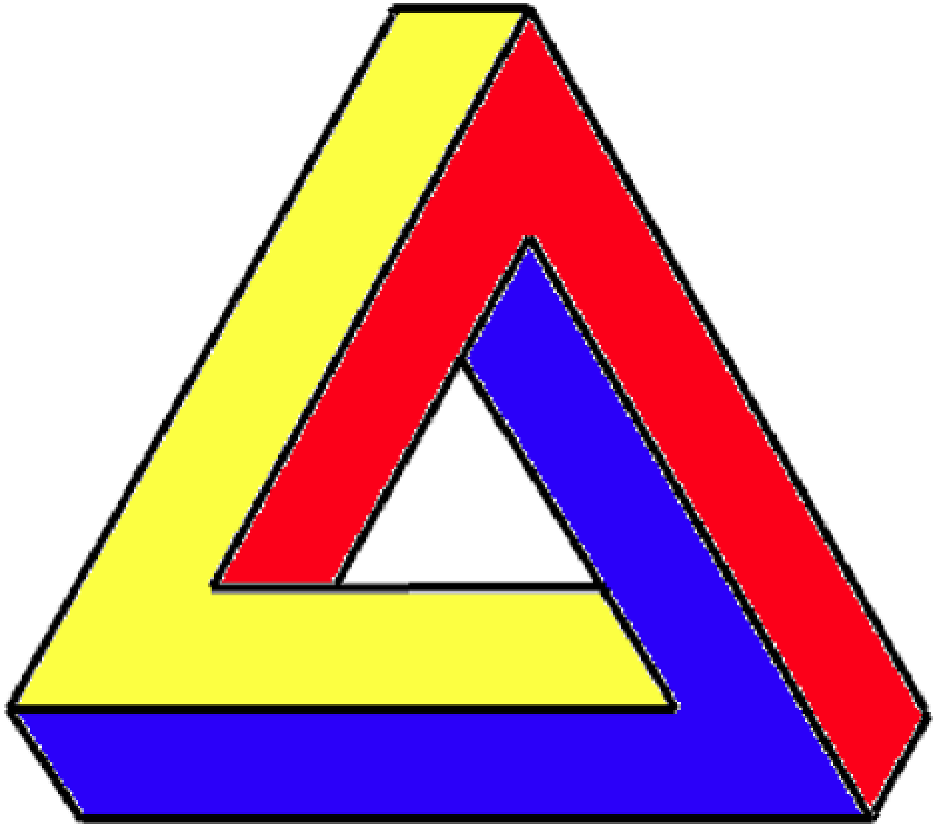
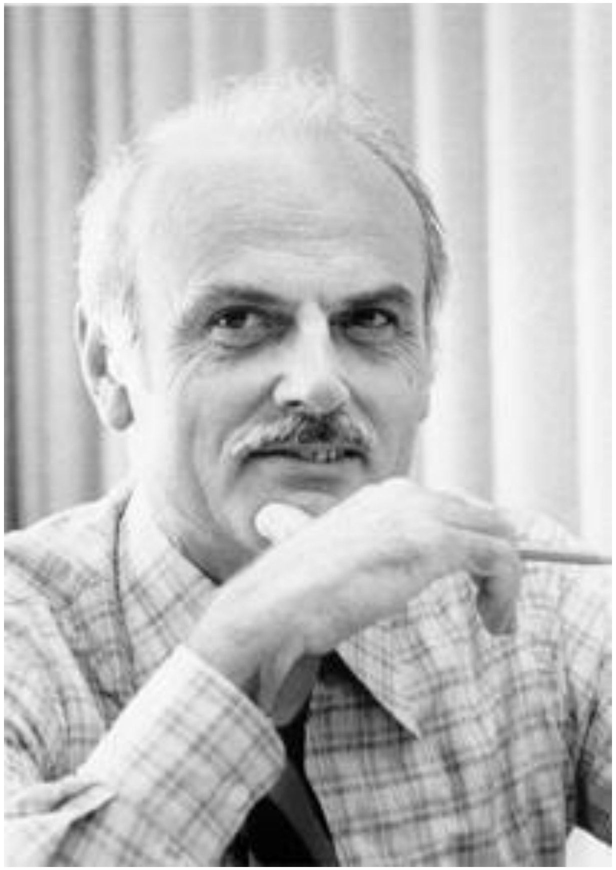


APPLIED DUALITY

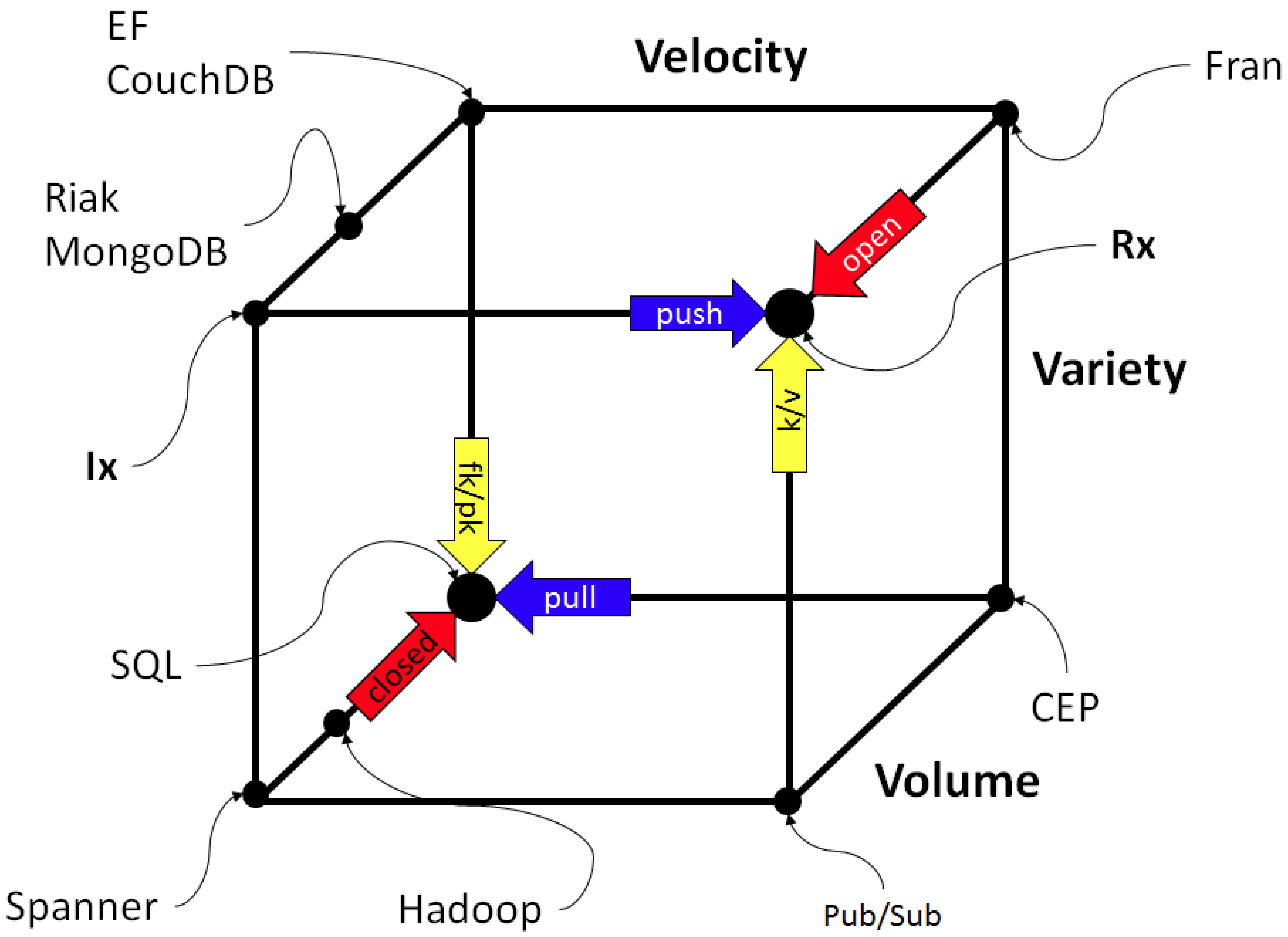
www.applied-duality.com

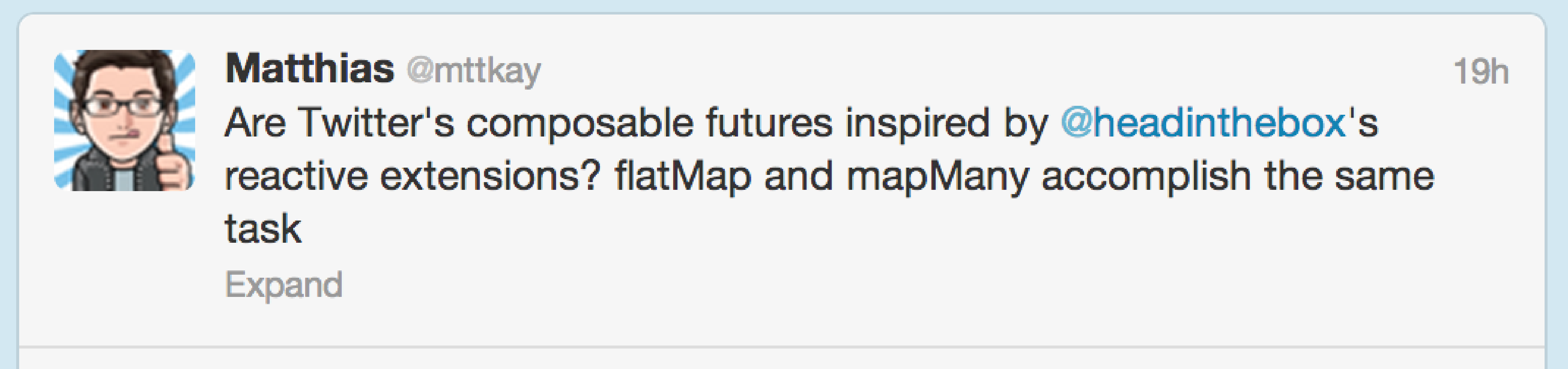
Ted Codd Was Not A Developer



Pencil

Asynchronous Programming With Rx


EF
CouchDB
Velocity
Fran
Riak
MongoDB
open
Rx
push
Variety
k/v
Ix
fk/pk
pull
SQL
closed
CEP
Volume
Spanner
Hadoop
Pub/Sub



Matthias @mttkay 19h

Are Twitter's composable futures inspired by @headinthebox's reactive extensions? flatMap and mapMany accomplish the same task

The Four Essential Effects of Modern Applications

	One Result	**Many Results**
Synchronous	T	Iterable<T>
Asyncronous	Future<T>	IObservable<T>

blocking

```
int x = Foo.Bar(4711);
int y = Bar.Qux(x);
```

used once

blocking

```
Iterable<int> xs = Foo.Bar(4711);
for(int x : xs)
{
   System.out.println(x);
}
```

blocking

used many times

non-blocking

```
Task<int> x = Foo.Bar(4711);
int y = await Bar.Qux(await x);
```

used once

non-blocking

```
IObservable<int> xs = Foo.Bar(4711);
IDisposable d = xs.Subscribe(int x ->
{
   System.out.println(x);
});
```

non-blocking

used many times

A traditional Futur ult of an asynchronous computation: a cor r may not have finished producing a result e a handle to an in-progress computation, a promise om a service to supply us with a result.

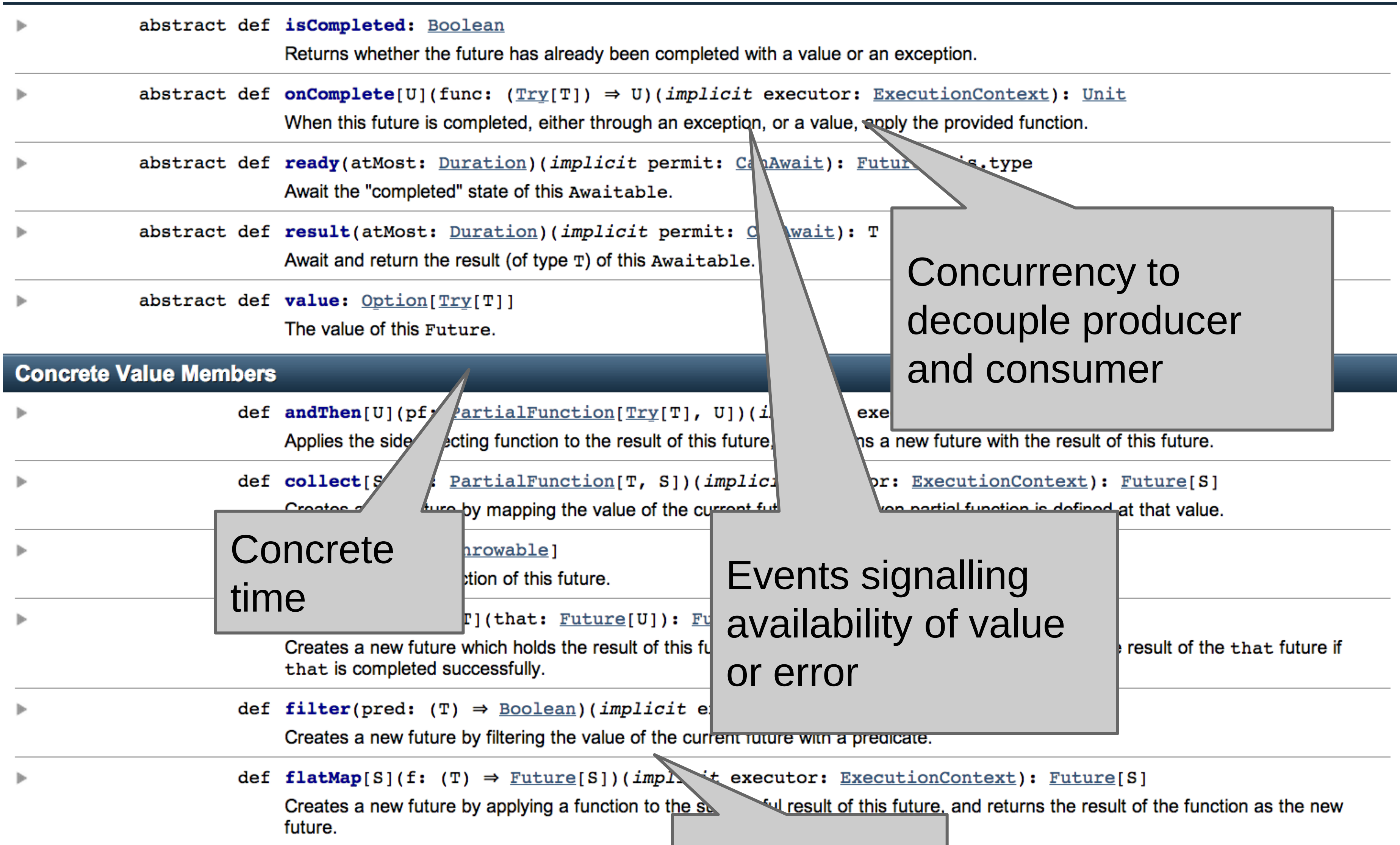
Events signalling availability of value or error

A ListenableFuture allows you to register callbacks to be executed once the computation is complete, or if the computation is already complete, immediately. This it possible to efficiently support many ope Future interface cannot support.

The basic operation added by Listenab e is addListener(Runnable, Executor), which specifies that when the computation represented by this Future is done, the specified Runnable will be run on the specified Executor.

Concurrency to decouple producer and consumer

https://code.google.com/p/guava-libraries/wiki/ListenableFutureExplained



`abstract def isCompleted: Boolean`

Returns whether the future has already been completed with a value or an exception.

`abstract def onComplete[U](func: (Try[T]) ⇒ U)(implicit executor: ExecutionContext): Unit`

When this future is completed, either through an exception, or a value, apply the provided function.

`abstract def ready(atMost: Duration)(implicit permit: CanAwait): Futur s.type`

Await the "completed" state of this Awaitable.

`abstract def result(atMost: Duration)(implicit permit: C wait): T`

Await and return the result (of type T) of this Awaitable.

`abstract def value: Option[Try[T]]`

The value of this Future.

Concrete Value Members

`def andThen[U](pf: PartialFunction[Try[T], U])(i exe`

Applies the side ecting function to the result of this future, ns a new future with the result of this future.

`def collect[S : PartialFunction[T, S])(implici or: ExecutionContext): Future[S]`

by mapping the value of the current fut at that value.

`hrowable]`

tion of this future.

`T](that: Future[U]): Fu`

Creates a new future which holds the result of this fu result of the that future if that is completed successfully.

`def filter(pred: (T) ⇒ Boolean)(implicit e`

Creates a new future by filtering the value of the current future with a predicate.

`def flatMap[S](f: (T) ⇒ Future[S])(impl it executor: ExecutionContext): Future[S]`

Creates a new future by applying a function to the s ful result of this future, and returns the result of the function as the new future.

Concurrency to decouple producer and consumer

Concrete time

Events signalling availability of value or error

Composition

http://www.scala-lang.org/archives/downloads/distrib/files/nightl ml#scala.concurrent.Future

Futures are "hot", i.e. a value of type Future<T> is already running.

Future<T> represents a single value.

Continuation passing style (callbacks) is painful.

Concurrency is important aspect.

Time is important aspect.

Cancellation?

In the .NET world

All asynchronous operations that return a single result are expressed as Task<T>

coMonadic

```
class Task<T>
  {
    Task<R> ContinueWith
          (Func<Task<T>, R>
  continuation)

      T Result { get; }
  }
```

In the .NET world

All asynchronous computations that return a single result use regular control structures via async await.

```
byte[] result;using(var SourceStream =
  File.Open(...))
{
      result = new byte[SourceStream.Length];
      await SourceStream.ReadAsync
         (result, 0, (int)SourceStream.Length);
  }
```

Compiler generates callback/statemachine

In the .NET world

Task-based asynchronous operations never implicitly introduce concurrency. Async-ness bubbles up the call-stack/return type.

```
async Task<int> FAsync()
  {
       ...
  var x = await G();
    ...
     return H(x);
  }
```

Inside async method

Can only use await

In the .NET world

Cancellation for asynchronous computations of at most one value and threads is cooperative using cancellation tokens.

```
async Task<int> FAsync(CancellationToken token)
  {
   if(!token.IsCancellationRequested) ...
  }

var s = new CancellationTokenSource();
var t = FAsync(s.Token);
  s.Cancel();
```

In any language

Writing CPS by hand is never acceptable. The compiler should take care of that.

Writing map, flatMap, filter, ... just for collections with at most one value is silly. That is what control structures are for.

Token-based Cancellation does not compose well (is not fluent).

What if you do not have async await?

Kill two birds with one stone.

* Generalize single asynchronous results to asynchronous data streams.

* Compose operations on data streams using map, flatMap, filter, ...

* Note you need asynchronous data streams even when you have async await.

Just one small change is needed ...

A FutureCallback<V> implements two methods:

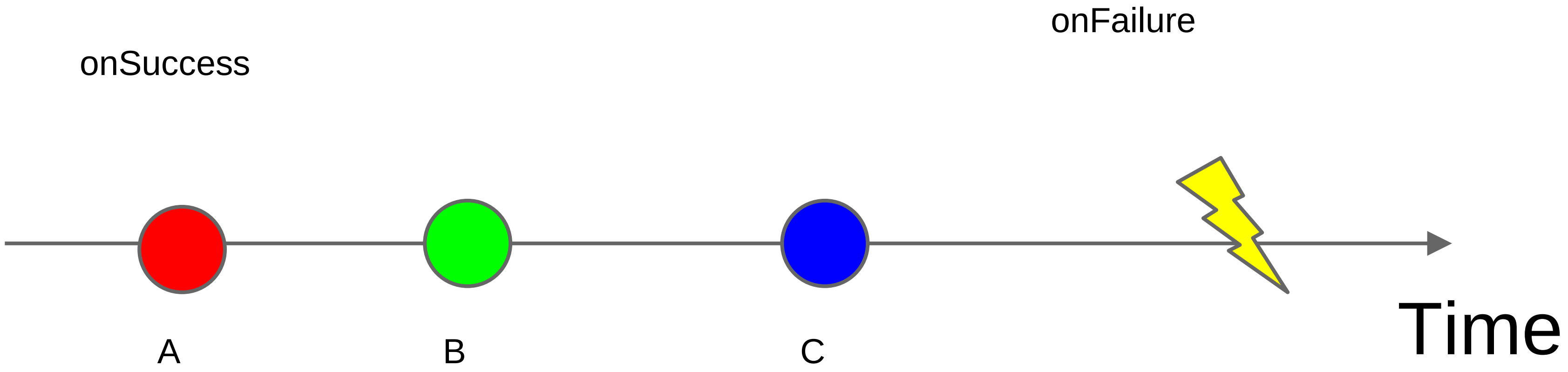
- onSuccess(V), the action to perform if the future succeeds, based on its result
- onFailure(Throwable), the action to perform if the future fails, based on the failure

Can be called multiple times, once for each value in the data stream

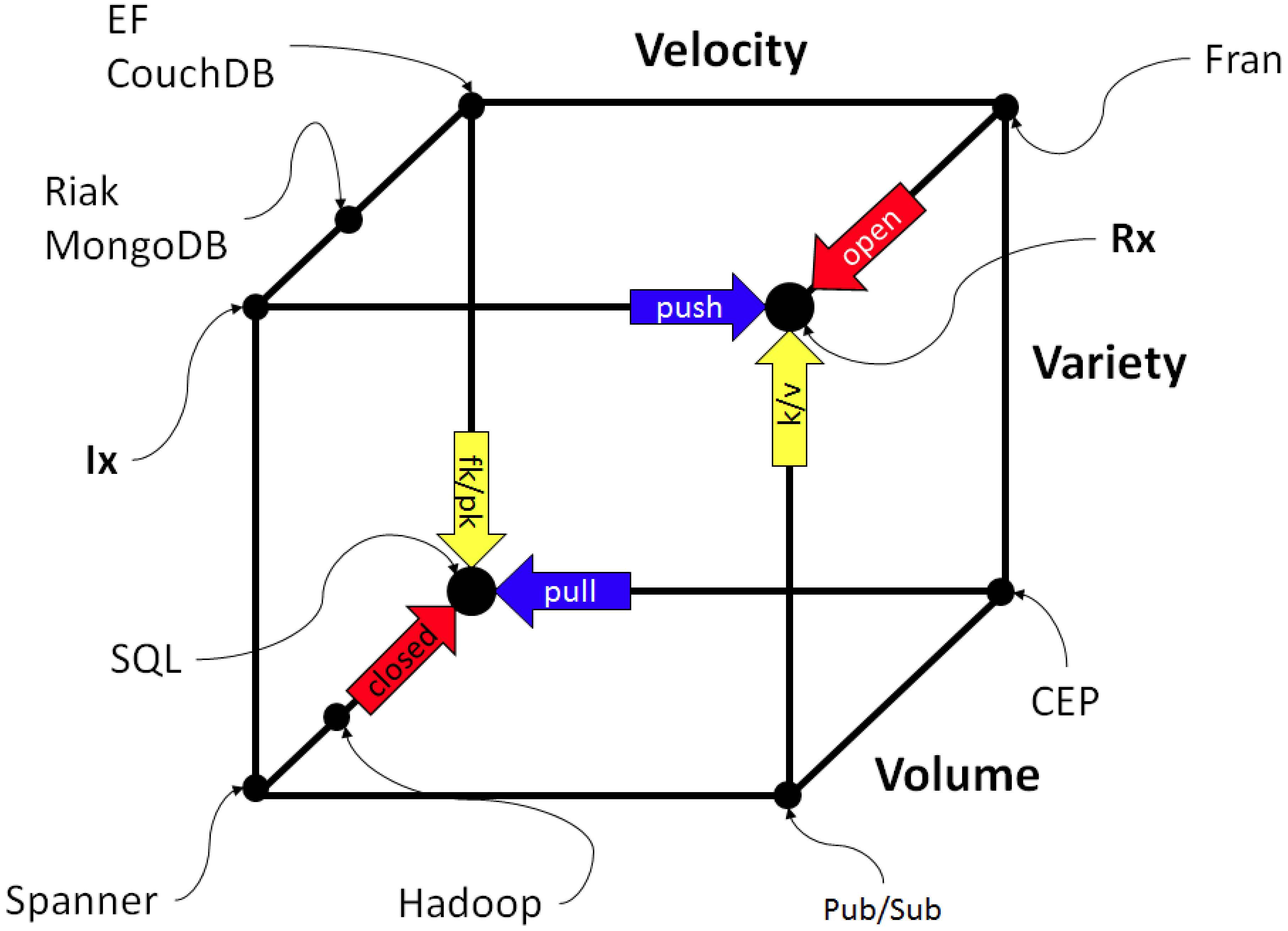
Add a third one

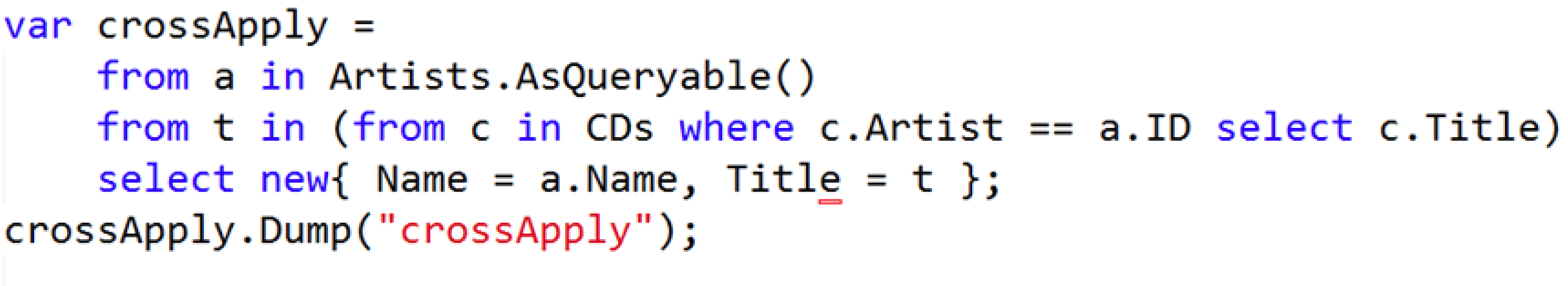
- onCompleted(), the action to perform if the future has terminated successfully.

Marble Diagram

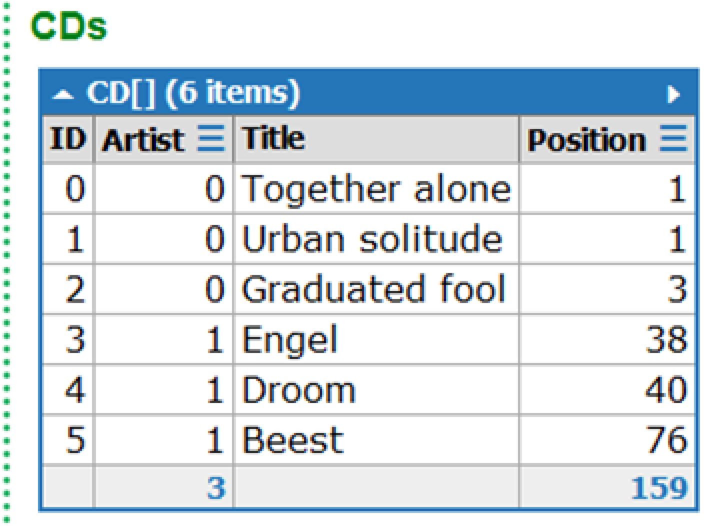


We can define all the standard collection operators over such asynchronous data streams.


EF
CouchDB
Velocity
Fran
Riak
MongoDB
open
Rx
push
Variety
k/v
Ix
fk/pk
pull
SQL
closed
CEP
Volume
Spanner
Hadoop
Pub/Sub



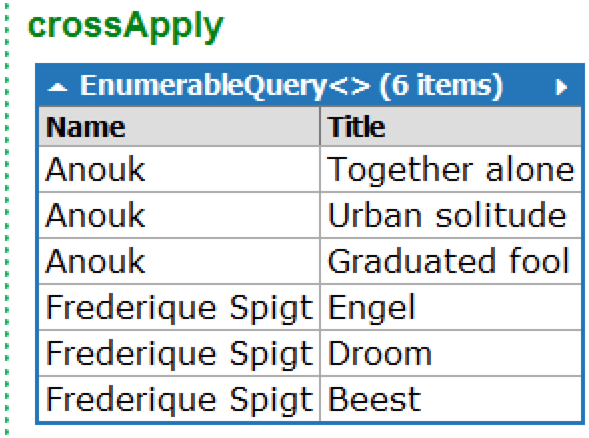
```
var crossApply =
    from a in Artists.AsQueryable()
    from t in (from c in CDs where c.Artist == a.ID select c.Title)
    select new{ Name = a.Name, Title = t };
crossApply.Dump("crossApply");
```



CDs

CD[] (6 items)

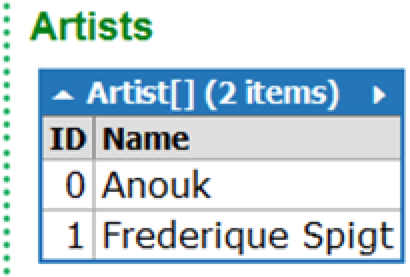
ID	Artist	Title	Position
0	0	Together alone	1
1	0	Urban solitude	1
2	0	Graduated fool	3
3	1	Engel	38
4	1	Droom	40
5	1	Beest	76
	3		159



crossApply

EnumerableQuery<> (6 items)

Name	Title
Anouk	Together alone
Anouk	Urban solitude
Anouk	Graduated fool
Frederique Spigt	Engel
Frederique Spigt	Droom
Frederique Spigt	Beest



Artists

Artist[] (2 items)

ID	Name
0	Anouk
1	Frederique Spigt

Ted Codd's Relational Algebra

σ	$\in$	$\{S\}\times(S\rightarrow \mathrm{bool})$	$\rightarrow$	$\{S\}$
π	$\in$	$\{S\}\times(S\rightarrow T)$	$\rightarrow$	$\{T\}$
X	$\in$	$\{S\}\times\{T\}$	$\rightarrow$	$\{S\times T\}$
$@$	$\in$	$\{S\}\times(S\rightarrow\{T\})$	$\rightarrow$	$\{T\}$
$\varnothing$	$\in$			$\{S\}$
$\cup$	$\in$	$\{S\}\times\{S\}$	$\rightarrow$	$\{S\}$
$\{\}$	$\in$	S	$\rightarrow$	$\{S\}$

map

filter

flatMap

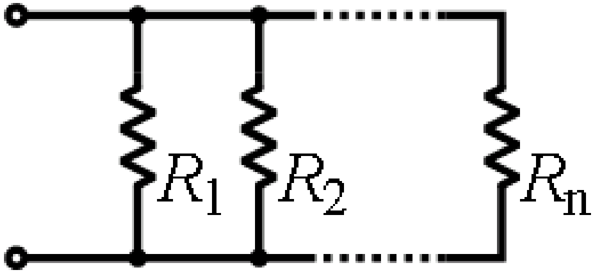
Ix interfaces (.NET Version)

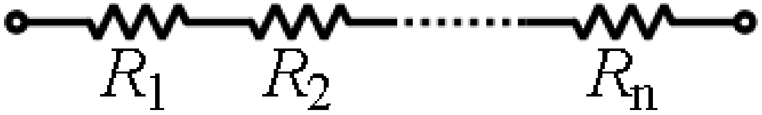
```
interface IEnumerable<T>
  {
    IEnumerator<T> GetEnumerator()
  }

interface IEnumerator<T>
{
    bool MoveNext()
    T Current { get; }
}
```

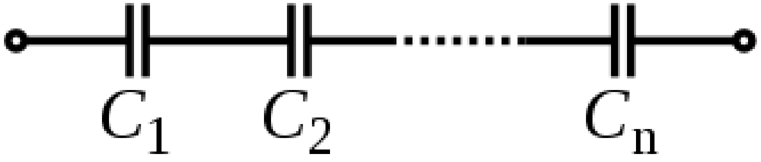
Duality

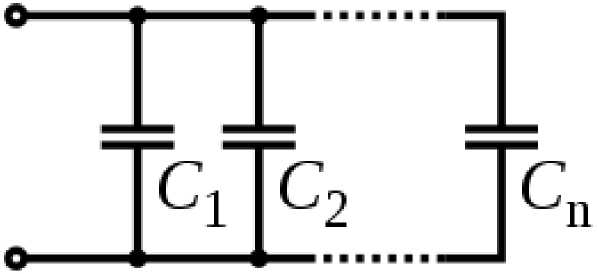
$1/R = 1/R_1 + \ldots + 1/R_n$

$R = R_1 + \ldots + R_n$









$1/C = 1/C_1 + \ldots + 1/C_n$

$C = C_1 + \ldots + C_n$

Rx Interfaces (.NET version)

```
interface IObservable<T>
  {
   IDisposable Subscribe(IObserver<T> observer)
  }


interface IObserver<T>
  {
   void OnNext(T value)
     void OnError(Exception e)
   void OnCompleted()
  }
```

subscription (to unsubscribe from further notifications)

Callbacks for each possible event

Rx == multi-valued ListenableFuture/Scala Future

We did not address cancellation, concurrency and time yet.

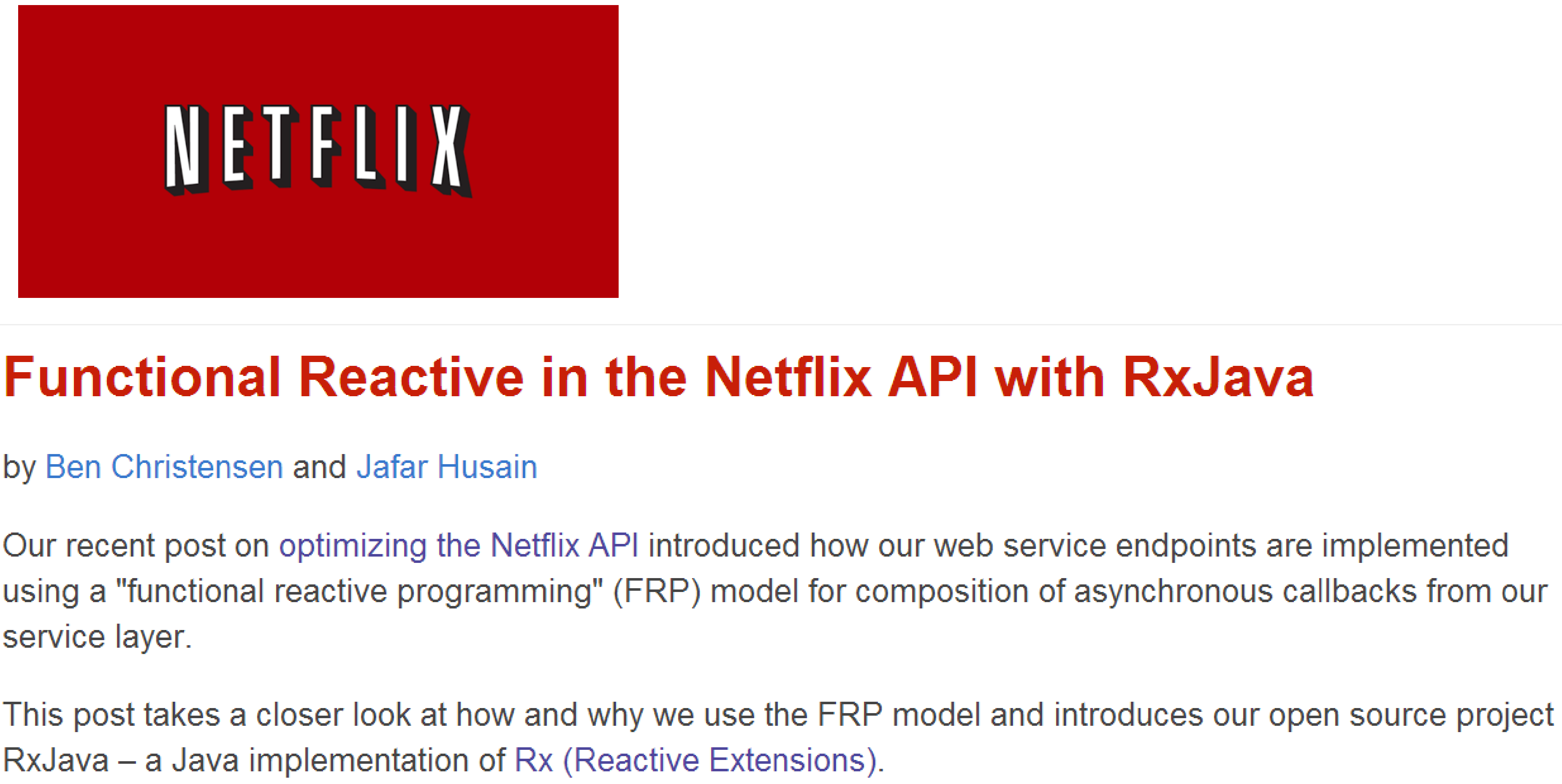
```
Observable<T> xs = ...;
Closable d = xs.subscribe
    (onNext, onError, onCompleted);
  ....
  d.close();
```

Unsubscription vs Cancellation

Multiptiple observers can be subscribed to the same observable.

Disposing the subscription stops delivering new events/values to that subscriber (best effort).

Could mean underlying computation is cancelled, or not.



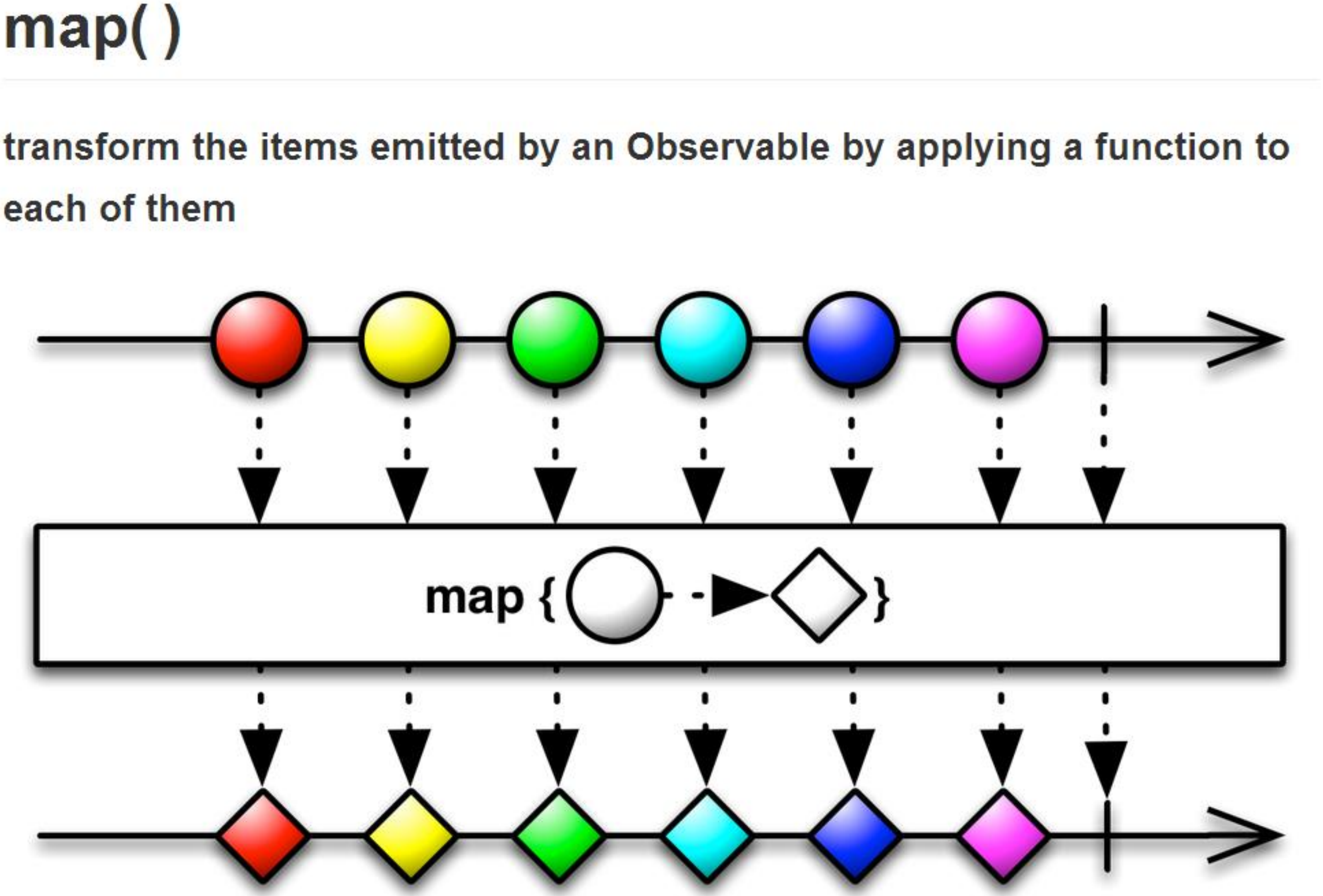
Functional Reactive in the Netflix API with RxJava

by Ben Christensen and Jafar Husain

Our recent post on optimizing the Netflix API introduced how our web service endpoints are implemented using a "functional reactive programming" (FRP) model for composition of asynchronous callbacks from our service layer.

This post takes a closer look at how and why we use the FRP model and introduces our open source project RxJava – a Java implementation of Rx (Reactive Extensions).

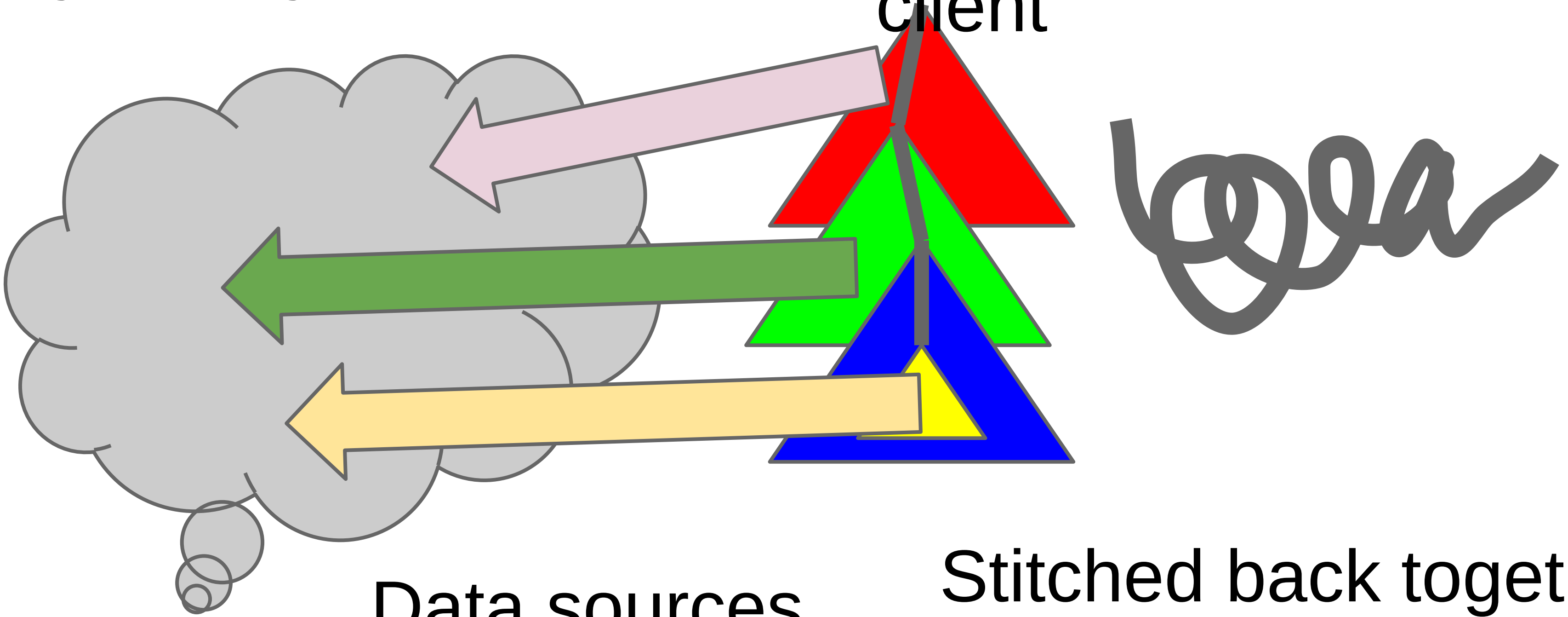
https://github.com/Netflix/RxJava



map( )

transform the items emitted by an Observable by applying a function to each of them

http://channel9.msdn.com/posts/YOW-2012-Jafar-Husain-Rx-and-Netflix-A-Match-Made-in-Composable-Asynchrony



Query results streamed asynchronously to client

Data sources virtualized as document

Stitched back together using path fragments

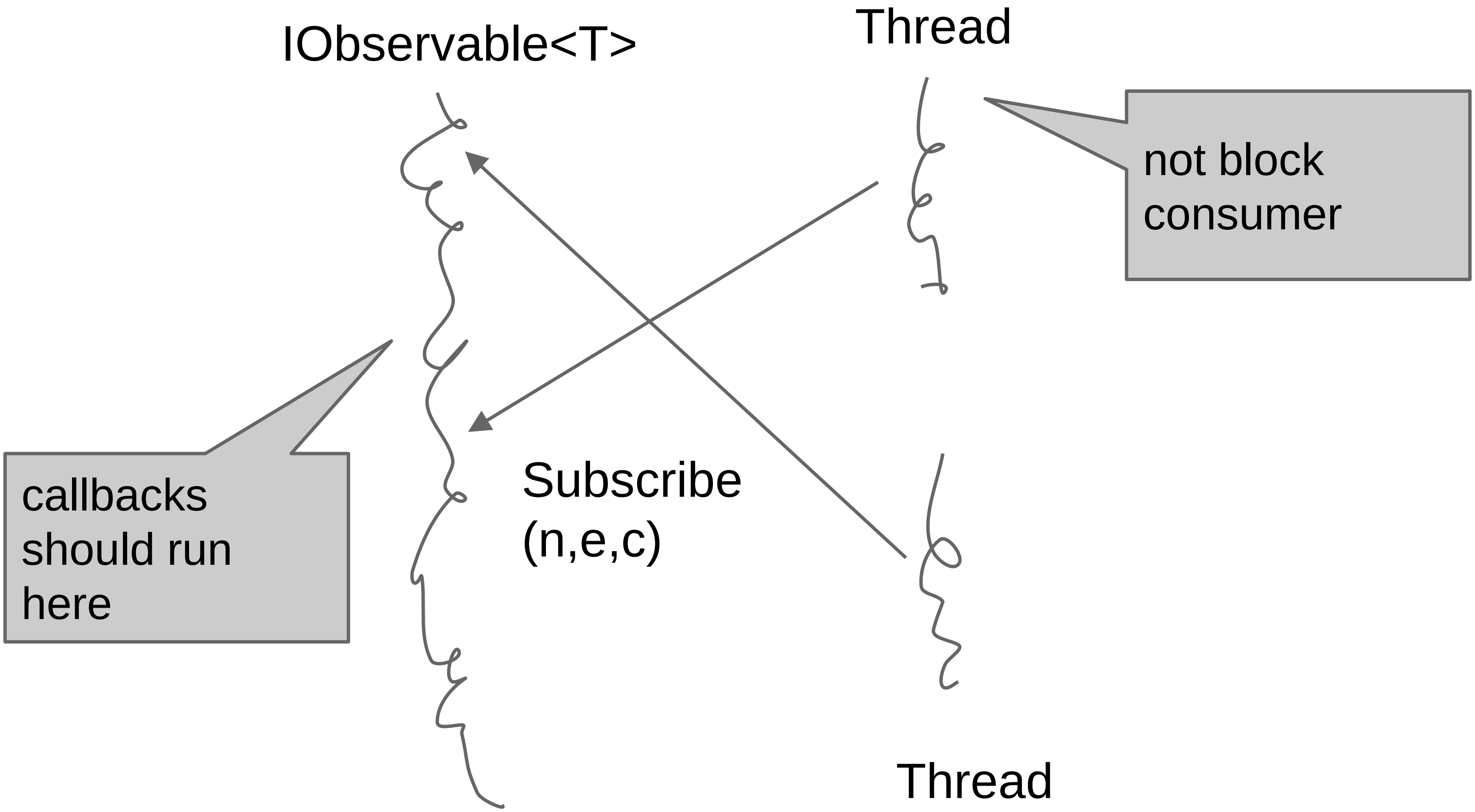
https://github.com/jhusain/learnrx

You'll be surprised to learn that most of the operations you perform on collections can be accomplished with five simple functions:

1. map
2. filter
3. mergeAll
4. reduce
5. zip

Here's my promise to you: if you learn these 5 functions your code will become shorter, more self-descriptive, and more durable. Also, for reasons that might not be obvious right now, you'll learn that these five functions hold the key to simplifying asynchronous programming. Once you've finished this

Schedulers



IObservable<T>

Thread

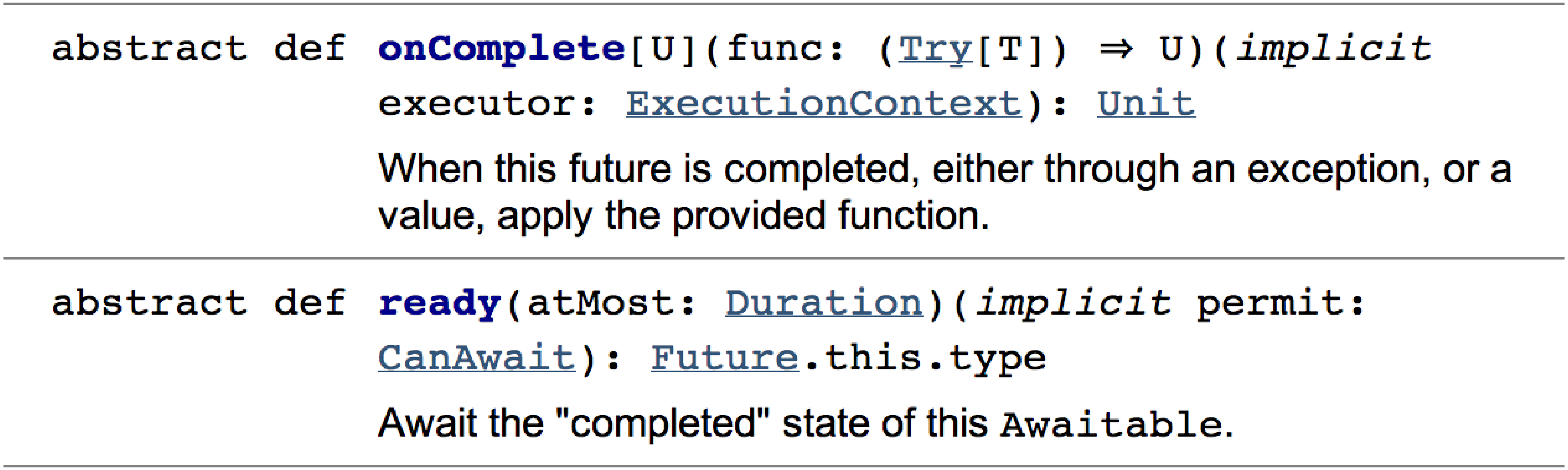
not block consumer

callbacks should run here

Subscribe (n,e,c)

Thread

Time and Concurrency



```
abstract def onComplete[U](func: (Try[T]) ⇒ U)(implicit
             executor: ExecutionContext): Unit
```

When this future is completed, either through an exception, or a value, apply the provided function.

```
abstract def ready(atMost: Duration)(implicit permit:
             CanAwait): Future.this.type
```

Await the "completed" state of this `Awaitable`.

Java executor abstracts from concurrency only, but not from clock/time.

Thread switching should be anywhere in the query chain.

Schedulers .NET version

```
interface IScheduler
{
   DateTimeOffset Now { get; }

   IDisposable Schedule<T>
   ( T state
   , DateTimeOffset delta,
   , Func<IScheduler, T, IDisposable> work
   )
}
```

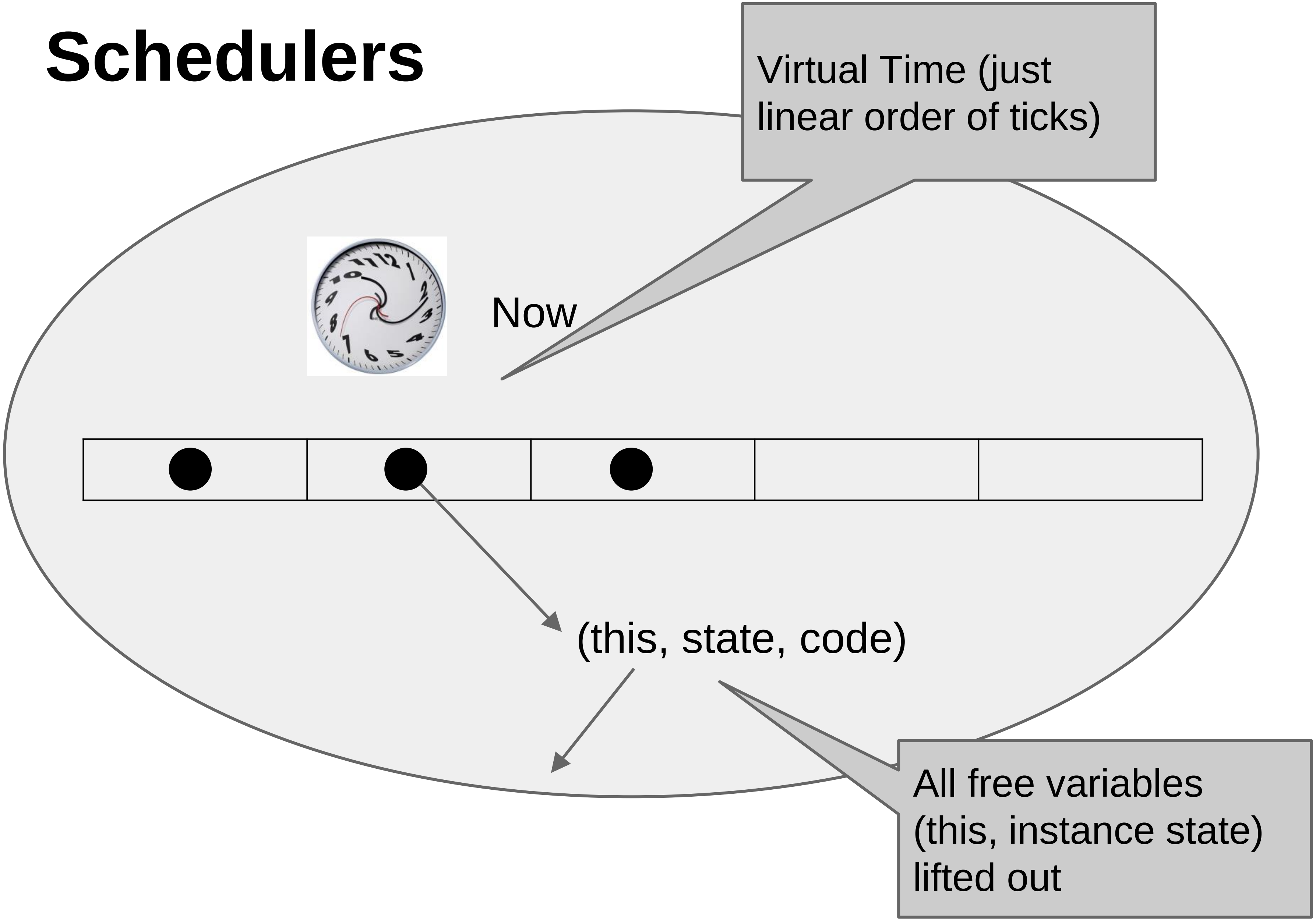
Generalized executors

Easy serialization

Recursion

Delay

Schedulers



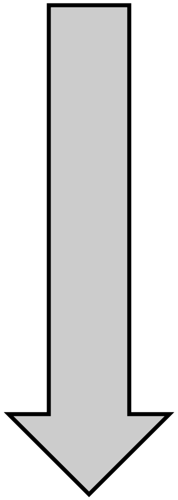
Virtual Time (just linear order of ticks)

Now

(this, state, code)

All free variables (this, instance state) lifted out

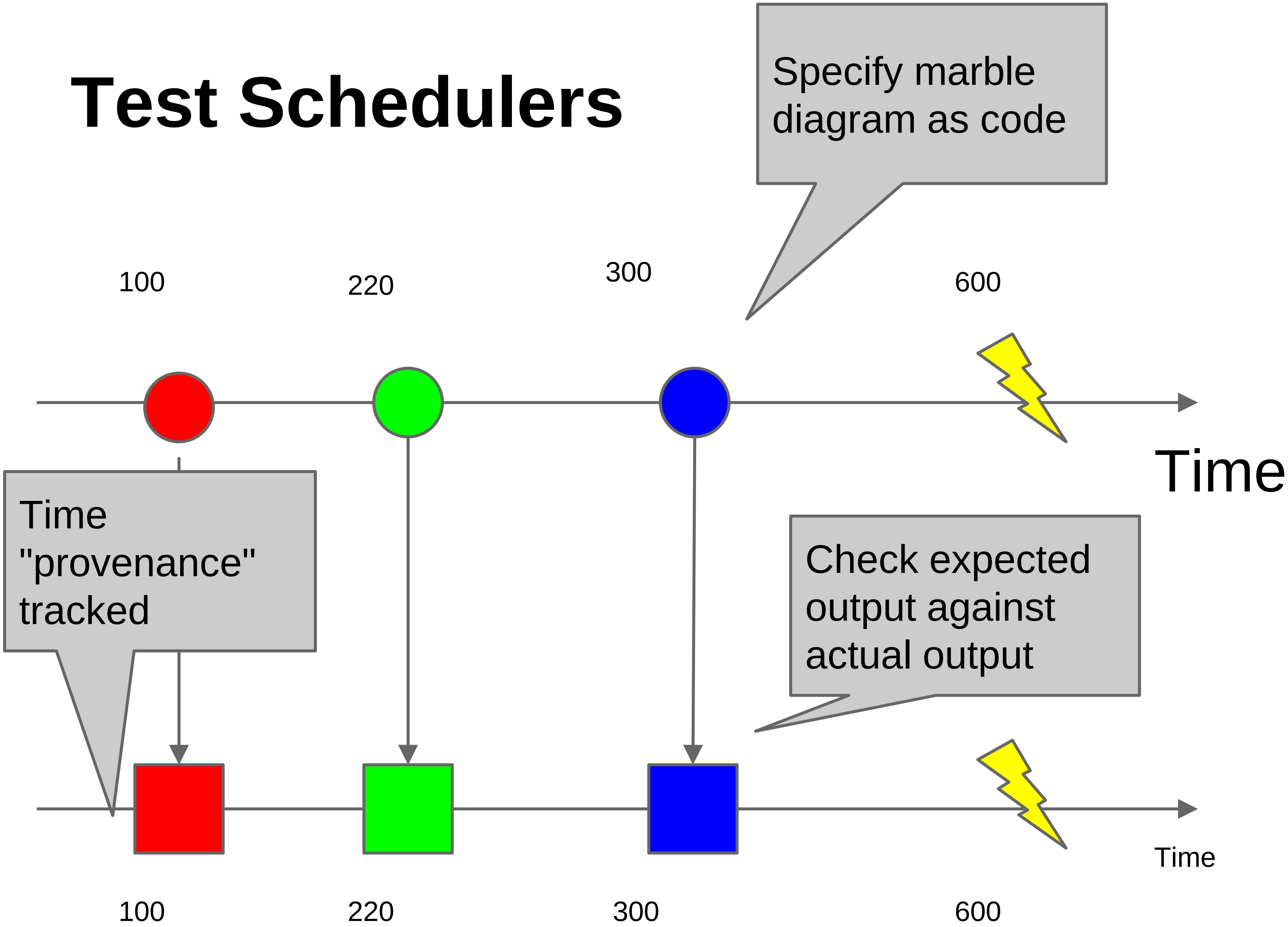
Virtual Time in Log Files (Tx)



05/01/2013/...	{ foo: { bar: 4711 }, baz : "Django Unchained" }
05/01/2013/...	{ foo: { bar: 42 }, bar : "Die Hard 3" }
05/03/2013/...	{ foo: { bar: 1024 }, baz : "Titanic" }
...	

As you are processing each line in the log file, increment clock to latest time-stamp seen.

Test Schedulers

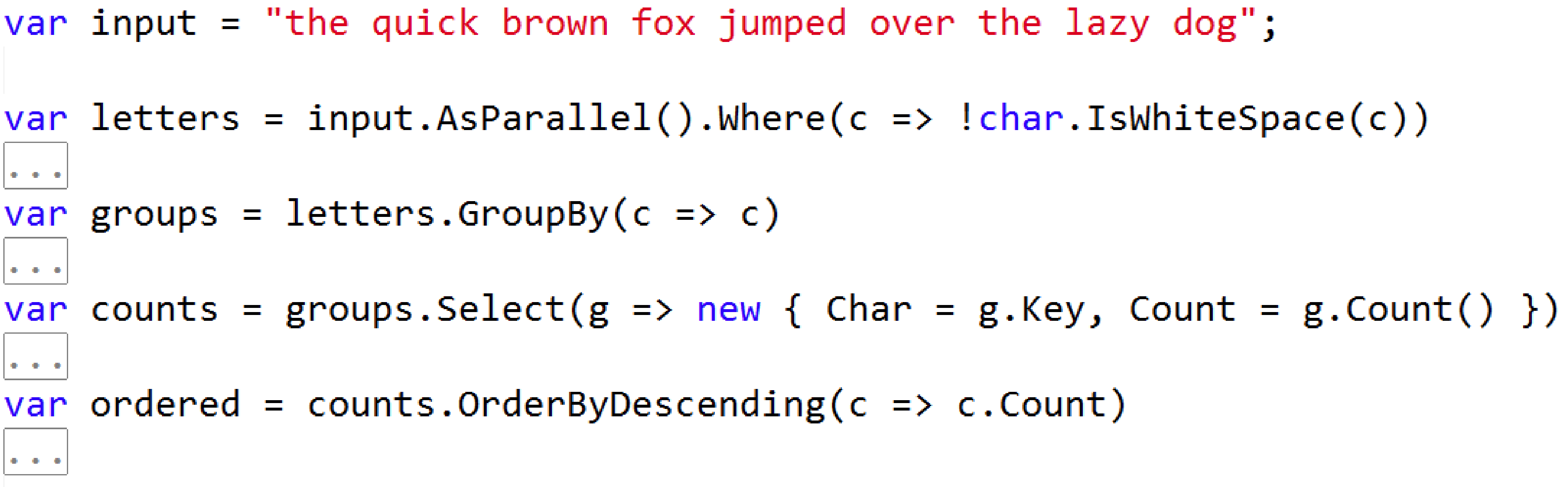


Specify marble diagram as code

Time "provenance" tracked

Check expected output against actual output

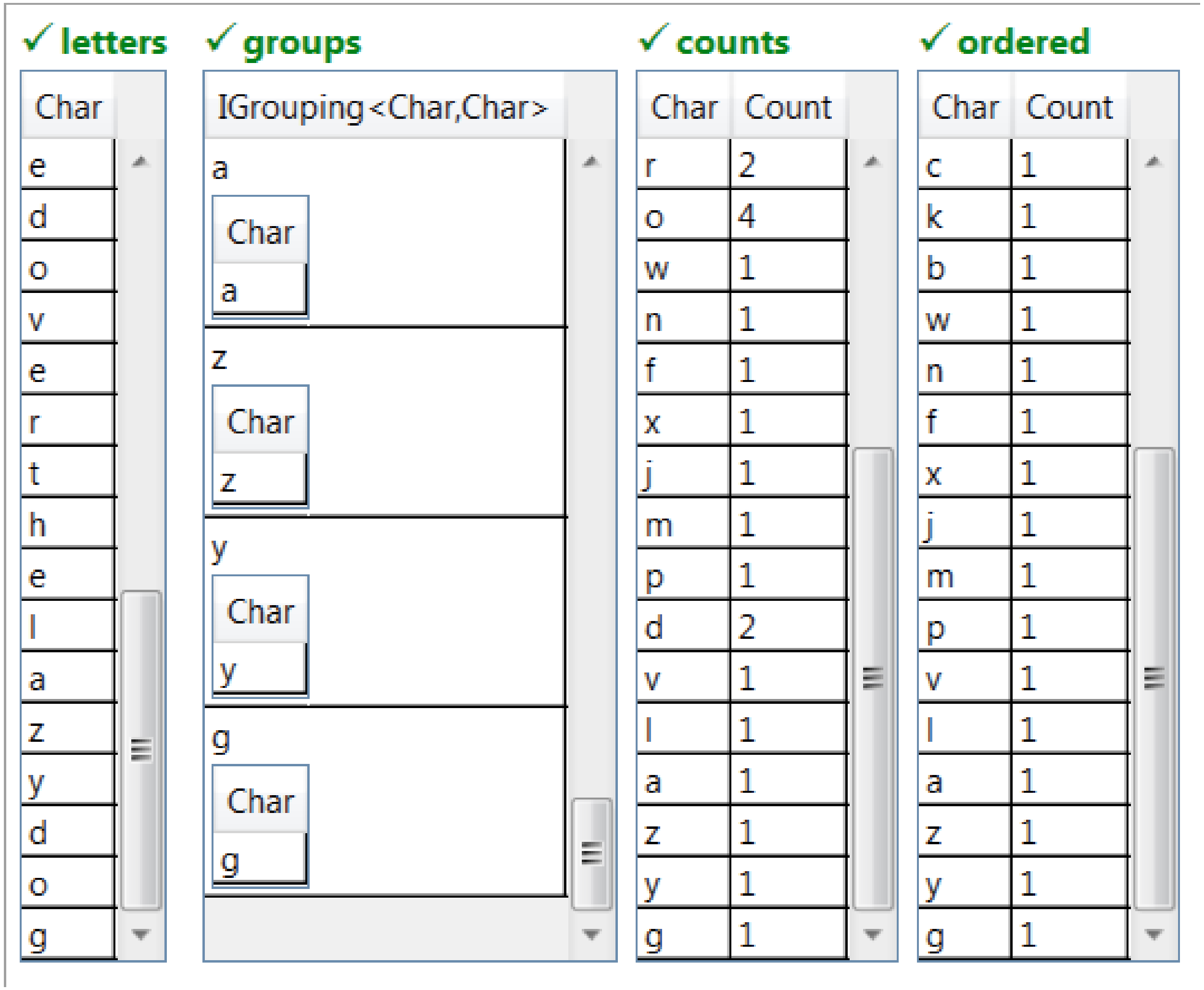
MapReduce



```
var input = "the quick brown fox jumped over the lazy dog";

var letters = input.AsParallel().Where(c => !char.IsWhiteSpace(c))
...
var groups = letters.GroupBy(c => c)
...
var counts = groups.Select(g => new { Char = g.Key, Count = g.Count() })
...
var ordered = counts.OrderByDescending(c => c.Count)
...
```

MapReduce is a query engine with just one fixed query plan.
Pull-based (batch).
First-order (no nesting)
No iteration
Simple (primitive?) job scheduling.

MapReduce



✓ letters

Char
e
d
o
v
e
r
t
h
e
l
a
z
y
d
o
g

✓ groups

IGrouping<Char,Char>
a Char a
z Char z
y Char y
g Char g

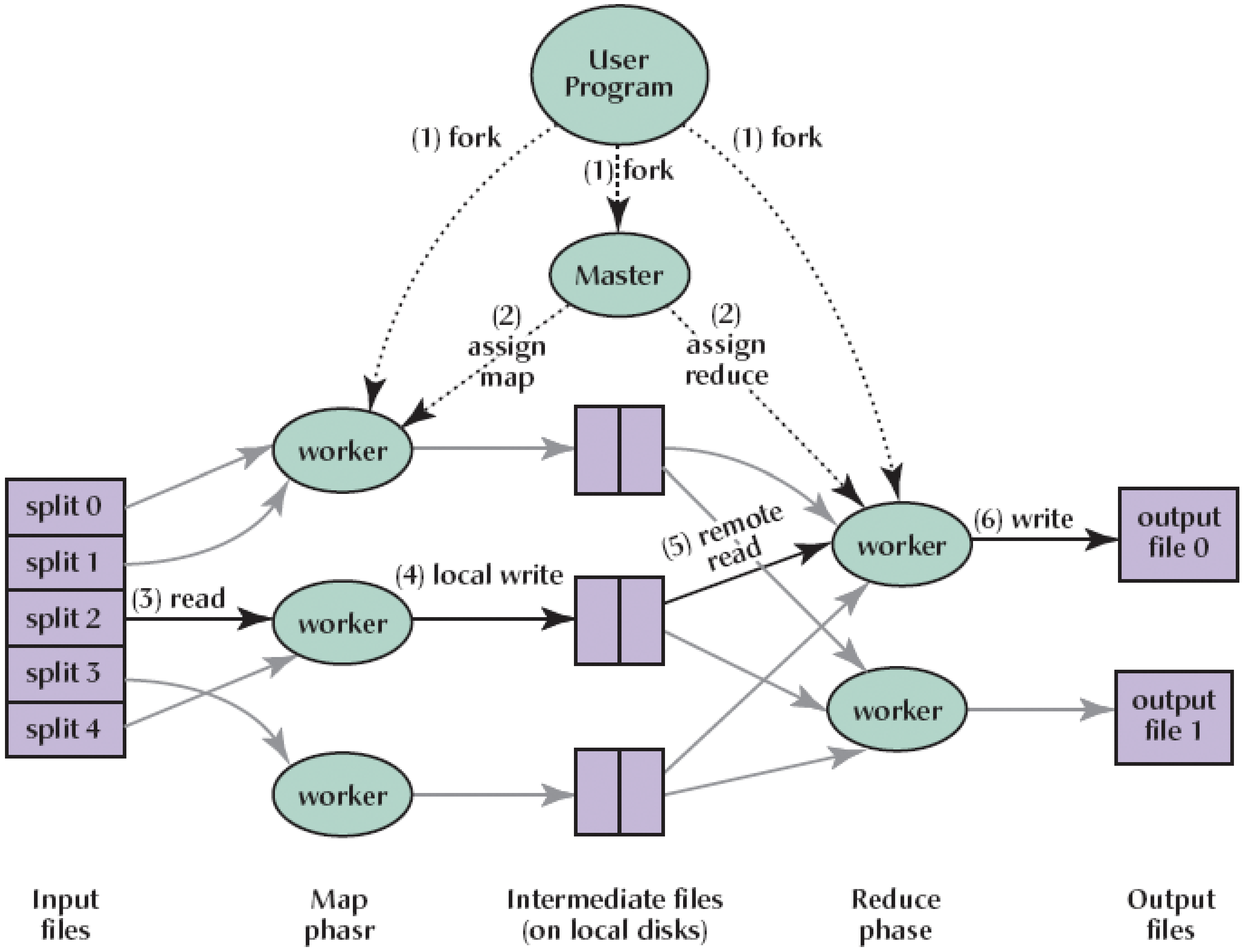
✓ counts

Char	Count
r	2
o	4
w	1
n	1
f	1
x	1
j	1
m	1
p	1
d	2
v	1
l	1
a	1
z	1
y	1
g	1

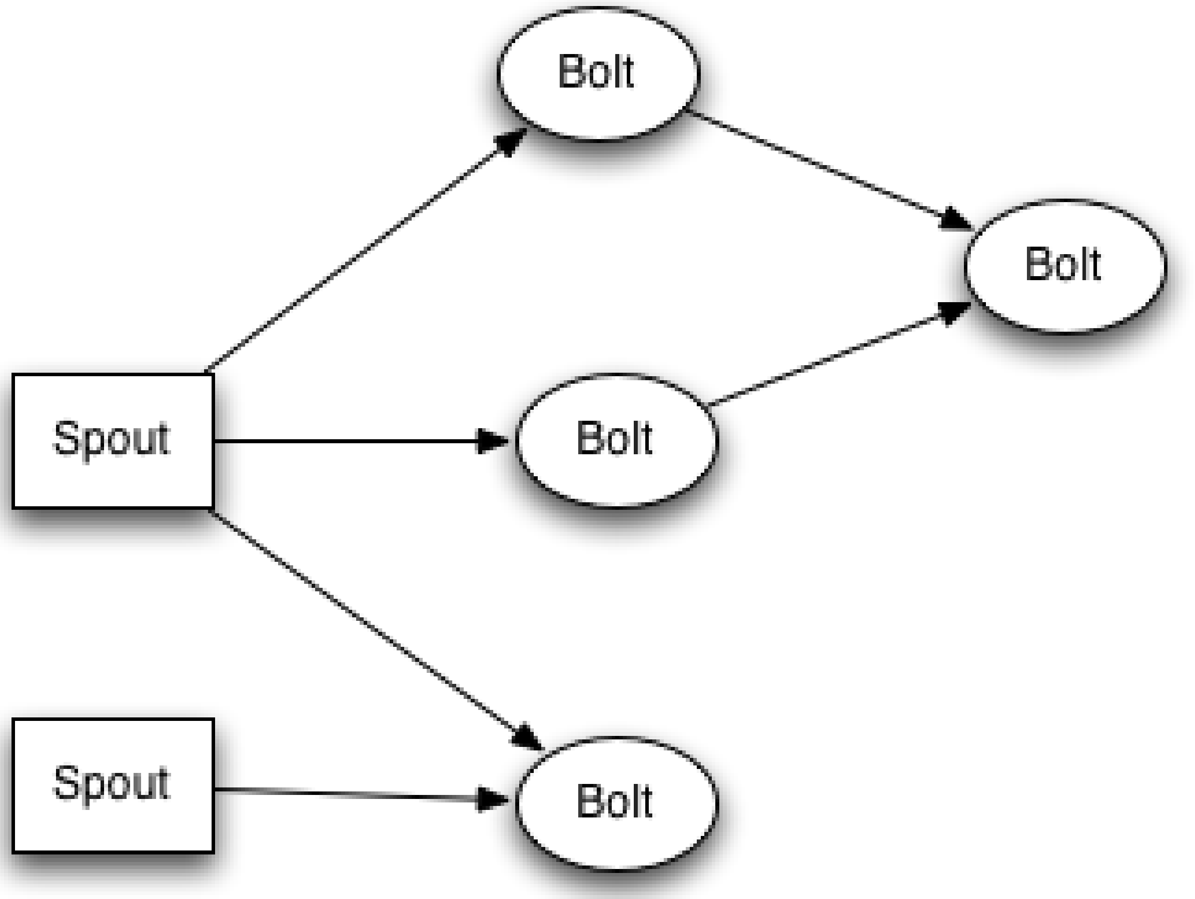
✓ ordered

Char	Count
c	1
k	1
b	1
w	1
n	1
f	1
x	1
j	1
m	1
p	1
v	1
l	1
a	1
z	1
y	1
g	1

Map Reduce


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

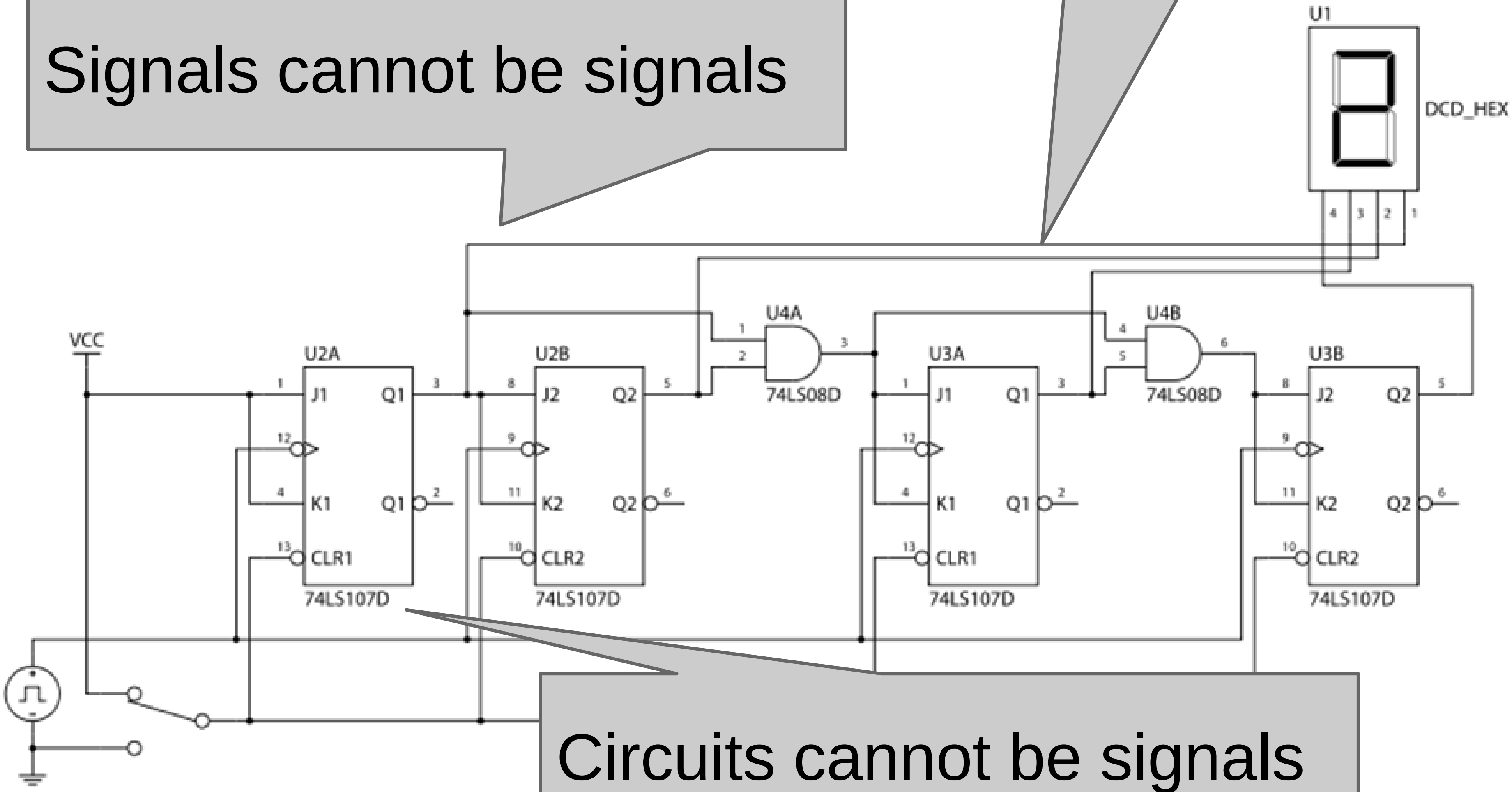
Storm



Dataflow

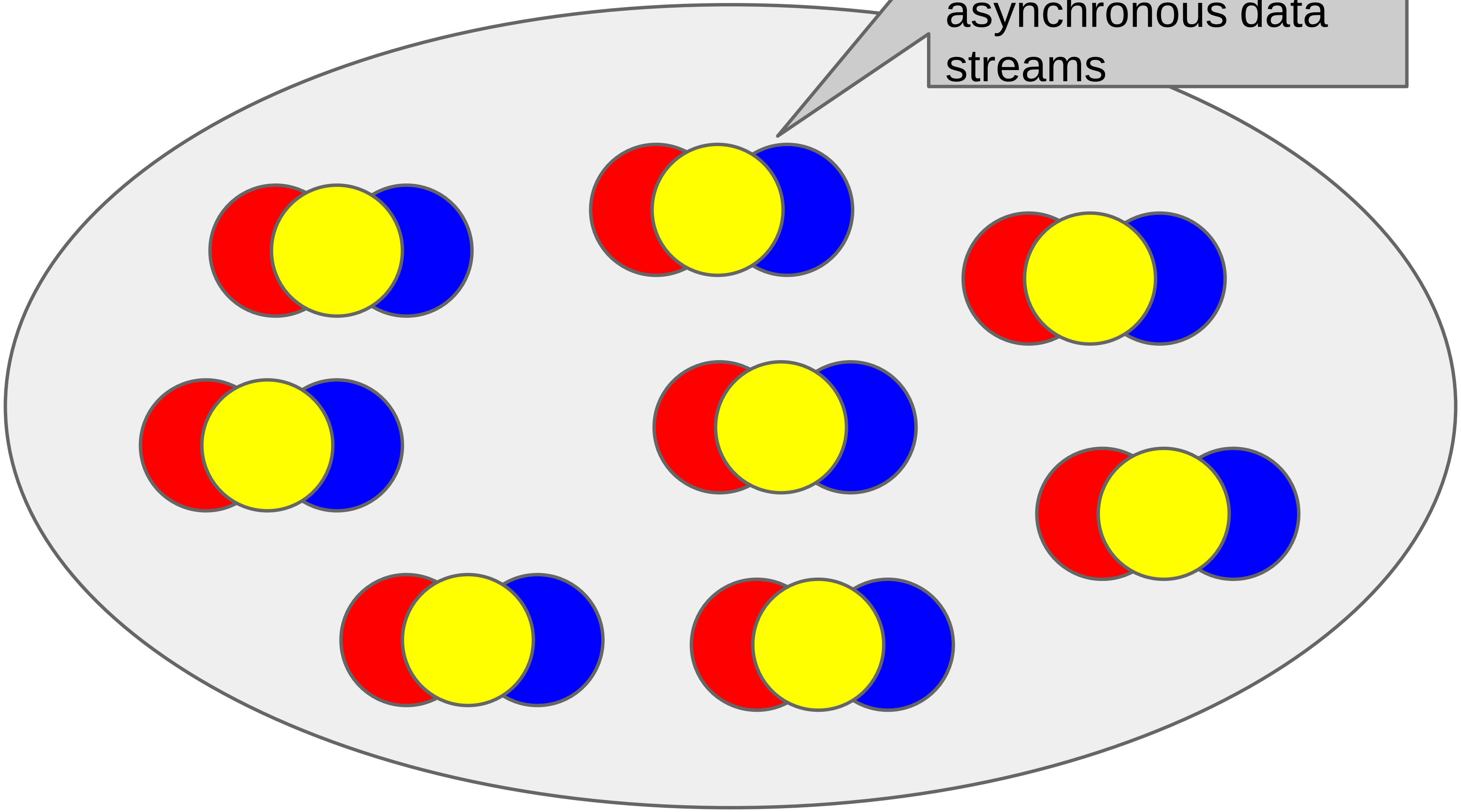
Topology is static

Signals cannot be signals



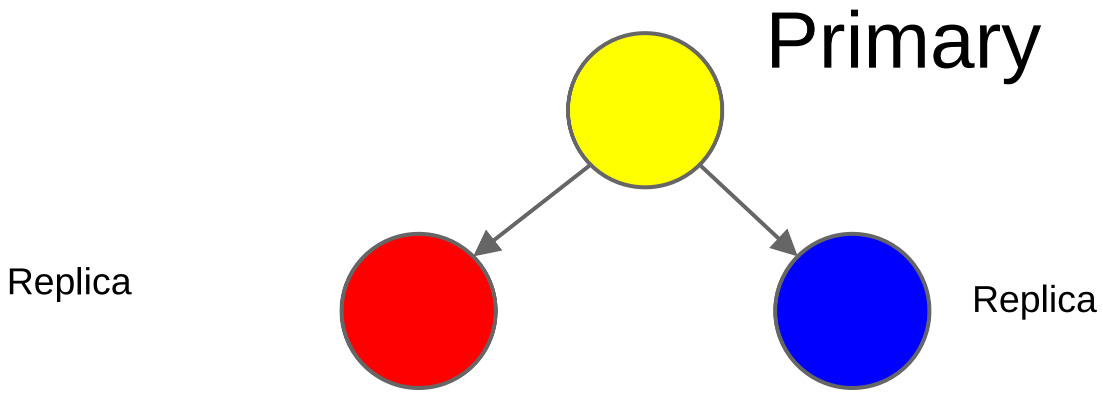
Circuits cannot be signals

ActorFx (Ax)



Highly available,
replicated,
stateful services
communicate via
asynchronous data
streams

ActorFx

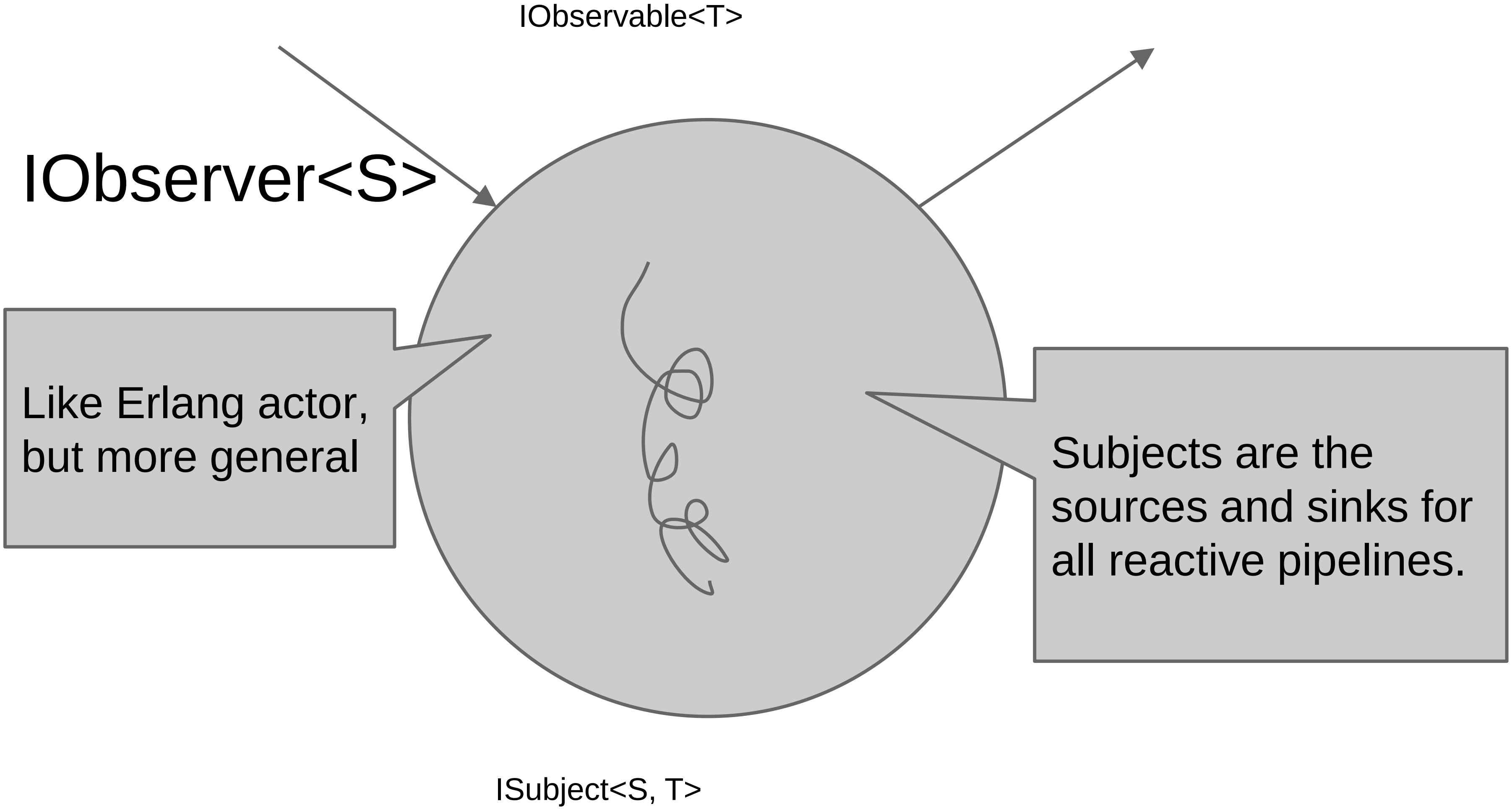


Intercept all state changes via virtual this pointer.

Replicate all mutations to replicas (using your favorite distributed compute fabric).

Behavior is just a special instance of state (monkey patching).

ActorFx for building Subjects



IObservable<T>

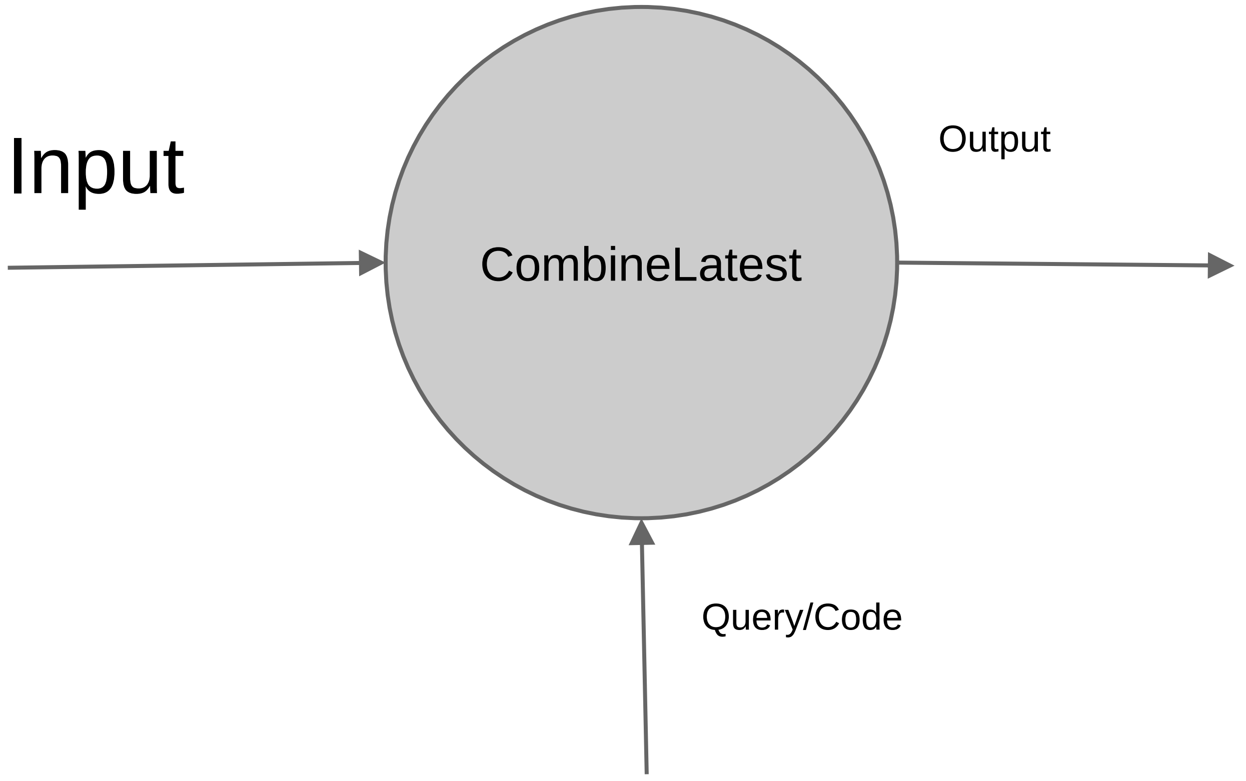
IObserver<S>

Like Erlang actor,
but more general

Subjects are the
sources and sinks for
all reactive pipelines.

ISubject<S, T>

CombineLatest for Deployment



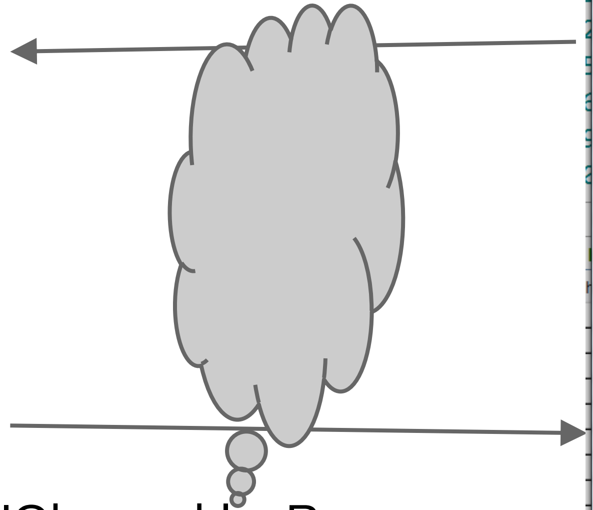
Input

CombineLatest

Output

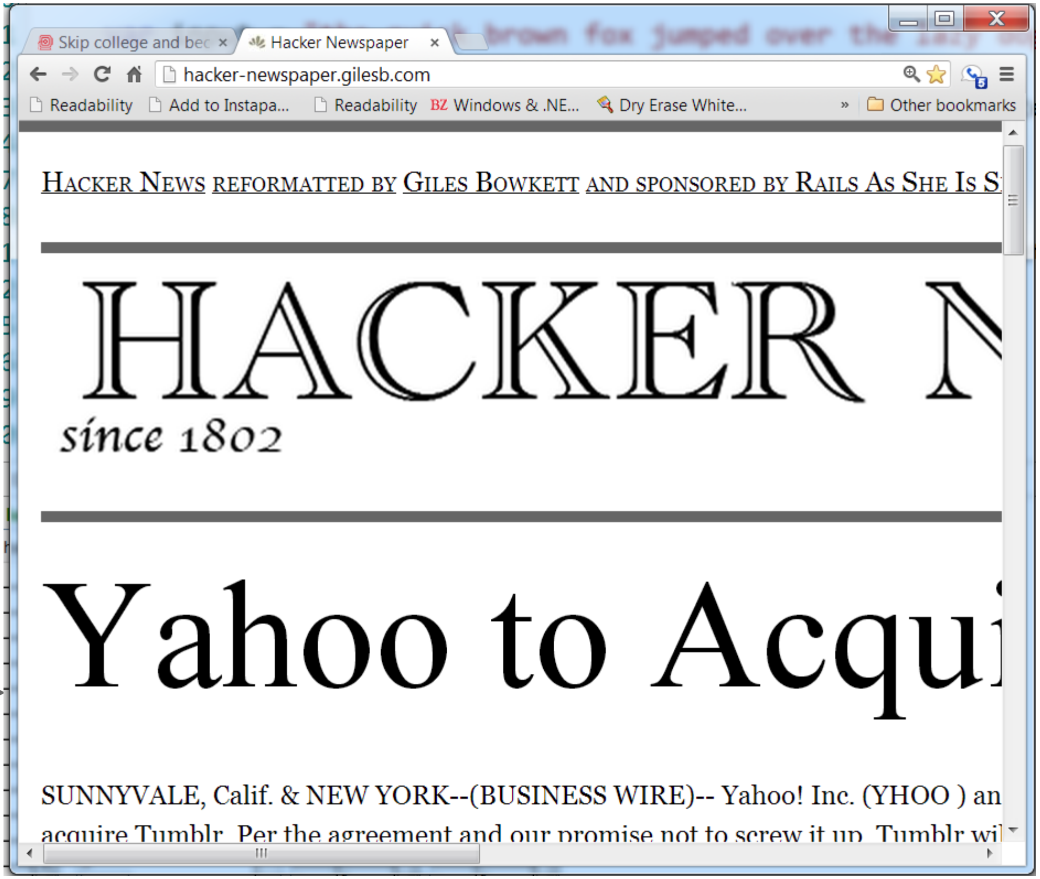
Query/Code

Communicating Concurrent Subjects



IObserver<Request>

IObservable<Response>



Hacker News reformatted by Giles Bowkett and sponsored by Rails As She Is S

HACKER N

since 1802

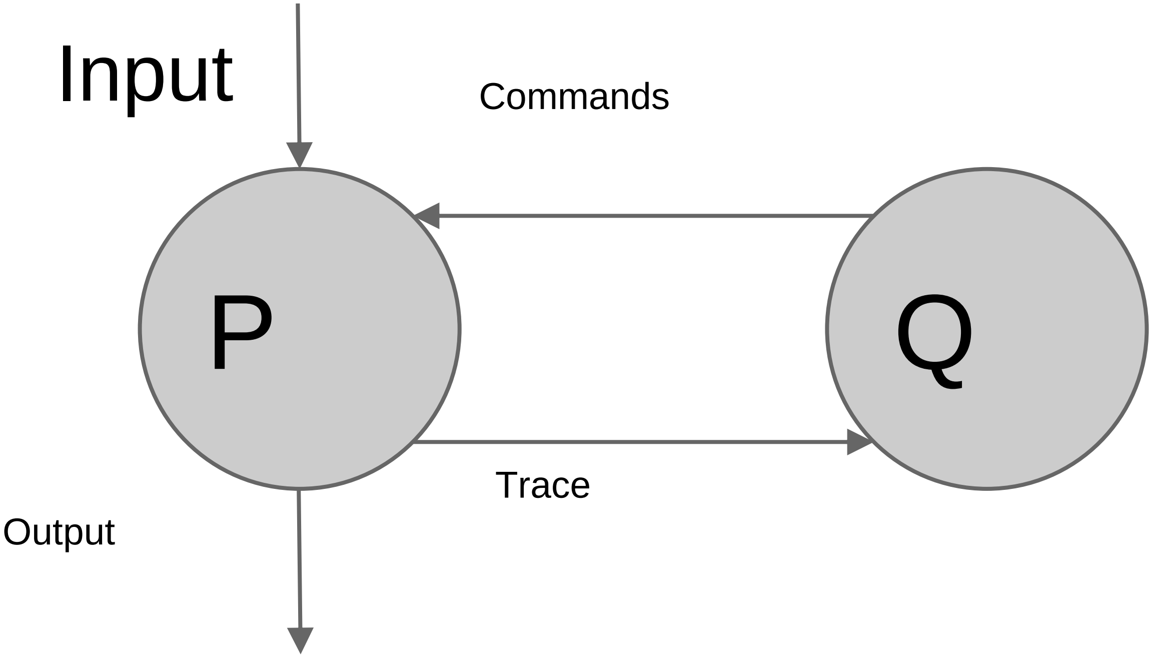
Yahoo to Acqui

SUNNYVALE, Calif. & NEW YORK--(BUSINESS WIRE)-- Yahoo! Inc. (YHOO) an acquire Tumblr. Per the agreement and our promise not to screw it up, Tumblr wil

Model

View

RealTime Monitoring of Queries Uisng Queries



Input

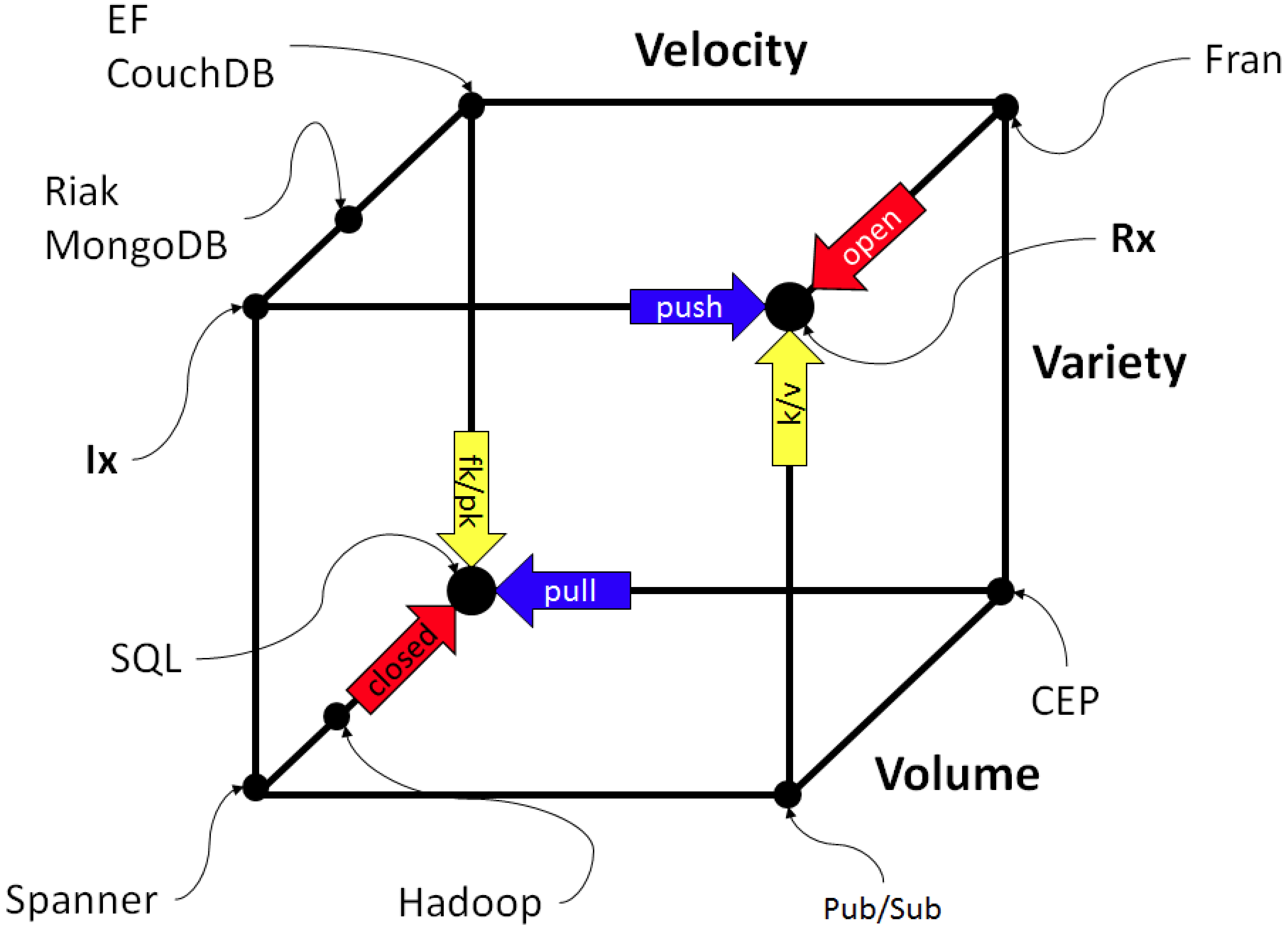
Commands

P

Q

Trace

Output


EF
CouchDB
Velocity
Fran
Riak
MongoDB
open
Rx
push
Variety
k/v
Ix
fk/pk
pull
SQL
closed
CEP
Volume
Spanner
Hadoop
Pub/Sub