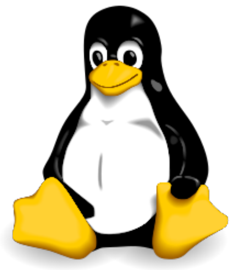
Scatter

Programming Support for an Integrated Multi-Party Computation and MapReduce Infrastructure

Scatter

Programming Support for an Integrated **Multi-Party Computation** and **MapReduce** Infrastructure



Scatter

Programming Support for an Integrated **Multi-Party Computation** and **MapReduce** Infrastructure

Scatter

Programming Support for an Integrated Multi-Party Computation and **MapReduce** Infrastructure

Scatter

Programming Support for an Integrated **Multi-Party Computation** and MapReduce Infrastructure

Scatter

Programming Support for an Integrated Multi-Party Computation and **MapReduce** Infrastructure

So what's MapReduce?

Programming paradigm to specify data analytics tasks.

Backend infrastructure as a highly-distributed, parallel execution environment for those tasks.

It’s a really big deal!

Programming paradigm to specify data analytics tasks.

Backend infrastructure as a highly-distributed execution environment for those tasks.

Largest Apache Spark cluster is **8000 nodes.**

200 node Spark cluster sorted **100TB** of data in **23 minutes.**

Word count in five lines

```
text_file = spark.textFile("hdfs://...")

counts = text_file.flatMap(lambda line: line.split(" ")) \

             .map(lambda word: (word, 1)) \

             .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://...")
```

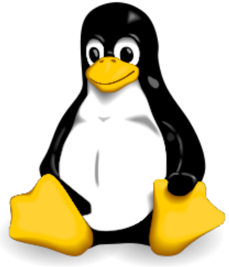
Programmer doesn't have to worry about

How the data is distributed across the cluster

Managing data operations performed by each machine

Fault tolerance

The distributed nature of the platform



MapReduce is a great example of separation of concerns.

Scatter

Programming Support for an Integrated **Multi-Party Computation** and MapReduce Infrastructure

What does multi-party computation give us?

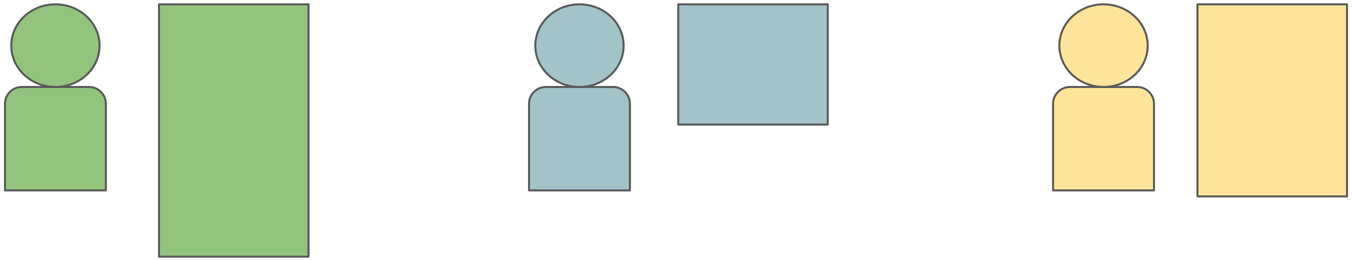
Given multiple parties $p_1, p_2, \ldots, p_n$ with private inputs $x_1, x_2, \ldots, x_n$

Need to compute $f(x_1, x_2, \ldots, x_n)$

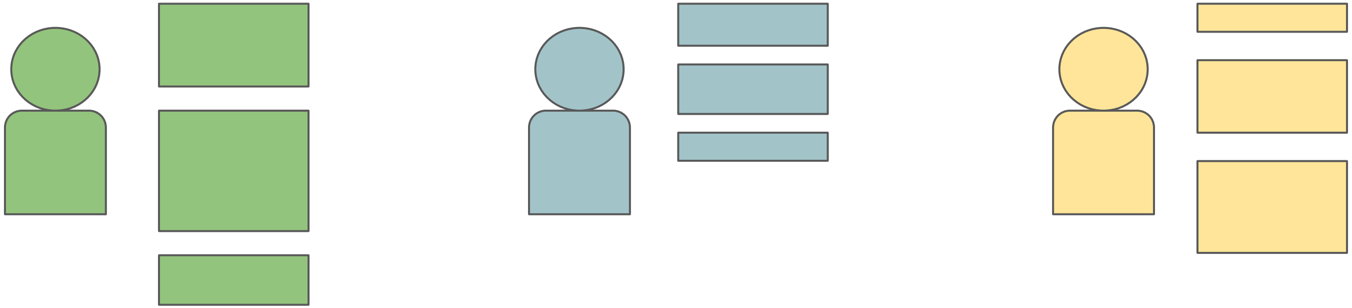
Without revealing more than the outputs of f

Sounds a bit like a magic trick...

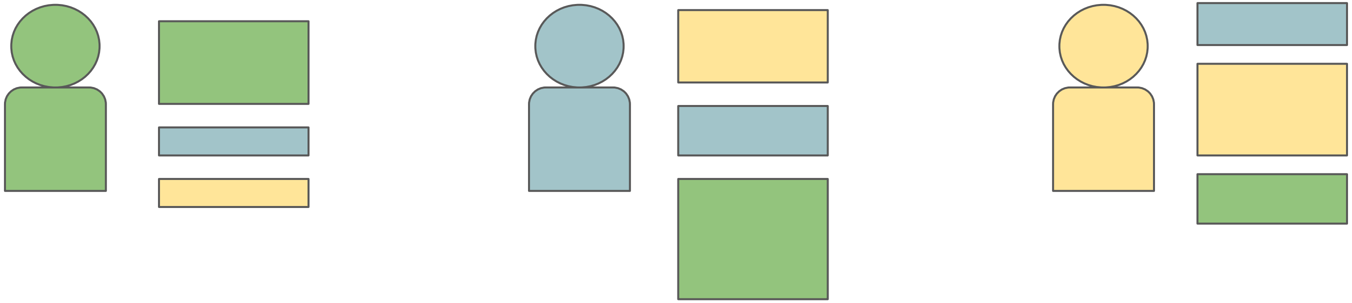
Quick example: the sum of secrets



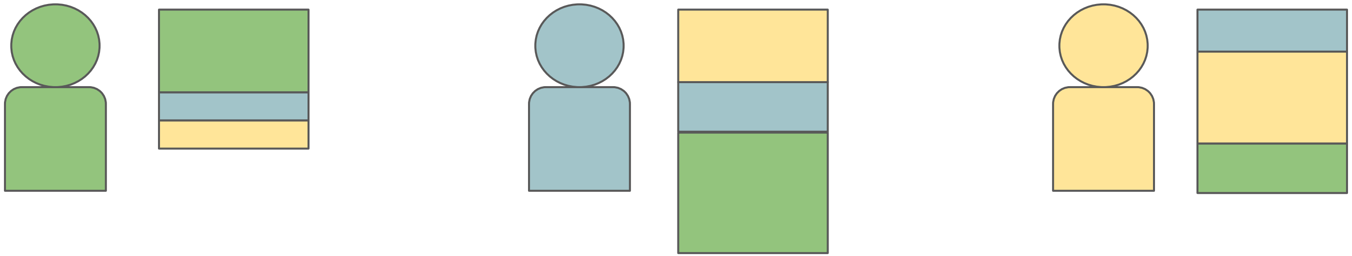
Split secrets into “shares”



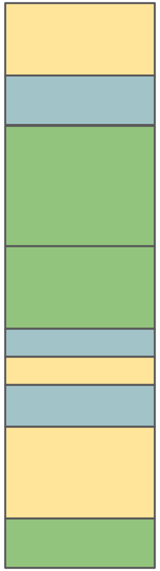
Distribute shares



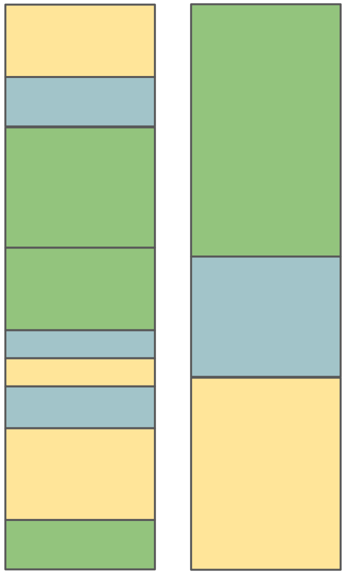
Add shares, (this results in more shares)

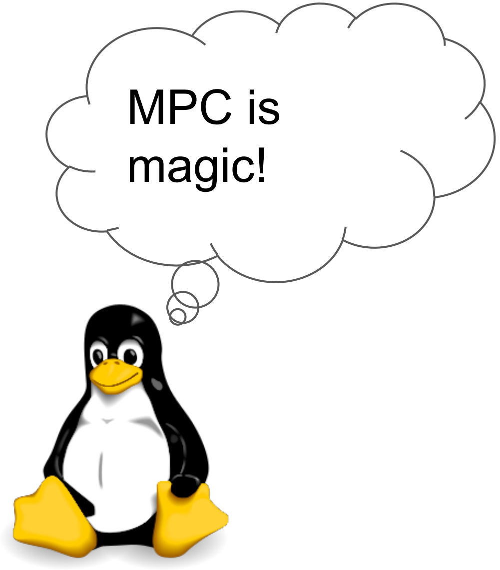


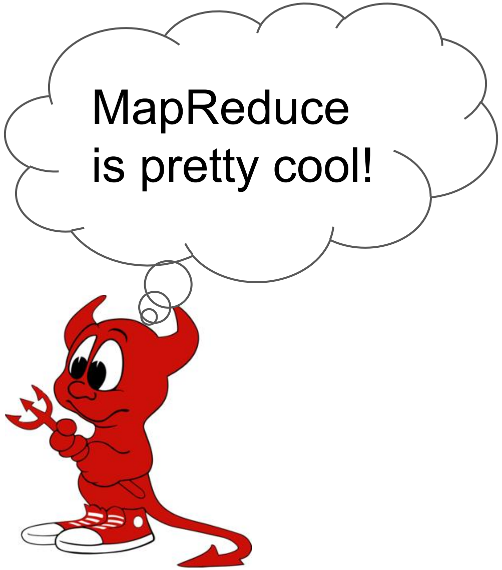
Recombine shares

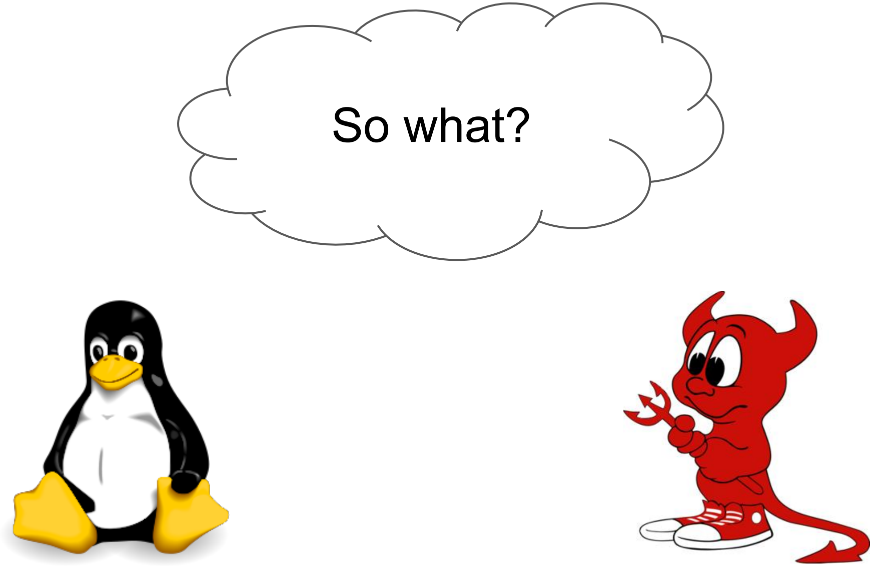


Lo and behold

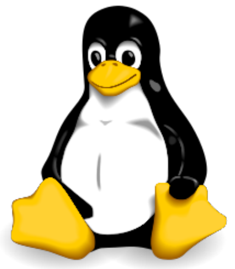



MPC is
magic!


MapReduce
is pretty cool!


So what?

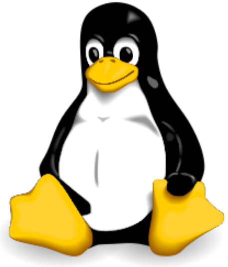
Let’s think about pay (in)equity for a moment



Each company can use **MapReduce** to find the salary differences in their own data.

The companies can use **MPC** to find the collective difference without revealing their data.

Let’s think about pay (in)equity for a moment



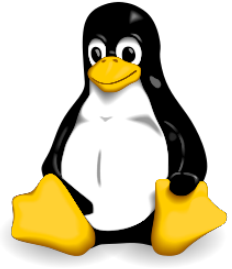
Each company can use **MapReduce** to find the salary differences in their own data.

→ Lots of computation

The companies can use **MPC** to find the collective difference without revealing their data.

→ Just one addition

Why bother splitting tasks up like that?

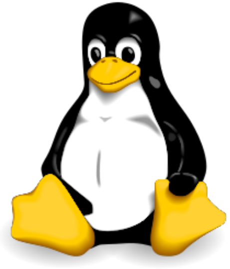


Performance!

Practical MPC frameworks are **slow**.

MPC frameworks optimize MPC, they don’t optimize local computation.

Usability



Practical MPC frameworks are **slow**.

MPC frameworks optimize MPC, they don’t optimize local computation.

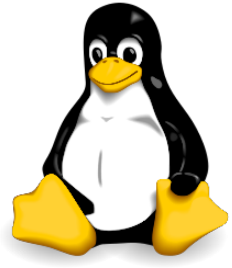
Data analysts don’t know about MPC (or think they do).

MPC frameworks require a steep learning curve (trust me...).

Direct disconnect between user expertise and available tools.

What about separation of concerns?

Let's put MapReduce and
MPC together!



Scatter

Programming Support for an Integrated Multi-Party Computation and MapReduce Infrastructure

The main components of Scatter

Programming language to specify MapReduce and MPC operations.

Compiler to convert Scatter programs to tasks that are executable in existing MapReduce and MPC frameworks.

Backend platform running those MapReduce and MPC frameworks to act as an execution environment for a compiled Scatter program.

Let's explore Scatter top-down

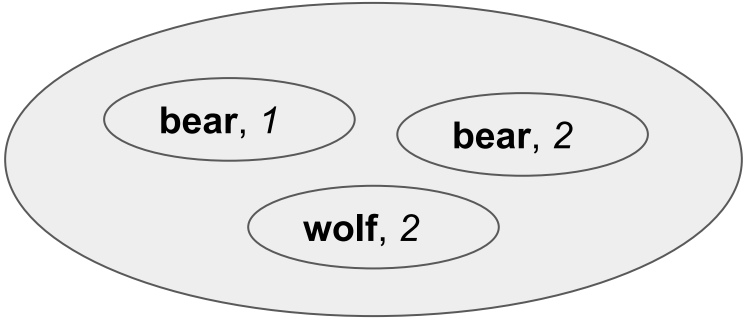
Programming language to specify MapReduce and MPC operations.

Compiler to convert Scatter programs to tasks that are executable in existing MapReduce and MPC frameworks.

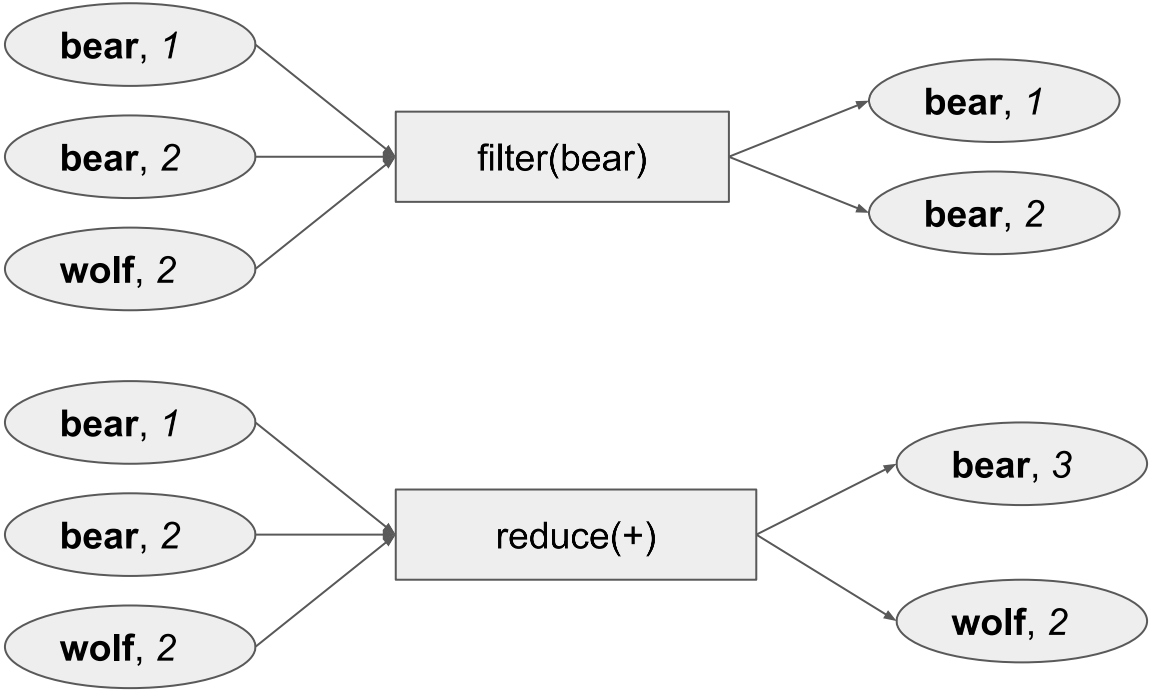
Backend platform running those MapReduce and MPC frameworks to act as an execution environment for a compiled Scatter program.

MapReduce primer coming up!

MapReduce is all about **key**-*value* stores



MR operations are functions on key-value stores



Pay equity in Scatter

```
m := reduce(+, filter("m", data))
f := reduce(+, filter("f", data))
d := m - f
```

Declaring the key-value store

```
type gender = str
type salary = int
data := store(gender, salary)

m := reduce(+, filter("m", data))
f := reduce(+, filter("f", data))
d := m - f
```

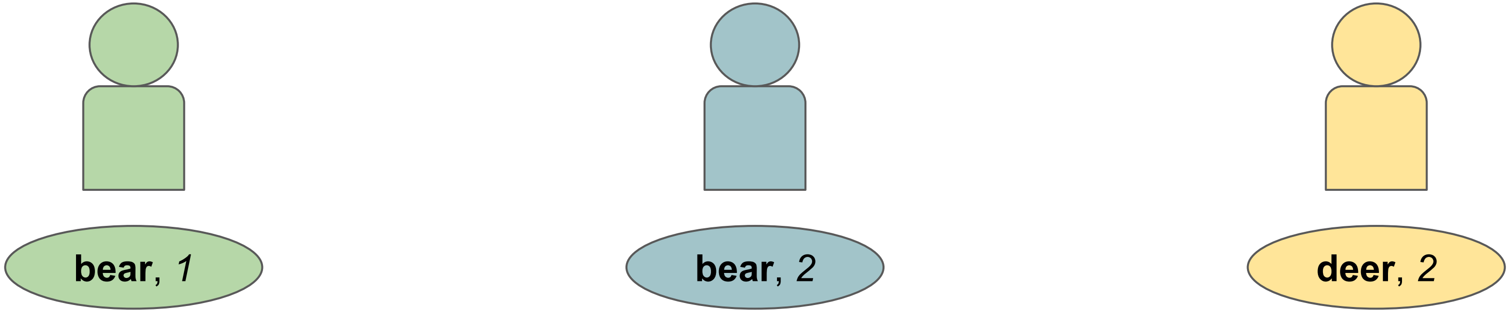
What about MPC?

What about MPC?

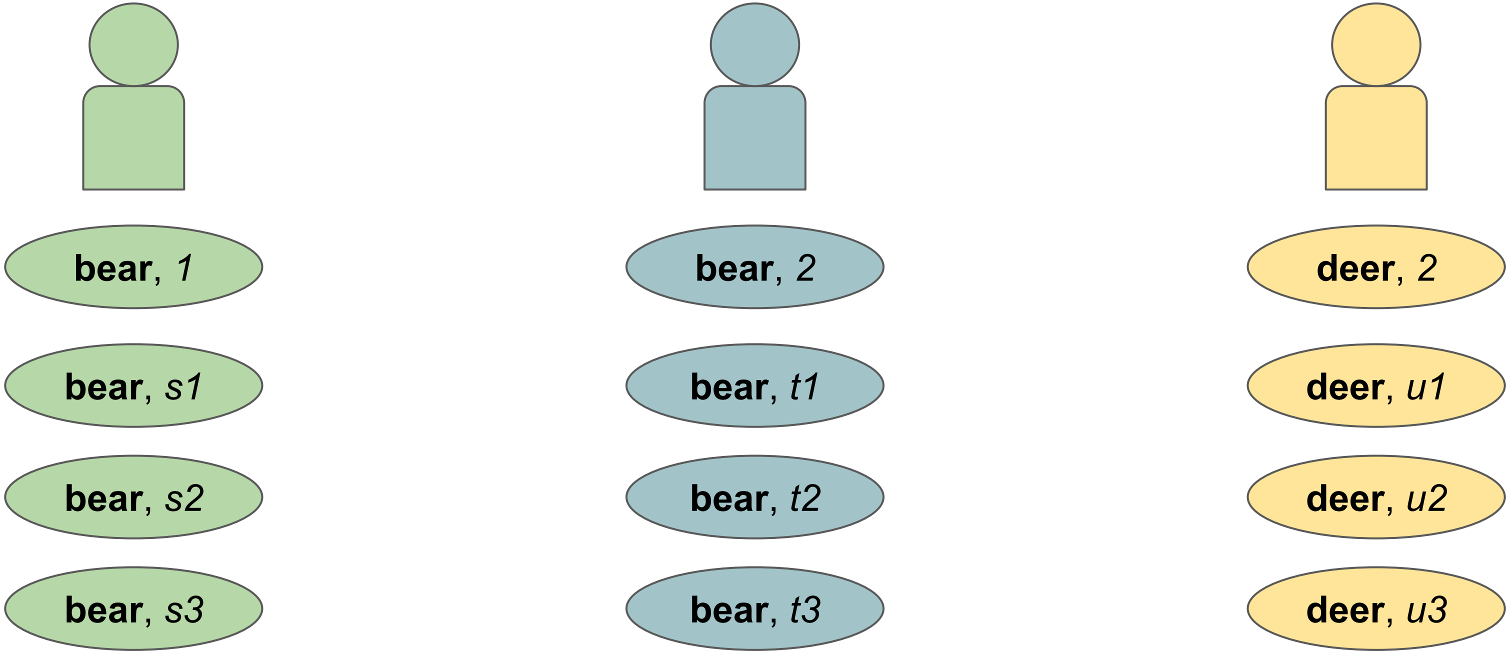
Two main constructs **Scatter**, and **Gather**.

(Finally, the mystery is lifted.)

Scatter: make secret and share

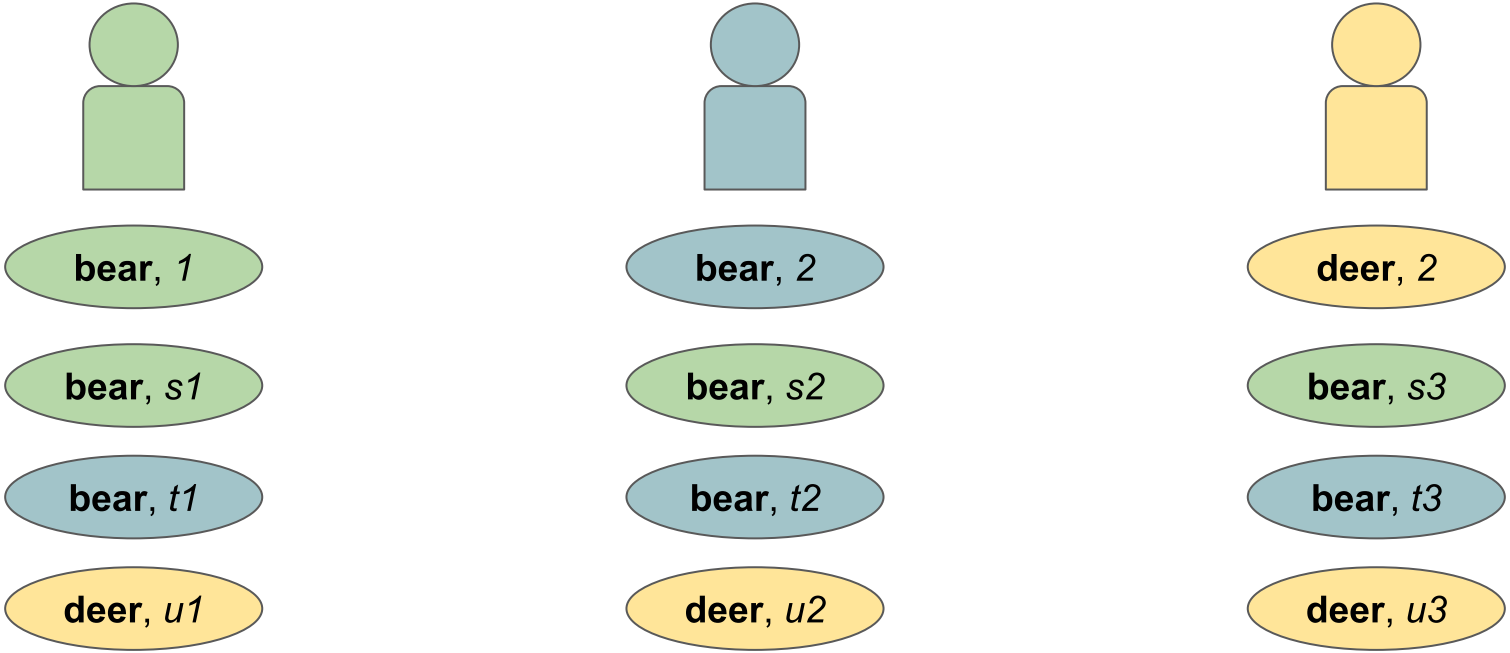


Split* values into shares

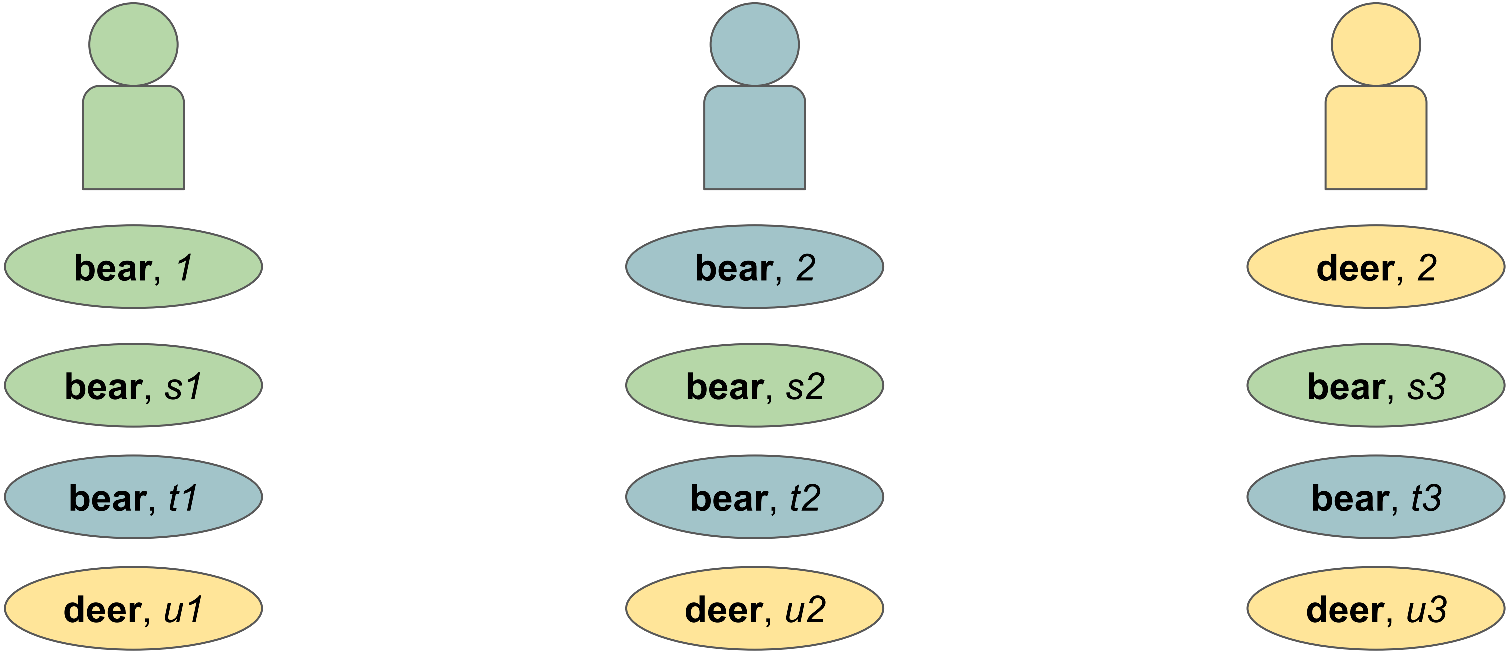


* using the MPC backend secret sharing implementation

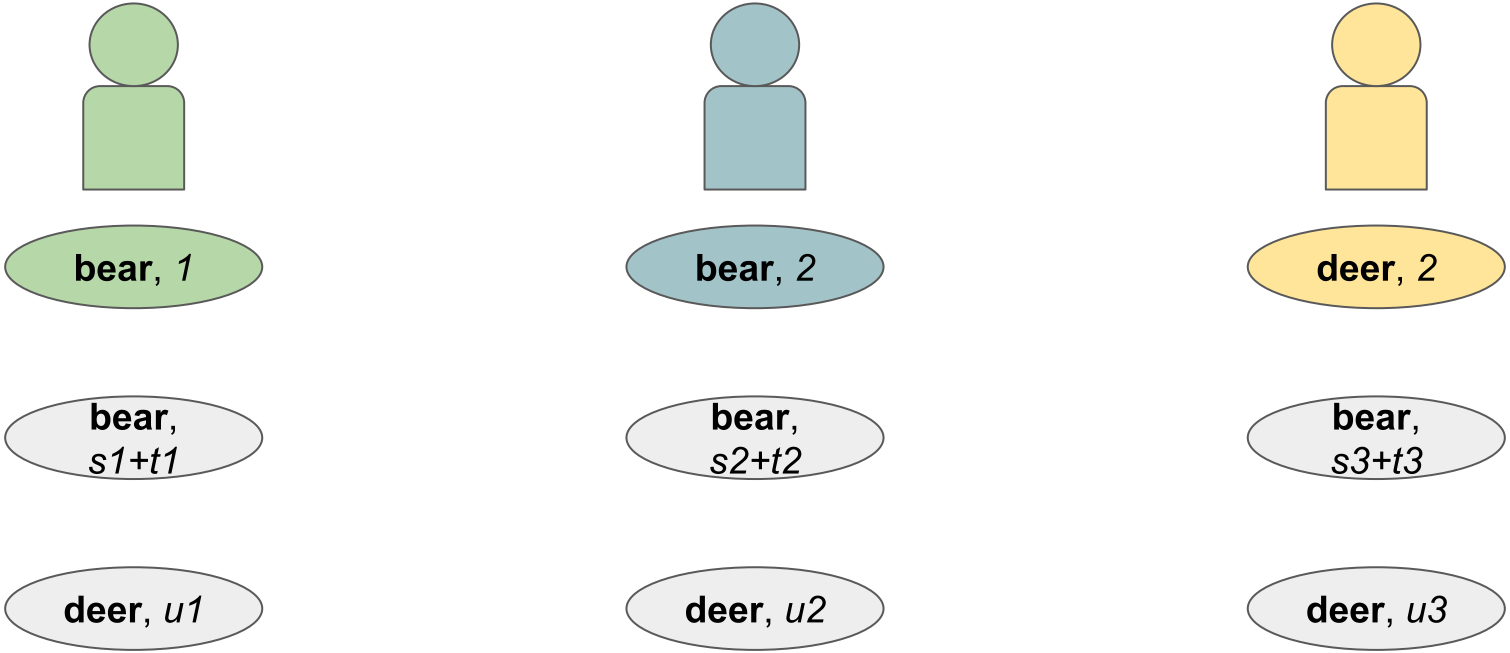
Send and receive shares



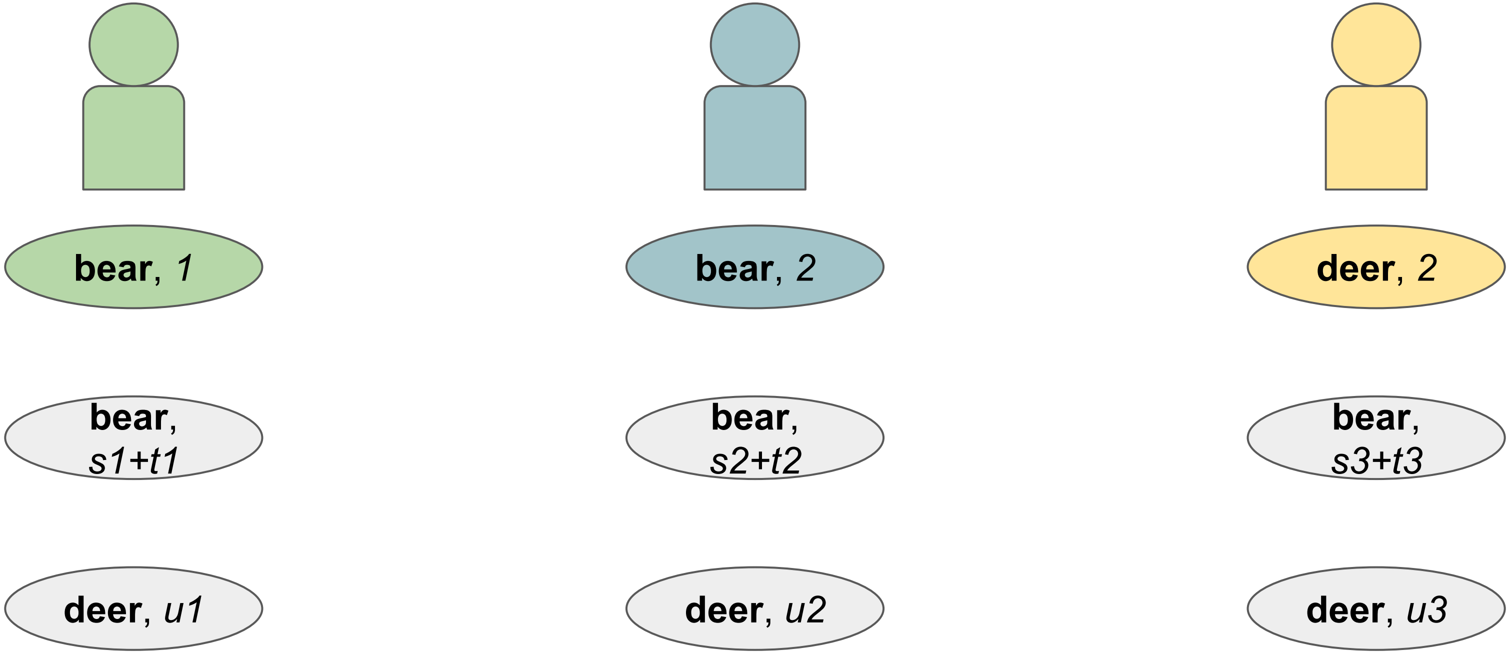
Now let's say we want to *reduce(+)*



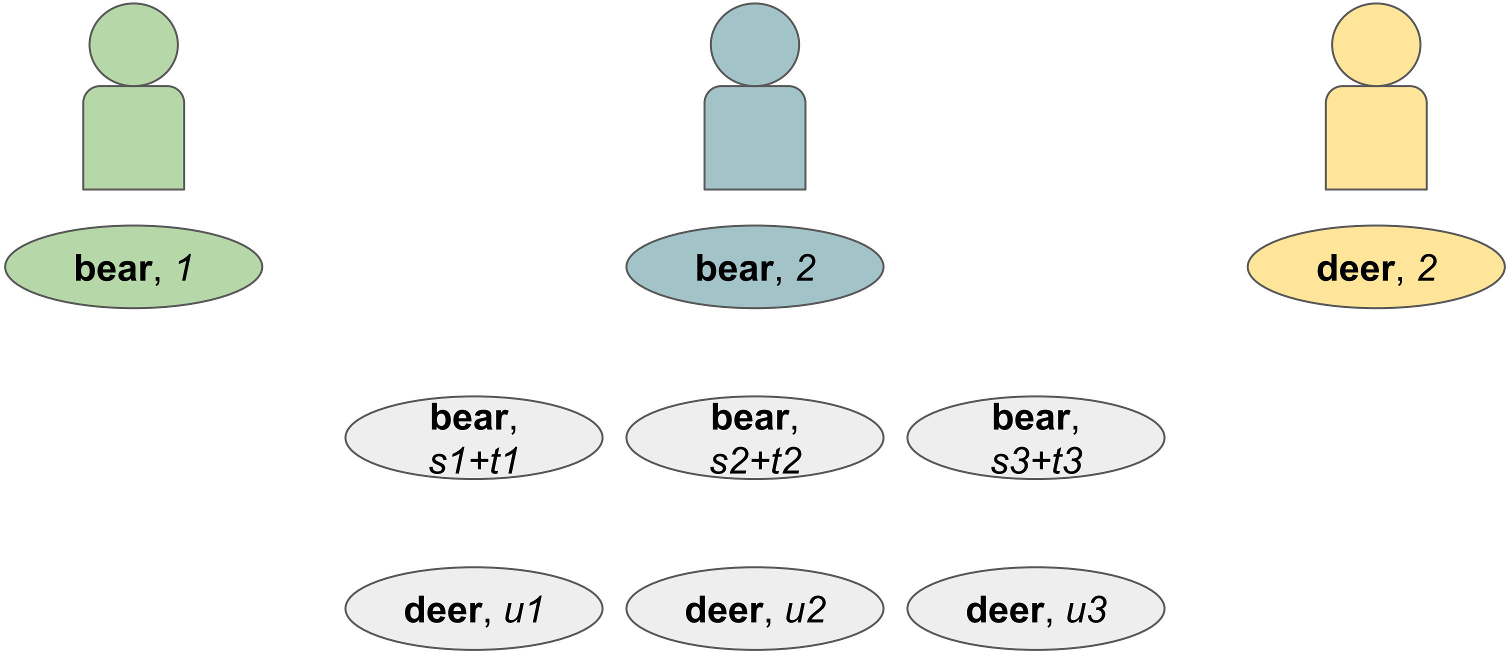
Now let's say we want to *reduce(+)*



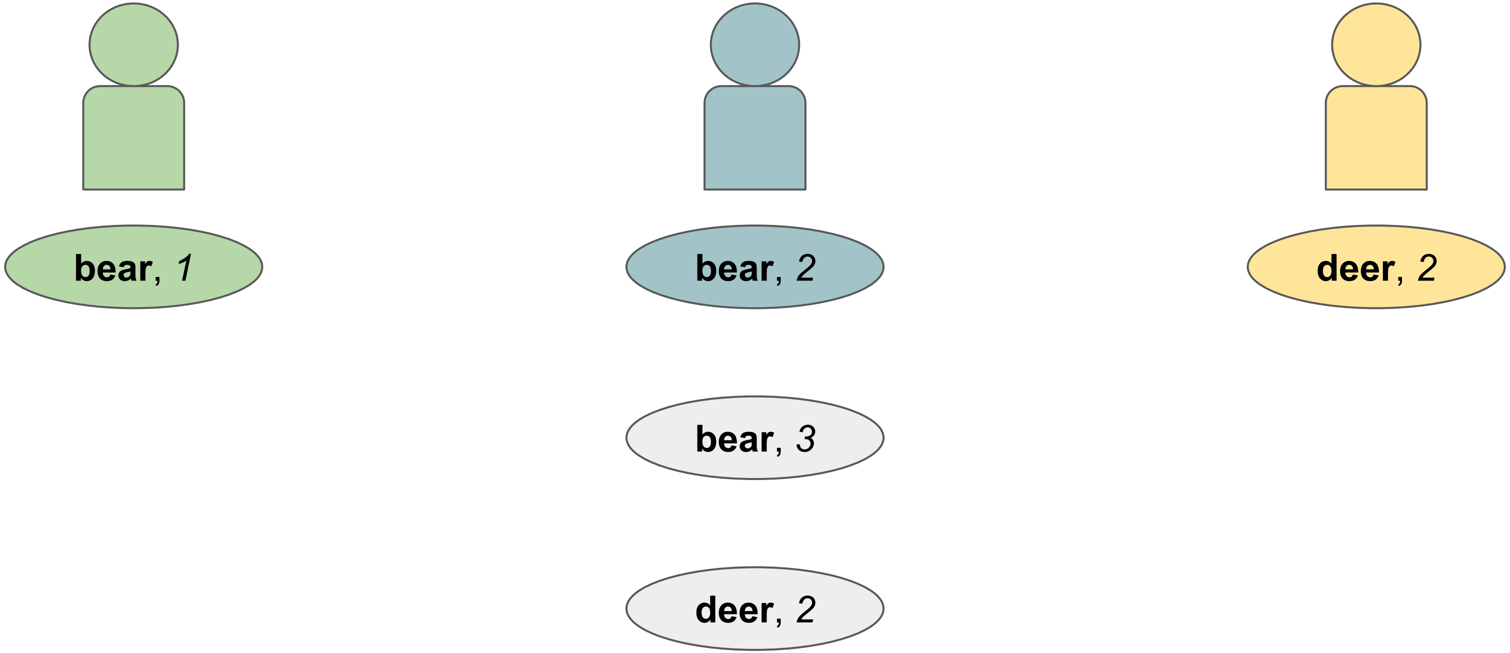
Gather: collect and reveal



Send shares to specified participant



Recombine shares



Complete pay equity Scatter program

```
type gender = str
type salary = int
data := store(gender, salary)

m := reduce(x, y, +, filter("m", data))
f := reduce(x, y, +, filter("f", data))
d := m - f

s := gather(reduce(lambda x,y: x+y, scatter(d)))
```

Each company will execute this locally

```
type gender = str
type salary = int
data := store(gender, salary)

m := reduce(lambda x,y: x+y, filter("m", data))
f := reduce(lambda x,y: x+y, filter("f", data))
d := m - f

s := gather(reduce(lambda x,y: x+y, scatter(d)))
```

The companies will need to perform an MPC

```
type gender = str
type salary = int
data := store(gender, salary)
m := reduce(lambda x,y: x+y, filter("m", data))
f := reduce(lambda x,y: x+y, filter("f", data))
d := m - f
s := gather(reduce(lambda x,y: x+y, scatter(d)))
```

What to do with a Scatter program?

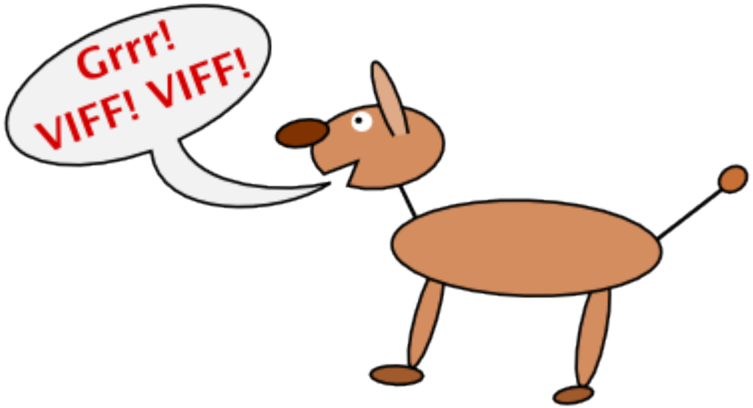
Programming language to specify MapReduce and MPC operations.

Compiler to convert Scatter programs to tasks that are executable in existing MapReduce and MPC frameworks.

Backend platform running those MapReduce and MPC frameworks to act as an execution environment for a compiled Scatter program.

Our current target frameworks

“a fast and general engine for large-scale data processing”



MPC framework that allows for Shamir secret sharing, arithmetic, and comparison over secret shares

Let’s compile Scatter code to PySpark*

```
5: own m := reduce(lambda x,y: x+y,
                   filter("m", data))
6: own f := reduce(lambda x,y: x+y,
                   filter("f", data))
7: own d := m - f
```

```
m = data.filter(lambda x: x[0] == 'm')\
        .reduceByKey(lambda x, y: x + y)\
        .collect()
f = data.filter(lambda x: x[0] == 'f')\
        .reduceByKey(lambda x, y: x + y)\
        .collect()
d = ('d', m[0][1] - f[0][1])
```

* Apache Spark’s Python API

Let’s compile Scatter code to Viff

9: own s := gather(reduce(lambda x,y: x+y,
scatter(d)))

```
def input(in_handle):
  return in_handle.read()
def output(result, out_handle):
  out_handle.write(result)
def reduceByKey(lmbd, kv_store):
  distinct_keys = set(map(lambda x: x[0], kv_store))
  res = []
  for k in distinct_keys:
    pairs_for_key = filter(lambda x: x[0] == k, kv_store)
    values_for_key = map(lambda x: x[1], pairs_for_key)
    v = reduce(lmbd, values_for_key)
    res.append((k, v))
  return res
def run(id, players, in_handle, out_handle):
  Zp = GF(104729)
  kv_store = input(in_handle)
  filtered = filter(lambda x: x[0] == 'diff', kv_store)
  mapped = map(lambda x: x[1], filtered)
  private_kv_store = sorted(mapped, key=lambda x: x[0])
def protocol(rt):
    def exchange_key_stores(rt):
      def create_shared_key_stores(keys, player_mask):
        keys_to_sharers = zip(
          [chr(k) for k in keys], player_mask)
        return keys_to_sharers
```

Let's compile Scatter code to Viff

9: own s := gather(reduce(lambda x,y: x+y,
scatter(d)))

```
def key_store_sizes_ready(key_store_sizes):
        return [int(ks) for ks in key_store_sizes]
      def share_keys(key_store_sizes, rt, Zp):
        sorted_keys = []
        player_mask = []
        for player in rt.players:
          key_store_size = key_store_sizes[player - 1]
          for i in xrange(key_store_size):
            if rt.id == player:
              key = ord(private_kv_store[i][0])
            else:
              key = None
            sorted_keys.append(rt.shamir_share(
              [player], Zp, key, threshold=0))
            player_mask.append(player)
        gathered_keys = gather_shares(
          [rt.open(k) for k in sorted_keys])
        return gathered_keys.addCallback(create_shared_key_stores,
                                         player_mask)
      shared_key_store_sizes = rt.shamir_share(
        players, Zp, len(private_kv_store), threshold=0)
      opened_key_store_sizes = map(rt.open, shared_key_store_sizes)
      key_store_sizes = gather_shares(
        opened_key_store_sizes).addCallback(key_store_sizes_ready)
```

Let’s compile Scatter code to Viff

9: own s := gather(reduce(lambda x,y: x+y,
scatter(d)))

```
keys_to_sharers = key_store_sizes.addCallback(
        share_keys, rt, Zp)
    return keys_to_sharers
def distribute_shares(rt, Zp, private_kv_store, keys_to_sharers):
    private_value_queue = collections.deque(
        map(lambda x: x[1], private_kv_store))
    shared_kv_store = []
    for key, sharer in keys_to_sharers:
        if sharer == rt.id:
            value = rt.shamir_share(
                [sharer], Zp, private_value_queue.popleft())
        else:
            value = rt.shamir_share([sharer], Zp)
        shared_kv_store.append((key, value))
    return shared_kv_store
def open_shares(rt, kv_store, keys_to_owners, result_handler):
    owner_queue = collections.deque(
        map(lambda x: x[1], keys_to_owners))
    opened_res = filter(lambda x: bool(x[1]), [(k, rt.open(v, owner_queue.popleft()))
                                      for k, v in kv_store])
    expected_keys = sorted(map(lambda x: x[0], opened_res))
    result_kv_store = []
    for k, v in opened_res:
        v.addCallback(
            result_handler, k, result_kv_store, expected_keys)
```

Let’s compile Scatter code to Viff

```
9: own s := gather(reduce(lambda x,y: x+y,
                          scatter(d)))
```

Phew, we have executable code!

Programming language to (unified) specify MapReduce and MPC operations.

Support for actual MapReduce and MPC frameworks that can execute those operations.

Backend platform running those MapReduce and MPC frameworks to act as an execution environment for a Scatter program.

Executable where?

Programming language to (unified) specify MapReduce and MPC operations.

Support for actual MapReduce and MPC frameworks that can execute those operations.

Backend platform running those MapReduce and MPC frameworks to act as an execution environment for a Scatter program.

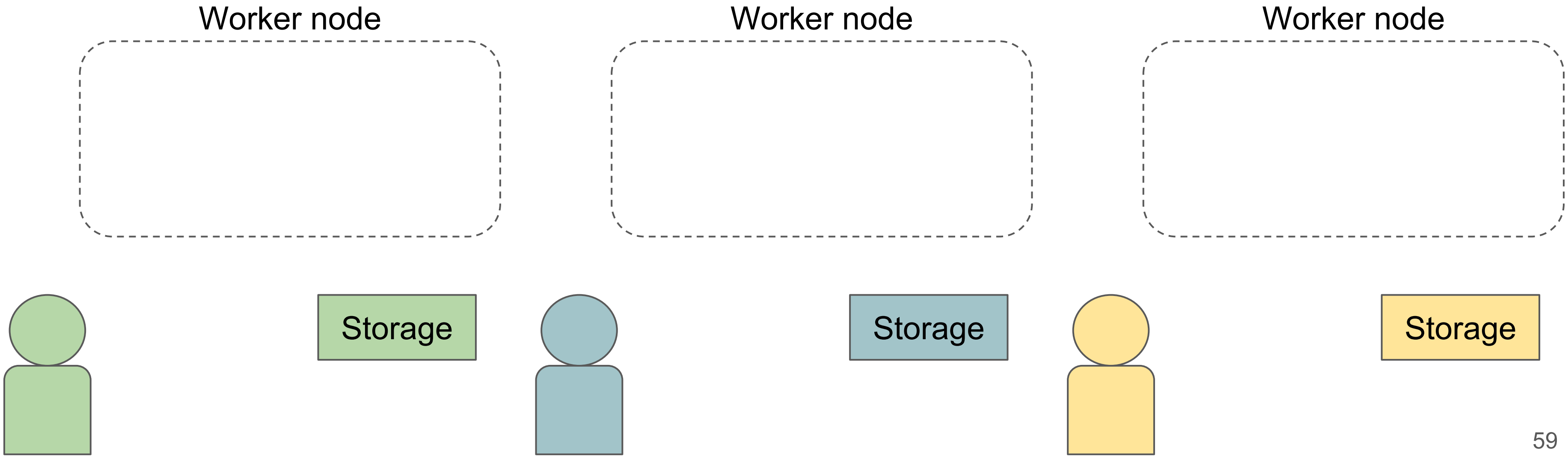
Let’s build our backend.

Give each client the computational resources to:

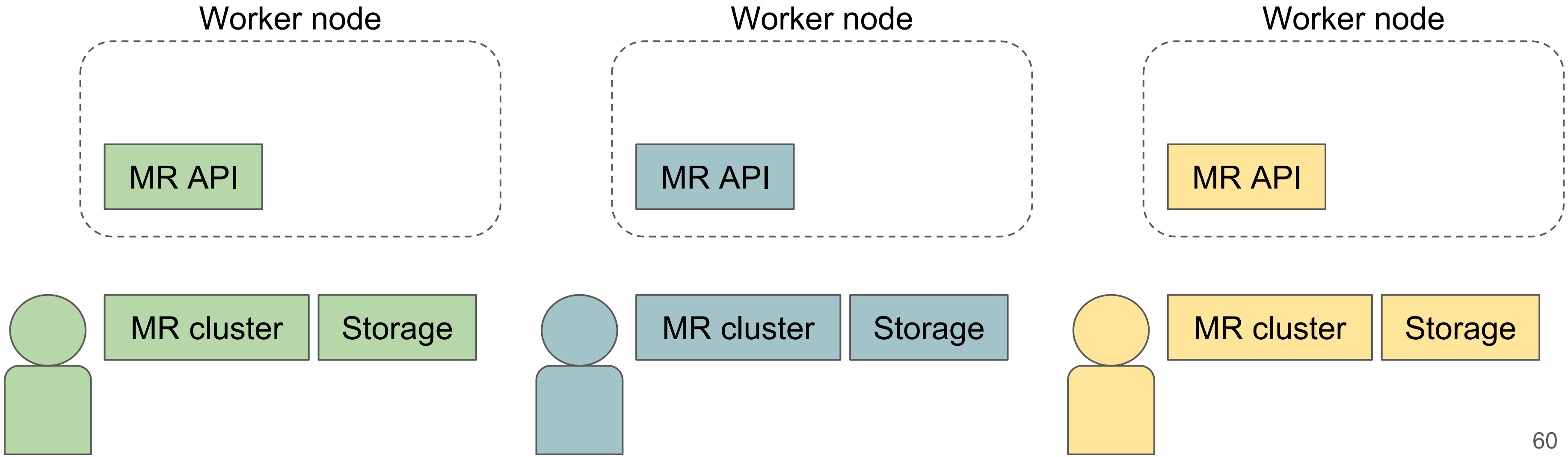
- run local MapReduce tasks on their data
- participate in MPC rounds to process data across companies
- coordinate those two actions

Let’s build a **worker node**.

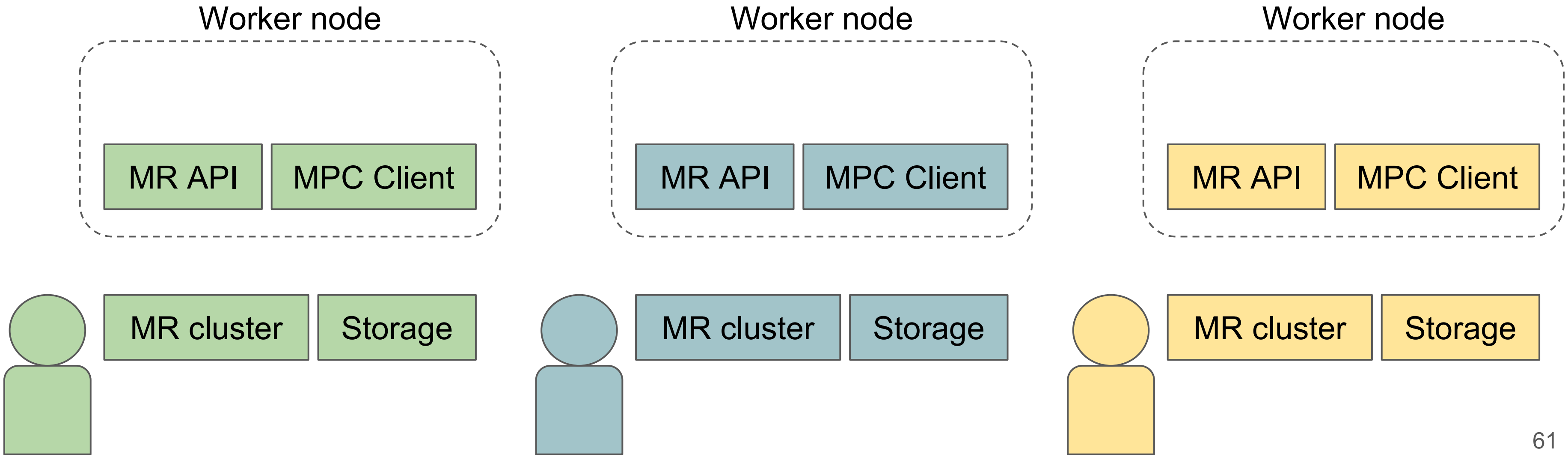
What does each company start with?



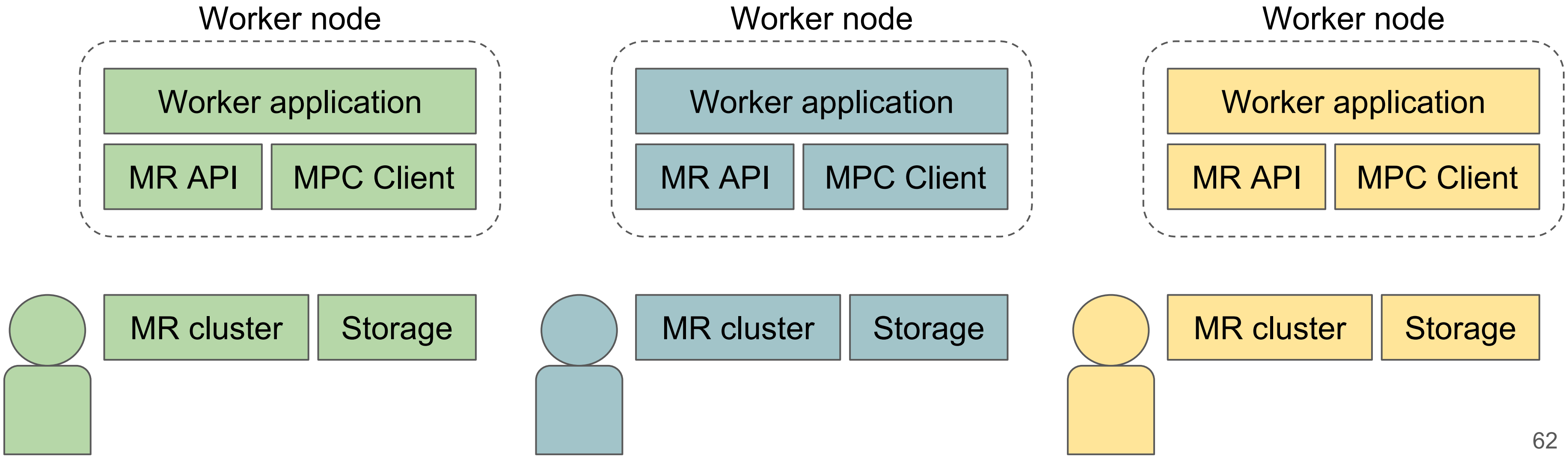
What do companies need to run MapReduce code?



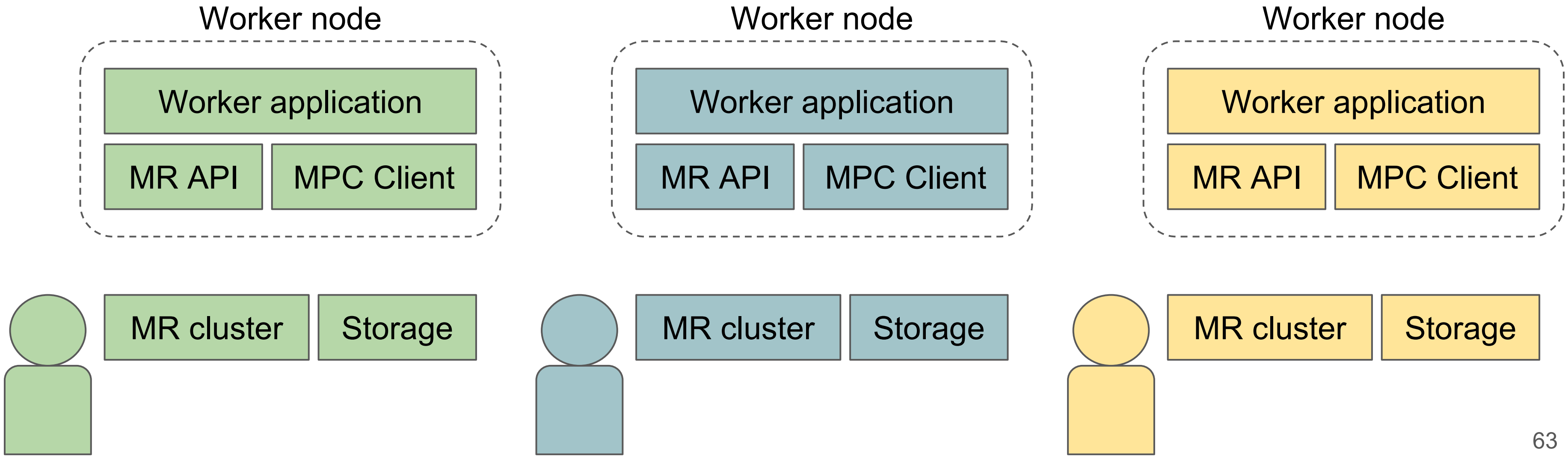
What do companies need to run MPC code?



What about coordinating program execution?



Bundle up the software, we have our **worker nodes**!



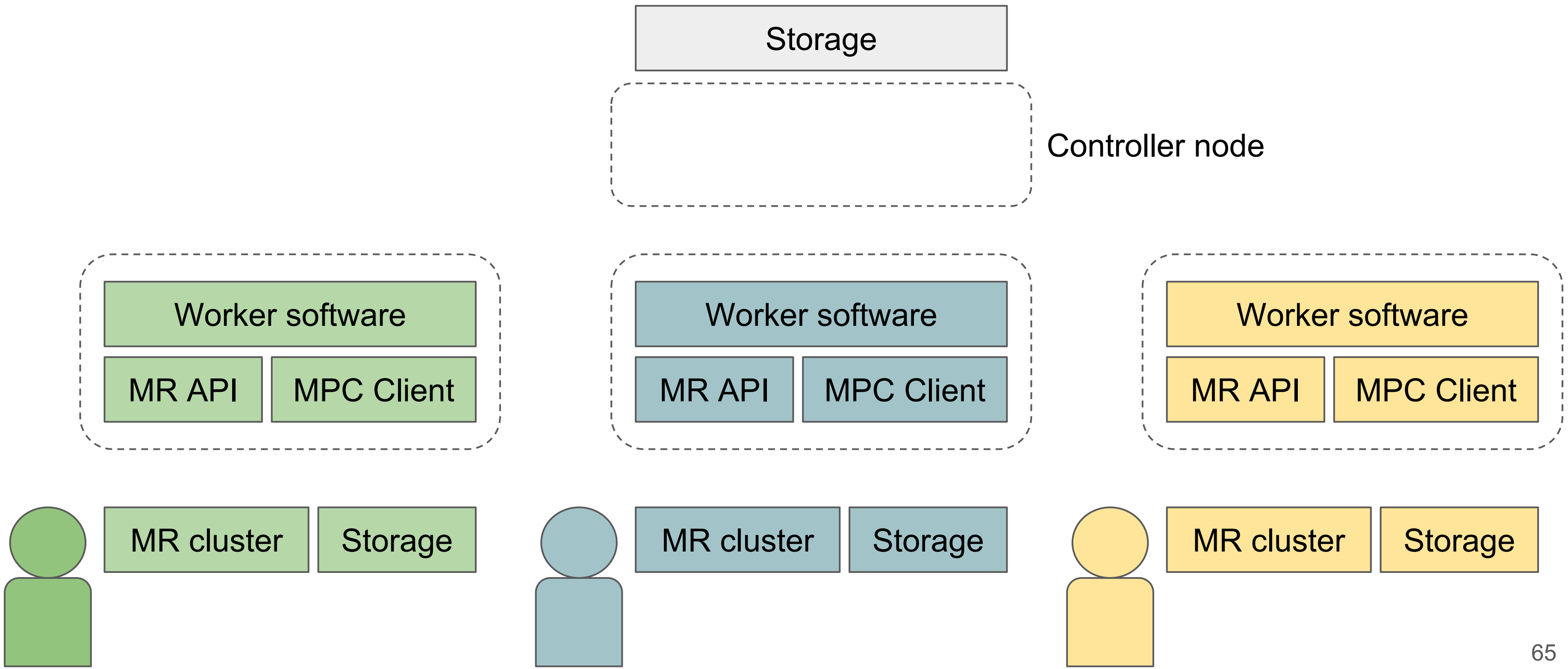
Almost done...

We have a distributed system.

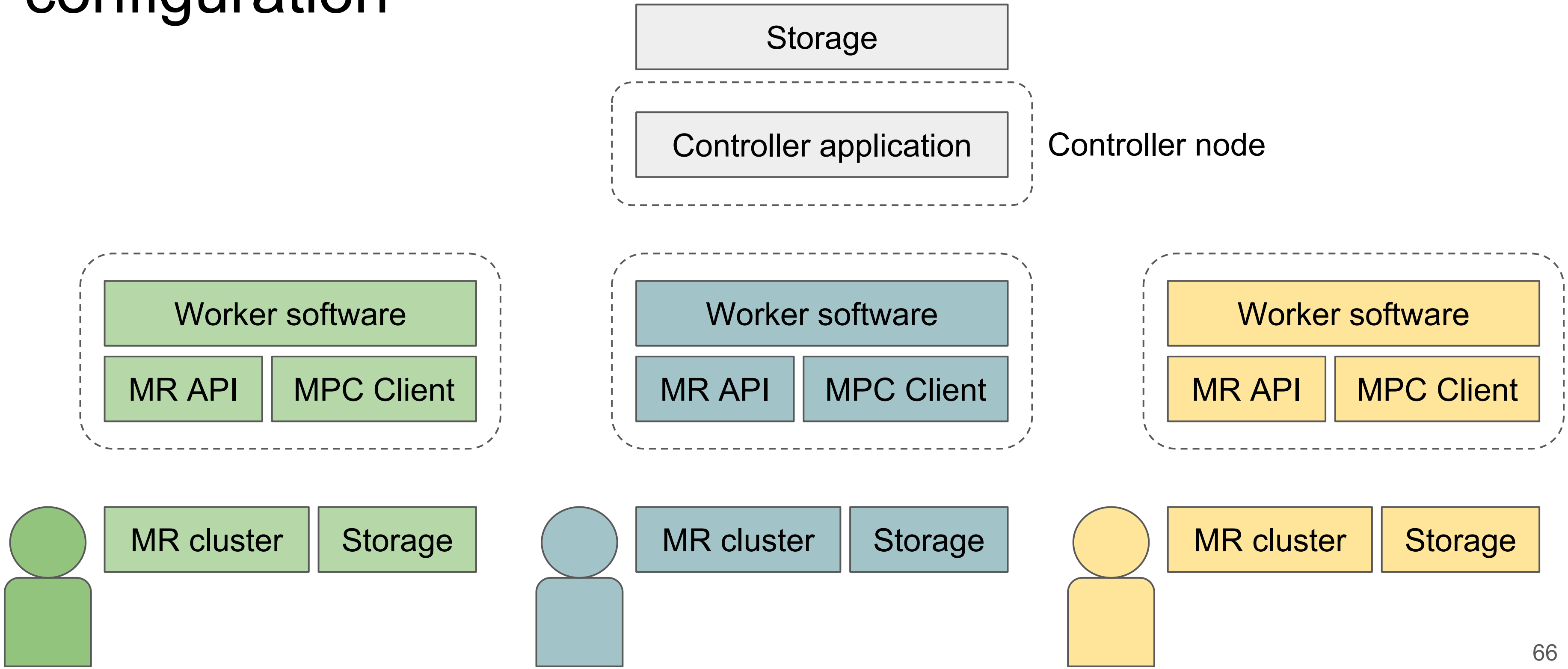
We need to coordinate task execution not only within worker nodes but also **across** worker nodes. (Why?)

Let’s build a **controller node**.

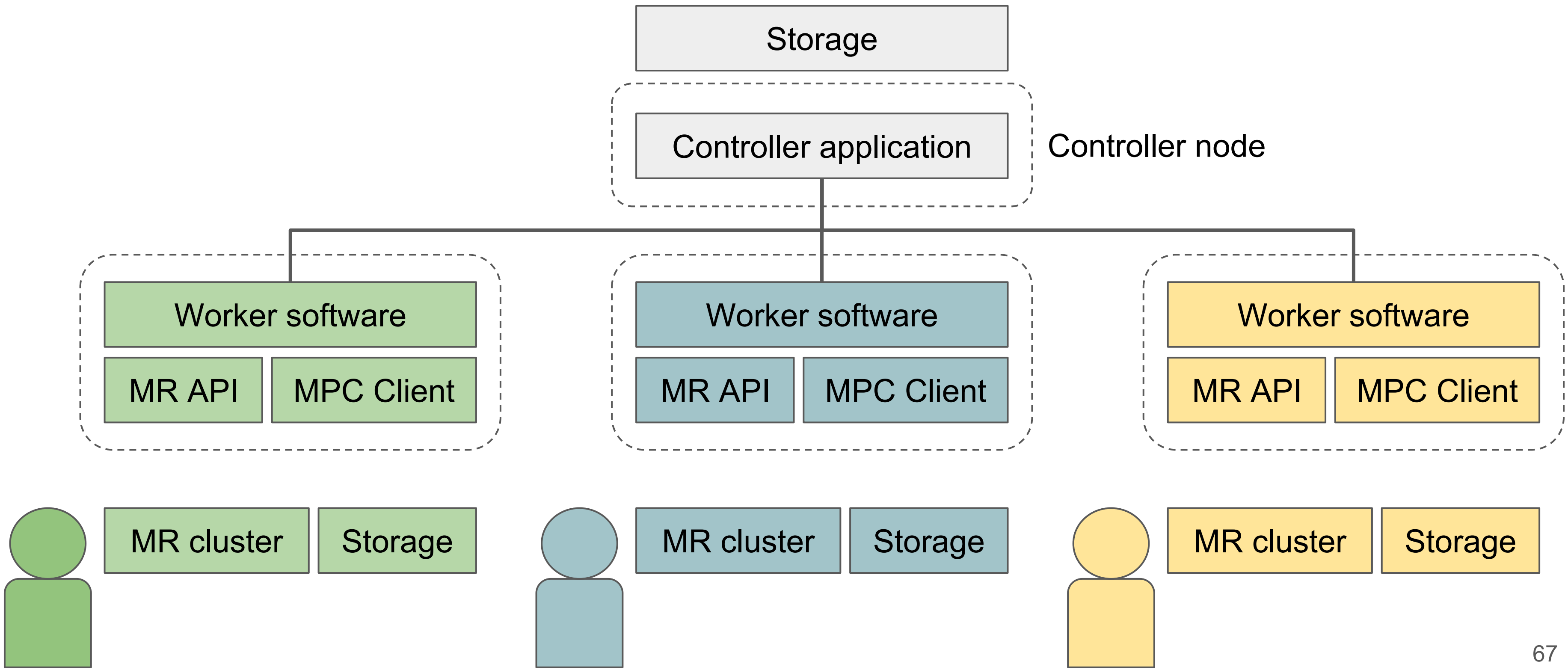
What goes inside a controller node?



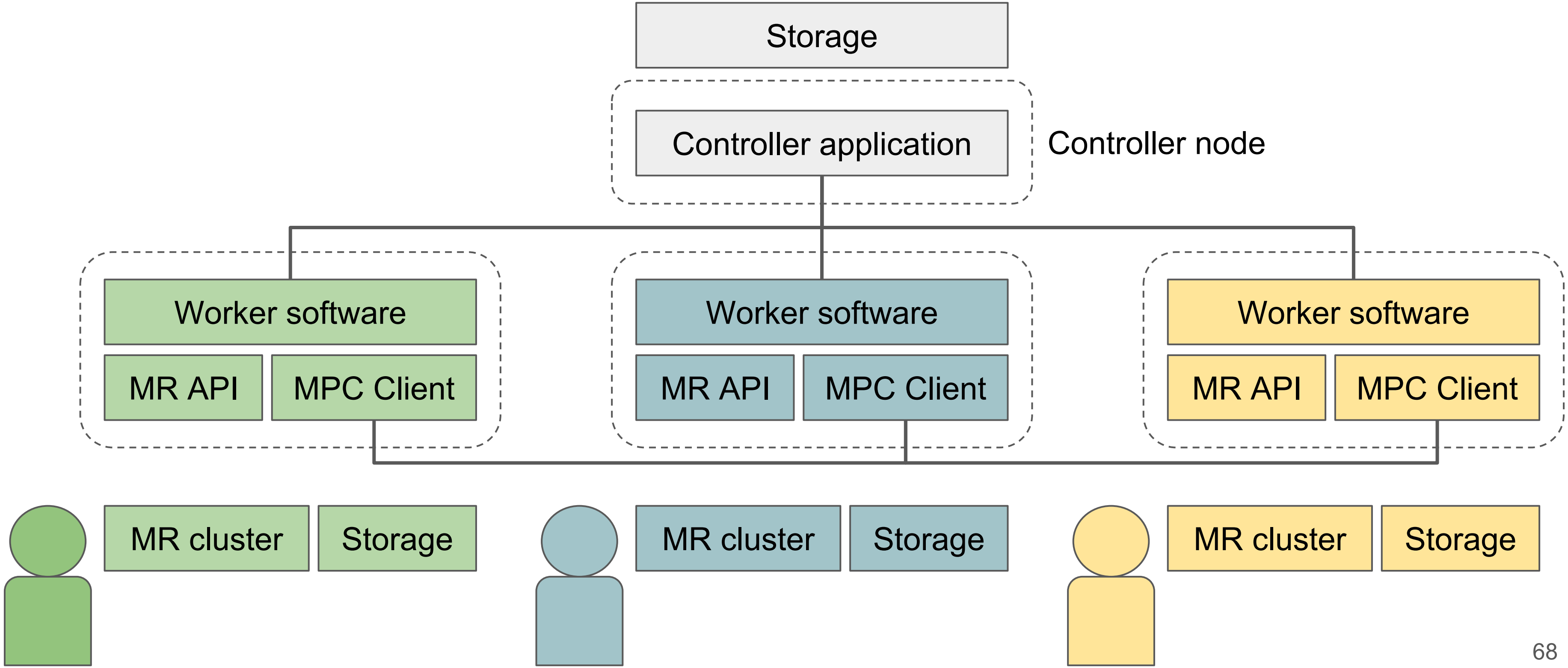
Software to orchestrate task execution + worker configuration



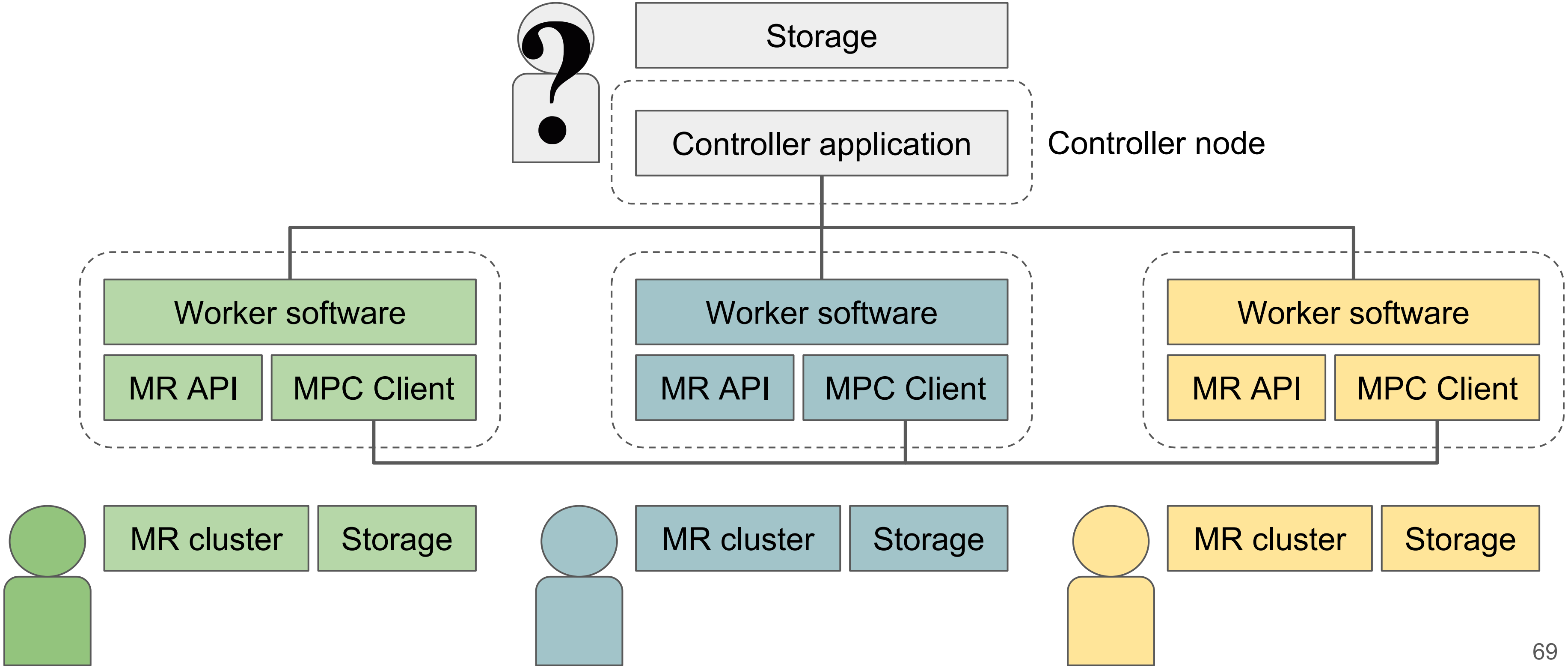
Workers connect to Controller over HTTPS



Use controller to configure and connect MPC clients



Who controls the controller?



And there we have it

Programming language to specify MapReduce and MPC operations.

Compiler to convert Scatter programs to tasks that are executable in existing MapReduce and MPC frameworks.

Backend platform running those MapReduce and MPC frameworks to act as an execution environment for a compiled Scatter program.

Open future

Extend compiler with static analysis tools.

And more!