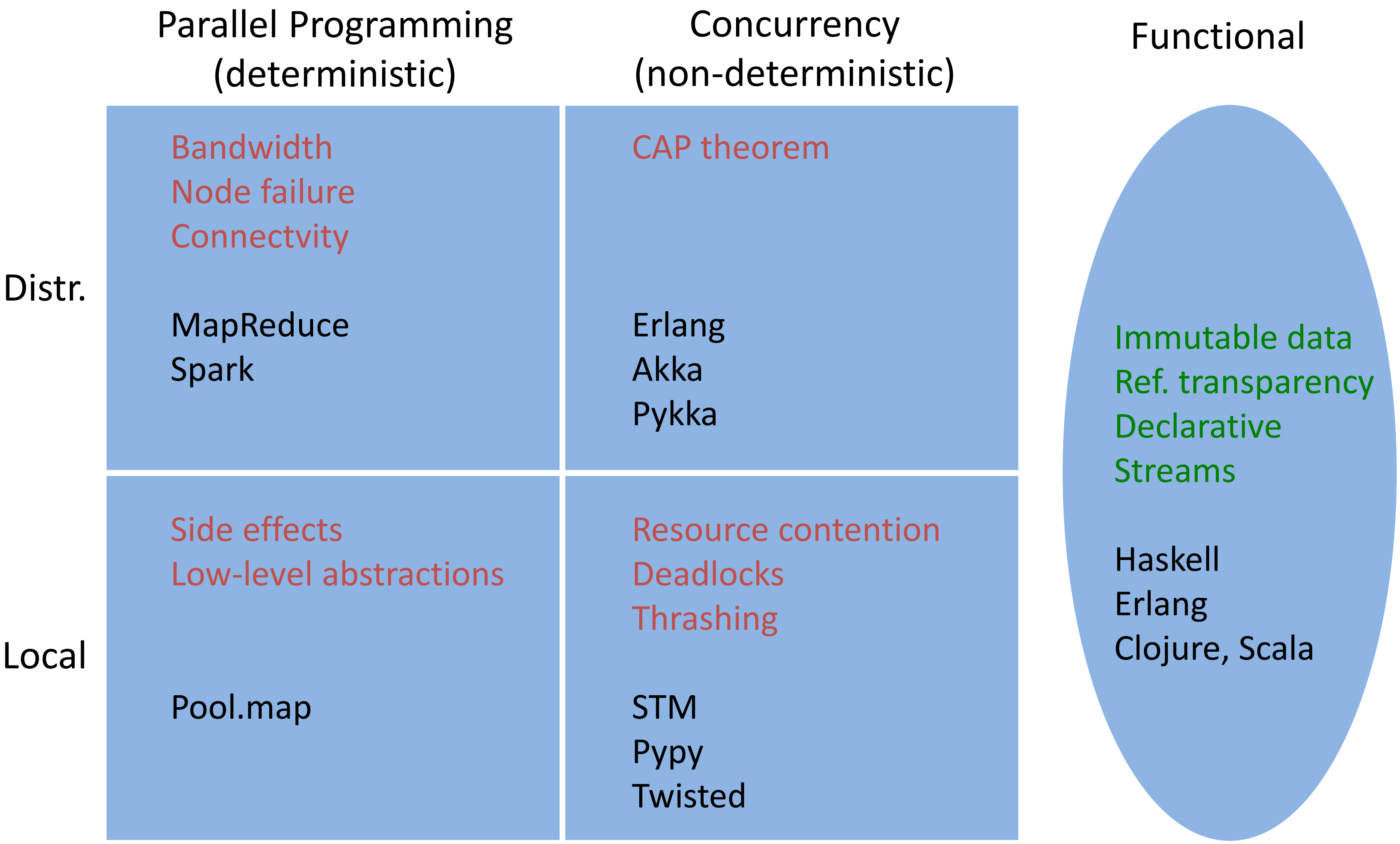
PySpark Distributed Computing

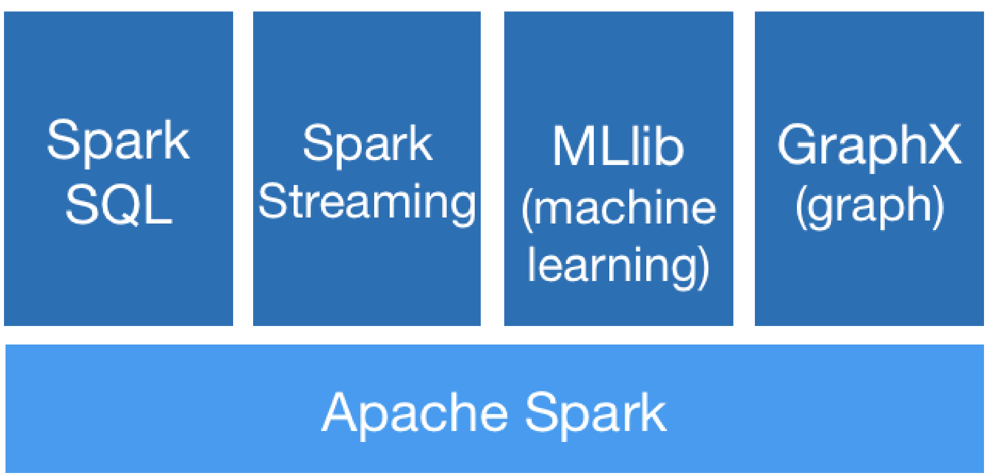
Leveraging the Functional Model

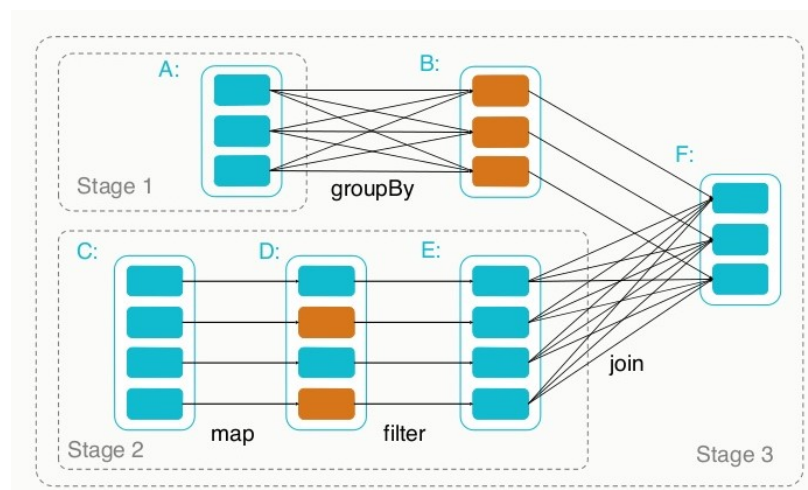
Johannes Ahlmann, PyCon Ireland, 2015-10-24


Parallel Programming (deterministic)
Concurrency (non-deterministic)
Functional
Distr.
Local
Bandwidth
Node failure
Connectvity
MapReduce
Spark
CAP theorem
Erlang
Akka
Pykka
Side effects
Low-level abstractions
Pool.map
Resource contention
Deadlocks
Thrashing
STM
Pypy
Twisted
Immutable data
Ref. transparency
Declarative
Streams
Haskell
Erlang
Clojure, Scala

What is Apache Spark

- Fast and general engine for large-scale data processing
- Multi-stage in-memory primitives
- Supports Iterative Algorithms
- High-Level Abstractions
- Extensible; integrated stack of libraries



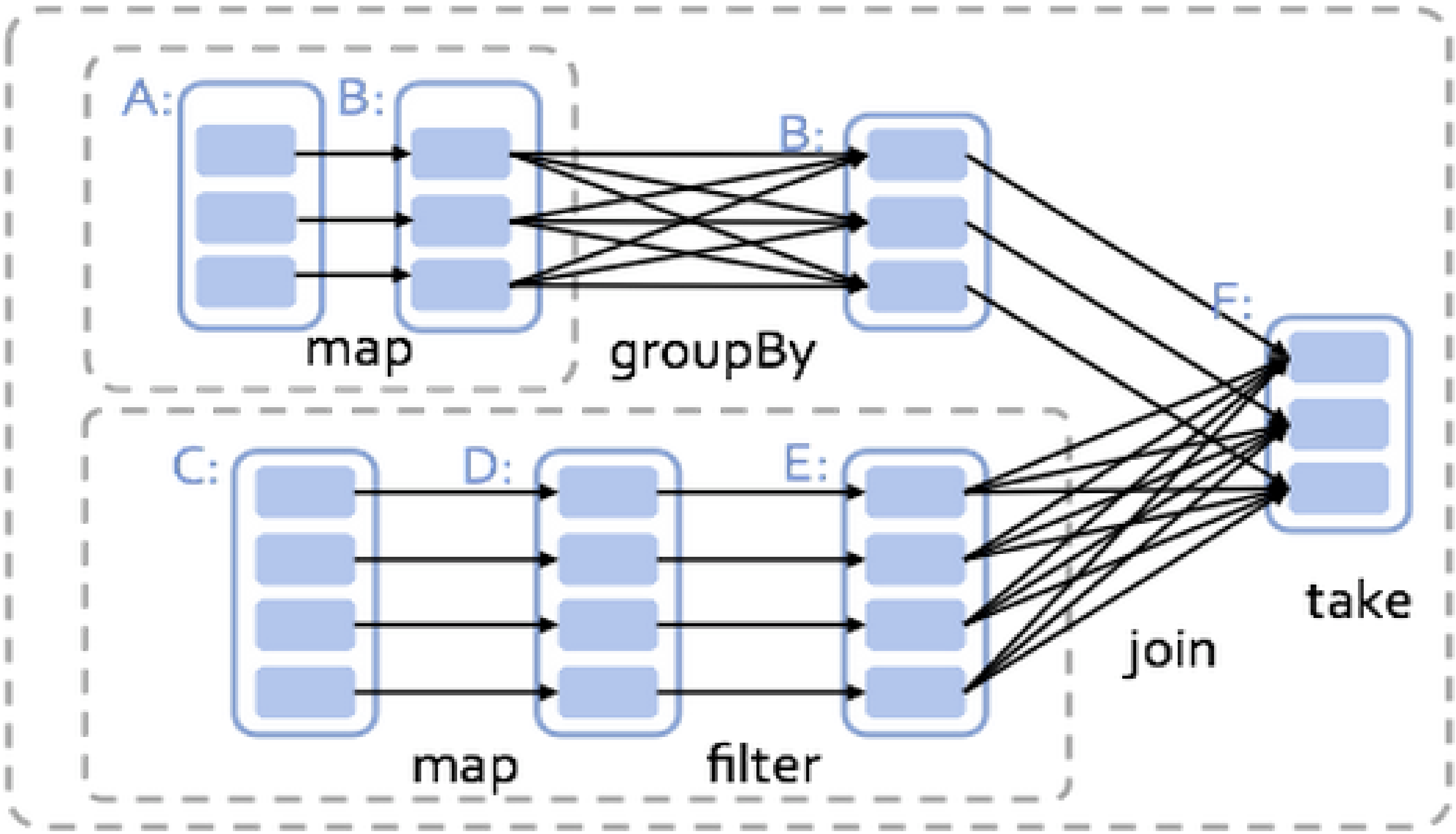


Spark Example

```
from pyspark import SparkContext
sc = SparkContext("local", "Log Analyzer")

rdd1 = sc.textFile('testfile1.txt')
rdd2 = sc.textFile('testfile2.txt')


def splitLine(line):
    key,val = line.split('|')
    return (key, int(val))

a = rdd1.filter(lambda line: not "ERROR" in line).map(splitLine)
b = rdd2.map(splitLine).reduceByKey(lambda a,b: a+b)

res = a.join(b)
#print res.toDebugString()
res.take(10)
```

```
[(u'key3', (50, 30)), (u'key1', (20, 60))]
```

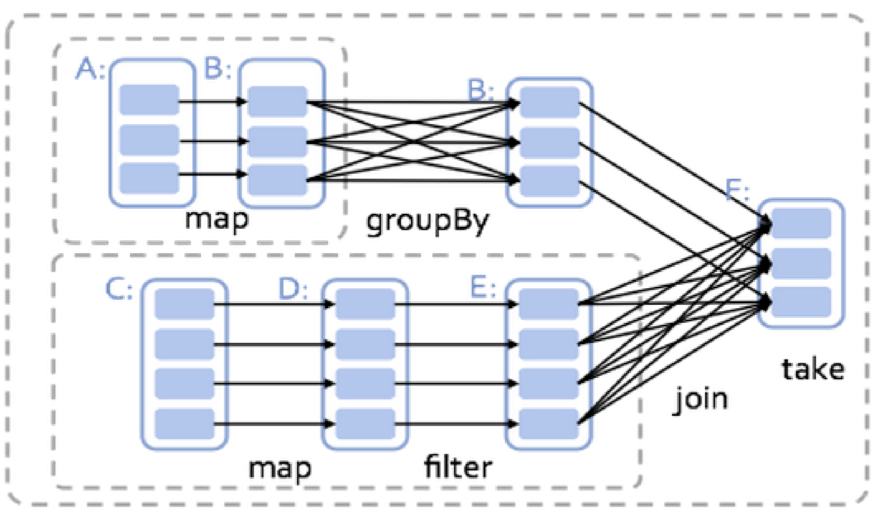
Operator Graph



vs.

MapReduce

- Arbitrary operator graph
- Lazy eval of lineage graph => optimization
- Off-heap use of large memory
- Native integration with python



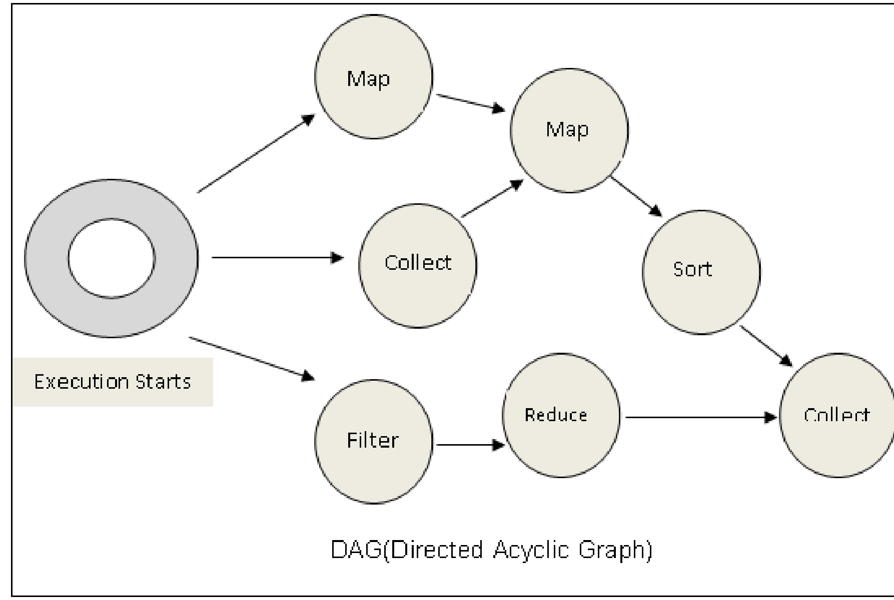
RDD

- Resilient Distributed Datasets are primary abstraction in Spark
- fault-tolerant collection
  - parallelized collections
  - hadoop datasets
- can be cached for reuse
- extensions (SchemaRDD)

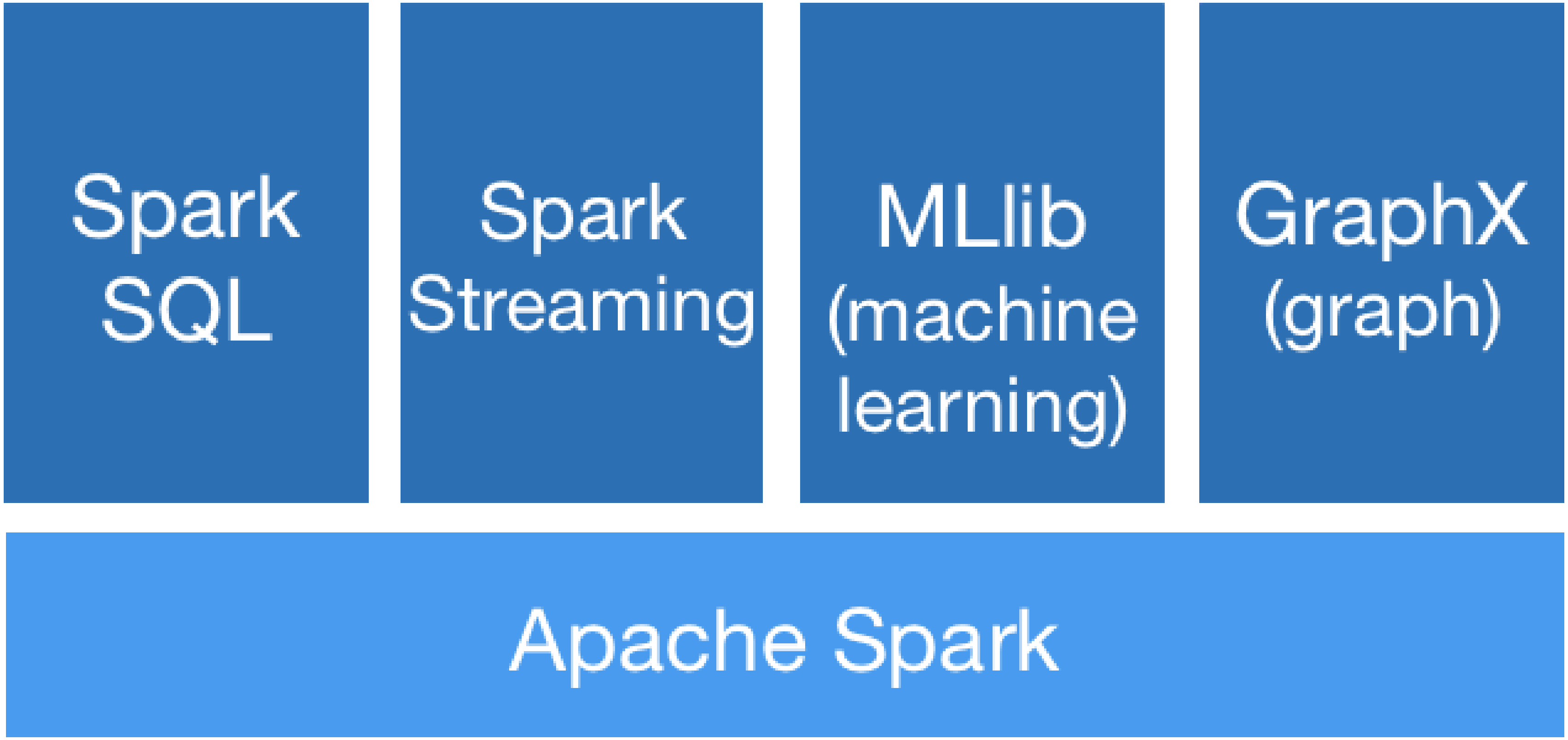
Transformations	**Actions**
map()	reduce()
filter()	collect()
flatMap()	count()
mapPartitions()	take()
sample()	takeSample()
union()	takeOrdered()
intersection()	saveAsTextFile()
distinct()	saveAsSequenceFile()
groupByKey()	countByKey()
reduceByKey()	foreach()
aggregateByKey()	
sortByKey()	
join()	
cogroup()	
cartesian()	
coalesce()	
repartition()	

Lifetime of an RDD

1. create from data
   - local collection
   - hadoop data set
2. lazily combine RDDs using transformations
   - map()
   - join()
   - etc.
3. call an RDD 'action' on it (collect(), count(), etc.) to "collapse" tree:
   1. Operator DAG is constructed
   2. Split into stages of tasks
   3. Launch tasks via cluster manager
   4. Execute tasks on local machines
4. store/consume results



Integrated Libraries



Takeaways

Spark...

- feels like native python, very nice API
- adds awesome Distributed Computing and Parallel Programming capabilities to python
- comes with batteries included (SQL, GraphX, MLLib, Streaming, etc.)
- can be used from the start for exploratory programming

Getting Started

- Download Spark; ./bin/pyspark
- docker-spark
- Spark on Amazon EMR
- Berkeley MOOC setup
  (vagrant, virtualbox, notebook)

Backups

Pure Functions

- f: a -> b
- Takes an “a” and returns a “b”

- Does not access global state and has no side-effects
- Function invocation can be substituted with the function body
- Can be used in an expression
- Can be “memoized”
- Is idempotent

Pure

- stateless
- no sequence, no time
- non-strict
- x = 1+4 (equality)
- “x” can be substituted by the expression (referential transparency)
- idempotent
- expressions, algebra

Effects

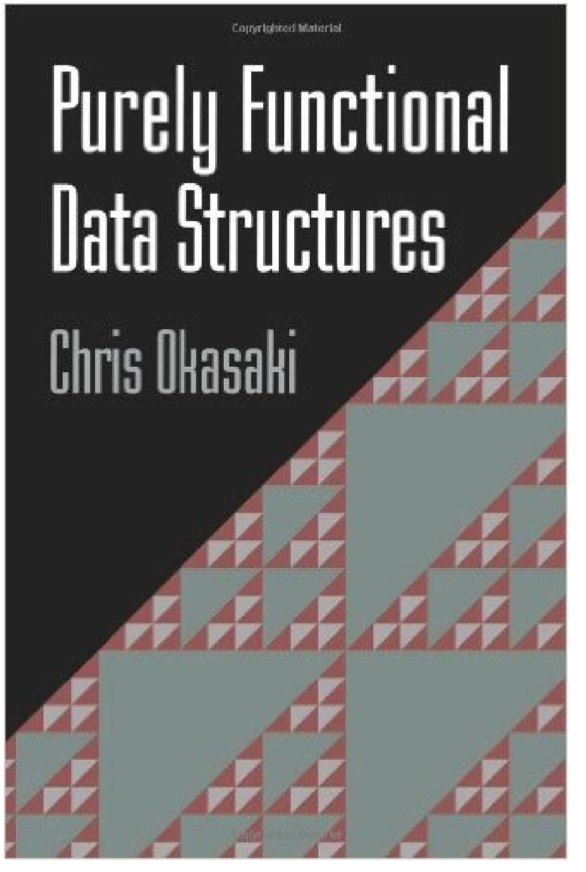
- stateful
- fixed sequence, time
- strict
- x := x + 1 (assignment)
- “x” = changeable memory “slot”

Pure functions by themselves are useless.
We want to interact with storage, network, screen etc.
We need both pure functions and (controlled, contained) effects

Immutable State

append([1, 2, 3], 4) => [1, 2, 3, 4]

- [1, 2, 3] remains unchanged
- Inherently thread-safe
- Can be shared freely
- “Everything is atomic”



Streams (Generators, Iterators)

Declarative

```
xs = [1, 2, 3];
return xs.map(x => x+1);
```

Imperative

```
xs = [1, 2, 3];
res = []
for (int i = 0; i < 3; i++) {
    res.append(xs[i] + 1);
}
return res;
```

Which do you think is easier to parallelize?

Stream Fusion

```
xs
.map(x => x+1)
.map(y => y*2)
```

```
xs
.map(x => (x+1)*2)
```

Iff functions are pure, we can

- combine
- reorder
- optimize the entire chain

If application is lazy, we can optimize across functions as well