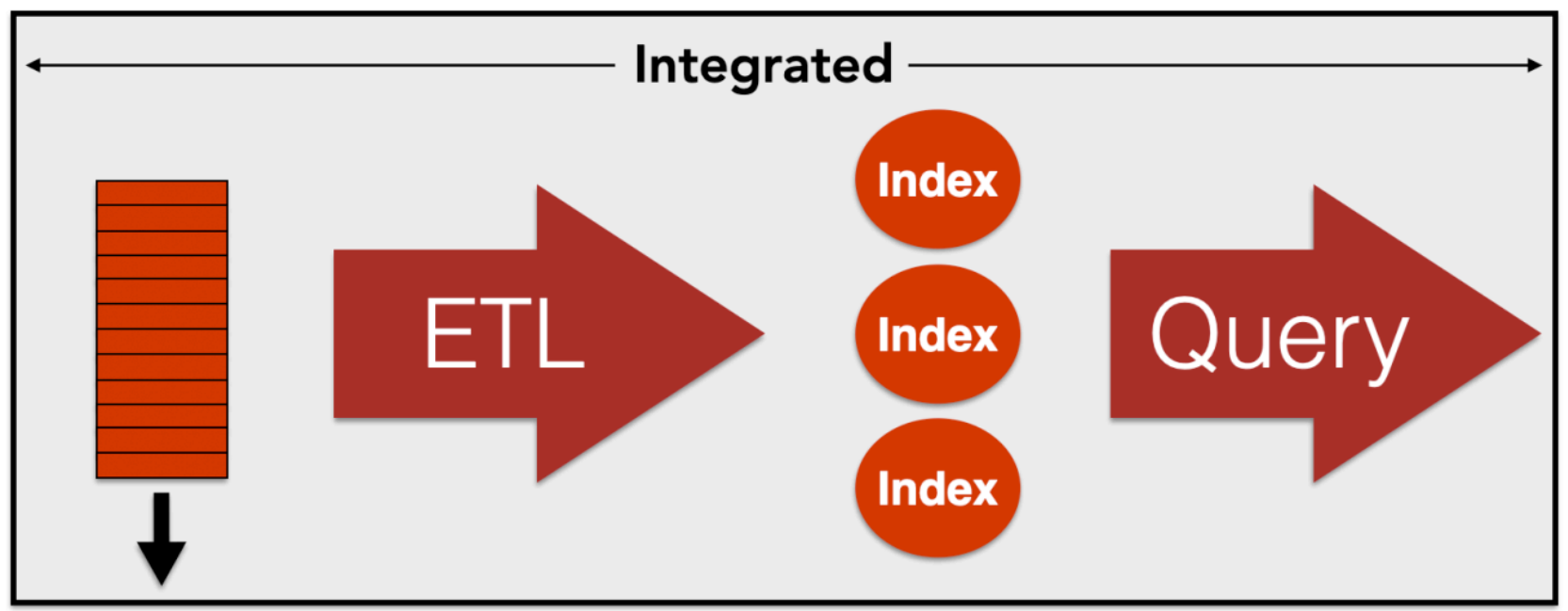
Rama workshop

Rama overview

- Platform for building end-to-end backends: ingestion, processing, indexing, and querying
- Supports both synchronous and asynchronous use cases
- Massively scalable

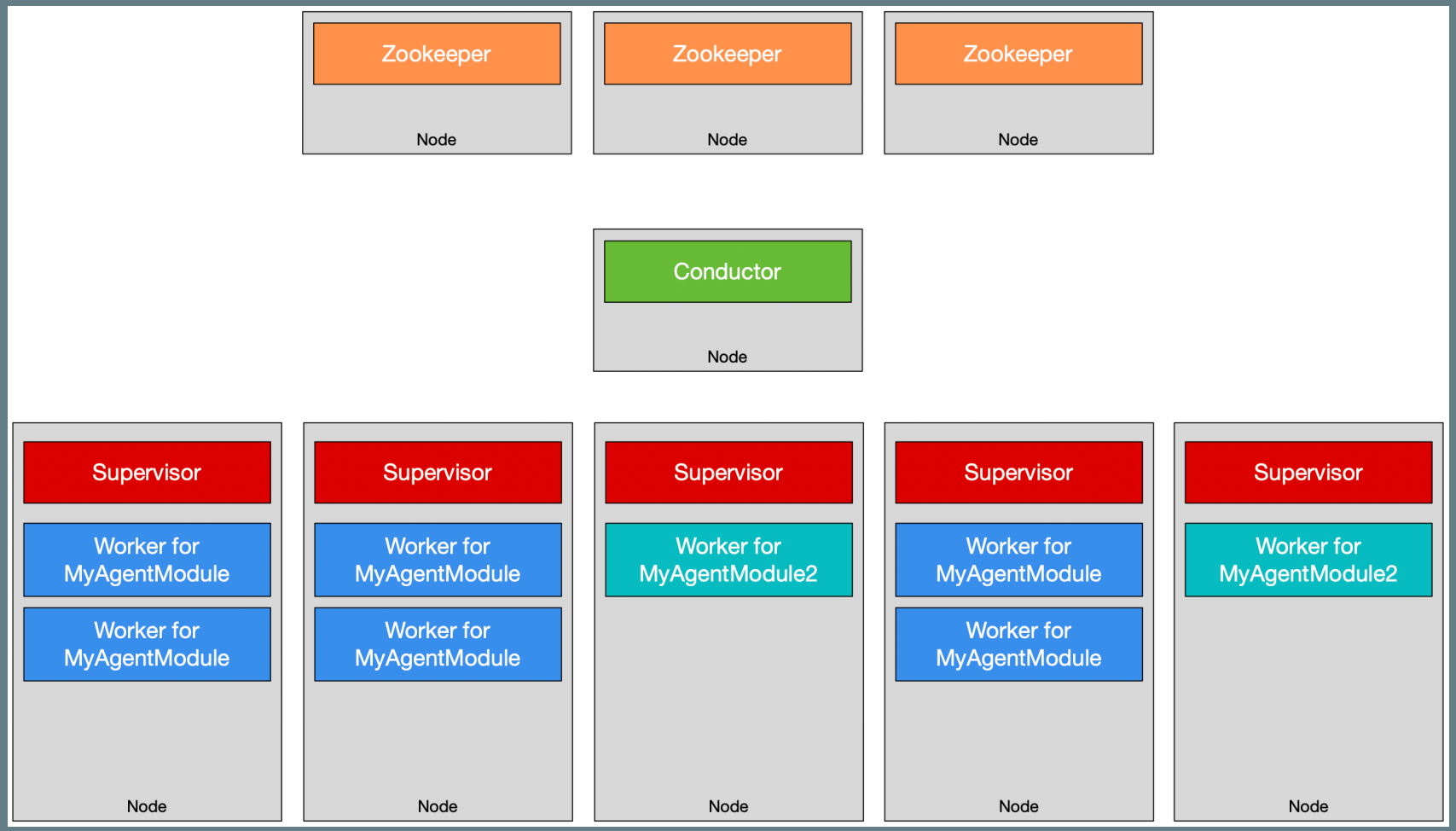
Rama overview



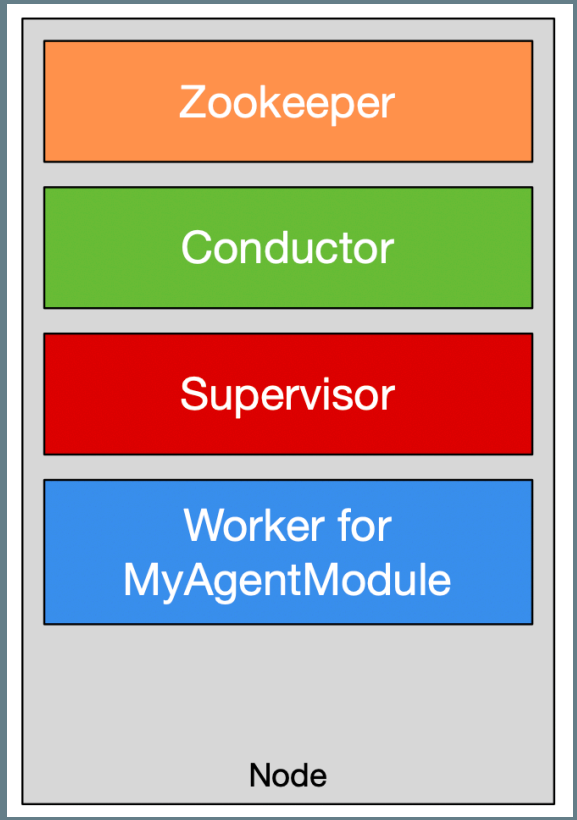
Rama overview

- Rama applications are called “modules”
- Module contain any number of depots, ETL topologies, PStates, and query topologies
- Modules are the deployment unit onto a Rama cluster
- Rama CLI used to deploy, update, and scale modules with one line commands

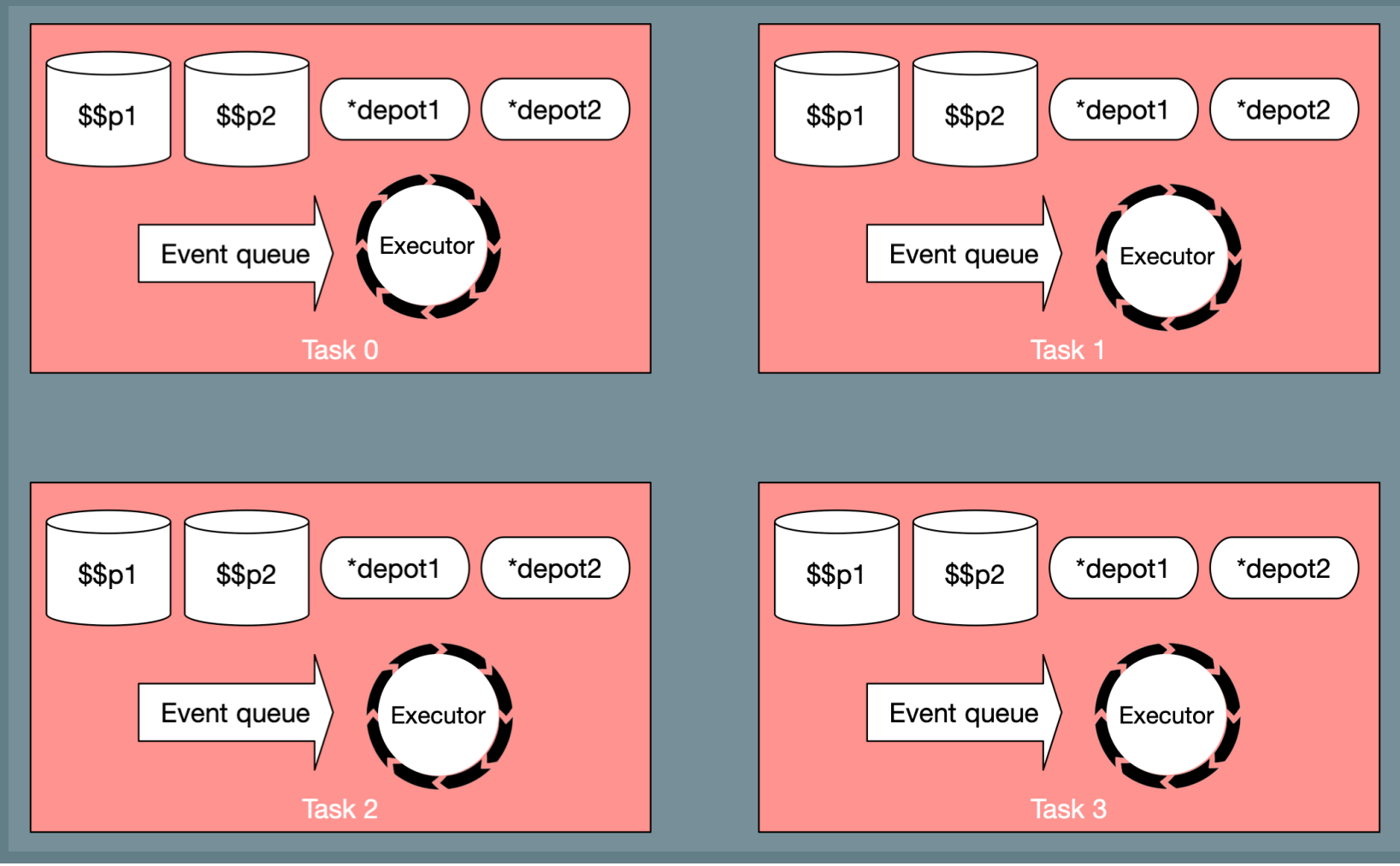
Rama overview



Rama overview



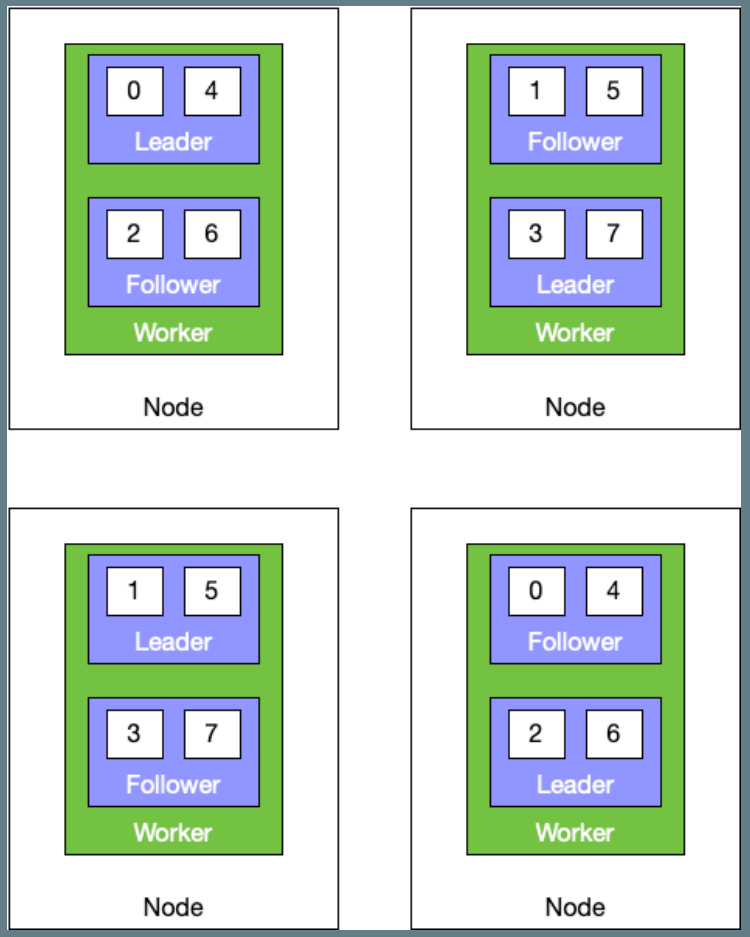
Module structure



Module structure

- Modules are deployed with a fixed number of tasks (must be power of 2)
- Each task has one partition of every depot and PState for the module
  - Except for objects defined as “global”, which are only on task 0
- Tasks run on “task threads”
- A task thread runs one or more tasks
- Task threads run inside “workers”
- Workers are assigned to nodes
- A module can be replicated

Module structure



Workshop goal

- Build the backend for an ultra-scalable collaborative todo list application
- Features:
  - User registration
  - Create and edit user profiles
  - Create new lists
  - Share lists
  - Add/remove from lists
  - Edit list items
  - Mark items as complete
  - Reorder list
  - Remove list
  - Basic analytics on user activity

Defining a module

```
(defmodule MyModule
  [setup topologies]
  ;; implementation here
  )
```

- “setup” is for depots and task globals
- “topologies” is for:
  - ETL topologies along with the PStates they own
  - Query topologies

In-process cluster (IPC)

```
(rtest/create-ipc)
```

- Testing/experimentation environment
- Runs full Rama cluster in process

Depots

```
(declare-depot setup *depot (hash-by :id))
```

- “Depot partitioner” routes appends to tasks
  - hash-by
  - :random
  - :disallow
  - Custom partitioners
- Option to declare as global

Depots

- (require ‘[workshop.depot :as depot] :reload)
- Run the code in the comment to append the 10 maps of the form {:a <num>}
- Find the partition that has {:a 6}

Exercise

Depots

- Fill in MultiDepotModule
- Append a bunch of maps to *hash-depot with key :id
- Check depot partitions to see what data is on which partition

- Append to global depot and inspect the object info

Exercise

PStates

```
(declare-pstate
  s
  $$kv
  {String Long})
```

Owning topology

PState name

Schema

- “Partitioned state”
- Indexed datastore of any data model
- Defined as compound data structures

PStates

```
(declare-pstate
  s
  $$profiles
  {Long (fixed-keys-schema
          {:name String
           :created-at-millis Long})})
```

PStates

```
(declare-pstate
  s
  $$profiles
  {Long (fixed-keys-schema
          {:name String
           :created-at-millis Long
           :friends (set-schema Long {:subindex? true})})})
```

PStates

```
{String {String String}}
```

vs.

```
{String (map-schema String
                    String
                    {:subindex? true})}
```

Main table

"alice"	{"k1" "a" "k2" "b" "k3" "c"}
"bob"	{"k1" "a2" "k4" "b2"}

Main table

"alice"	0
"bob"	1

Subindexing table

[0,"k1"]	"a"
[0,"k2"]	"b"
[0, "k3"]	"c"
[1, "k1"]	"a2"
[1, "k4"]	"b2"

Subindexing

PStates

- map-schema / {}
- fixed-keys-schema
- Top-level class (value PState)
  - e.g. “Map”, “Long”
- vector-schema / []
  - Cannot be used at top-level
- set-schema / #{}
  - Cannot be used at top-level
- Use *-schema functions when need to declare as subindexed

PStates

```
(use 'com.rpl.rama)
(use 'com.rpl.rama.path)
(require '[com.rpl.rama.test :as rtest])

(def p (rtest/create-test-pstate {String Long}))
(rtest/test-pstate-transform [(keypath "a") (termval 1)] p)
(rtest/test-pstate-transform [(keypath "b") (termval 2)] p)
(rtest/test-pstate-transform [(keypath "b") (termval "v")] p) ;; schema error
(println (rtest/test-pstate-select (keypath "a") p))
(println (rtest/test-pstate-select MAP-KEYS p))
(close! p)
```

- rtest/create-test-pstate
- rtest/test-pstate-select
- rtest/test-pstate-transform

Learning helpers

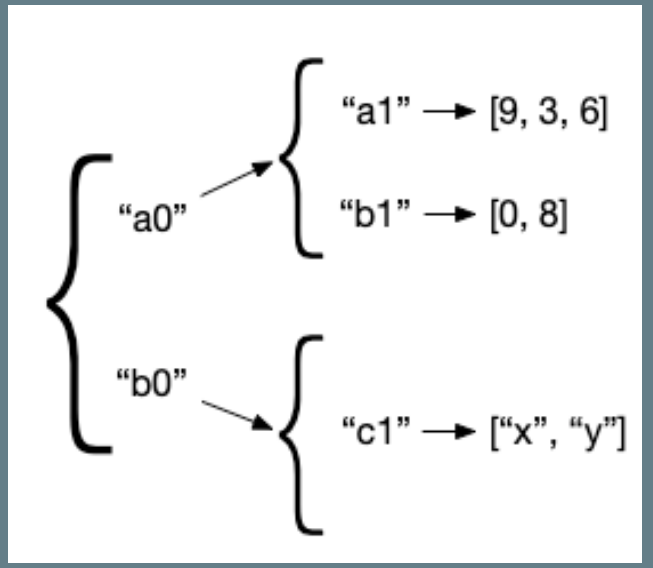
PStates

- PStates are read and written to with Specter paths
- A path is a list of “navigators”

- keypath
- MAP-VALS
- ALL
- nthpath
- pred
- NONE-ELEM
- AFTER-ELEM
- set-elem
- term
- termval
- NONE>

PStates

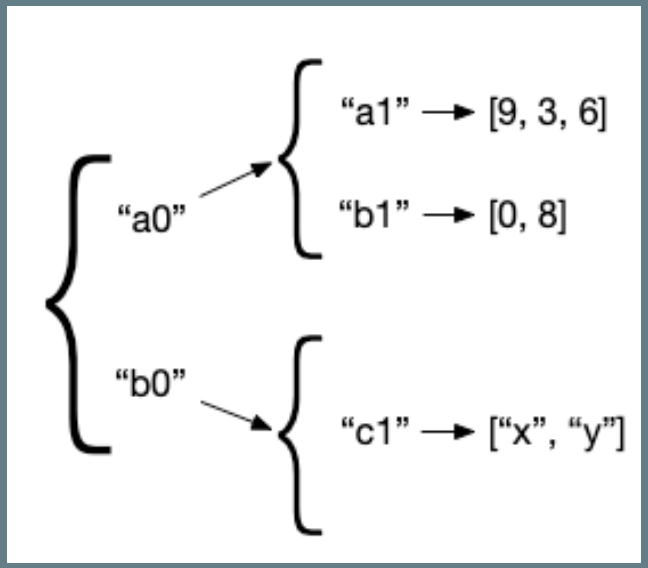
- (require '[workshop.pstate :as pstate] :reload)
- (def p (pstate/test-pstate))
- Use (rtest/test-pstate-select STAY p) to see current
- Make p have this data in it:



Exercise

PStates

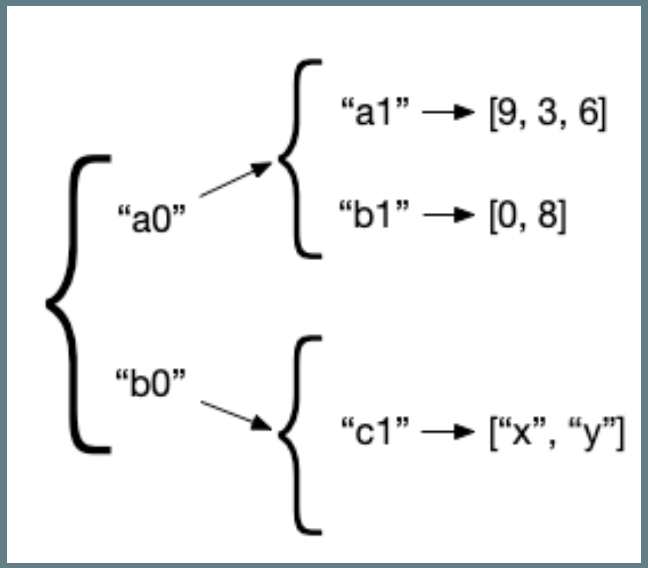
- Query for leaf values
- Query for all even numbers
- Query for keys nested one level ([“a1”, “b1”, “c1”])
- Query for all lists with more than 2 elements



Exercise

PStates

- Remove 3
- Append “z” to list at “c1”
- Append “!” to every list
- Remove the nested key “b1”



Exercise

Dataflow

- Programmed with Rama's dataflow language
- Dataflow language is a new programming paradigm
  - Based on generalizing the concept of a function
- Exposed via macros, so it's still Clojure
- Can invoke dataflow from Clojure and vice-versa

Dataflow

```
(source> *depot :> {:keys [*op *data]})
(<<cond
  (case> (= *op "all"))
  (|all)
  (println "Running on every task")

  (case> (= *op "loop"))
  (loop<- [*i *data]
    (<<if (> *i 0)
      (|shuffle)
      (:>)
      (continue> (dec *i))))
  (default>)
  (|hash *data)
  (println "Running on one task"))
(println "Continuation")
(|global)
(println "Running on task 0")
```

Unifies “what” with “where”

Dataflow

Enables “what” and “where” to be unified

Dataflow

- (use ‘com.rpl.rama)
- (use ‘com.rpl.rama.path)
- Copy/paste from workshop.dataflow

Dataflow

```
(deframafn foo []
  (:> 1))
```

Dataflow

```
(deframafn bad-foo []
  (:> 1)
  (:> 2))
```

Dataflow

```
(deframafn bad-foo []
  )
```

Dataflow

```
(deframafn bad-foo []
  (:> 1)
  (println "B"))
```

Dataflow

```
(deframaop foo []
  (:> 1)
  (:> 2)
  (println "A")
  (:> 3))

(?<-
  (foo :> *v)
  (println "B" *v))
```

Dataflow

```
(?<-
 (println "B" (foo)))
```

Dataflow

- Variables are symbols beginning with *, %, or $$
- * = “value var”
- % = “frag var”
- $$ = “PState var”

Dataflow

```
(deframaop foo []
  (:> 1 2 3)
  (:> 4 5 6)))

(?<-
  (foo :> *v1 *v2 *v3)
  (println "OUT" *v1 *v2 *v3))
```

Dataflow

- Implement foo(x, y) = (x * 10 + y * 3) * (x * 2 - y * 5)
  - (foo 1 2) = -128
  - (foo 10 3) = 545

Exercise

Dataflow

```
(deframafn foo [*x *y]
  (+ (* 10 *x) (* 3 *y) :> *v1)
  (- (* 2 *x) (* 5 *y) :> *v2)
  (:> (* *v1 *v2)))
```

Exercise

Dataflow

```
(deframafn foo [{:keys [*a *b] :as *m}]
  (println "A" *a)
  (println "B" *b)
  (println "M" *m)
  (:>))

(foo {:a 1 :b 2})
```

Dataflow

```
(deframafn foo [*v]
  (<<if *v
    (println "true path")
    (:> 1)
   (else>)
    (println "false path")
    (:> 2)))
```

Dataflow

```
(deframafn foo [*v]
  (<<cond
    (case> (= *v 1))
    (println "case 1")
    (:> 1)

    (case> (= *v 2))
    (println "case 2")
    (:> 2)

    (default>)
    (println "default case")
    (:> 3)))
```

Dataflow

```
(deframafn foo [*x]
  (<<if (> *x 100)
    (:> *x)
   (else>)
    (:> (%self (* 2 *x)))))
```

Exercise

- Implement recursive version of fib(n)
  - fib(0) = 1
  - fib(1) = 1
  - fib(2) = 2
  - fib(3) = 3
  - fib(4) = 5
  - fib(5) = 8

Exercise

Exercise

```
(deframafn fib [*n]
  (<<if (contains? #{0 1} *n)
    (:> 1)
   (else>)
    (:> (+ (%self (dec *n)) (%self (- *n 2))))))
```

Exercise

Dataflow

```
(deframaop foo [*x]
  (loop<- [*i *x :> *v]
    (<<if (>= *i 0)
      (:> *i)
      (continue> (dec *i))
      ))
  (:> *v))

(?<-
  (foo 5 :> *v)
  (println "V" *v))
```

Exercise

- Implement (explode-map *m :> *k *v) using loop<- and seq

```
user=> (?<-
  #_=>   (explode-map {:a 1 :b 2 :c 3} :> *k *v)
  #_=>   (println "OUT" *k *v))
OUT :c 3
OUT :b 2
OUT :a 1
nil
```

Exercise

Exercise

```
(deframaop explode-map [*m]
  (loop<- [*seq (seq *m) :> *k *v]
    (<<if *seq
      (first *seq :> [*k *v])
      (:> *k *v)
      (continue> (next *seq))))
  (:> *k *v))
```

Exercise

Dataflow

```
(deframafn foo [*v]
  (<<ramafn %ret
    [*v2]
    (:> (+ *v *v2)))
  (:> %ret))

((foo 10) 5)
```

Dataflow

```
(deframafn foo [*v]
  (<<ramaop %ret
    [*v2]
    (:> (ops/range> 0 (* *v *v2))))
  (:> %ret))

(?<-
  (foo 5 :> %f)
  (println (%f 2)))
```

ETL topologies

- Subscribe to depots
- Own any number of PStates
  - Only the owning topology can write to a PState
- Dataflow code that runs reactively to new data
- Change tasks with “partitioners”
- Interact with depot and PState partitions local to the current task

Stream topologies

- One-at-a-time processing model
- At-least once or at-most once semantics
- Can send information back to appenders with ack-return>

Stream topologies

- (require '[workshop.stream :as stream] :reload)
- (def m (stream/launch!))
- (foreign-append! (:depot m) (stream/->Item “a” “b”))

Stream topologies

- Do a bunch of (foreign-append! (:depot m) (stream/->Item “a” “b”))
- Query (foreign-select-one (keypath “a”) (:counts m))
- See that the counts are off

Exercise

Partitioners

- Fragment that emits asynchronously some number of times, potentially to different tasks/nodes
- Composes with any other dataflow code like any other operation

Partitioners

- (|hash *k)
- (|global)
- (|all)
- (|direct *task-id)
- (|custom afn & args)

Stream topologies

- Fill in the TODOs to fix the module
- Verify it works with foreign selects

Exercise

PStates in topologies

- local-select>
  - Emits per navigated value as opposed to emitting a list of navigated values
  - (local-select> [:a nil?] $$p)
    - This emits once if the value for :a is nil, and doesn't emit otherwise
  - (local-select> ALL $$p :> *v)
    - This emits zero or more times
  - (local-select> (subselect ALL) $$p :> *l)
    - This emits one list of navigated values
- local-transform>

Stream topologies

- Try (foreign-select-one ALL (:counts m))
- Foreign queries error when they don’t know how to route the path
- By default routes by the key in the leading keypath
- Try (foreign-select ALL (:counts m) {:pkey “a”})

Exercise

Stream topologies

- Remove the first partitioner and fix by updating the depot partitioner
- Verify it works with foreign selects

Exercise

TodoAppModule

- Fill in data handlers one by one
- Uncomment test assertions as we go

TodoAppModule

- CreateUser
  - If user already exists: (ack-return> {:error “Username already exists”})
  - Otherwise, set :name and :created-at-millis for that user’s profile

Exercise

TodoAppModule

- EditProfileField
  - Set that key to that value for the user

Exercise

TodoAppModule

- CreateList
  - Initialize :name, :created-at-millis, and :owners for the list
  - Add list to the user's :lists key in their profile
  - NONE-ELEM is useful here

Exercise

TodoAppModule

- ShareList
  - No-op if user doesn't own the list
  - No-op if list doesn't exist
  - Add *to-username to the list's owners
  - Add list to *to-username's lists
  - set-elem, NONE-ELEM, and must are useful here

Exercise

TodoAppModule

- AddTodo
  - Append new TodoItem to that list
  - AFTER-ELEM and must are useful here

Exercise

TodoAppModule

- EditTodo
  - Set the value for that key in that TodoItem
  - ALL is useful here
  - This path expression is useful here:
    - (selected? :todo-id (pred= *todo-id))

Exercise

TodoAppModule

- MoveTodo
  - Move the todo with that ID to the given index in the list
  - No-op if the target index is out of bounds
  - No-op if list or item don’t exist
  - Can be done in a single transform with INDEXED-VALS

Exercise

TodoAppModule

- DeleteTodo
  - Remove the todo from the list if it exists
  - Use NONE> as a terminal navigator to remove an element from a collection

Exercise

TodoAppModule

- RemoveList
  - Remove that user from that list's owners
  - Delete the list if there are no more owners
  - Remove that list from that user's lists

Exercise

Microbatch topologies

- Process small batches of data off of depots at same time
- Exactly-once semantics
- Higher latency than stream topologies
- Higher throughput than stream topologies
- No ack-return>
- Supports <<batch blocks

Microbatch topologies

```
(<<sources mb
 (source> *depot :> %mb)
  ;; on task 0
  (%mb :> *data)
  ;; on every task
  ;; ...
  )
```

Aggregation

- Serves multiple purposes:
  - Higher-level, more concise, less flexible way to write to PStates compared to paths
  - Enables huge two-phase optimization for global aggregations using combiner aggregators

Aggregation

- Common aggregations defined in com.rpl.rama.aggs
- Can define custom ones with “accumulator” and “combiner”

Aggregation

```
(require '[com.rpl.rama.aggs :as aggs])

(aggs/+count $$p)
(local-transform> (term inc) $$p)

(aggs/+vec-agg $$p "abc")
(local-transform> [AFTER-ELEM (termval "abc")] $$p)

(+compound $$p {*some-key (aggs/+count)})
(local-transform> [(keypath *some-key) (nil->val 0) (term inc)] $$p)

(+compound $$p {*some-key {*another-key [(aggs/+count) (aggs/+sum *v)]}})
(local-transform> [(keypath *some-key *another-key)
                   (nil->val [0 0])
                   (multi-path
                     [(nthpath 0) (term inc)]
                     [(nthpath 1) (term inc)])]
                  $$p)
```

TodoAppModule

- Implement analytics microbatch topology
- Use compound aggregation to write to the PState

Exercise

Deploy on real cluster

```
./rama devZookeeper &
./rama conductor &
./rama supervisor &

./rama deploy \
  --action launch \
  --systemModule monitoring \
  --tasks 1 \
  --threads 1 \
  --workers 1

./rama deploy \
  --action launch \
  --jar <path to uberjar> \
  --module workshop.todo/TodoAppModule \
  --tasks 4 \
  --threads 2 \
  --workers 1
```

Deploy on real cluster

- Open http://localhost:8888

Deploy on real cluster

```
./rama repl --jar <path to uberjar>

(use 'com.rpl.rama)
(use 'com.rpl.rama.path)
(require '[workshop.todo :as todo])
(def cluster (open-cluster-manager {"conductor.host" "localhost"}))
(def user-depot (foreign-depot cluster "workshop.todo/TodoAppModule" "*user-depot"))
(def profiles (foreign-pstate cluster "workshop.todo/TodoAppModule" "$$profiles"))


(foreign-append! user-depot (todo/->CreateUser "alice" "Alice Smith"))
(foreign-select-one [(keypath "alice") (submap [:name :location :created-at-millis])] profiles)
```

Module update

```
./rama deploy \
  --action update \
  --jar <path to uberjar> \
  --module workshop.todo/TodoAppModule
```

Further learning

- Dataflow
  - Multiple output streams
  - Hooks/anchors/unify>
  - Segmacros
  - Batch blocks
- Tick depots
- Query topologies
- Mirrors
- Reactivity (foreign-proxy)
- Depot and PState migrations