

facebook

Remote data access made easy and fast with Haskell

Simon Marlow
20 November 2014

Agenda

- 1** Motivation: efficient data fetching
- 2** Example: a blog
- 3** How to define a data source
- 4** Example data sources

Section 1: Introduction

Section 2: Overview

Section 3: Details

Section 4: Conclusion

Section 5: Summary

Section 6: Appendix

Section 7: References

Section 8: Index

Section 9: Introduction

Section 10: Overview

Section 11: Details

Section 12: Conclusion

Section 13: Summary

Section 14: Appendix

Section 15: References

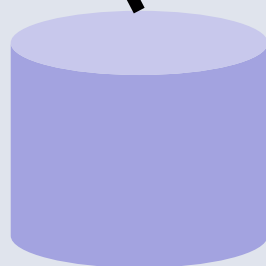
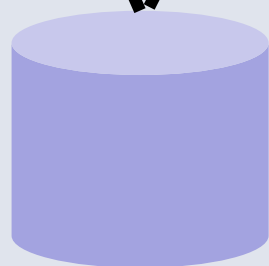
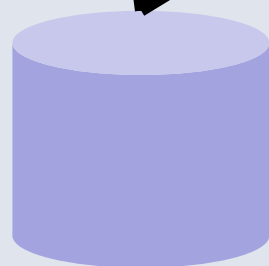
Section 16: Index

Section 1: Introduction

Section 2: Overview

Section 3: Details

Section 4: Conclusion



Data sources

For efficiency, we would like:

For efficiency, we would like:

- Concurrency

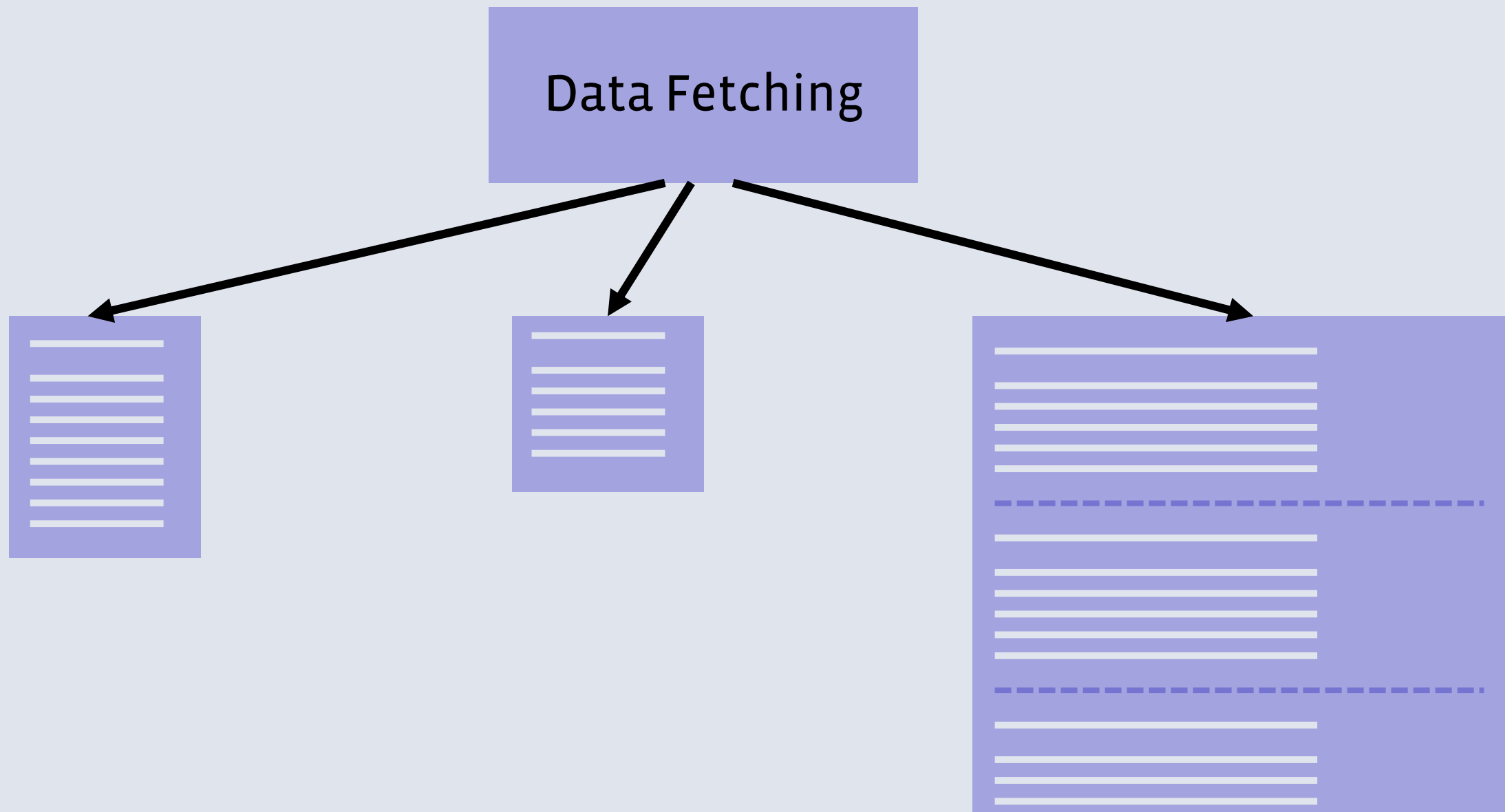
For efficiency, we would like:

- Concurrency
- Batching

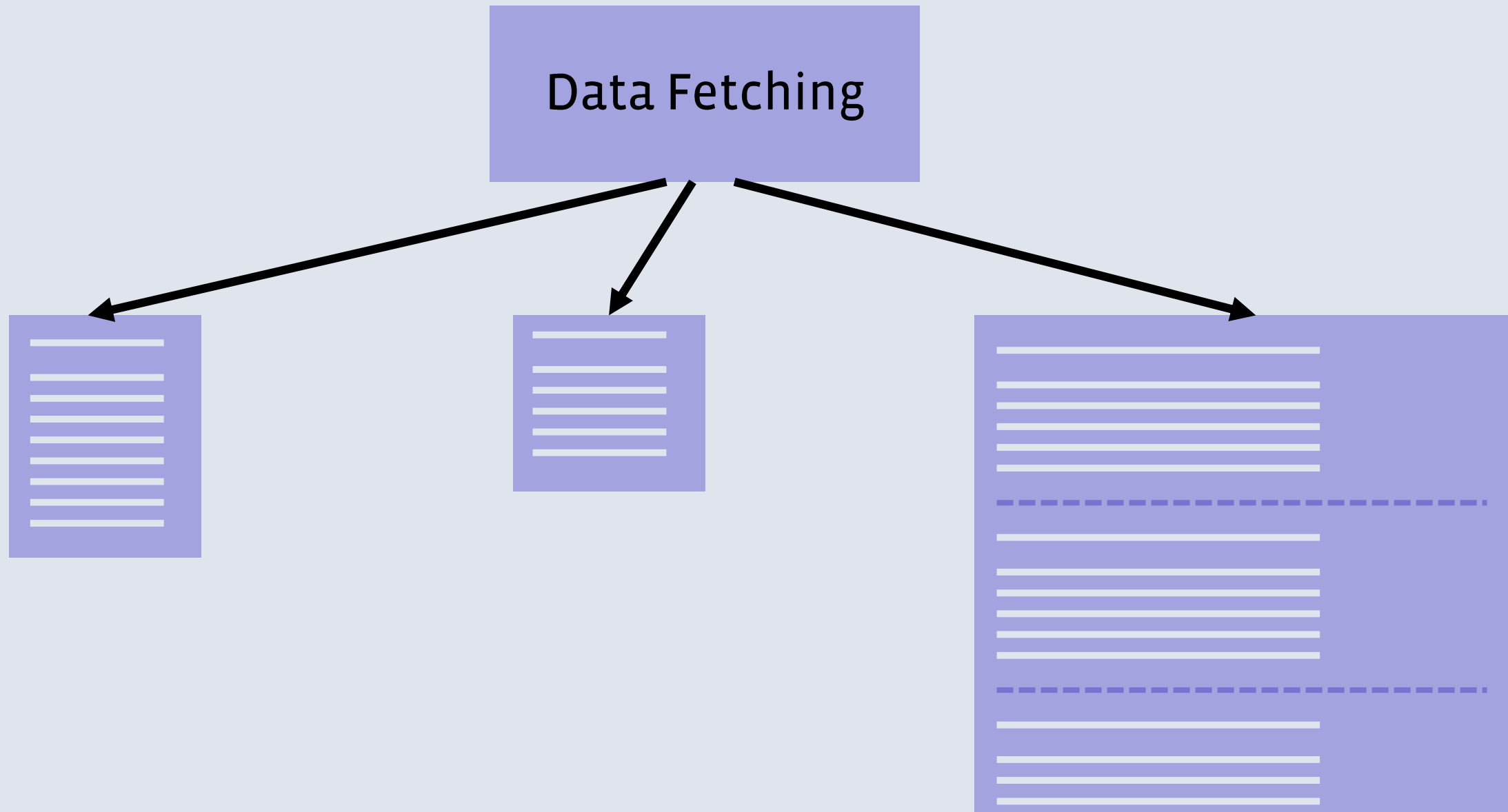
For efficiency, we would like:

- Concurrency
- Batching
- Caching

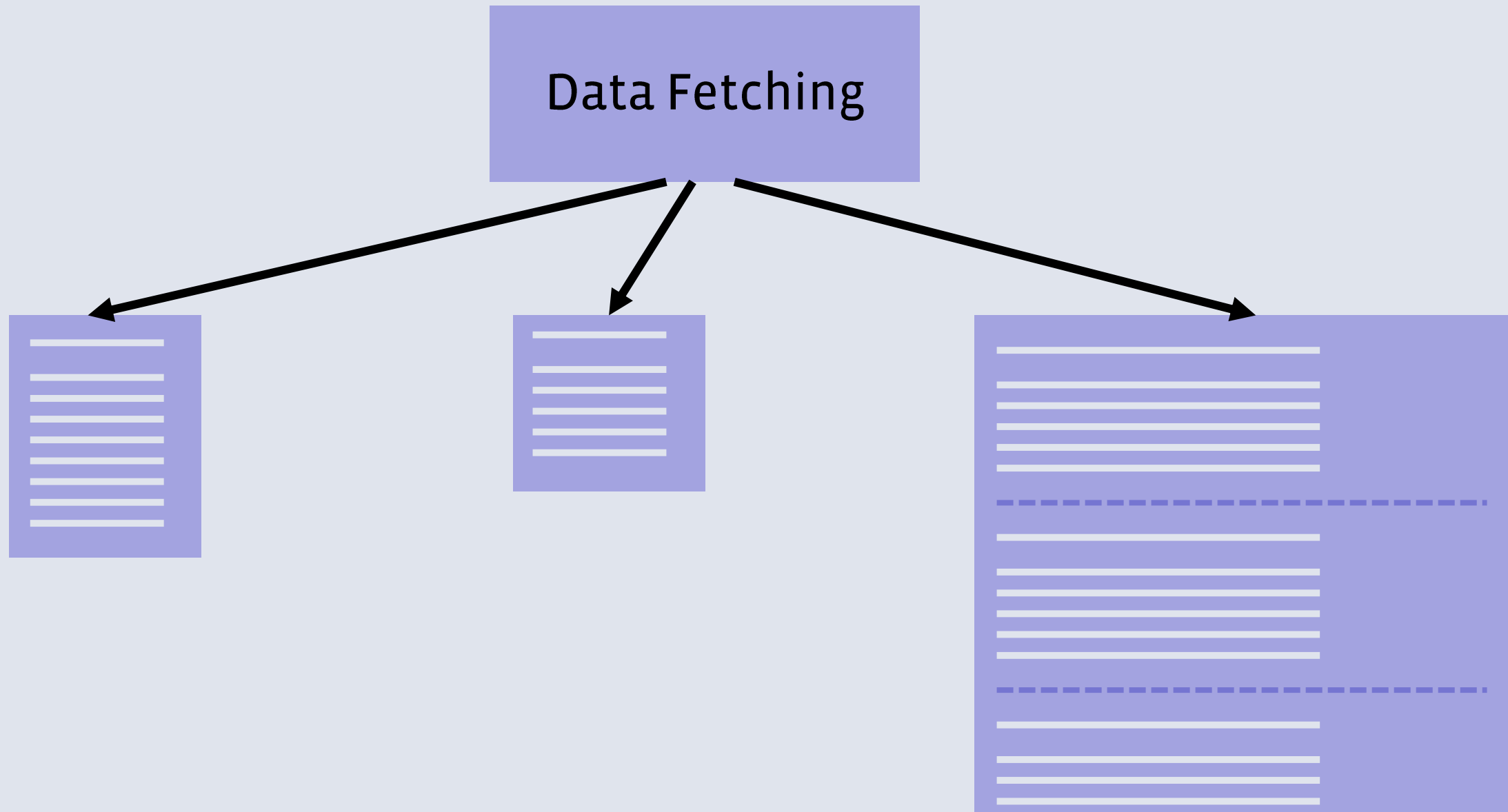
We could do all our data fetching up front...



But that destroys modularity.



And we cannot always extract the data fetching from the business logic.



What about explicit concurrency?

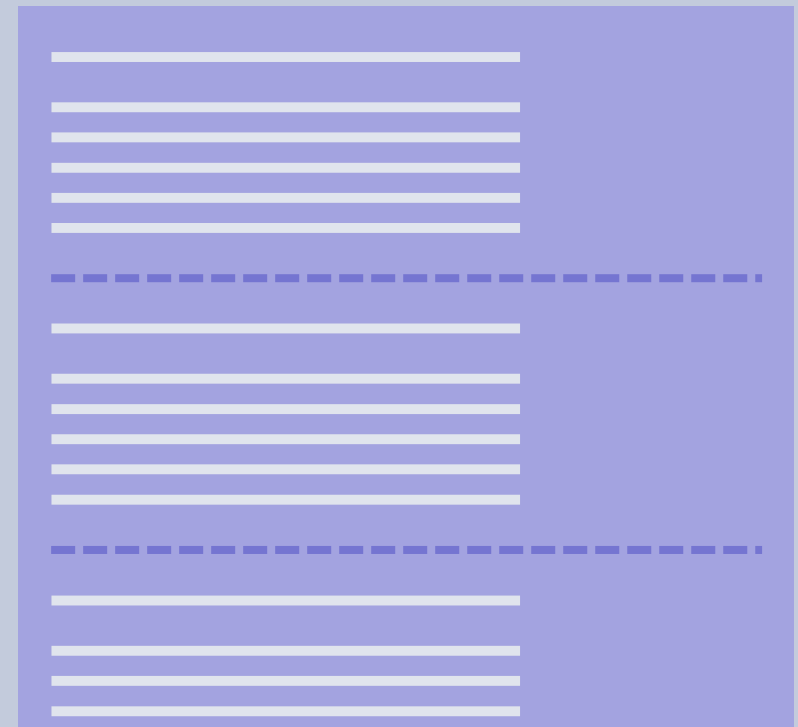
Thread 1



Thread 2



Thread 3



Explicit concurrency relies too much on the programmer.

- Must remember to fork (or async)
- Must not demand results too early
- Don't fork unnecessary work

Explicit concurrency is too explicit for this application.

- Explicit concurrency is good when the programmer needs to ask for non-determinism
- We're only fetching data – no side effects
- Programmer doesn't care about ordering

Concrete example: a blog

- Main pane: posts
- Left pane:
 - Top-10 most popular posts
 - Post topics, with post counts




```
blog :: Haxl Html  
blog = renderPage <$> leftPane <*> mainPane
```

```
blog :: Haxl Html  
blog = renderPage <$> leftPane <*> mainPane
```

Haxl is our
Monad that
provides data-
fetching.

```
blog :: Haxl Html  
blog = renderPage <$> leftPane <*> mainPane
```

```
leftPane :: Haxl Html  
leftPane = renderSidePane <$> popularPosts <*> topics  
  
renderPage      :: Html -> Html -> Html  
renderSidePane :: Html -> Html -> Html
```

```
data PostId      -- identifies a post
data PostContent -- the content of a post

-- metadata about a post
data PostInfo = PostInfo
  { postId      :: PostId
  , postDate    :: Date
  , ...
  }

-- data-fetching operations
getPostIds      :: Haxl [PostId]
getPostInfo     :: PostId -> Haxl PostInfo
getPostContent  :: PostId -> Haxl PostContent
```

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

```
data PostId      -- identifies a post
data PostContent -- the content of a post

-- metadata about a post
data PostInfo = PostInfo
  { postId      :: PostId
  , postDate    :: Date
  , ...
  }

-- data-fetching operations
getPostIds      :: Haxl [PostId]
getPostInfo     :: PostId -> Haxl PostInfo
getPostContent  :: PostId -> Haxl PostContent

getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

```
data PostId      -- identifies a post
data PostContent -- the content of a post
```

```
-- metadata about a post
data PostInfo = PostInfo
  { postId      :: PostId
  , postDate    :: Date
  , ...
  }
```

```
-- data-fetching operations
getPostIds      :: Haxl [PostId]
getPostInfo     :: PostId -> Haxl PostInfo
getPostContent  :: PostId -> Haxl PostContent
```

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

```
data PostId      -- identifies a post
data PostContent -- the content of a post

-- metadata about a post
data PostInfo = PostInfo
  { postId      :: PostId
  , postDate    :: Date
  , ...
  }
```

```
-- data-fetching operations
getPostIds      :: Haxl [PostId]
getPostInfo     :: PostId -> Haxl PostInfo
getPostContent  :: PostId -> Haxl PostContent
```

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

```
data PostId      -- identifies a post
data PostContent -- the content of a post

-- metadata about a post
data PostInfo = PostInfo
  { postId      :: PostId
  , postDate    :: Date
  , ...
  }

-- data-fetching operations
getPostIds      :: Haxl [PostId]
getPostInfo     :: PostId -> Haxl PostInfo
getPostContent  :: PostId -> Haxl PostContent
```

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```



```
mainPane :: Haxl Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered = take 5 $ sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)
```

First we fetch
all the
metadata

```
mainPane :: Haxl Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered = take 5 $ sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)
```

```
mainPane :: Haxl Html
```

```
mainPane = do
```

```
  posts <- getAllPostsInfo
```

```
  let ordered = take 5 $ sortBy (flip (comparing postDate)) posts
```

```
  content <- mapM (getPostContent . postId) ordered
```


```
  return $ renderPosts (zip ordered content)
```

Sort it by date,
and take the
latest 5

```
mainPane :: Haxl Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered = take 5 $ sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)
```

Fetch the
content for
those 5

```
mainPane :: Haxl Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered = take 5 $ sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)
```



And finally
render the
output

Things to note

```
mainPane :: Haxl Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered = take 5 $ sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)
```

- No explicit concurrency constructs
 - Just standard structuring tools: do-notation, <*>, mapM
 - No concurrency bugs

No manual batching

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

No manual batching

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

This line
performs many
data fetches

No manual batching

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

This line
performs many
data fetches

Unbatched

```
SELECT postinfo FROM posts
  WHERE postid = id1
```

```
SELECT postinfo FROM posts
  WHERE postid = id2
```

...

No manual batching

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

This line
performs many
data fetches

Unbatched

```
SELECT postinfo FROM posts
  WHERE postid = id1
```

```
SELECT postinfo FROM posts
  WHERE postid = id2
```

...

Batched

```
SELECT postinfo FROM posts
  WHERE postid IN {id1, id2, ...}
```

How *you* can use Haxl

Open Source:

<https://github.com/facebook/Haxl>

The screenshot shows the GitHub repository page for `facebook/Haxl`. The repository is described as "A Haskell library for efficient, concurrent, concise data access." It has 28 commits, 1 branch, 0 releases, and 12 contributors. The current branch is `master`. A merge pull request #12 from `zudov/patch-1` is shown, authored by `simonmar` on 23 Jul. The latest commit is `79d1f1eea9`. The file list includes `Haxl`, `example`, `tests`, `.gitignore`, `LICENSE`, `PATENTS`, `Setup.hs`, `TARGETS`, `haxl.cabal`, `logo.png`, `logo.svg`, and `readme.md`. The right sidebar shows the `Code` tab selected, with links to `Issues` (1), `Pull Requests` (0), `Wiki`, `Pulse`, and `Graphs`. The SSH clone URL is `git@github.com:facebook:Haxl`. There are buttons for `Clone in Desktop` and `Download ZIP`.

facebook / Haxl

Unwatch 126 Star 1,254 Fork 89

A Haskell library for efficient, concurrent, concise data access.

28 commits 1 branch 0 releases 12 contributors

branch: master Haxl / +

Merge pull request #12 from `zudov/patch-1`

`simonmar` authored on 23 Jul latest commit `79d1f1eea9`

<code>Haxl</code>	Rework memoization, fixing infinite loop problems	2 months ago
<code>example</code>	Added a note about the Prelude	2 months ago
<code>tests</code>	Update exposed-modules in <code>haxl.cabal</code>	2 months ago
<code>.gitignore</code>	Initial open source import	3 months ago
<code>LICENSE</code>	Initial open source import	3 months ago
<code>PATENTS</code>	Initial open source import	3 months ago
<code>Setup.hs</code>	Initial open source import	3 months ago
<code>TARGETS</code>	Initial open source import	3 months ago
<code>haxl.cabal</code>	Update exposed-modules in <code>haxl.cabal</code>	2 months ago
<code>logo.png</code>	Initial open source import	3 months ago
<code>logo.svg</code>	Add SVG logo	3 months ago
<code>readme.md</code>	Merge pull request #1 from <code>oreoshake/mixed_content_in_readme</code>	3 months ago

`<> Code`

`Issues` 1

`Pull Requests` 0

`Wiki`

`Pulse`

`Graphs`

SSH clone URL

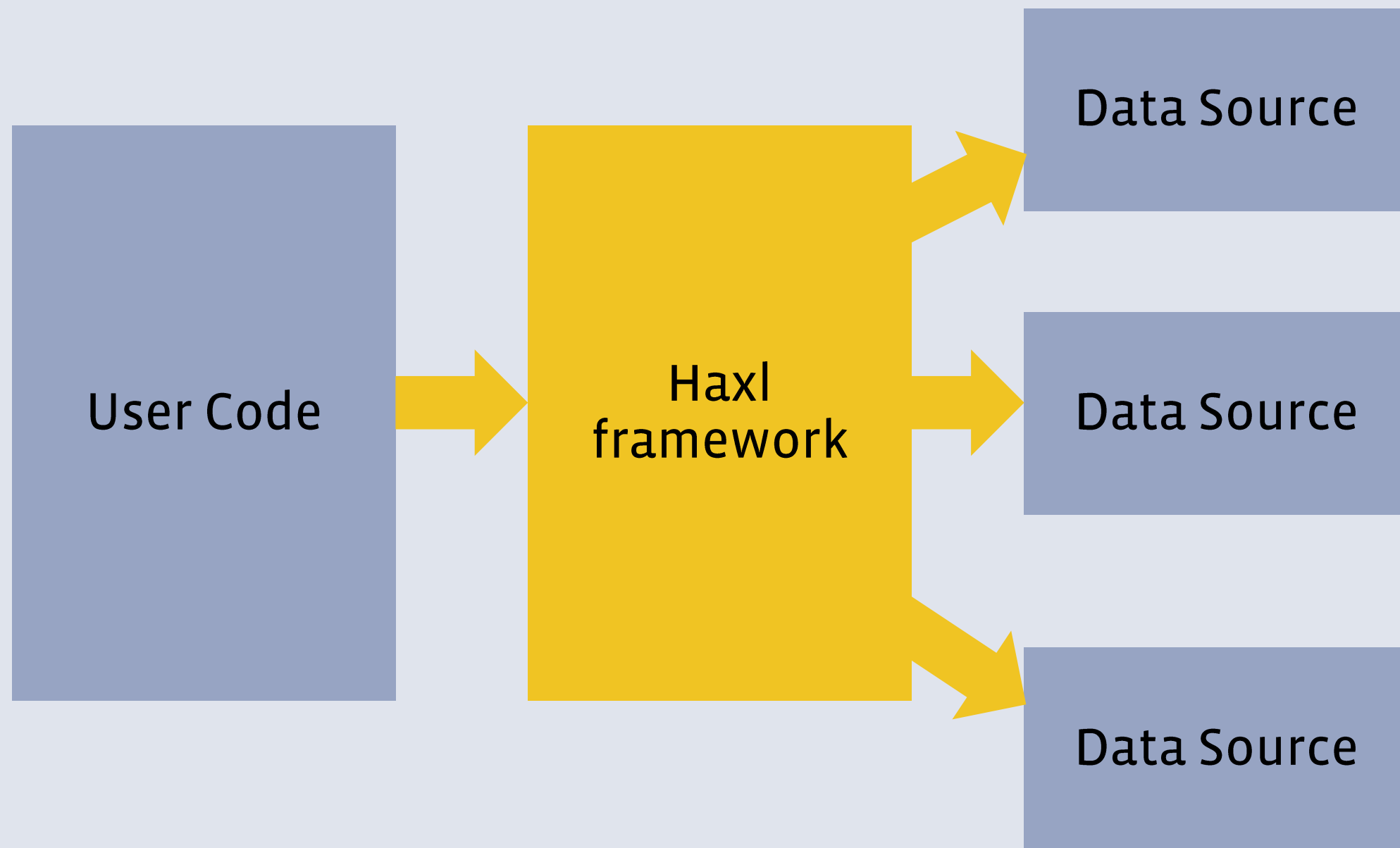
`git@github.com:facebook:Haxl`

You can clone with `HTTPS`, `SSH`, or `Subversion`.

`Clone in Desktop`

`Download ZIP`

```
cabal install haxl
```



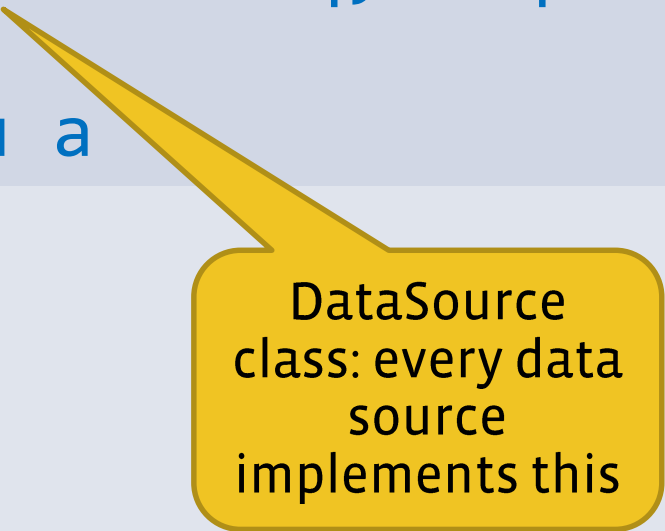
DataSource walk-through

- We'll walk through two complete data sources
 1. A data source for the Facebook Graph API
 - web API for querying the Facebook Graph
 - using Felipe Lessa's fb package to do the real work
 - Using threads to perform requests **concurrently**
 2. A data source to extract data from an SQL database
 - **Batching** multiple requests into a single SQL query

```
dataFetch :: (DataSource u req, Request req a)
           => req a
           -> GenHaxl u a
```



```
dataFetch :: (DataSource u req, Request req a)
           => req a
           -> GenHaxl u a
```



DataSource
class: every data
source
implements this

```
dataFetch :: (DataSource u req, Request req a)
           => req a
           -> GenHaxl u a
```

DataSource
class: every data
source
implements this

```
type Request req a =
  ( Eq (req a)
  , Hashable (req a)
  , Typeable (req a)
  , Show (req a)
  , Show a
  )
```

```
dataFetch :: (DataSource u req, Request req a)
           => req a
           -> GenHaxl u a
```

DataSource
class: every data
source
implements this

User state –
passed around,
can be accessed
by data sources

```
type Request req a =  
  ( Eq (req a)  
  , Hashable (req a)  
  , Typeable (req a)  
  , Show (req a)  
  , Show a  
  )
```

- Start with the request type:

```
data FacebookReq a where
  GetObject      :: Id -> FacebookReq Object
  GetUser        :: UserId -> FacebookReq User
  GetUserFriends :: UserId -> FacebookReq [Friend]
  deriving Typeable
```

- We also need some boilerplate:

```
deriving instance Eq (FacebookReq a)
deriving instance Show (FacebookReq a)

instance Show1 FacebookReq where show1 = show

instance Hashable (FacebookReq a) where ...
```

The DataSource class

```
class DataSourceName req where  
  dataSourceName :: req a -> Text
```

```
class (DataSourceName req, StateKey req, Show1 req)  
  => DataSource u req where
```

```
  fetch  
    :: State req  
    -> Flags  
    -> u  
    -> [BlockedFetch req]  
    -> PerformFetch
```

The Haxl monad collects requests from dataFetch calls, and passes them to fetch

```
class Typeable f => StateKey (f :: * -> *) where  
  data State f
```

The DataSource class

```
class DataSourceName req where  
  dataSourceName :: req a -> Text
```

```
class (DataSourceName req, StateKey req, Show1 req)  
  => DataSource u req where
```

```
  fetch  
    :: State req  
    -> Flags  
    -> u  
    -> [BlockedFetch req]  
    -> PerformFetch
```

The Haxl monad collects requests from dataFetch calls, and passes them to fetch

```
class Typeable f => StateKey (f :: * -> *) where  
  data State f
```

```
instance DataSourceName FacebookReq where  
  dataSourceName _ = "Facebook"
```

A data source has some state:

```
instance StateKey FacebookReq where
  data State FacebookReq =
    FacebookState
      { credentials :: Credentials
      , userAccessToken :: UserAccessToken
      , manager :: Manager
      , numThreads :: Int
      }
```

API keys

HTTP connection
manager

Concurrency
control

Initialise the state

```
initGlobalState
  :: Int
  -> Credentials
  -> UserAccessToken
  -> IO (State FacebookReq)

initGlobalState threads creds token = do
  manager <- newManager tlsManagerSettings
  return FacebookState
    { credentials = creds
    , manager = manager
    , userAccessToken = token
    , numThreads = threads
    }
```

- nothing surprising there.

Implementing fetch

```
class (DataSourceName req, StateKey req, Show1 req)
  => DataSource u req where
  fetch
    :: State req
    -> Flags
    -> u
    -> [BlockedFetch req]
    -> PerformFetch
```

Implementing fetch

```
class (DataSourceName req, StateKey req, Show1 req)
  => DataSource u req where
  fetch
    :: State req
    -> Flags
    -> u
    -> [BlockedFetch req]
    -> PerformFetch
```



Data source state

Implementing fetch

```
class (DataSourceName req, StateKey req, Show1 req)  
  => DataSource u req where
```

fetch

```
  :: State req
```

```
  -> Flags
```

```
  -> u
```

```
  -> [BlockedFetch req]
```

```
  -> PerformFetch
```

Data source state

Haxl monad flags (tracing etc.)

Implementing fetch

```
class (DataSourceName req, StateKey req, Show1 req)  
  => DataSource u req where
```

```
  fetch
```

```
    :: State req
```

```
    -> Flags
```

```
    -> u
```

```
    -> [BlockedFetch req]
```

```
    -> PerformFetch
```

Data source state

Haxl monad flags (tracing etc.)

User state (we'll use () here)

Implementing fetch

```
class (DataSourceName req, StateKey req, Show1 req)
  => DataSource u req where
  fetch
    :: State req
    -> Flags
    -> u
    -> [BlockedFetch req]
    -> PerformFetch
```

Data source state

Haxl monad flags (tracing etc.)

User state (we'll use () here)

The requests!

```
data BlockedFetch r = forall a. BlockedFetch (r a) (ResultVar a)
```

Implementing fetch

```
class (DataSourceName req, StateKey req, Show1 req)  
  => DataSource u req where
```

fetch

```
  :: State req
```

```
  -> Flags
```

```
  -> u
```

```
  -> [BlockedFetch req]
```

```
  -> PerformFetch
```

Data source state

Haxl monad flags (tracing etc.)

User state (we'll use () here)

The requests!

sync or async?

```
data BlockedFetch r = forall a. BlockedFetch (r a) (ResultVar a)
```

```
data PerformFetch
```

```
  = SyncFetch (IO ())
```

```
  | AsyncFetch (IO () -> IO ())
```

Implement fetch

```
data PerformFetch
  = SyncFetch  (IO ())
  | AsyncFetch (IO () -> IO ())
```

```
instance DataSource u FacebookReq where
  fetch = facebookFetch
```

```
facebookFetch
  :: State FacebookReq
  -> Flags
  -> ()
  -> [BlockedFetch FacebookReq]
  -> PerformFetch
```

```
facebookFetch FacebookState{..} _flags _user bfs =
  AsyncFetch $ \inner -> do
    sem <- newQSem numThreads
    asyncs <- mapM (async . fetchAsync credentials manager
                                     userAccessToken sem) bfs

    inner
    mapM_ wait asyncs
```

▪ Implement fetchAsync

```
fetchAsync
  :: Credentials -> Manager -> UserAccessToken -> QSem
  -> BlockedFetch FacebookReq
  -> IO ()

fetchAsync creds manager tok sem (BlockedFetch req rvar) =
  bracket_ (waitQSem sem) (signalQSem sem) $ do

    e <- Control.Exception.try $
      runResourceT $
      runFacebookT creds manager $
      fetchReq tok req

    case e of
      Left ex -> putFailure rvar (ex :: SomeException)
      Right a  -> putSuccess rvar a
```


- fetchReq maps FacebookReq to FacebookT computations

```
fetchReq
  :: UserAccessToken
  -> FacebookReq a
  -> FacebookT Auth (ResourceT IO) a

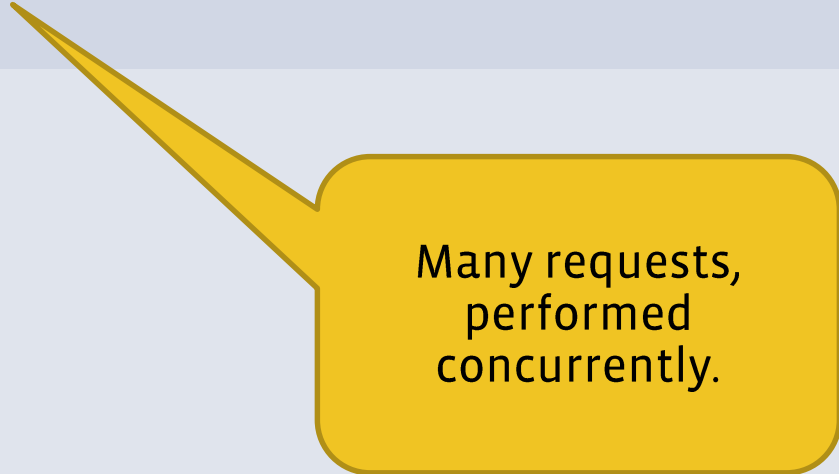
fetchReq tok (GetObject (Id id)) =
  getObject ("/" <> id) [] (Just tok)

fetchReq _tok (GetUser id) =
  getUser id [] Nothing

fetchReq tok (GetUserFriends id) = do
  f <- getUserFriends id [] tok
  source <- fetchAllNextPages f
  source $$ consume
```

Example

```
main :: IO ()
main = do
  (creds, access_token) <- getCredentials
  facebookState <- initGlobalState 10 creds access_token
  env <- initEnv (stateSet facebookState stateEmpty) ()
  r <- runHaxl env $ do
    likes <- getObject "me/likes"
    mapM getObject (likeIds likes)
  print r
```



Many requests,
performed
concurrently.

SQL example

Questions?

<https://github.com/facebook/Haxl>

```
cabal install haxl
```