
12
```

# Conduits

```
> runConduit (C.yieldMany [1,2,3] .| C.map (*2) .| C.sum)
12
```

```
C.yieldMany :: Monad m => [o] -> ConduitT i o m ()
C.map :: Monad m => (i -> o) -> ConduitT i o m ()
C.sum :: Monad m => ConduitT Int o m Int
```

# Conduits

```
> runConduit (C.yieldMany [1,2,3] .| C.map (*2) .| C.sum)
12
```

```
C.yieldMany :: Monad m => [o] -> ConduitT i o m ()
C.map :: Monad m => (i -> o) -> ConduitT i o m ()
C.sum :: Monad m => ConduitT Int o m Int
```

```
(.|) :: Monad m => ConduitT a b m ()
                -> ConduitT b c m r
                -> ConduitT a c m r
runConduit :: Monad m => ConduitT () Void m r -> m r
```



channable

# Adding parallelism

# Streams of work and values

We use a *mixed stream* of values and parallel work units.
Stream values use this type:

```
data WorkOr a = WOValue a | WOWork (IO ()) | WONothingYet
```

# Streams of work and values

We use a *mixed stream* of values and parallel work units.
Stream values use this type:

```
data WorkOr a = WOValue a | WOWork (IO ()) | WONothingYet
```

- For example, `ConduitT () (WorkOr Item) IO ()` produces both parallel work units and items
- The `IO ()` is a parallel work unit that:
  - Can run independent of everything else, in any order
  - Should be run at most once
- If a conduit has to wait for work to be completed, it can yield a `WONothingYet`

# Streams of work and values

We use a *mixed stream* of values and parallel work units. Stream values use this type:

```
data WorkOr a = WOValue a | WOWork (IO ()) | WONothingYet
```

- We don't need a separate task queue
- Work is produced on demand
- Supports deterministic ordering of values

# Example: Parallel streams

```
type ParallelStream = ConduitT () Item IO ()
yieldParallel :: ChunkSize -> [Item] -> ConduitT i ParallelStream IO ()
sinkItemsInParallel :: ConduitT ParallelStream (WorkOr o) IO [Item]
```

This `sinkItemsInParallel` function:

- Converts each `ParallelStream` to a `WOWork` that stores the output
- Main conduit waits on all outputs and concatenates

# Aggregations

# Aggregations

```
deduplicate :: Ord key => (Item -> key) -> [Item] -> [Item]
```

# Aggregations

```haskell
deduplicate :: Ord key => (Item -> key) -> [Item] -> [Item]
```

Deduplication deduplicates on the key.

```haskell
> deduplicate fst [(6, "a"), (3, "b"), (2, "c"), (5, "d"), (2, "e")]
[(2, "c"), (3, "b"), (5, "d"), (6, "a")]
```

For any duplicate key, takes the leftmost item.

# Aggregations

Arregations can be generalized

```
aggregation
  :: Ord key
  => (Item -> Item -> [Item])
  -> (Item -> key)
  -> [Item]
  -> [Item]
```

We only need to implement one (parallel) aggregation function

```
deduplication     = aggregation (\l _ -> [l])
sort              = aggregation (\l r -> [l, r])
deduplicateRemove = aggregation (\_ _ -> [])
sum               = aggregation (\l r -> [l + r])
```

deduplicateRemove has a bug, can you find it?

channable

# Parallel aggregations

Parallel deduplication can be implemented as a merge-sort.

1. Split the input in 2 blocks
2. Sort the individual blocks
3. Deduplicate the sorted blocks, using the provided function for any key collisions
4. Join the sorted, deduplicated blocks

# Parallel aggregations

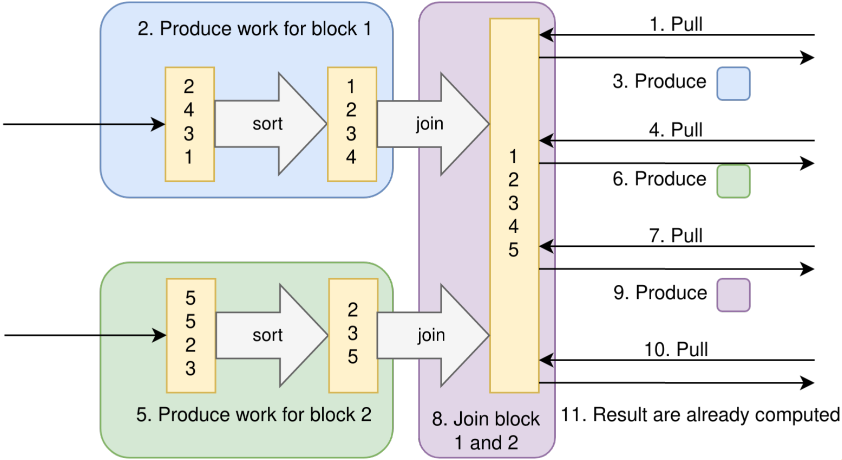
```haskell
type Block = Vector Item

processBlock
  :: MVar Block
  -> ParallelStream
  -> WorkOr a
processBlock resultPlaceholder stream = WOWork $ do
  result <- sort stream >>= deduplicate
  putMVar resultPlaceholder result
```

# Parallel aggregations

```haskell
joinBlocks
  :: MVar Block
  -> Block
  -> Block
  -> WorkOr a
joinBlocks resultPlaceholder leftBlock rightBlock = WOWork $ do
  result <- join leftBlock rightBlock
  putMVar resultPlaceholder result
```

# Parallel aggregations




channable

# Parallel aggregations

Summary

- Aggregations can be implemented as a parallel merge-sort
- We produce intermediate work units, only producing items after all work has completed
- Work is produced on demand

# Running the Conduit

# Running the Conduit

The `ConduitT` type is the core of the `conduit` library.

# Running the Conduit

The `ConduitT` type is the core of the `conduit` library.

A `ConduitT i o m r` is a stream processor:

- It *consumes* a stream of values of type `i`
- It *produces* a stream of values of type `o`
- It can run effects in monad `m`
- At the end, it produces a single `r`

# What’s in a Conduit?

Simplified answer: a `Pipe`

A `Pipe` represents the current state of the Conduit

# What’s in a Conduit?

Simplified answer: a **Pipe**

A **Pipe** represents the current state of the Conduit

```
data Pipe i o m r
  = HaveOutput (Pipe i o m r) o
  | NeedInput (i -> Pipe i o m r)
  | Done r
  | PipeM (m (Pipe i o m r))
```

```
runPipe :: Monad m => Pipe () Void m r -> m r
runPipe (HaveOutput _ o) = absurd o
runPipe (NeedInput c) = runPipe (c ())
runPipe (Done r) = return r
runPipe (PipeM mp) = mp >>= runPipe
```

# Running the Conduit in Parallel

Defining our own `runConduit` function

```
runConduitWithWork :: ConduitT () (WorkOr Void) IO r -> IO r
```

# Running the Conduit in Parallel

Defining our own `runConduit` function

```
runConduitWithWork :: ConduitT () (WorkOr Void) IO r -> IO r
```

We need one more case for `runPipe`:

```
runPipe :: Pipe () (WorkOr Void) IO r -> IO r
runPipe (HaveOutput _ (WOValue o)) = absurd o
runPipe (HaveOutput pipe (WOWork w)) = -- Handle work
runPipe (NeedInput c) = runPipe (c ())
runPipe (Done r) = return r
runPipe (PipeM mp) = mp >>= runPipe
```

# Running the Conduit in Parallel

How to parallelize `runPipe`?

1. Put the **`Pipe`** in an **`MVar`**.
2. Any thread can run the pipe until it finds a **`WOWork`**
3. Put the remaining **`Pipe`** back and evaluate the **`WOWork`**

Running the pipe should be cheap, as it happens in the *critical section*.

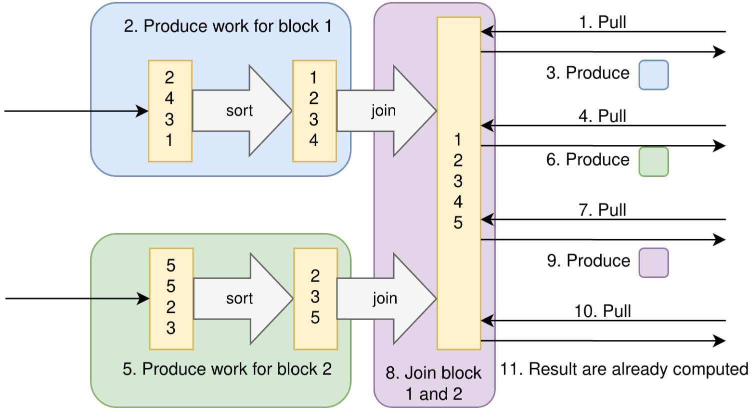
# Running the Conduit in Parallel

```
runConduitWithWork
  :: Int -> ConduitT () (WorkOr Void) IO r -> IO r
runConduitWithWork numThreads (ConduitT pipe) = do
  pipeVar <- newMVar $ injectLeftovers $ pipe Done
  threads <- replicateM numThreads $ Async.async $ runWorker pipeVar
  snd <$> Async.waitAnyCancel threads

runWorker :: MVar (Pipe () (WorkOr Void) IO r) -> IO r
runWorker pipeVar = loop
  where
    -- Take the pipe variable, so that no one else can.
    loop = takeMVar pipeVar >>= withPipe

    withPipe = \case
      HaveOutput pipe (WOWork w) -> do
        -- Put back the (modified) pipe for someone else to use, because we have work to do!
        putMVar pipeVar pipe
        w
        loop
      -- All the below is the same as 'runPipe' and is done within the critical section.
      -- This includes evaluation of upstream and monadic effects in the conduit.
      HaveOutput _ (WOValue o) -> absurd o
      NeedInput c -> withPipe (c ())
      Done r -> pure r
      PipeM mp -> mp >>= withPipe
```

# How do we know if there is work?






channable

# How do we know if there is work?

- Thread 1 deduplicates block 1
- Thread 2 deduplicates block 2
- Thread 3 cannot yet join the 2 blocks

What happens if we pull the join before the input is ready?

# How do we know if there is work?

- Thread 1 deduplicates block 1
- Thread 2 deduplicates block 2
- Thread 3 cannot yet join the 2 blocks

What happens if we pull the join before the input is ready?

`WONothingYet`

# How do we know if there is work?

- Thread 1 deduplicates block 1
- Thread 2 deduplicates block 2
- Thread 3 cannot yet join the 2 blocks

What happens if we pull the join before the input is ready?

`WONothingYet`



We don’t want to block.

- Makes time-measurement and core scheduling harder
- We cannot make explicit choices what work to forward

# Zipping streams

# Zipping streams

| LEFT | RIGHT | |
| --- | --- | --- |
| Has `WONothingYet` | Has `WONothingYet` | Forward a `WONothingYet` |
| Has `WOWork` | _ | Forward the `WOWork` |
| Has `WONothingYet` | Has `WOWork` | Forward the `WOWork` |
| Has a stream | _ | Forward the stream |
| Has `WONothingYet` | Has a stream | Forward a `WONothingYet` |
| Is `Done` | Has `WONothingYet` | Forward a `WONothingYet` |
| Has `WONothingYet` | Is `Done` | Forward a `WONothingYet` |

# Conclusion

Pros

- Simple, but covers most use cases
- Little overhead
- Evaluation strategy aligns with GHC runtime
- Your custom `runConduitWithWork` could dynamically scale the number of used cores

Cons

- Sometimes feels bolted on to conduit
- Components don't know if the downstream currently prefers values, or more work

# References


– Haskell jobs at channable.
https://jobs.channable.com/o/haskell-software-engineer-3.

– Parallel streaming in haskell, 2023.
https://www.channable.com/tech/parallel-streaming-in-haskell-part-1-fast-efficient-fun.

– Lessons in managing haskell memory, 2020.
https://www.channable.com/tech/lessons-in-managing-haskell-memory.

– Conduit.
https://hackage.haskell.org/package/conduit-1.3.5.

channable

# Parallel Streaming

Yorick Sijsling & Joris Burgers

June 5, 2023