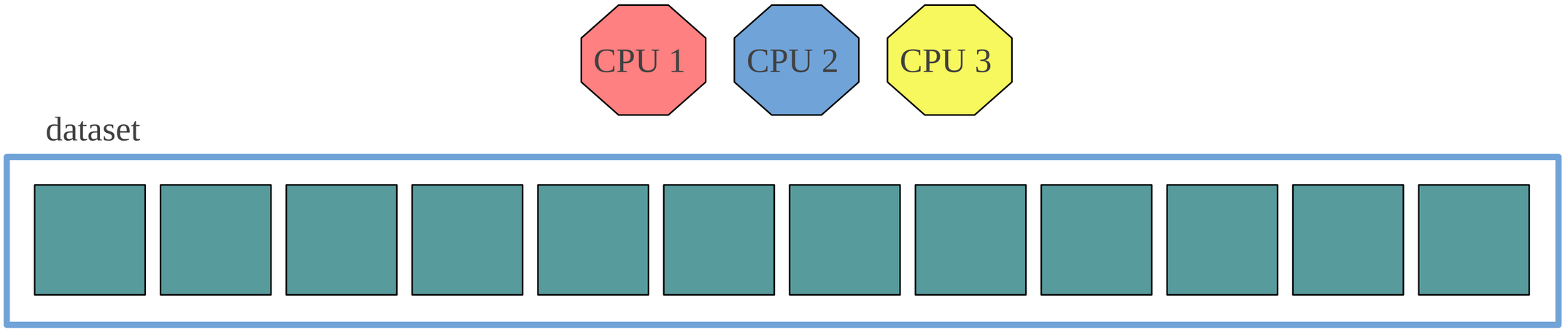
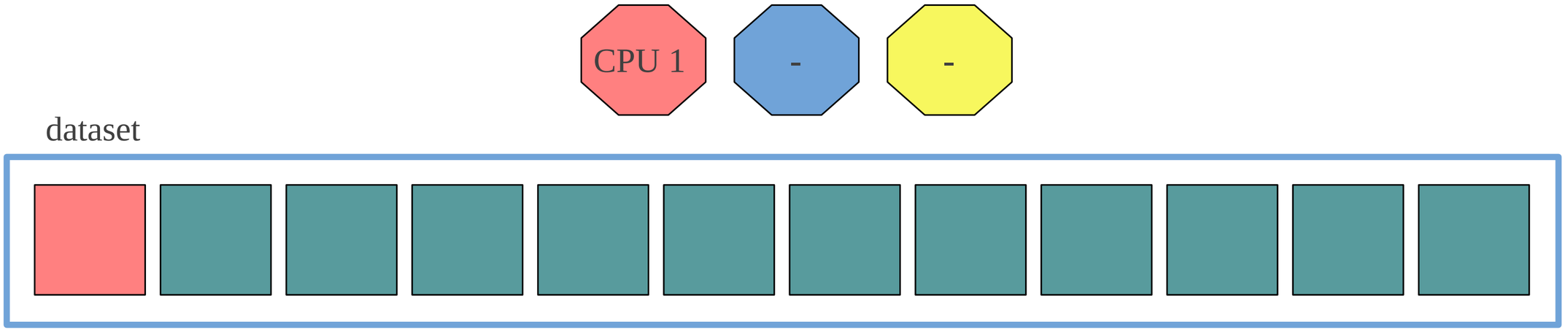
ANALYSIS OF GLOBAL TEMPERATURE

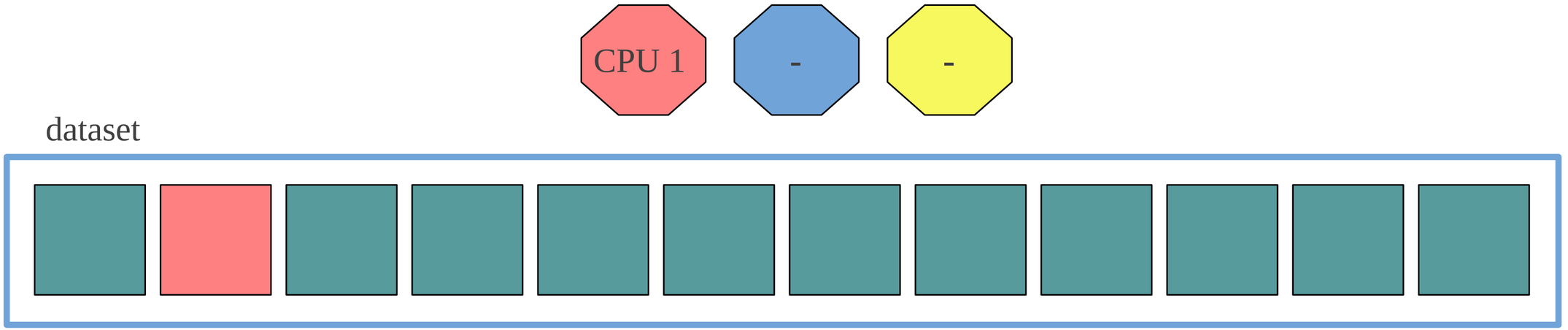
Data processing



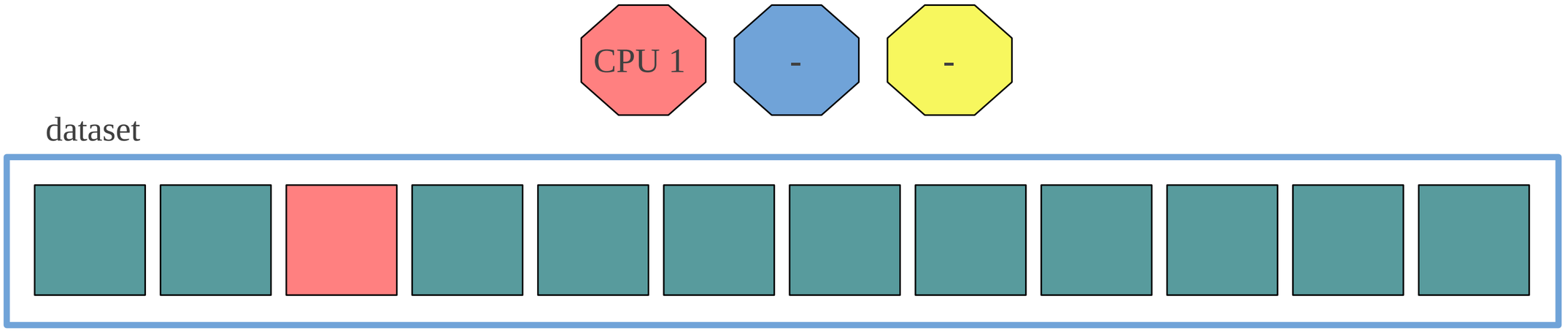
Sequential iteration



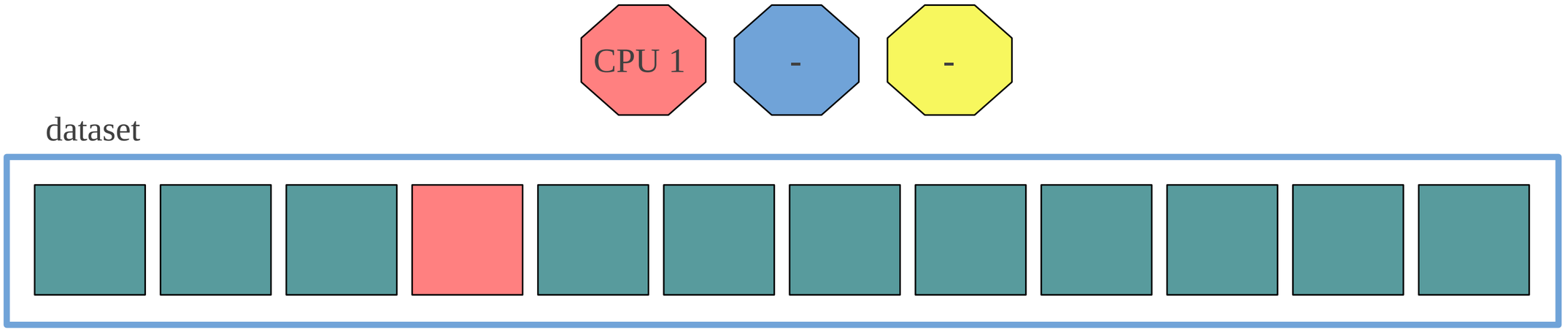
Sequential iteration



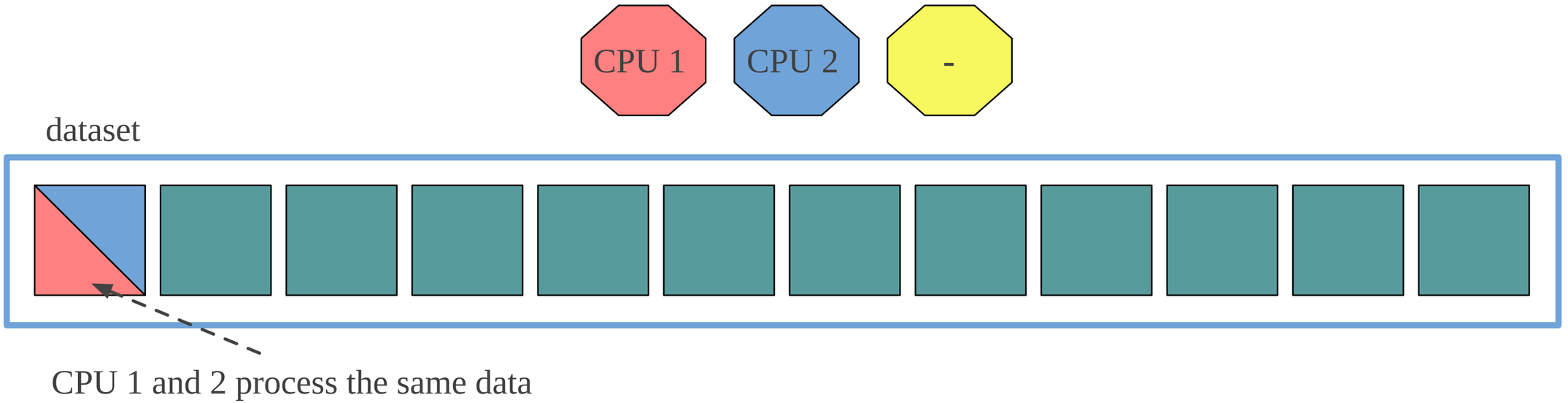
Sequential iteration



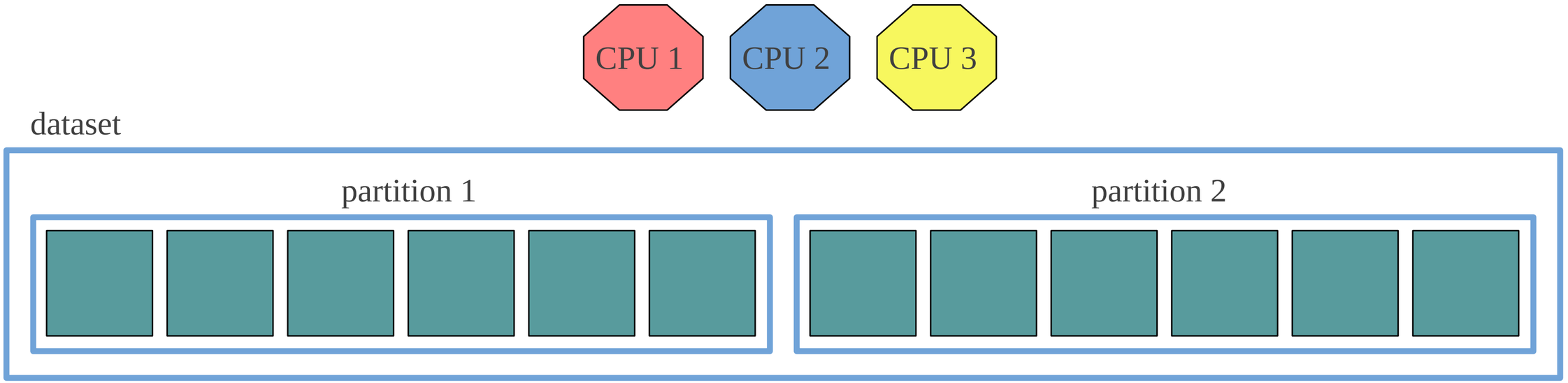
Sequential iteration



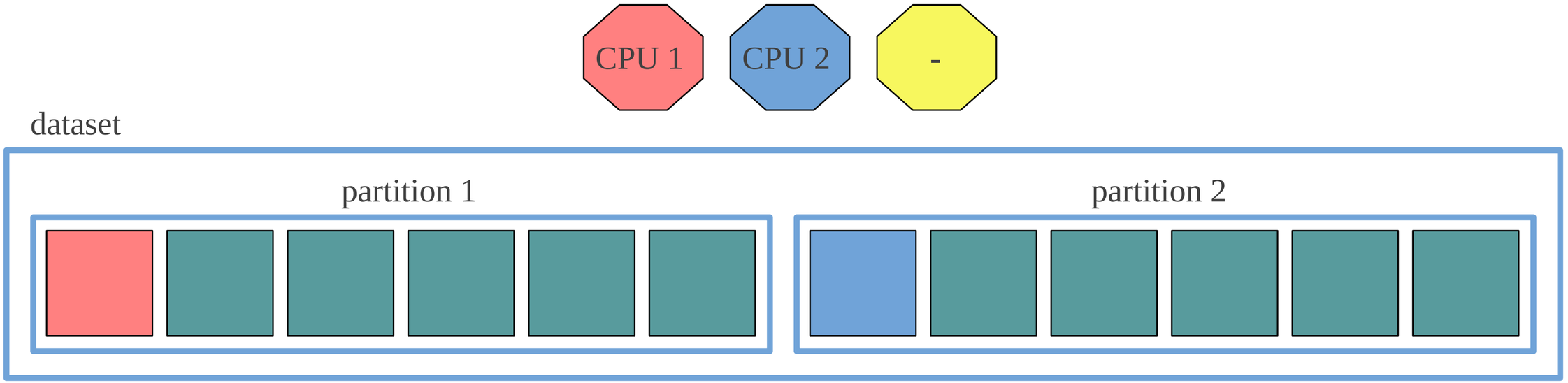
Useful work



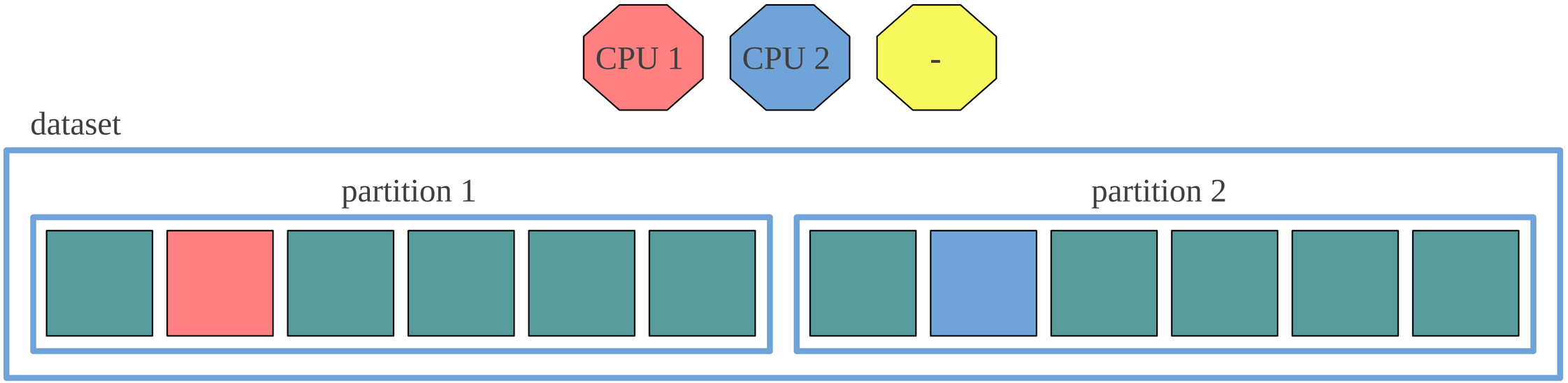
Partition the dataset



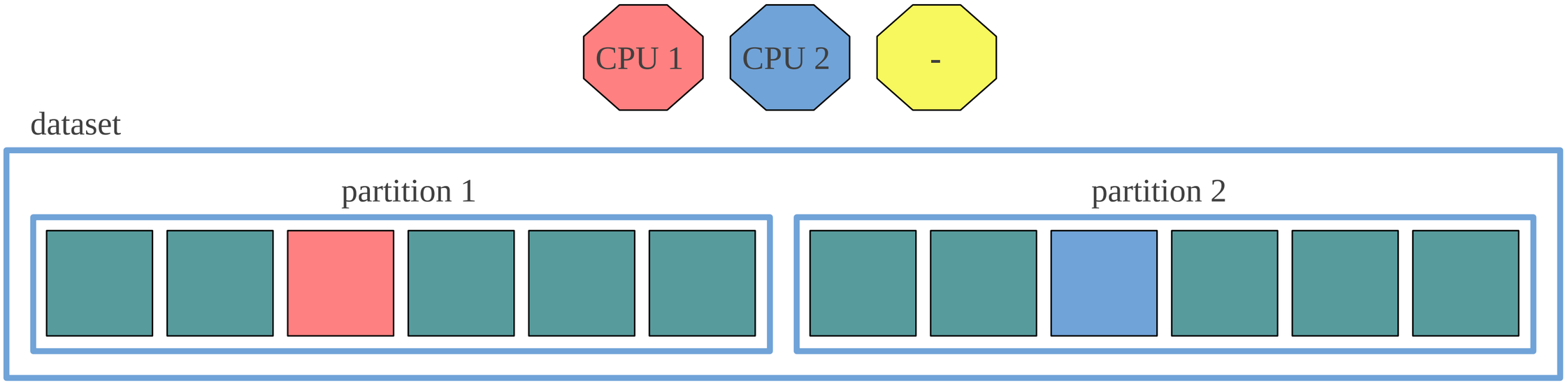
Partition the dataset



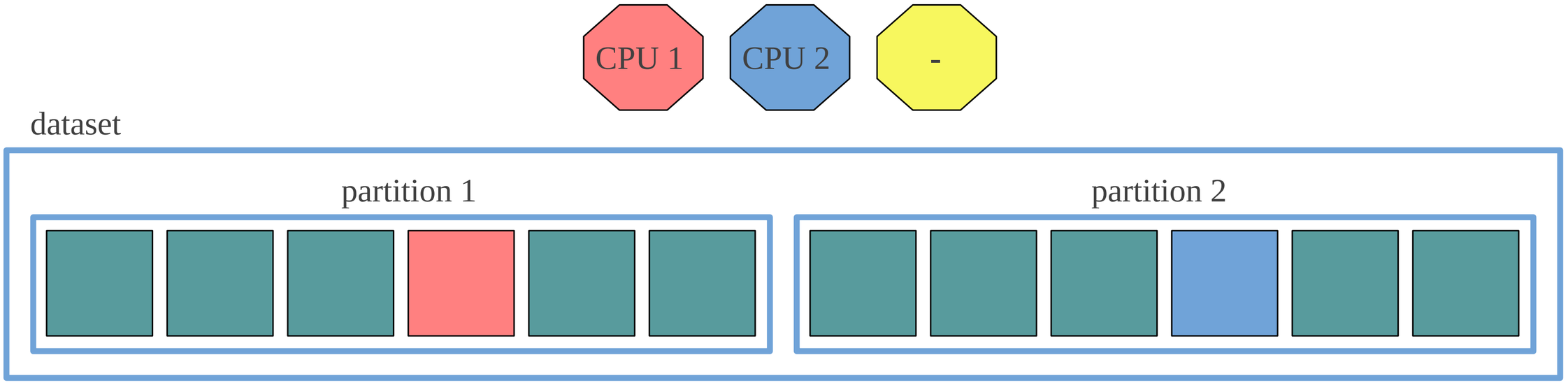
Partition the dataset



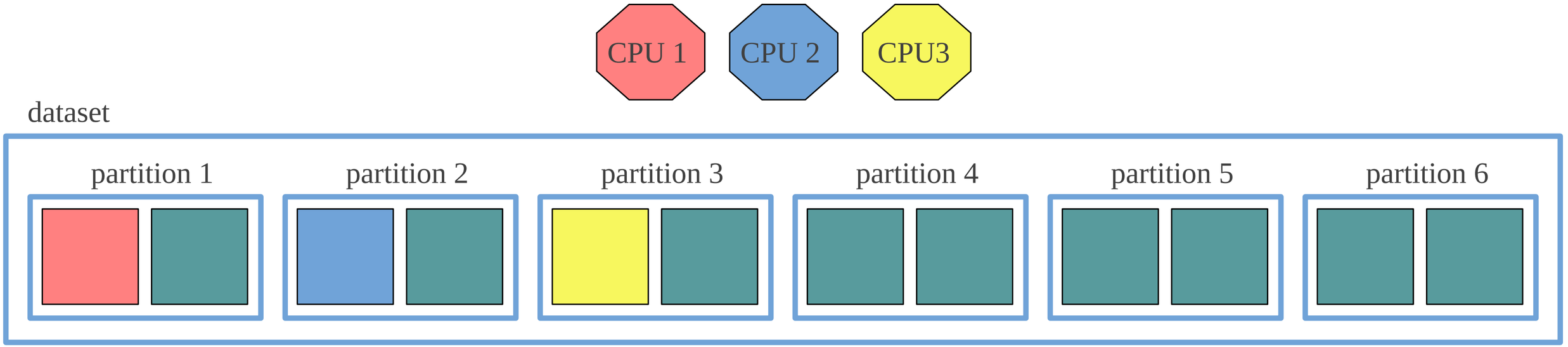
Partition the dataset



Partition the dataset

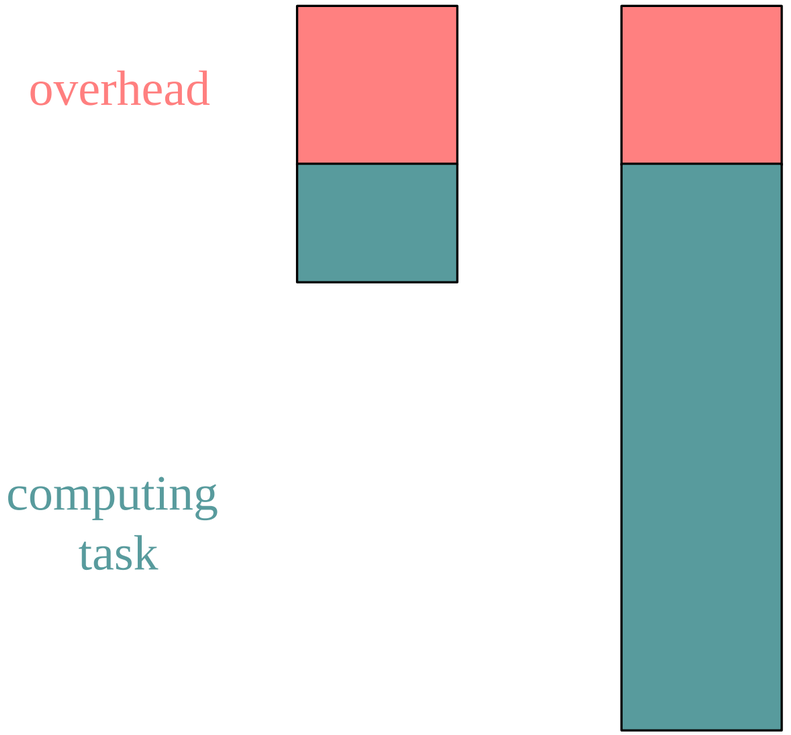


Partition the dataset

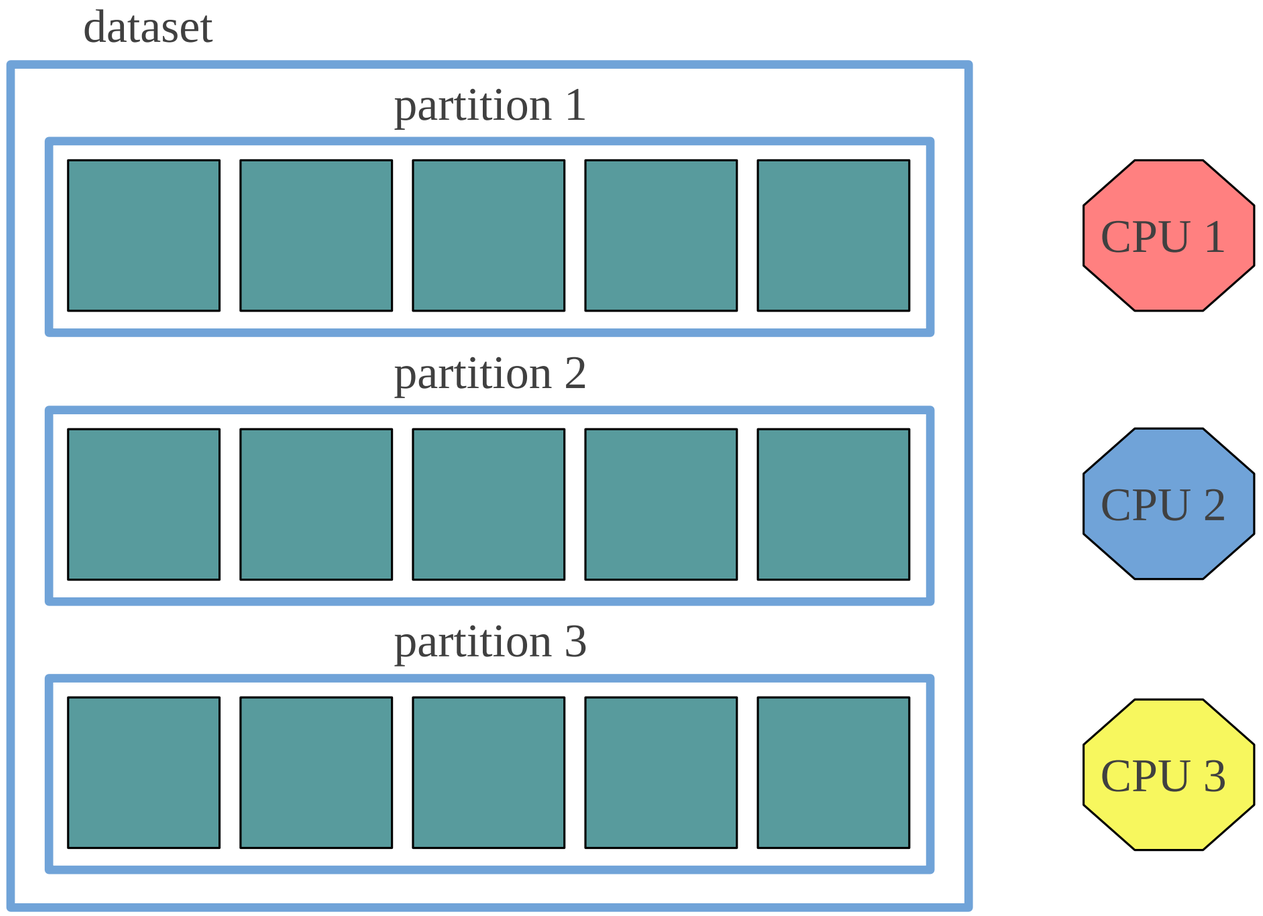


Important points

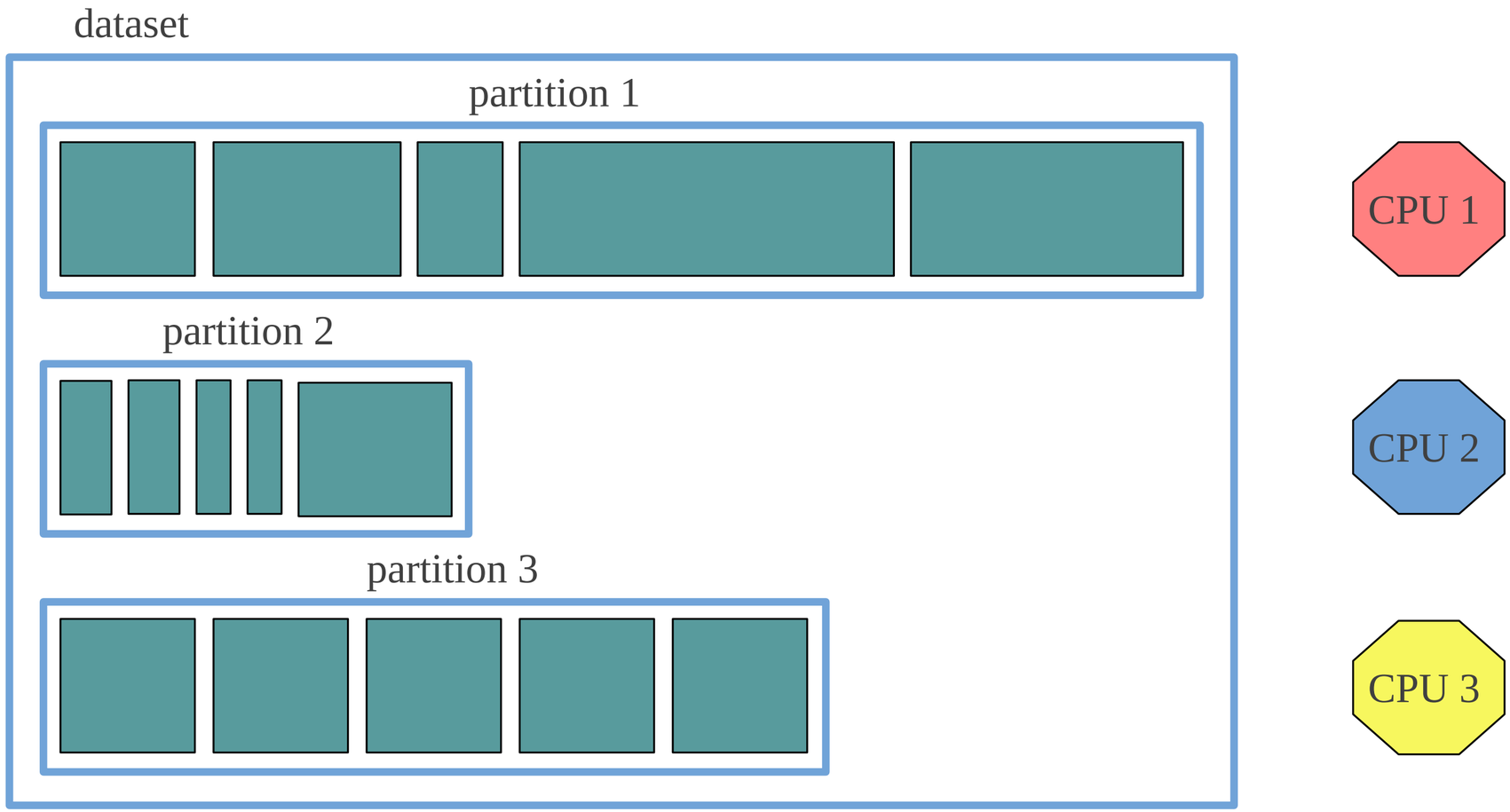
1. Size the partitions appropriately



Tasks are not always as demanding



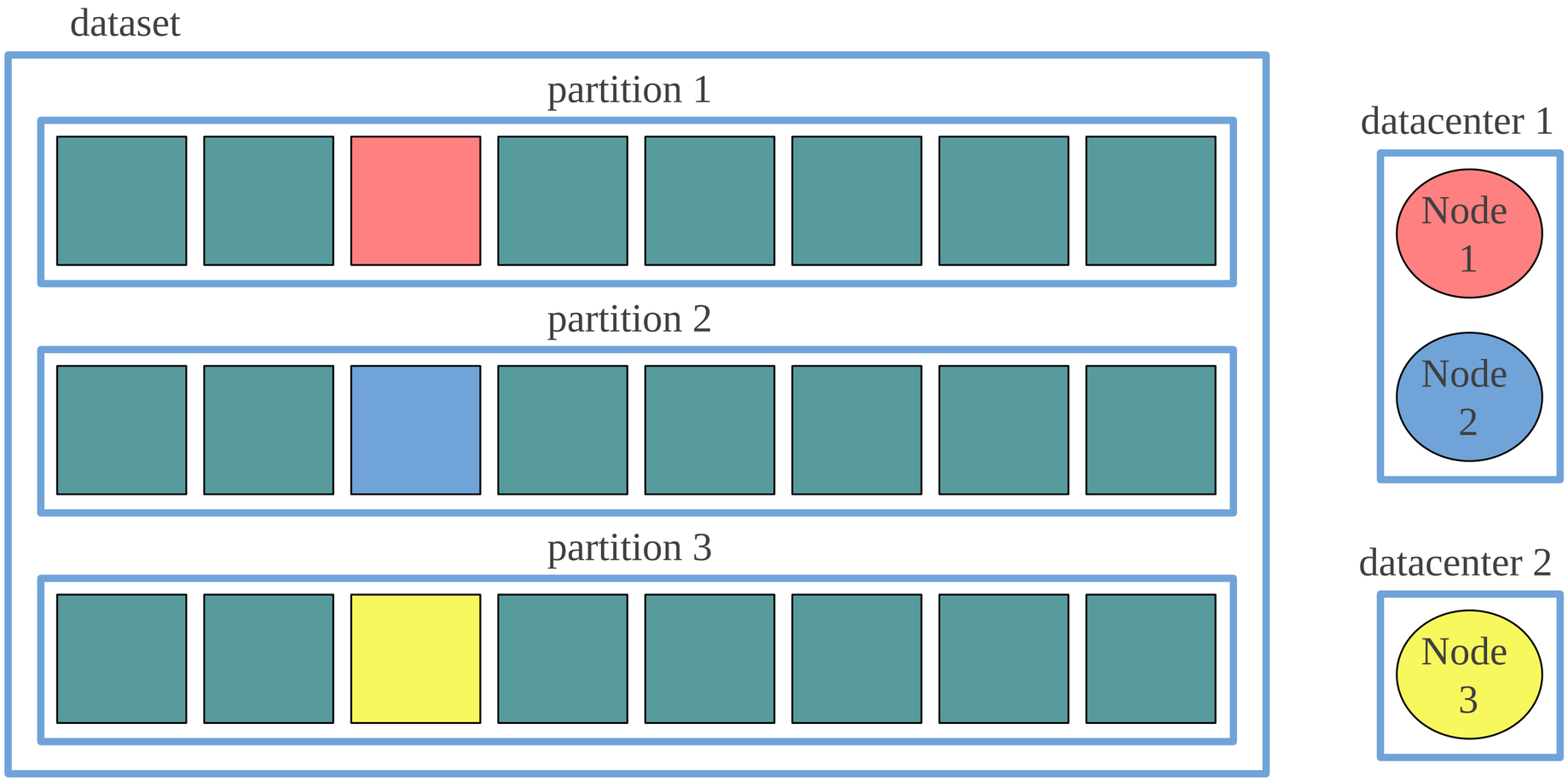
Tasks are not always as demanding



Important points

1. Size the partitions appropriately
2. Benchmark and tweak configuration for the task at hand

Scales to more than one computer

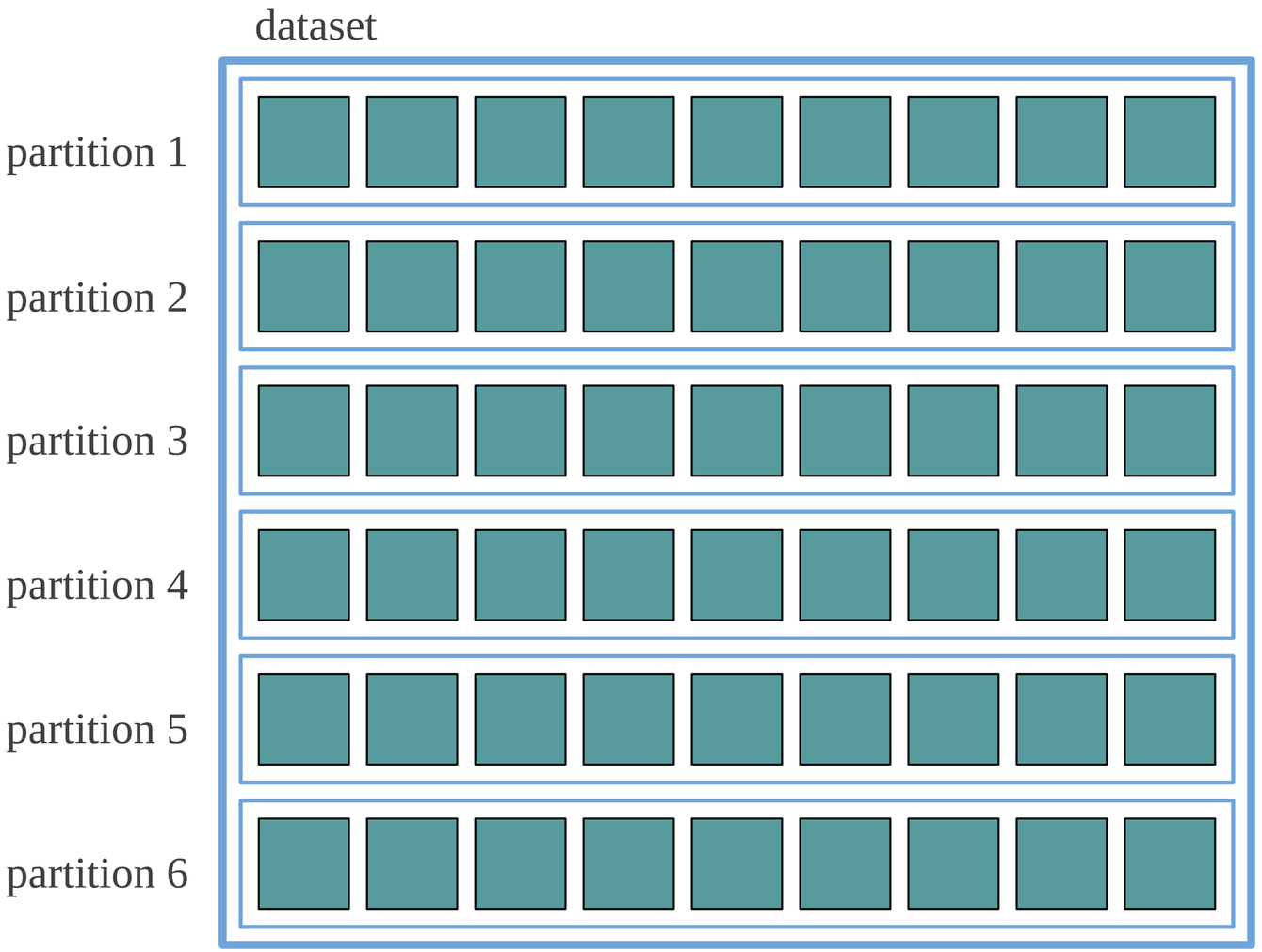


Important points

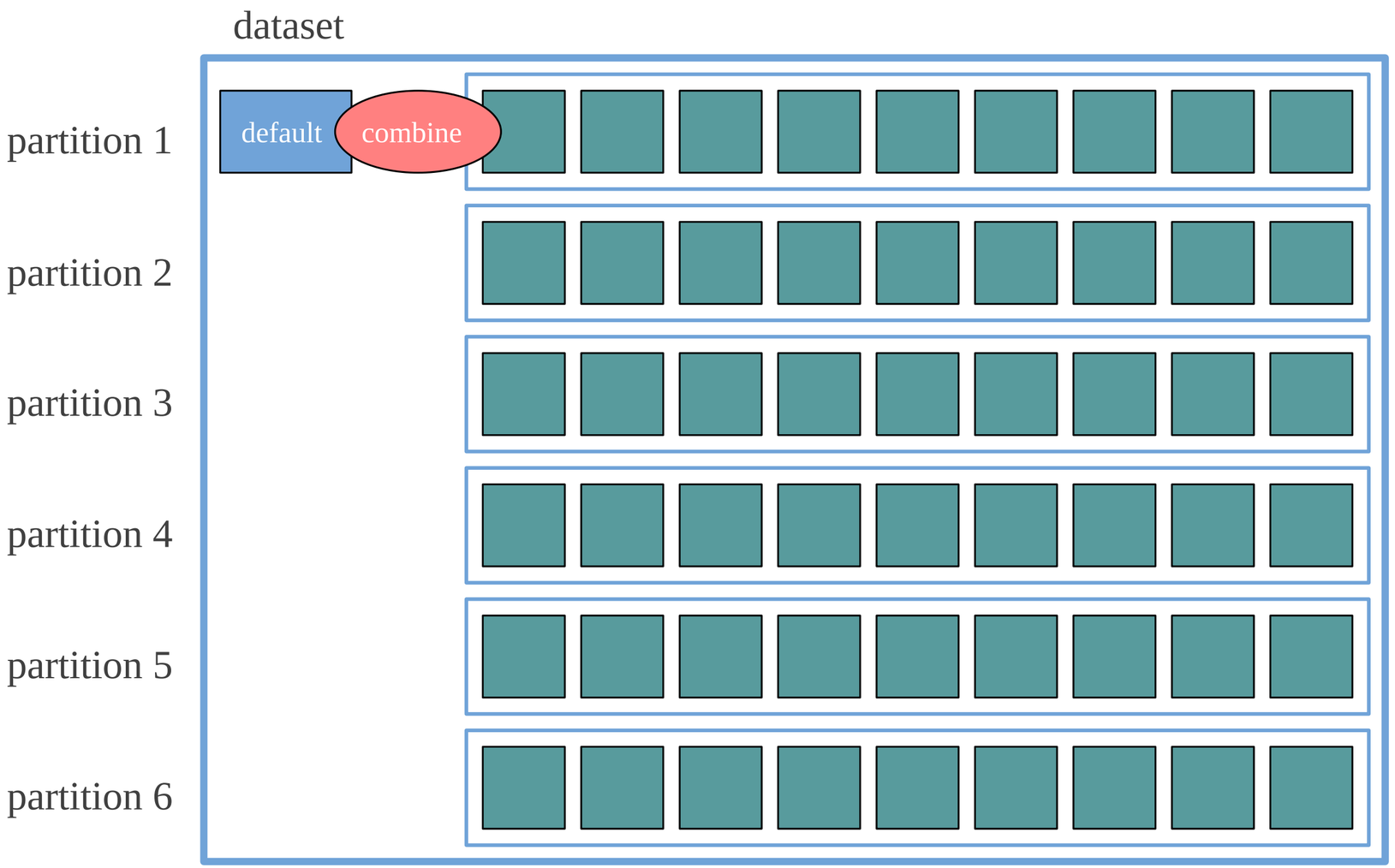
1. Size the partitions appropriately
2. Benchmark and tweak configuration for the task at hand
3. Parallel process must produce the same result as the sequential one

TemperatureNotebook.scala

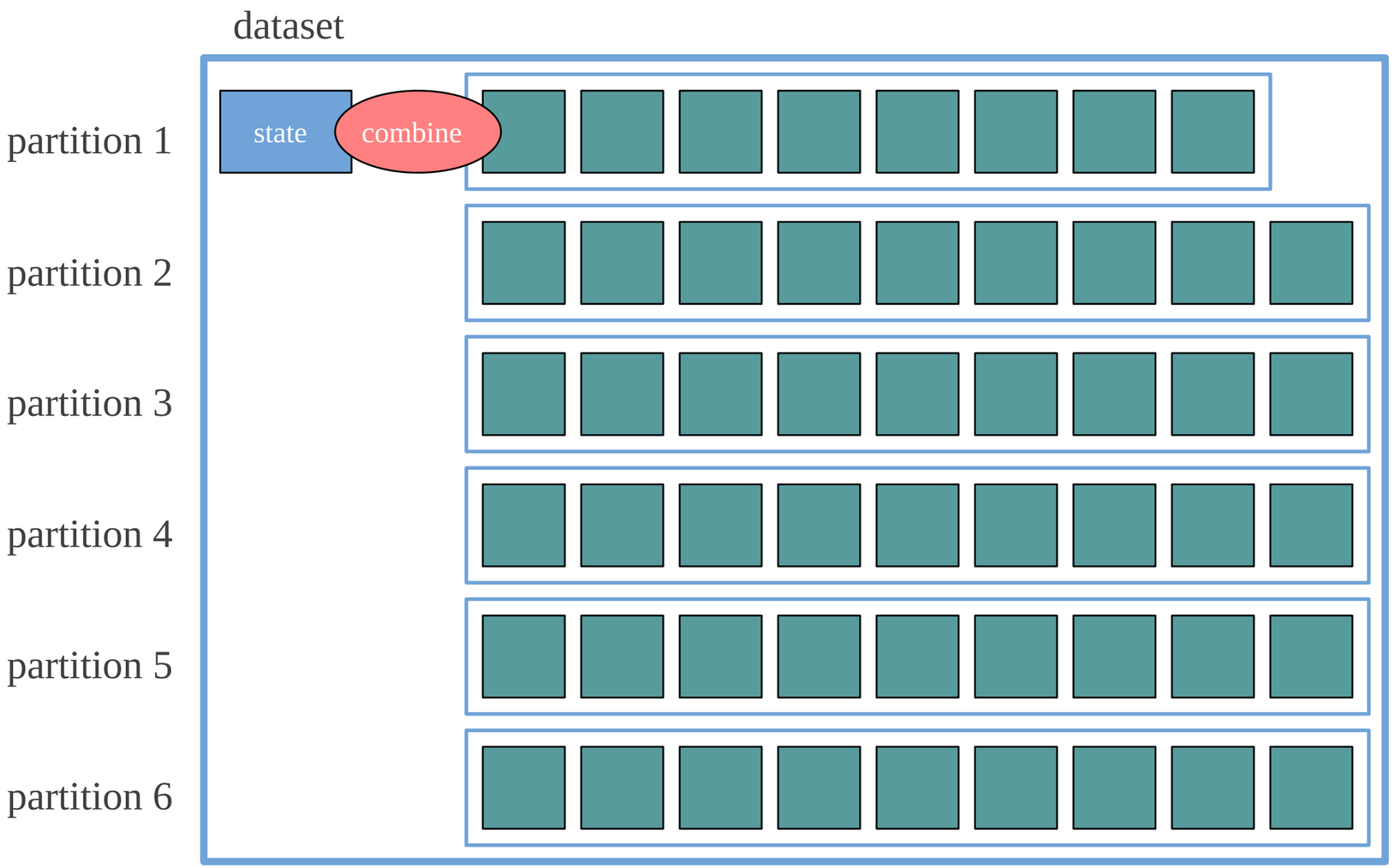
foldLeft


dataset
partition 1
partition 2
partition 3
partition 4
partition 5
partition 6

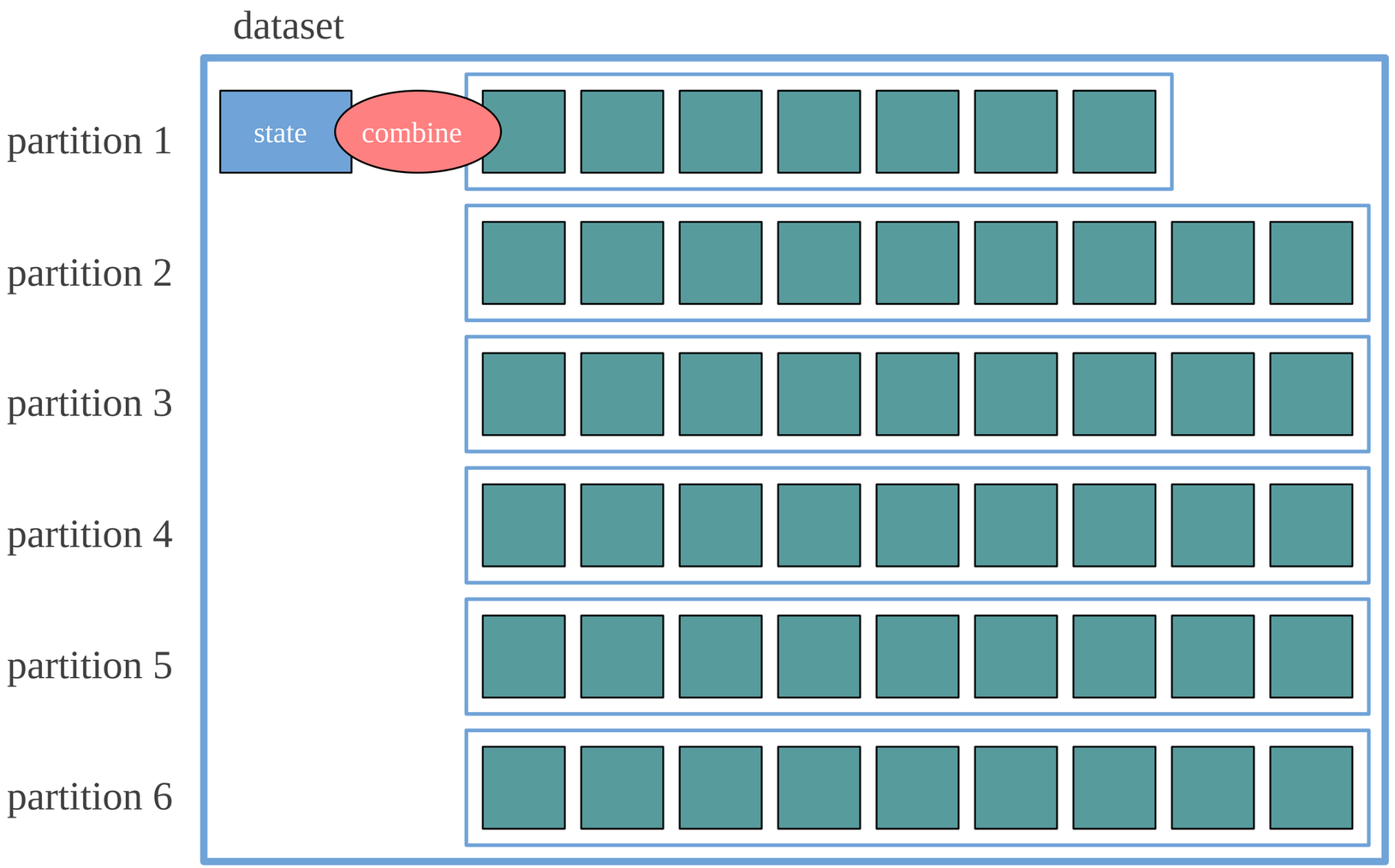
foldLeft per partition



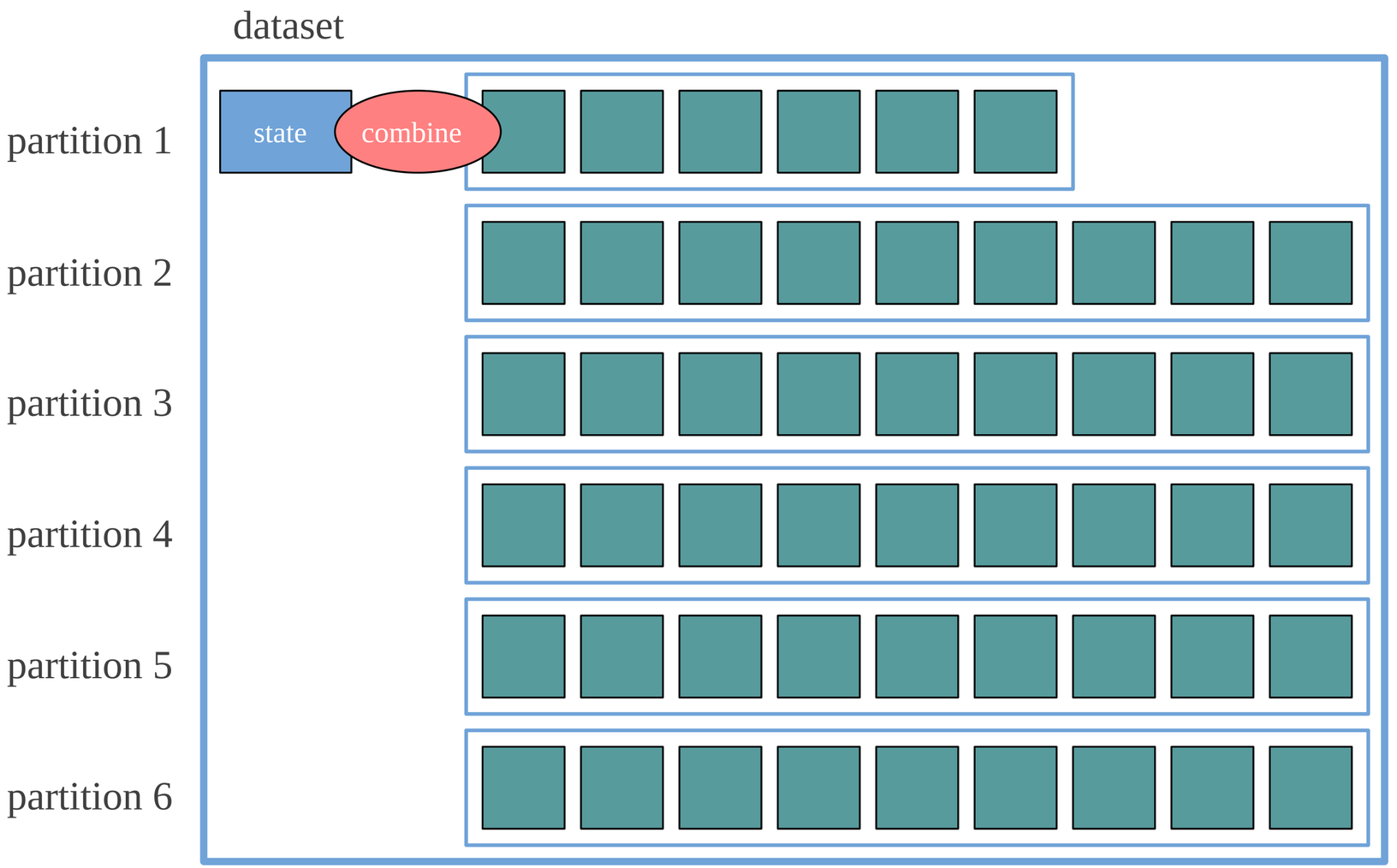
foldLeft per partition



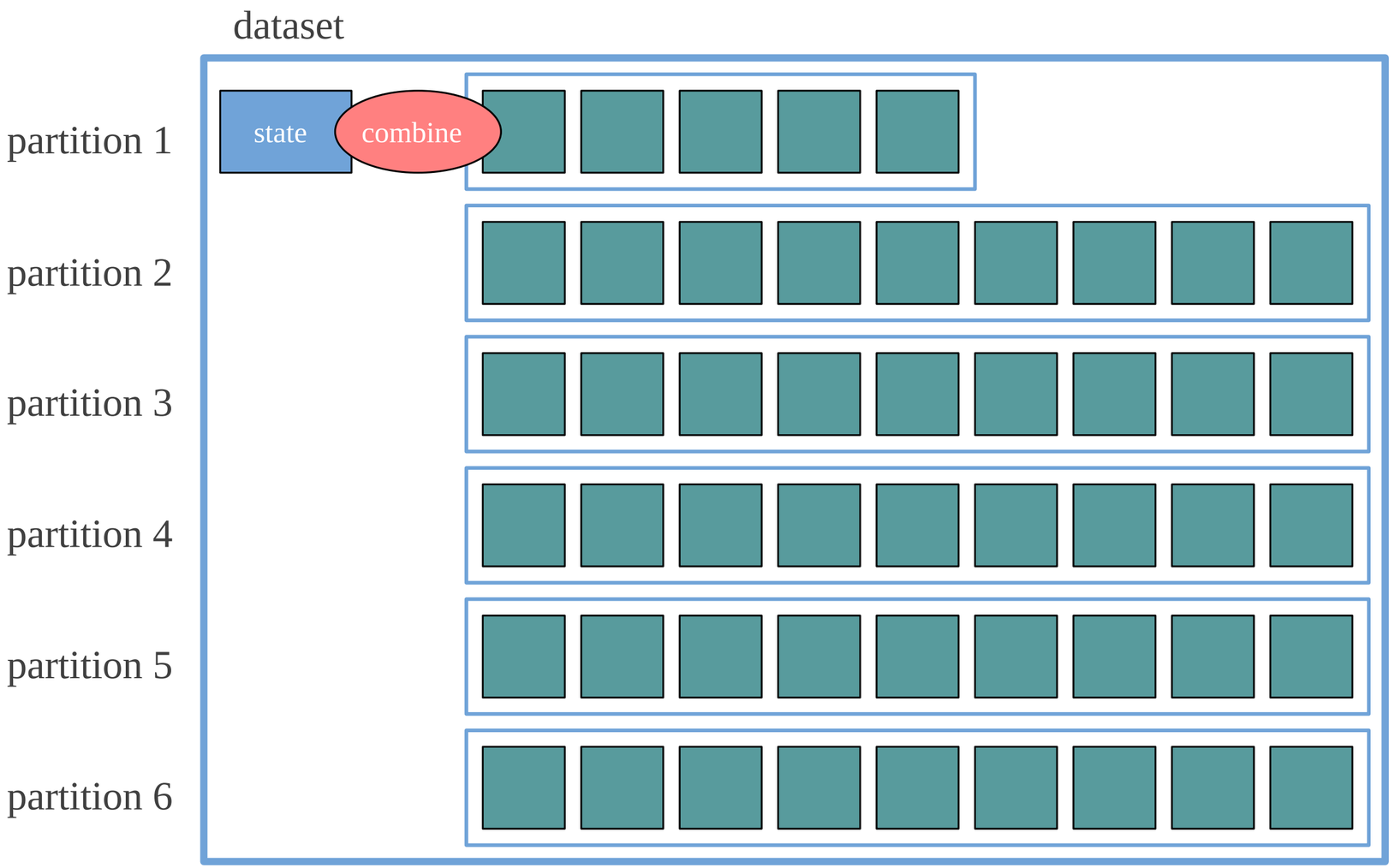
foldLeft per partition



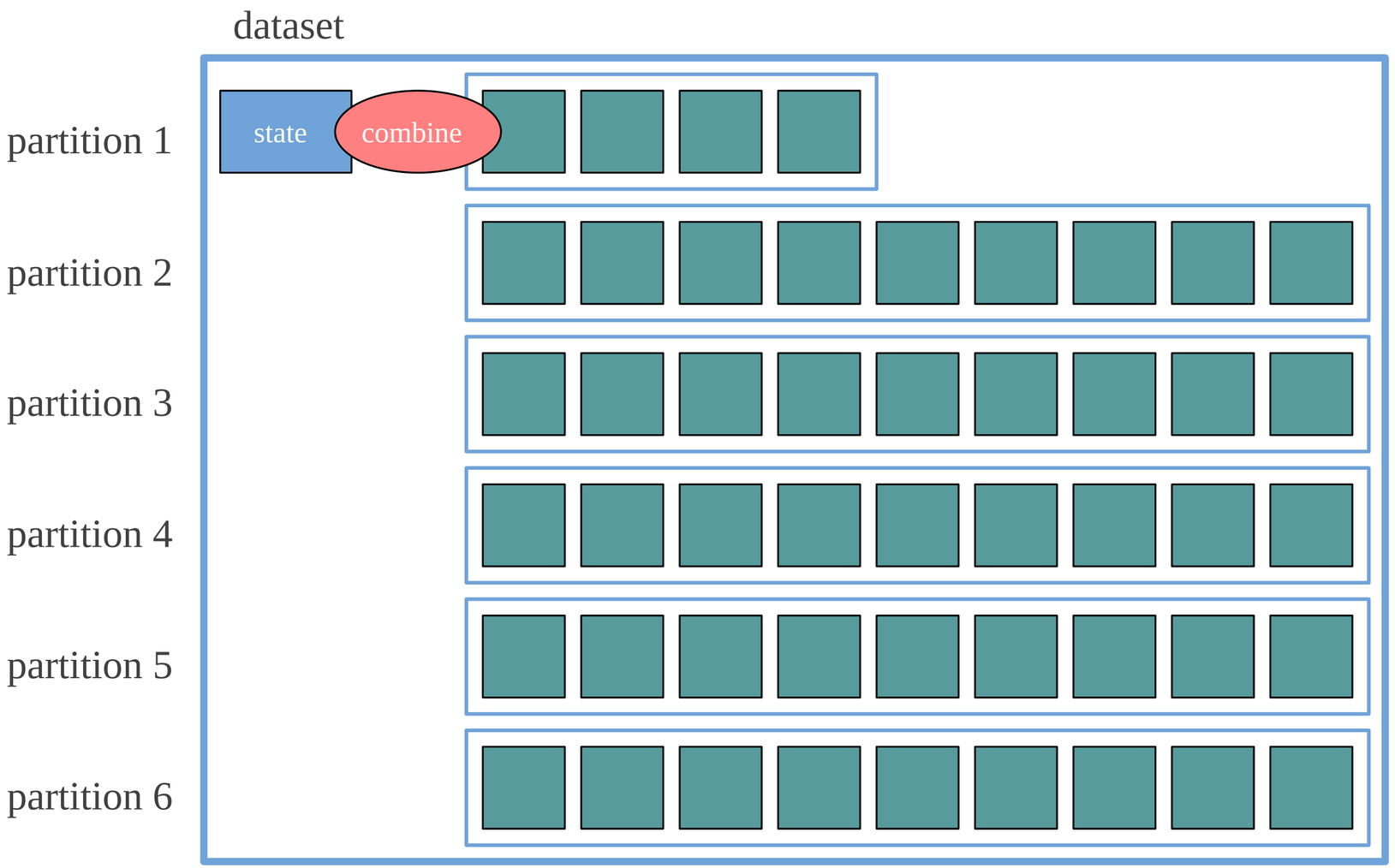
foldLeft per partition



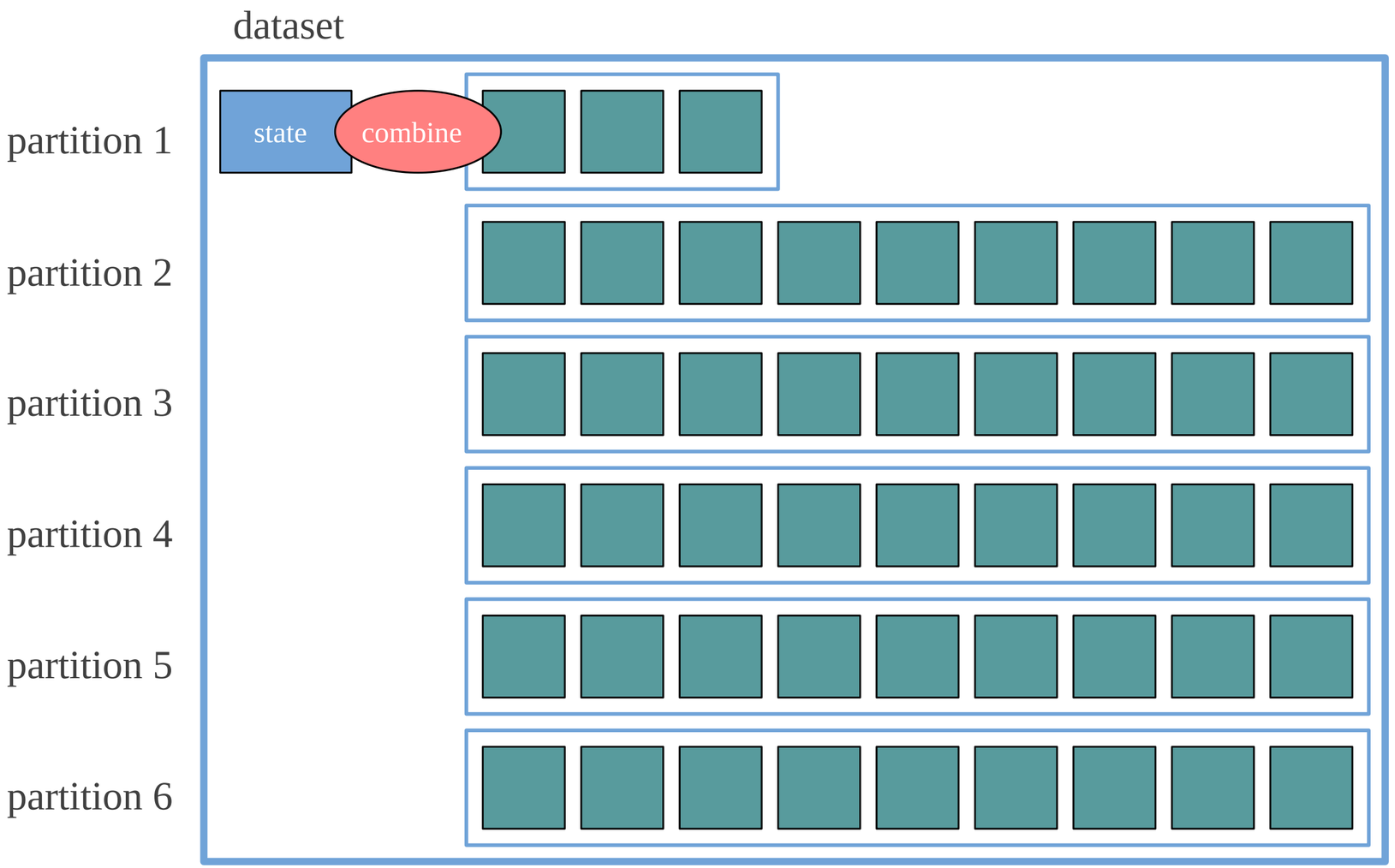
foldLeft per partition



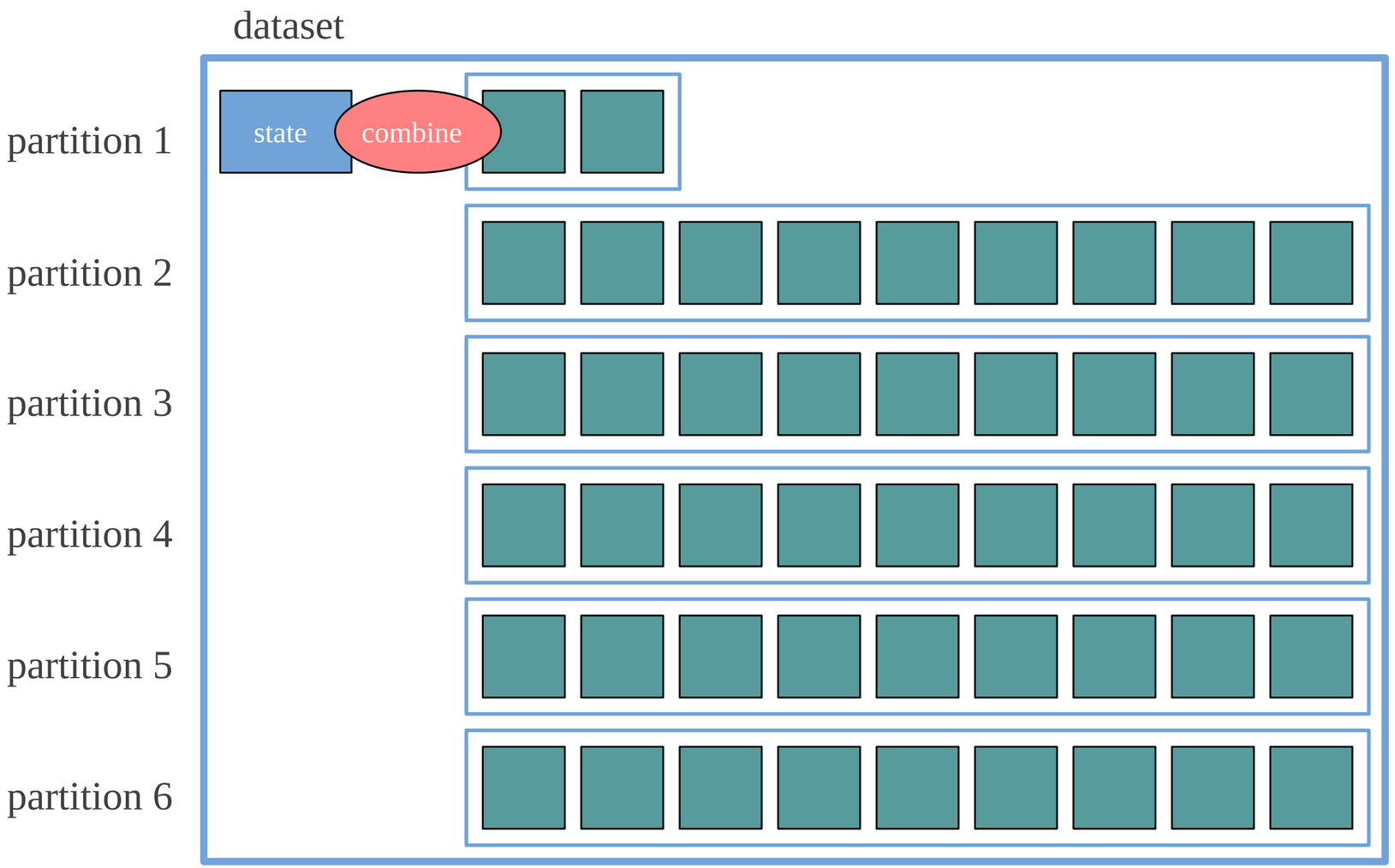
foldLeft per partition



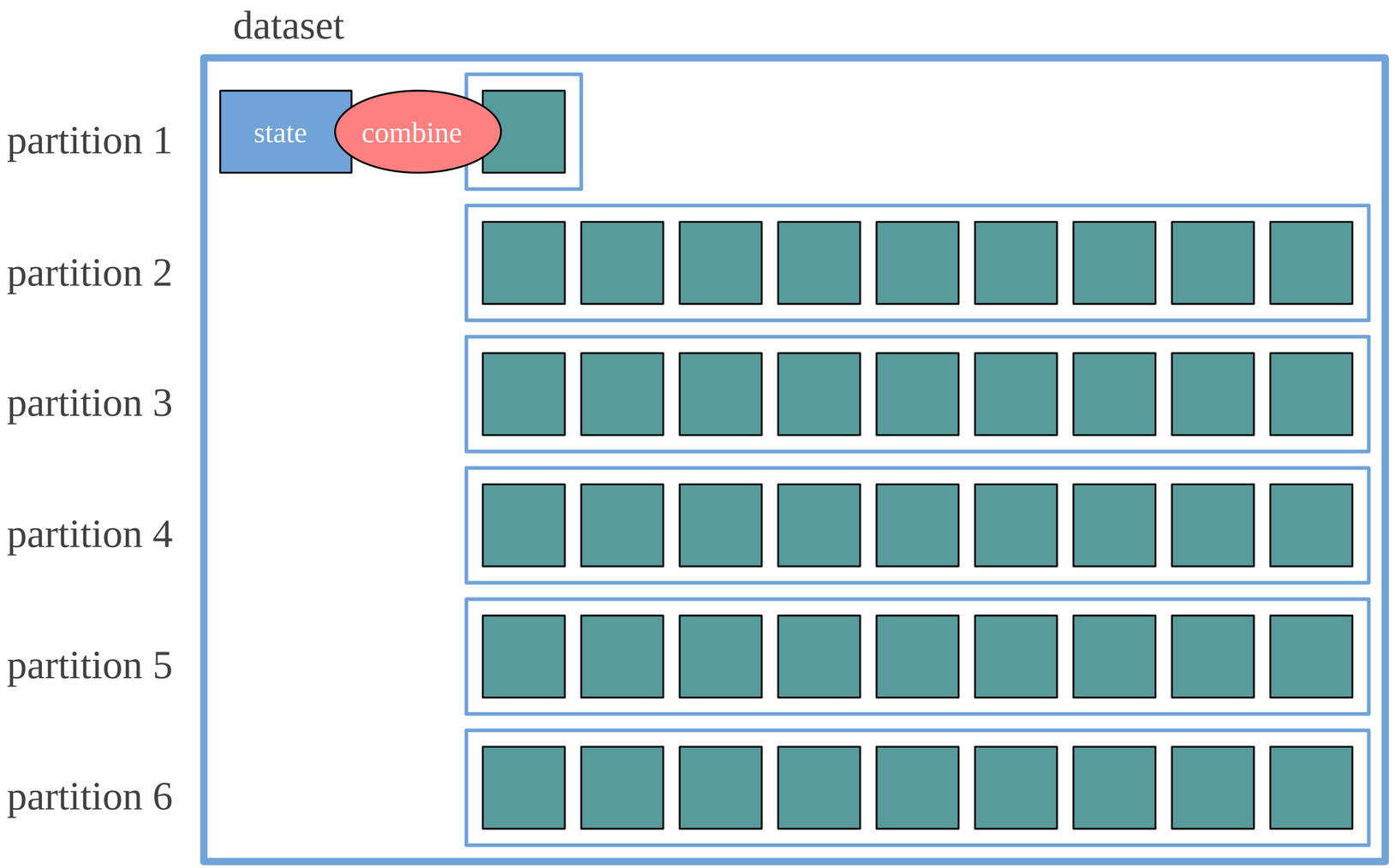
foldLeft per partition



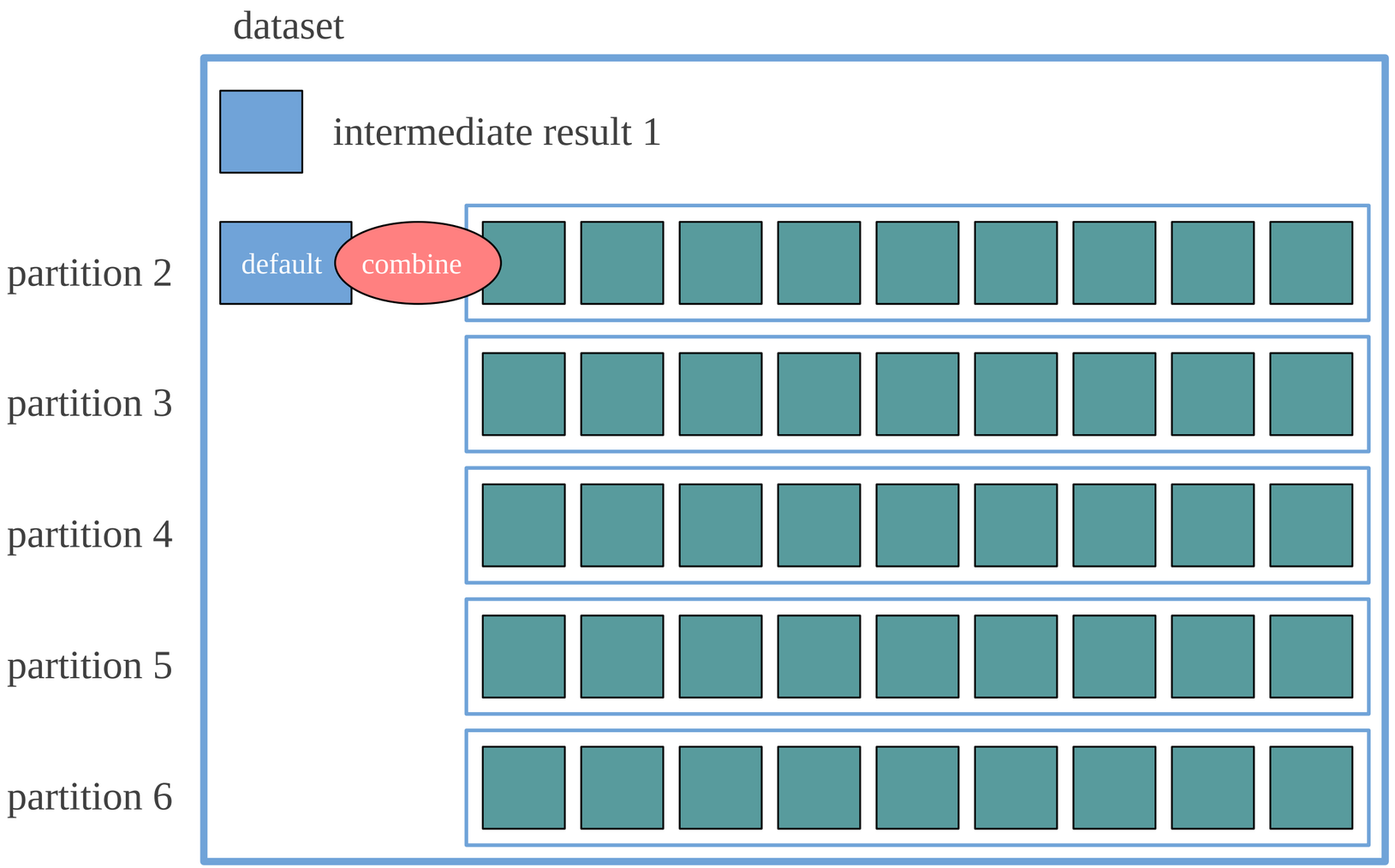
foldLeft per partition



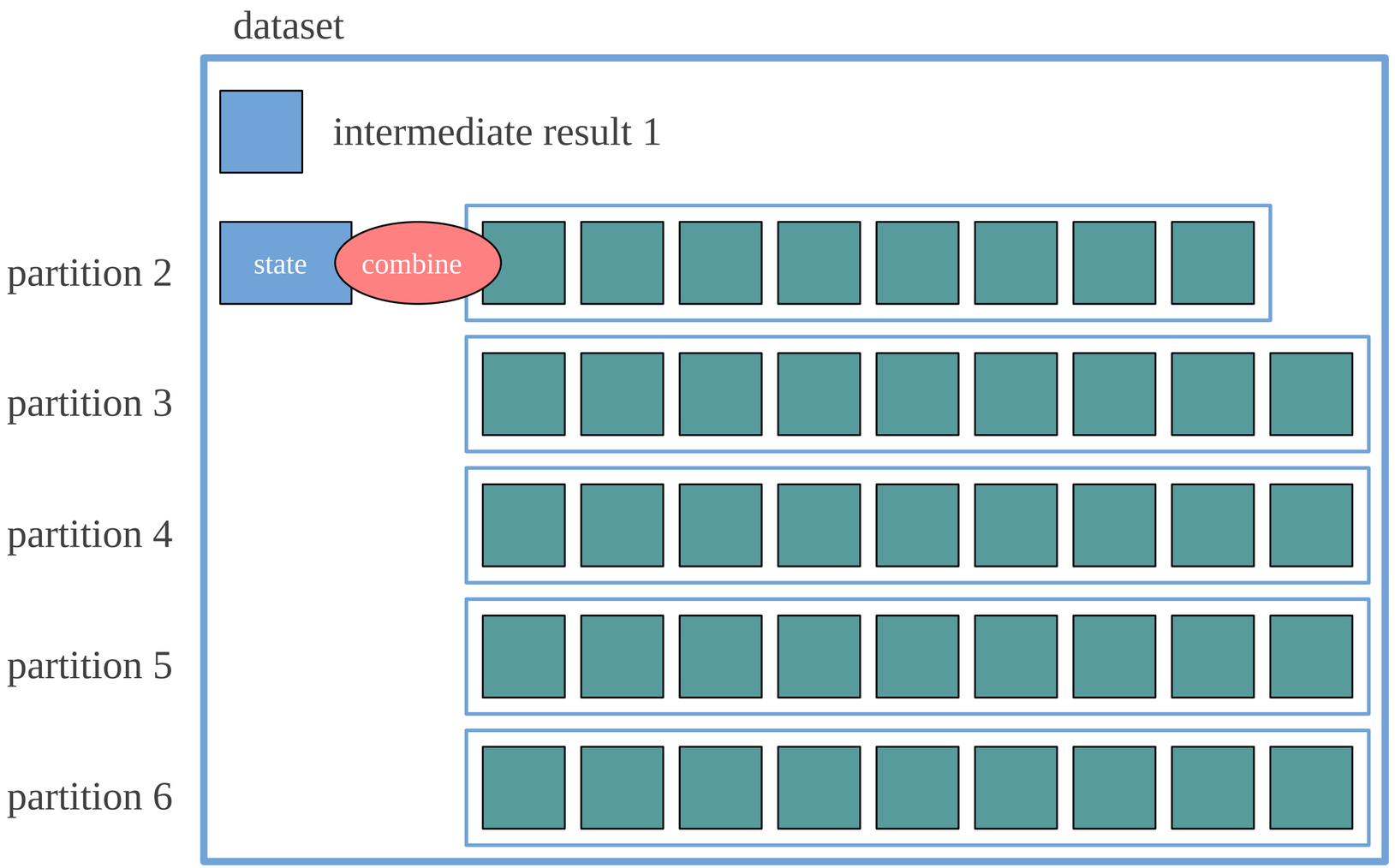
foldLeft per partition



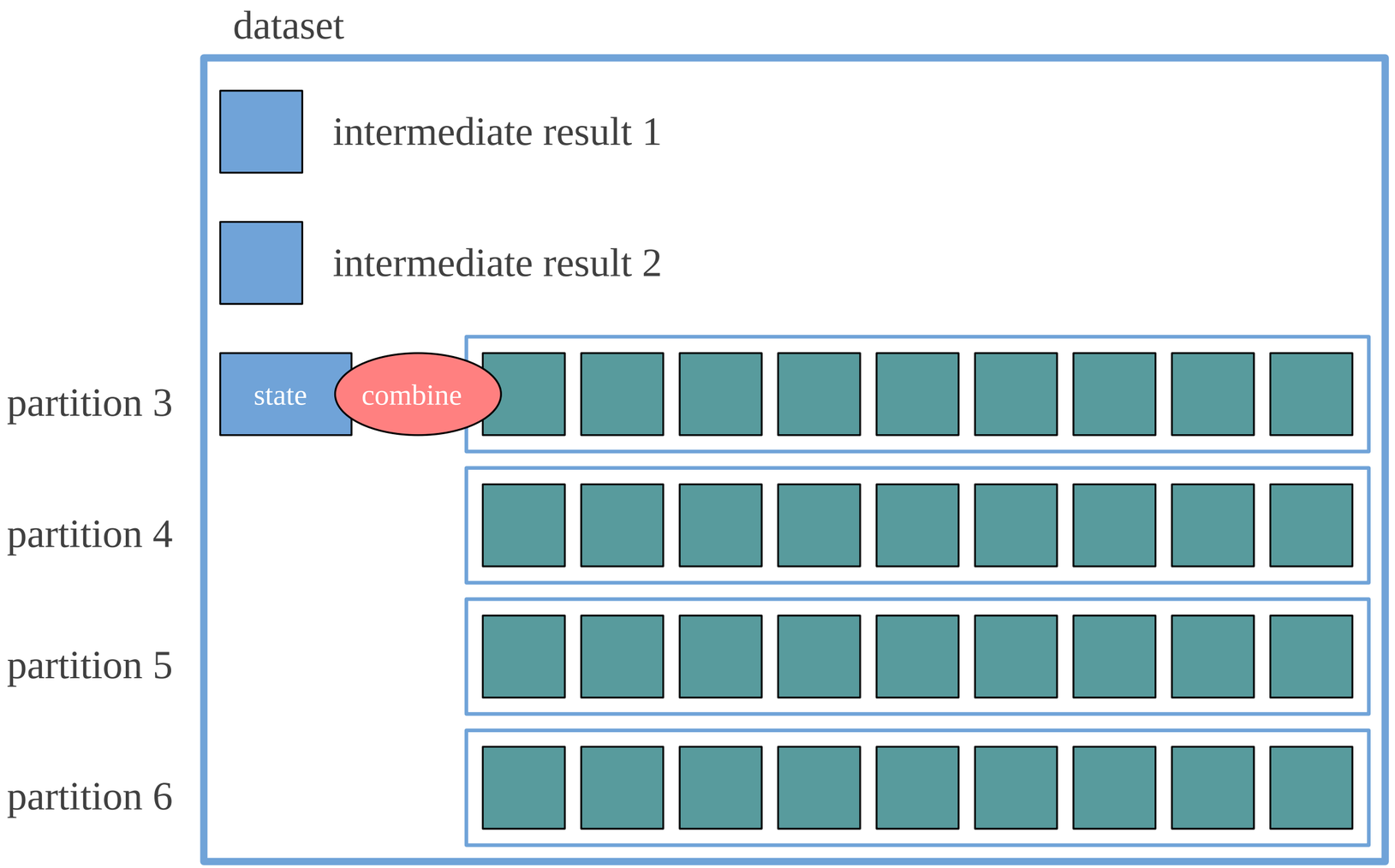
foldLeft per partition



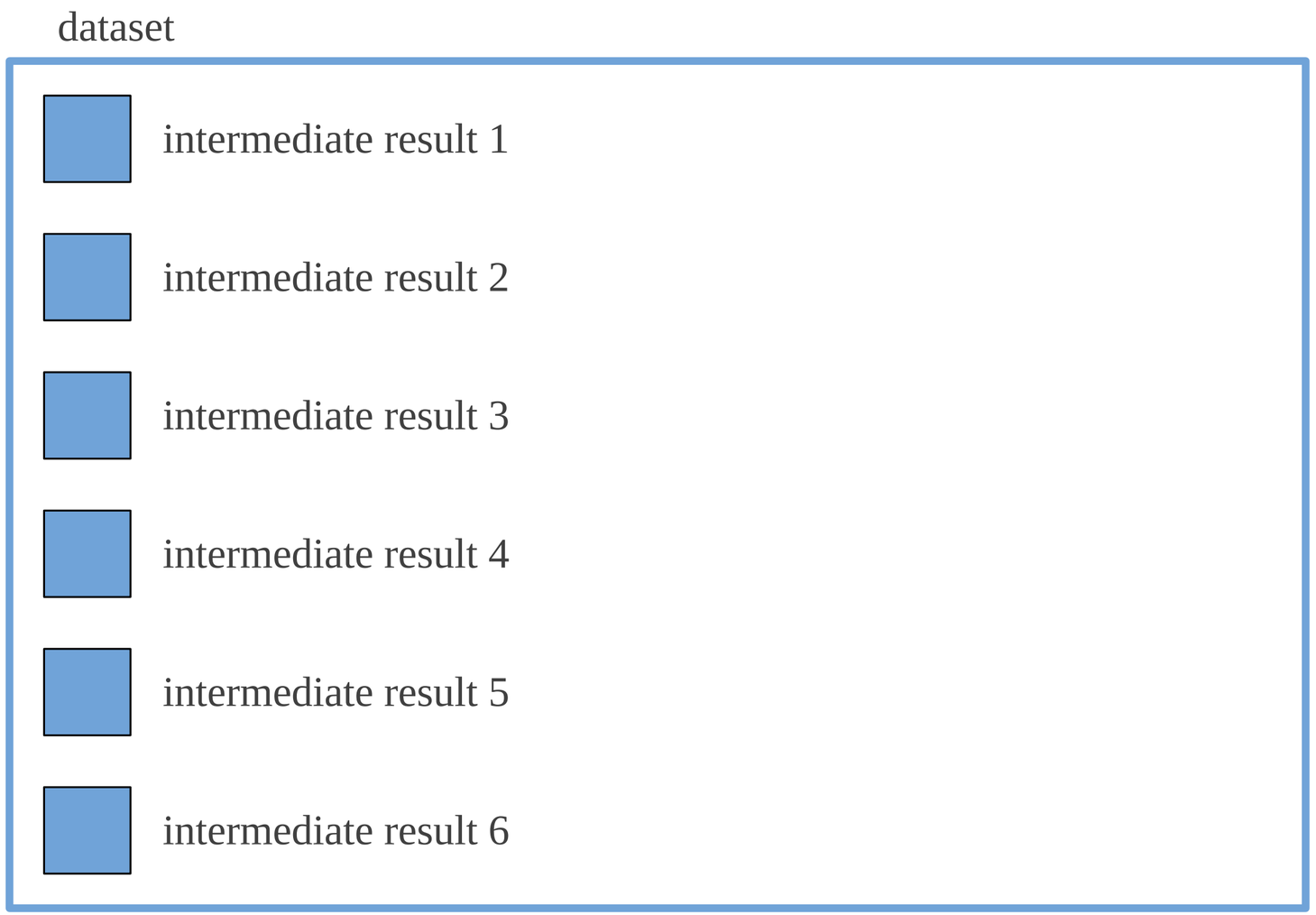
foldLeft per partition



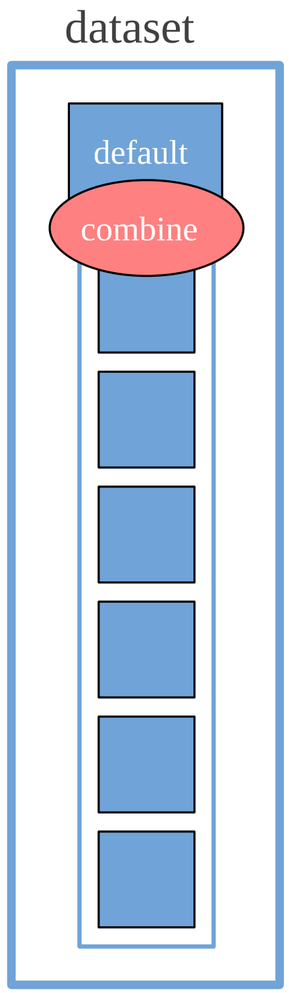
foldLeft per partition



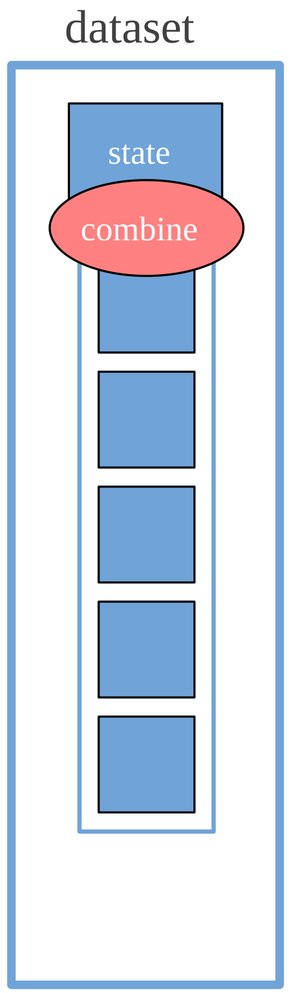
All partitions folded



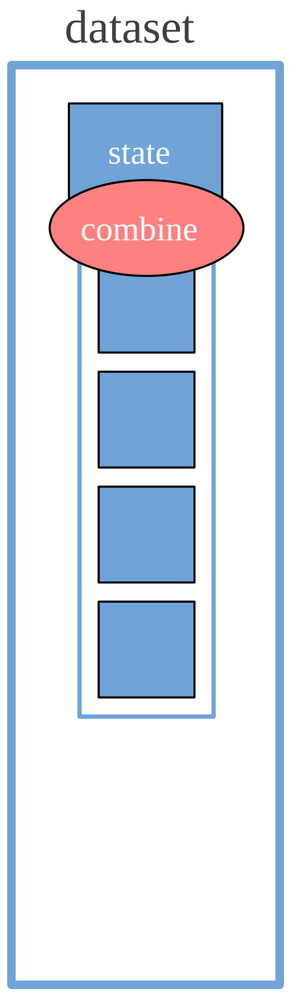
foldLeft intermediate results



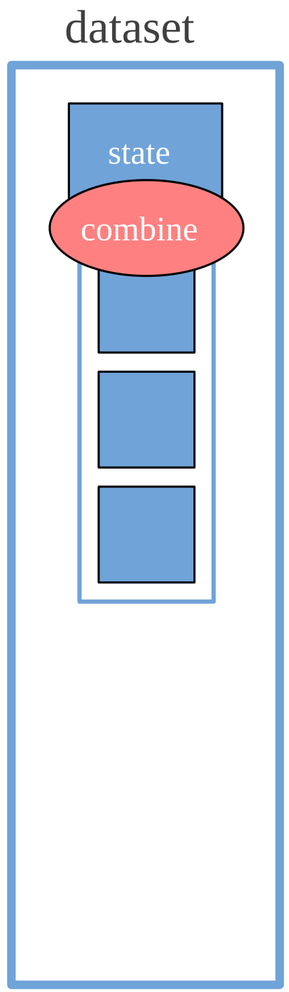
foldLeft intermediate results



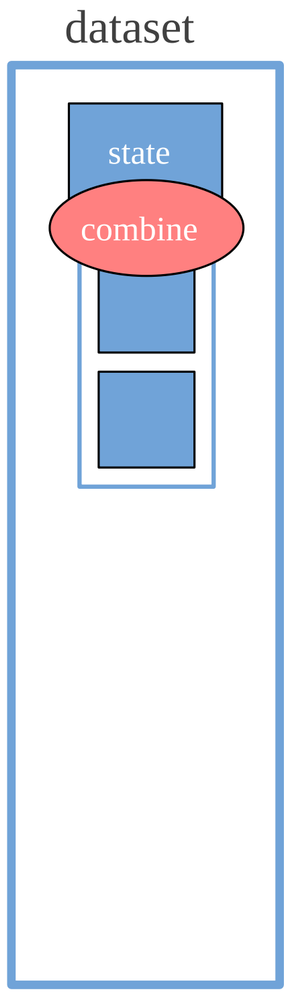
foldLeft intermediate results



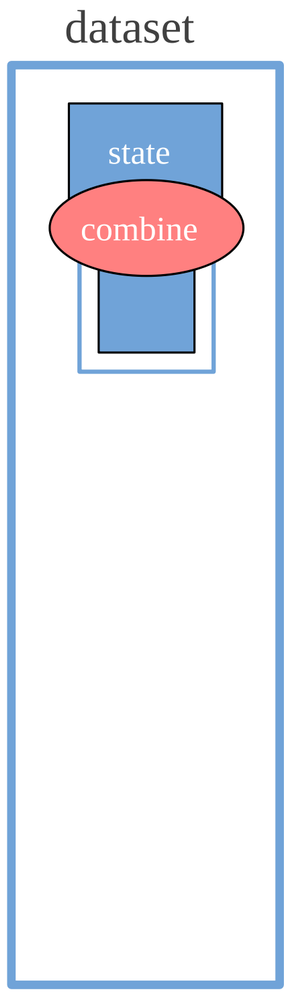
foldLeft intermediate results



foldLeft intermediate results



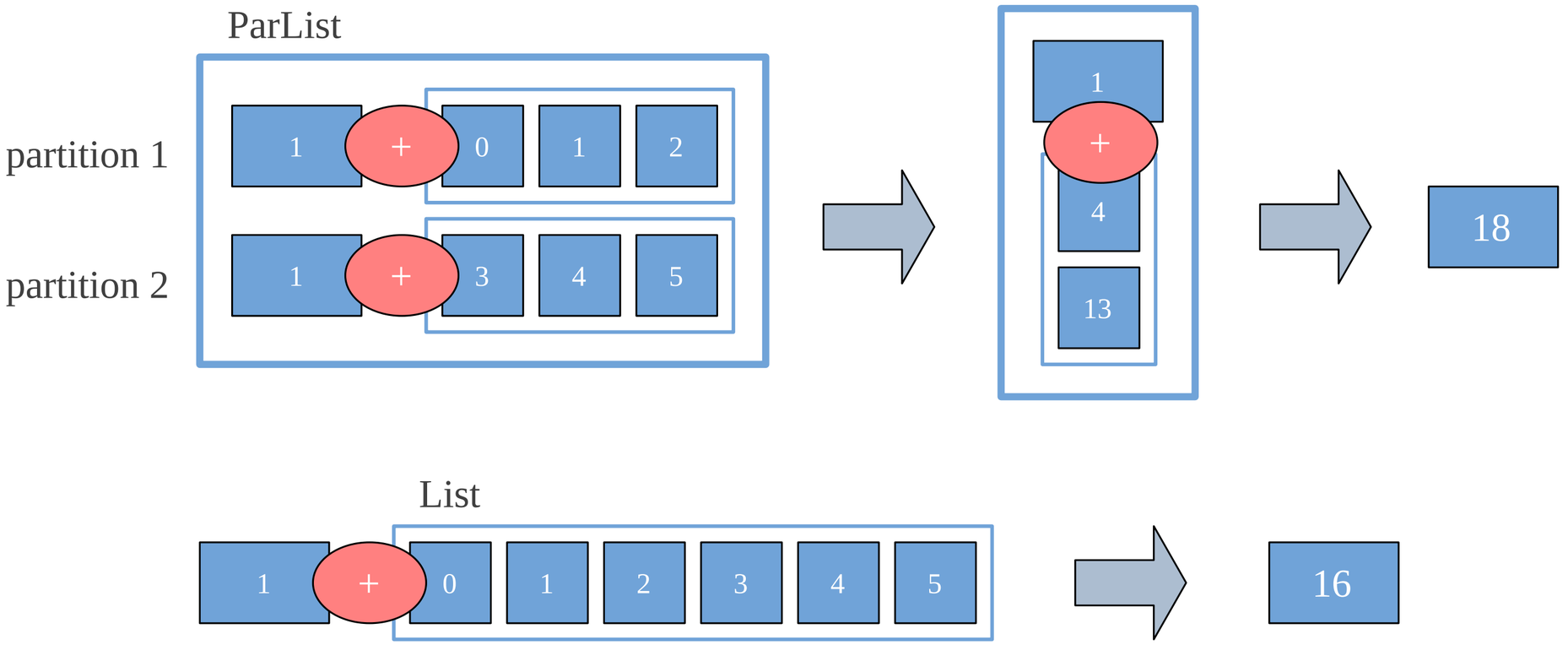
foldLeft intermediate results



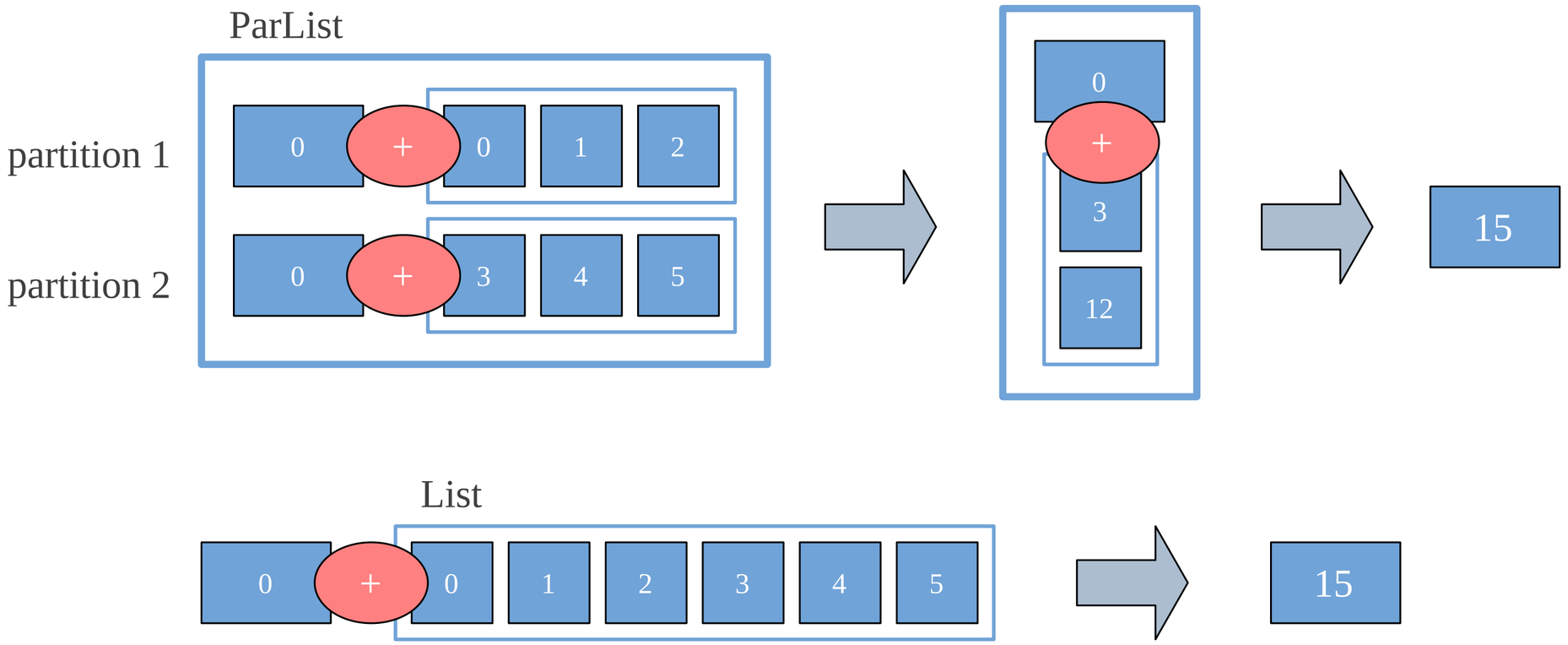
foldLeft

TemperatureExercises.scala

monoFoldLeft vs List foldLeft



monoFoldLeft vs List foldLeft



combine(default, x) == x

combine(default, x) == x == combine(x, default)

```
average(10, 12) = ???
```

```
average(10, 12) = (10 + 12) / 2
                = 11
```

```
average(10, 12, 14) = (10 + 12 + 14) / 3
                    = 12
```

```
average(average(10, 12), 14) = average(11, 14)
                             = (11 + 14) / 2
                             = 12.5
```

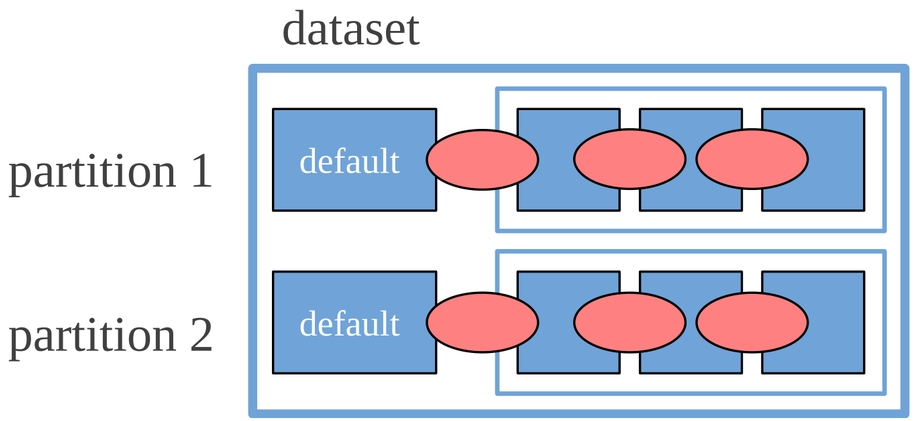
```
average(10, average(12, 14)) = average(10, 13)
                             = (10 + 13) / 2
                             = 11.5
```

```
average(10, 12, 14) = 12
```

Associative functions

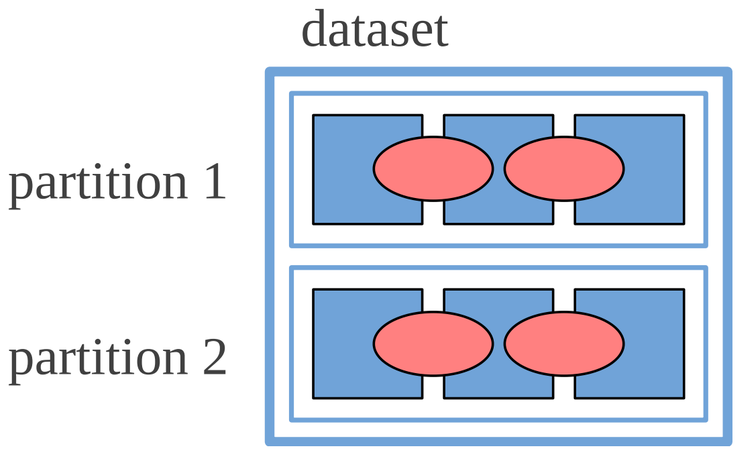
```
(1 + (2 + 3)) == ((1 + 2) + 3)
// res0: Boolean = true

(1 min (2 min 3)) == ((1 min 2) min 3)
// res1: Boolean = true
```

monoFoldLeft



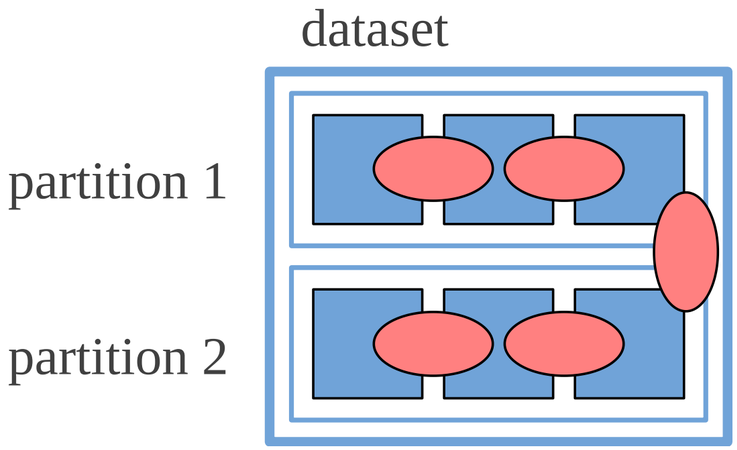

```
(default combine a1 combine a2 combine a3)
(default combine b1 combine b2 combine b3)
```

monoFoldLeft



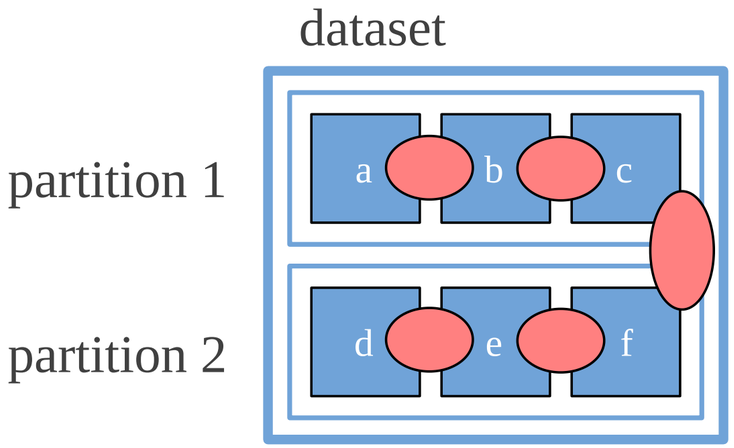

```
(a1 combine a2 combine a3)
(b1 combine b2 combine b3)
```

monoFoldLeft

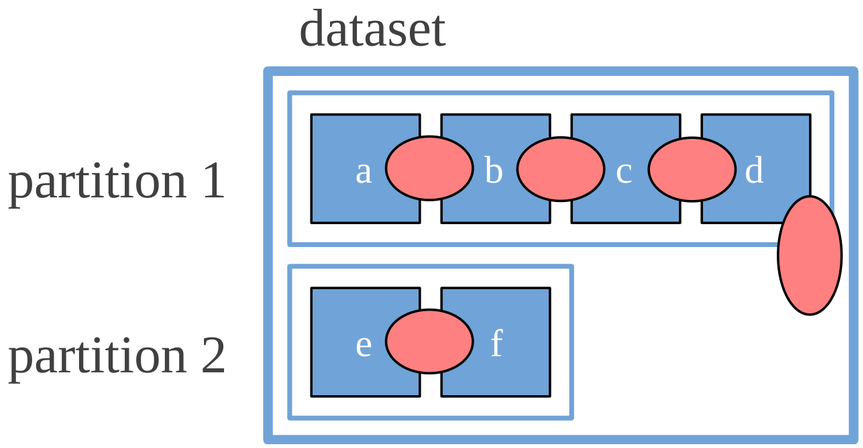



```
(a1 combine a2 combine a3) combine
(b1 combine b2 combine b3)
```

monoFoldLeft requires combine to be associative

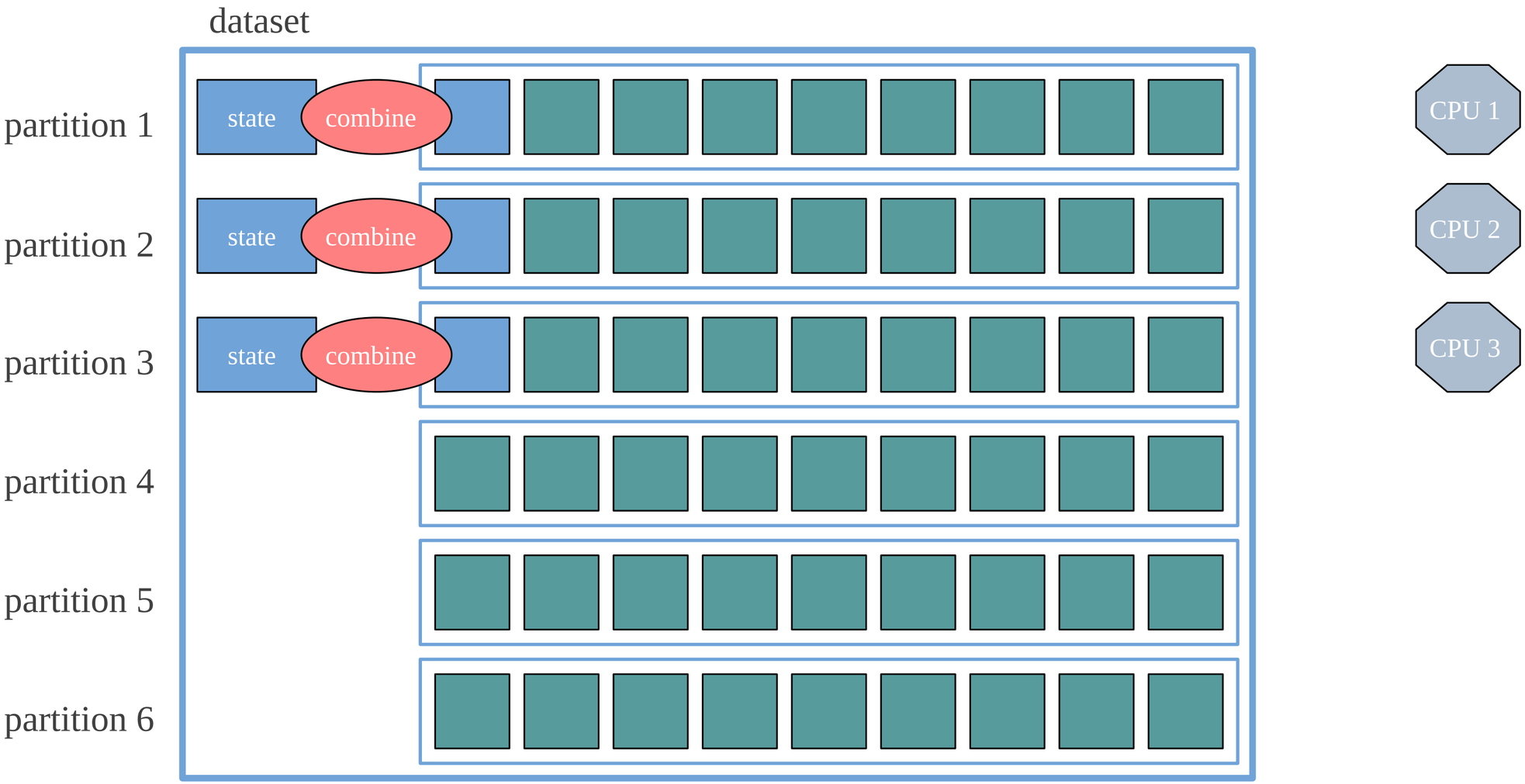



```
(a combine b combine c) combine
(d combine e combine f)
```

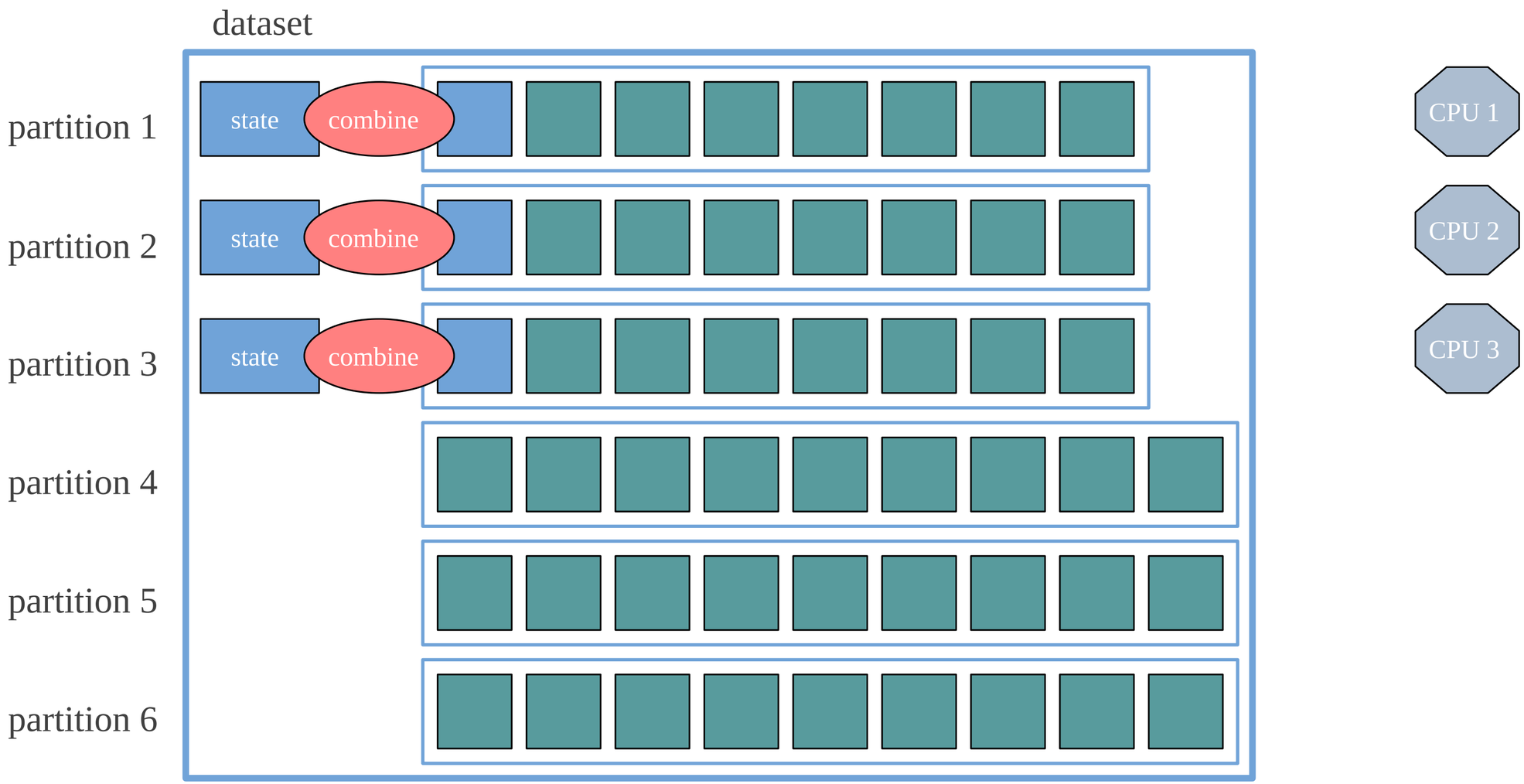



```
(a combine b combine c combine d) combine
(e combine f)
```

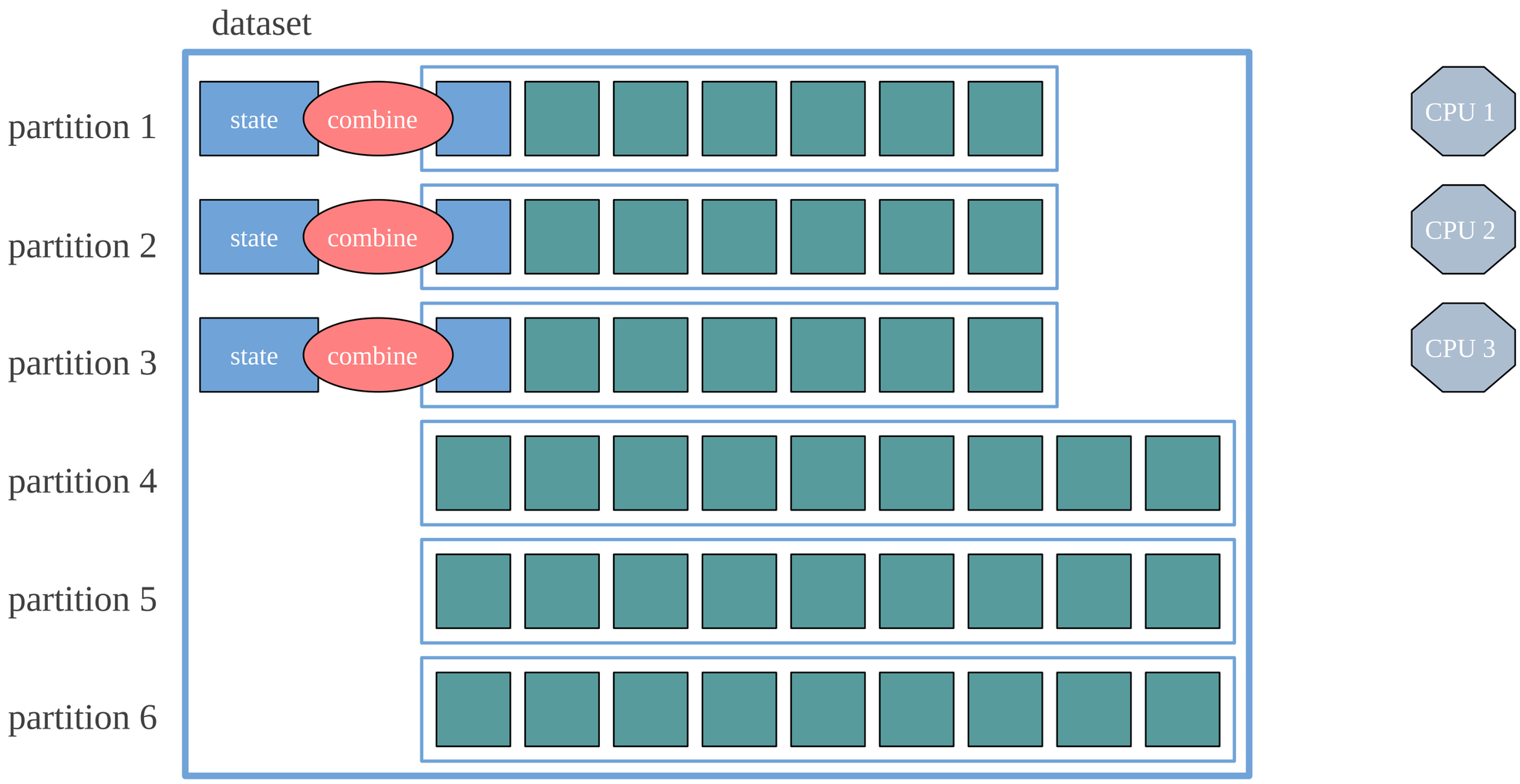
foldMap in parallel



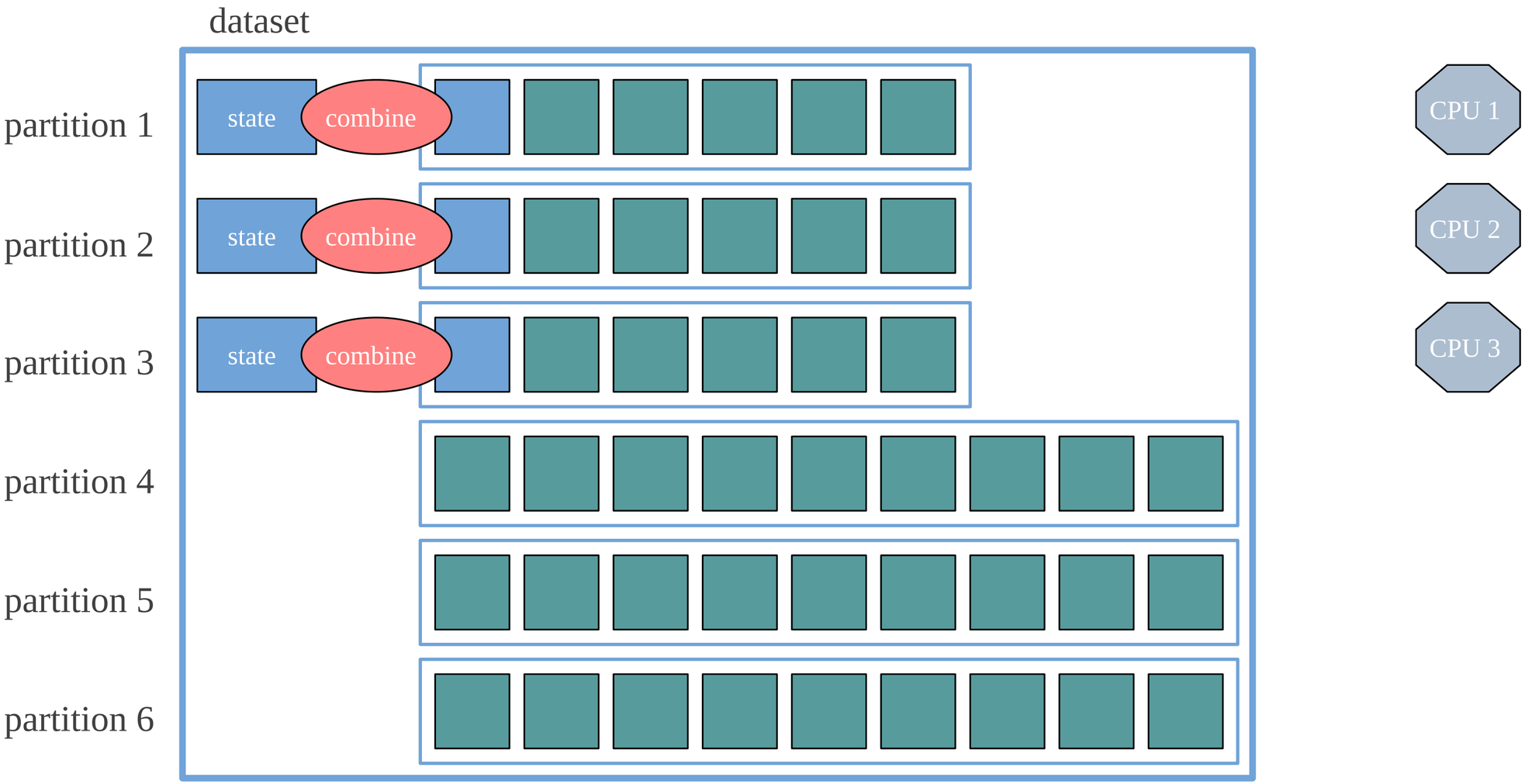
foldMap in parallel



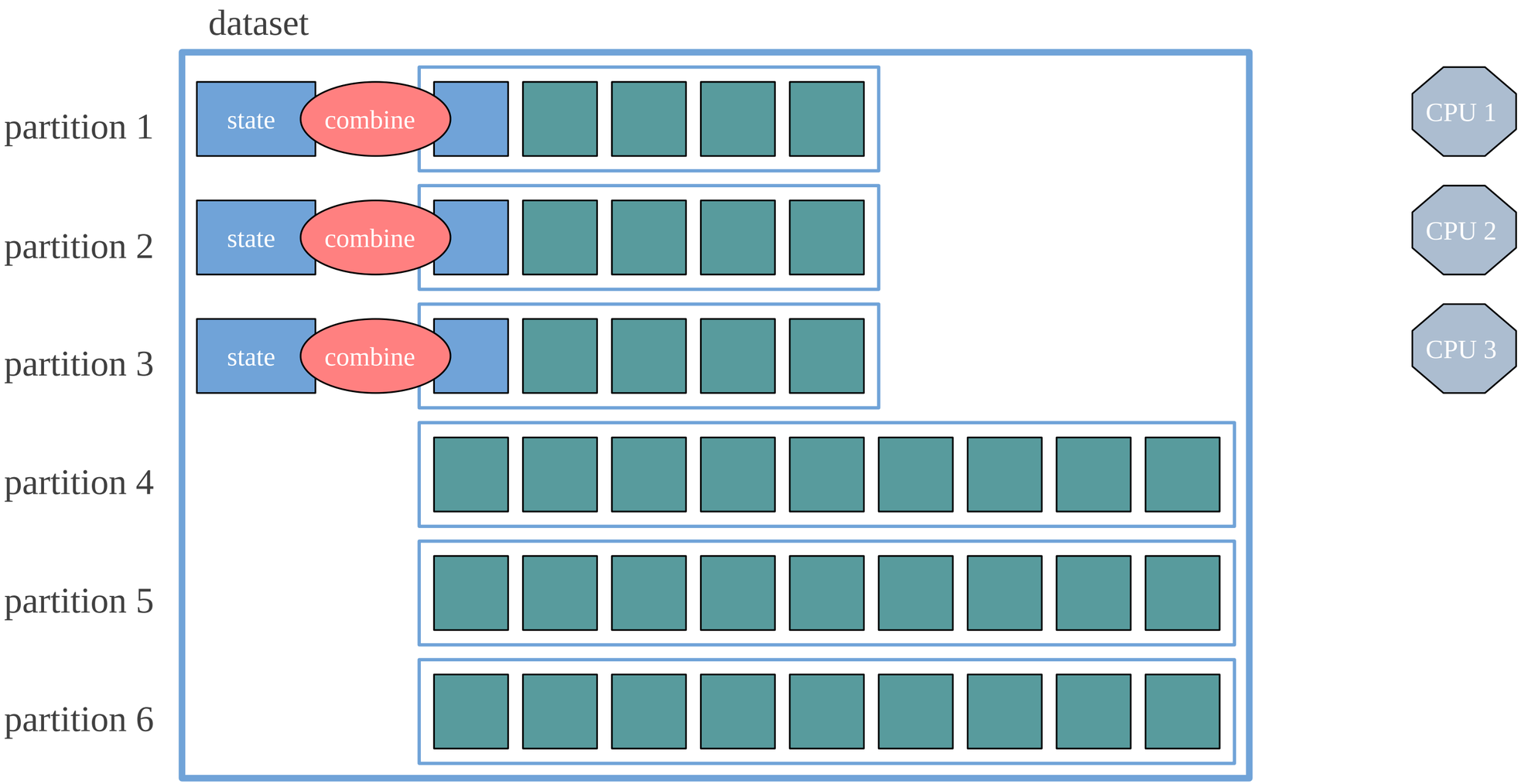
foldMap in parallel



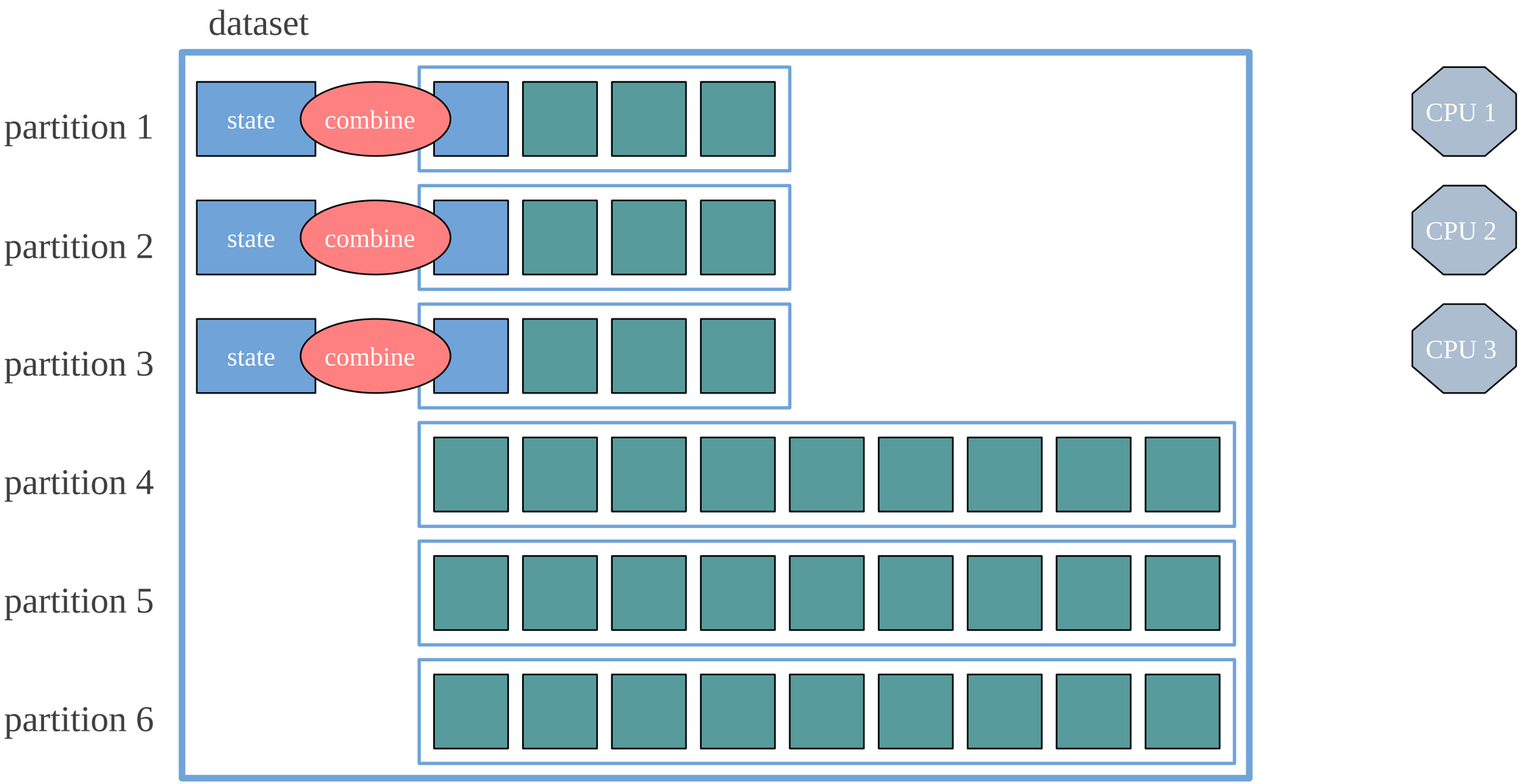
foldMap in parallel



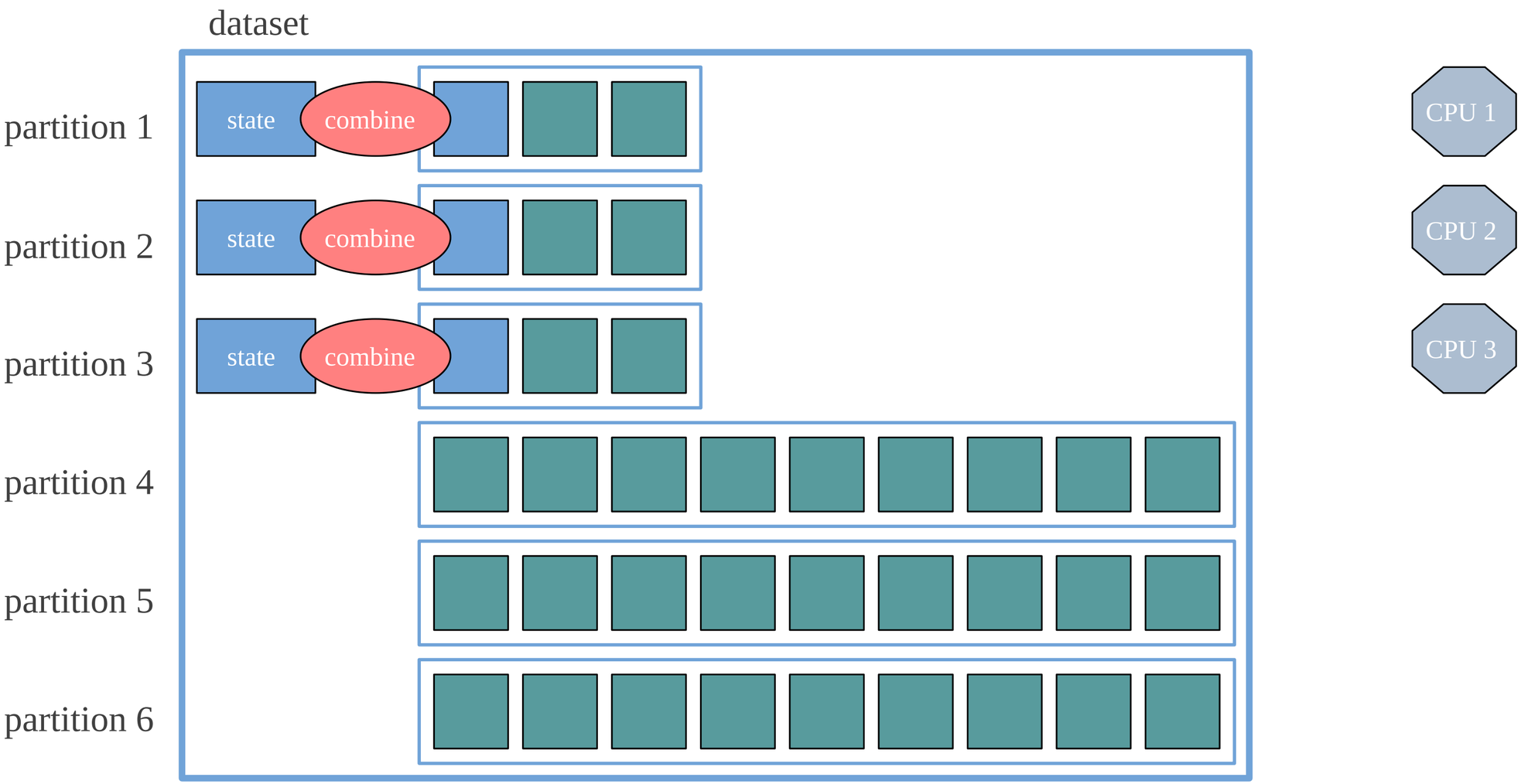
foldMap in parallel



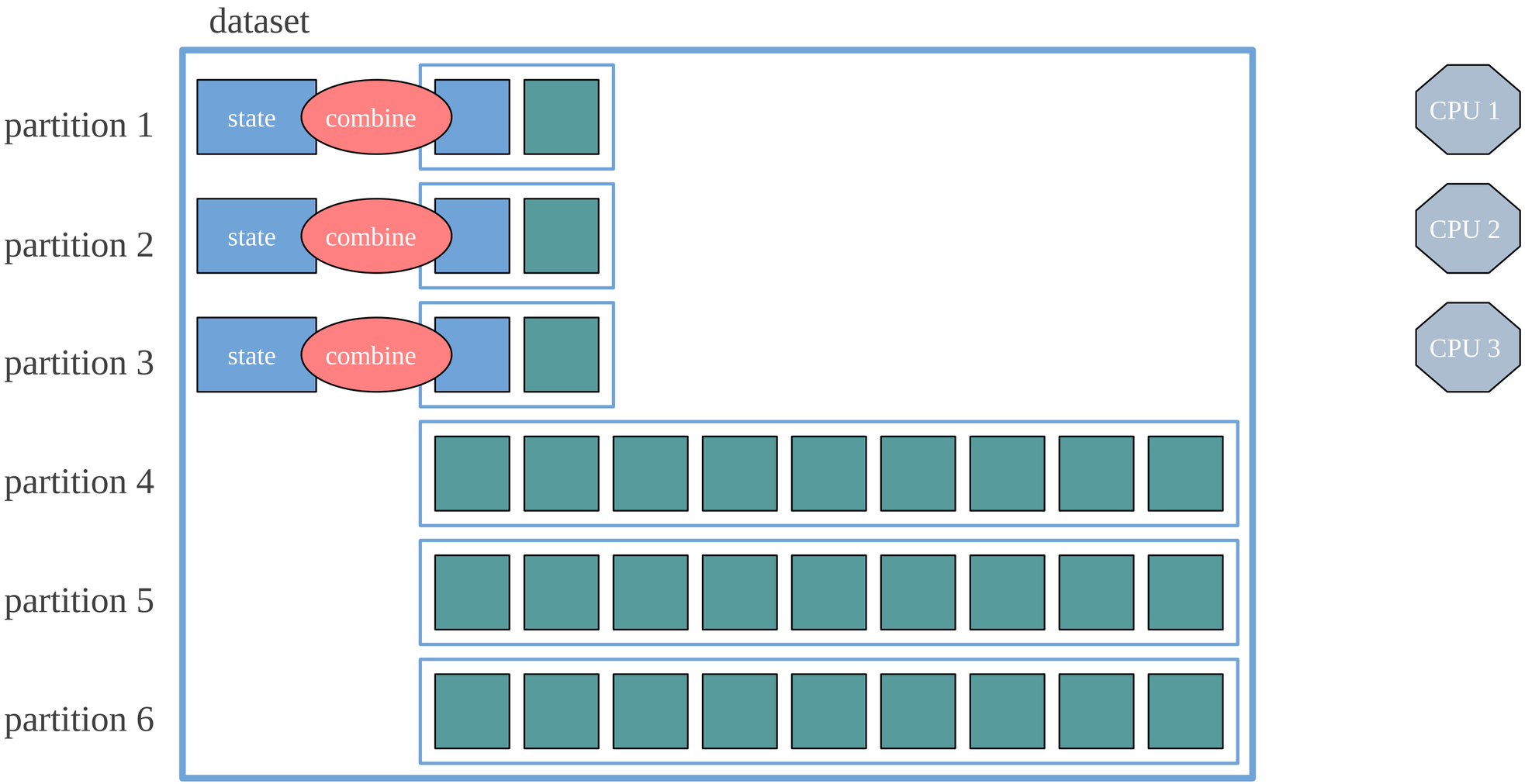
foldMap in parallel



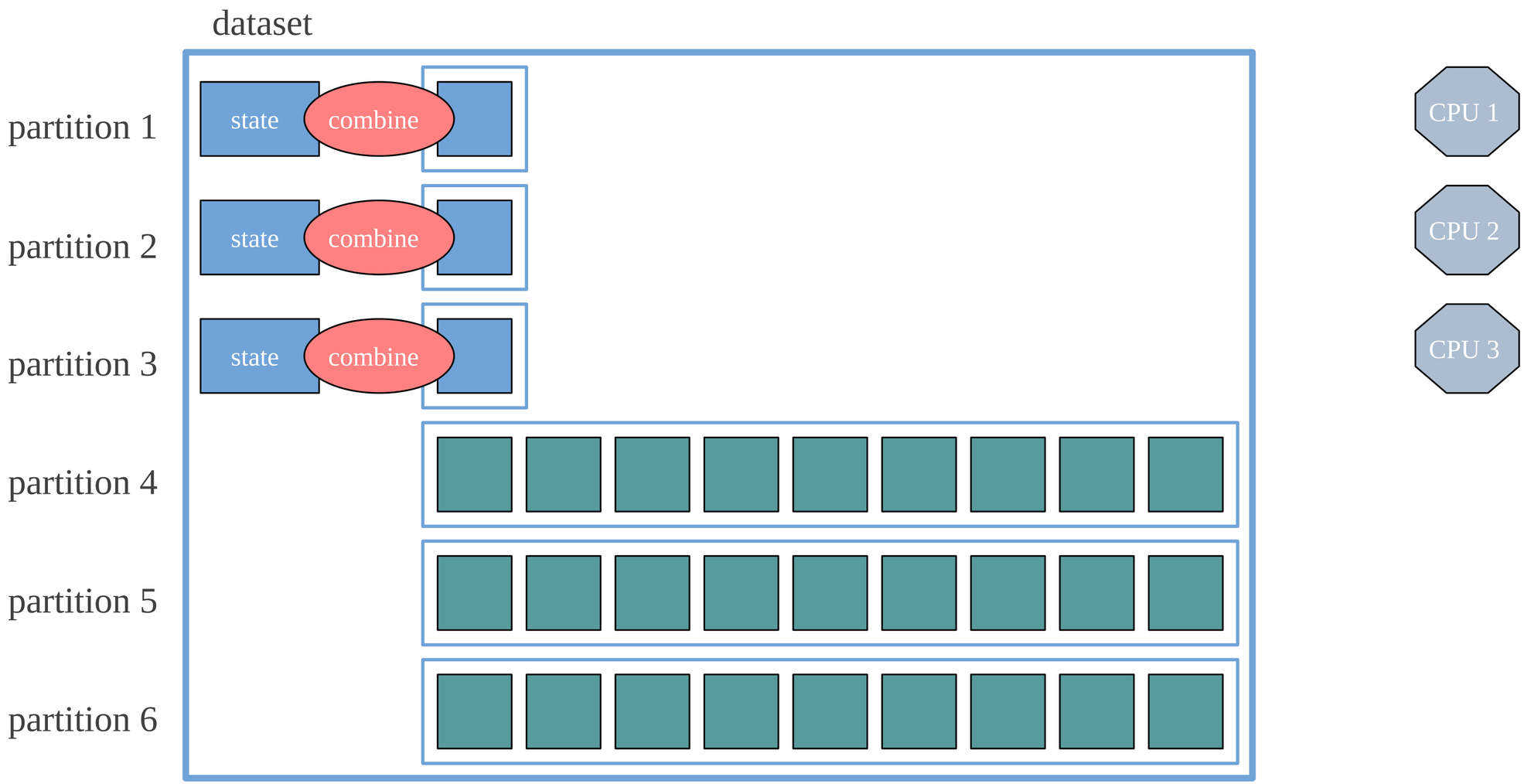
foldMap in parallel



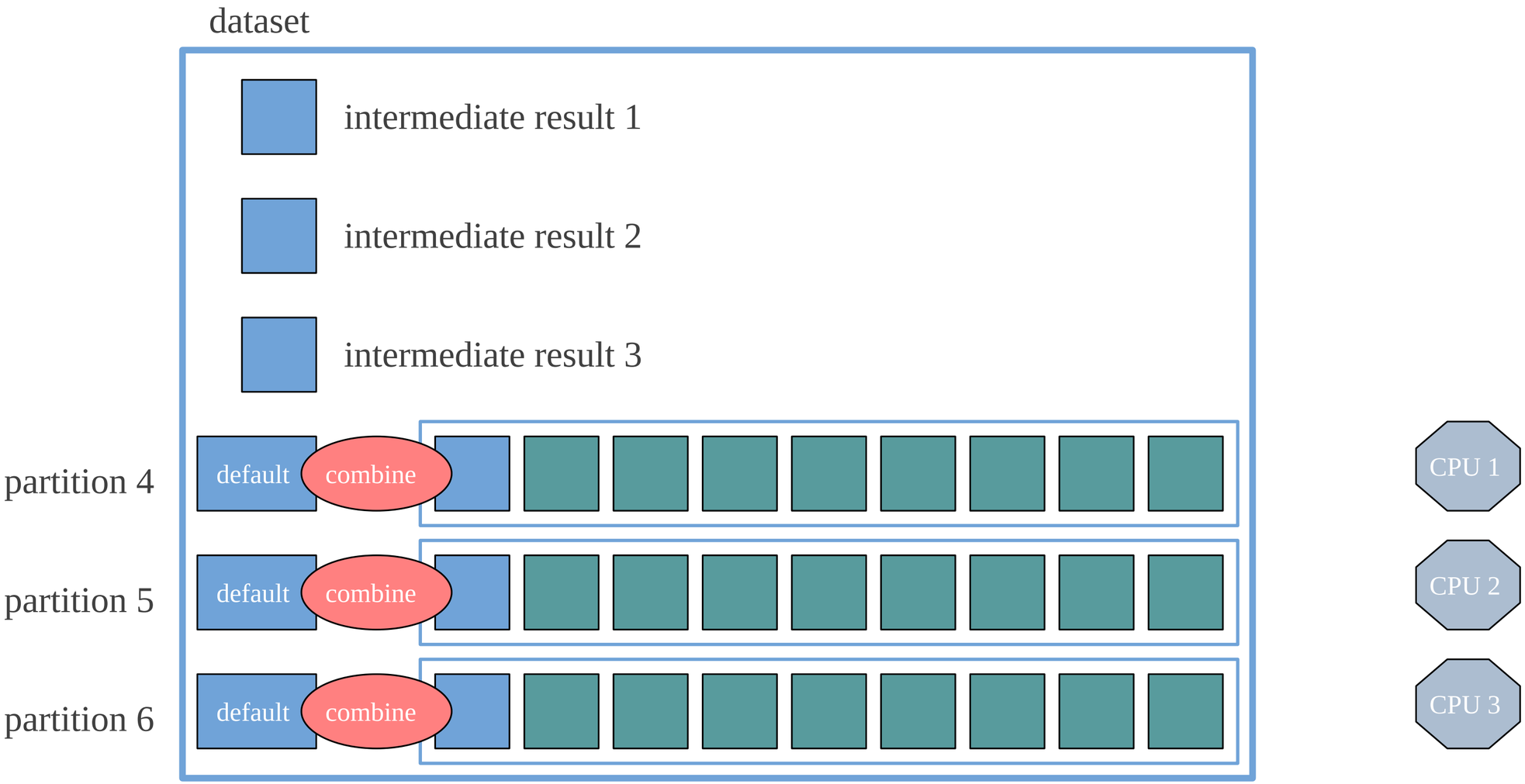
foldMap in parallel



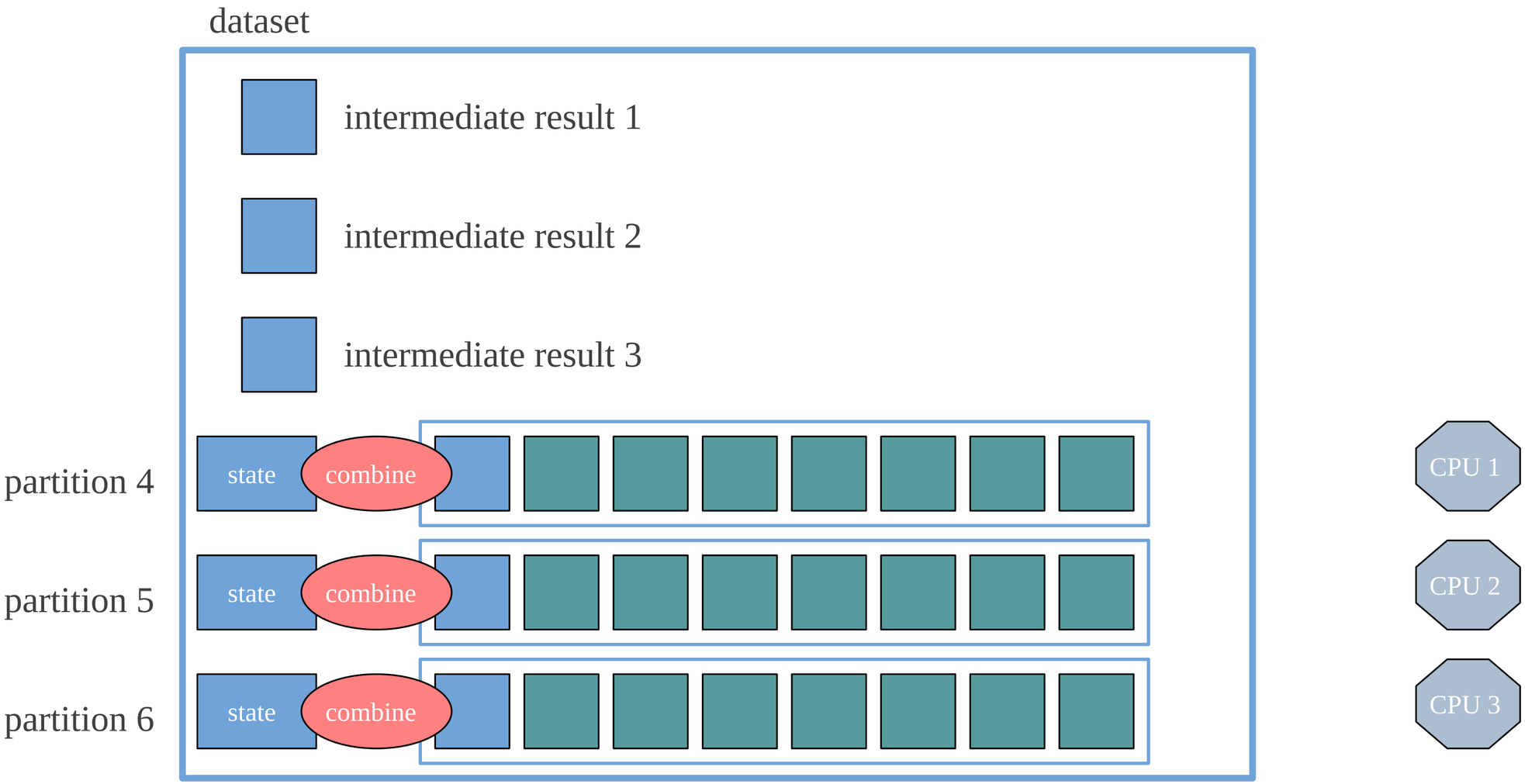
foldMap in parallel



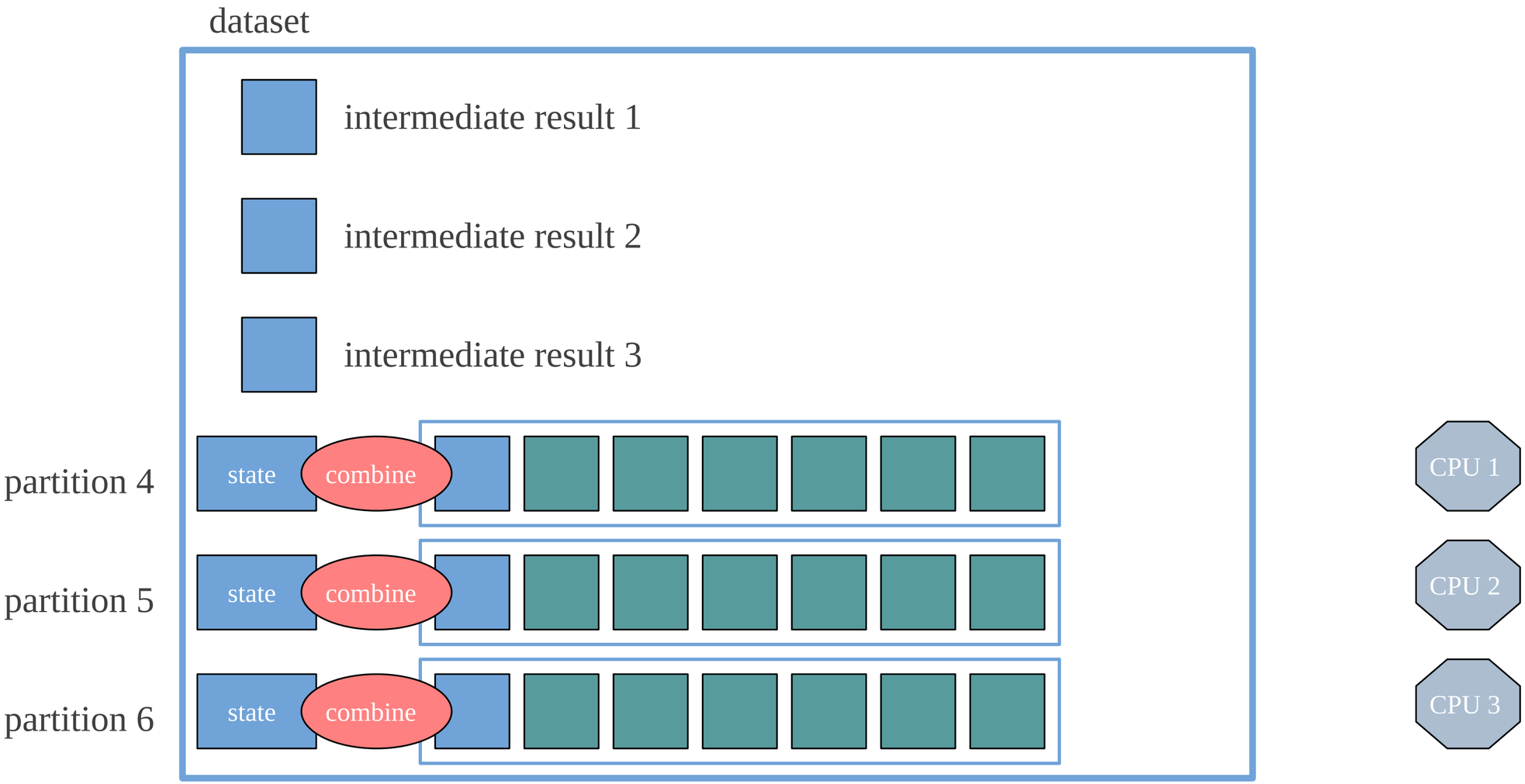
foldMap in parallel



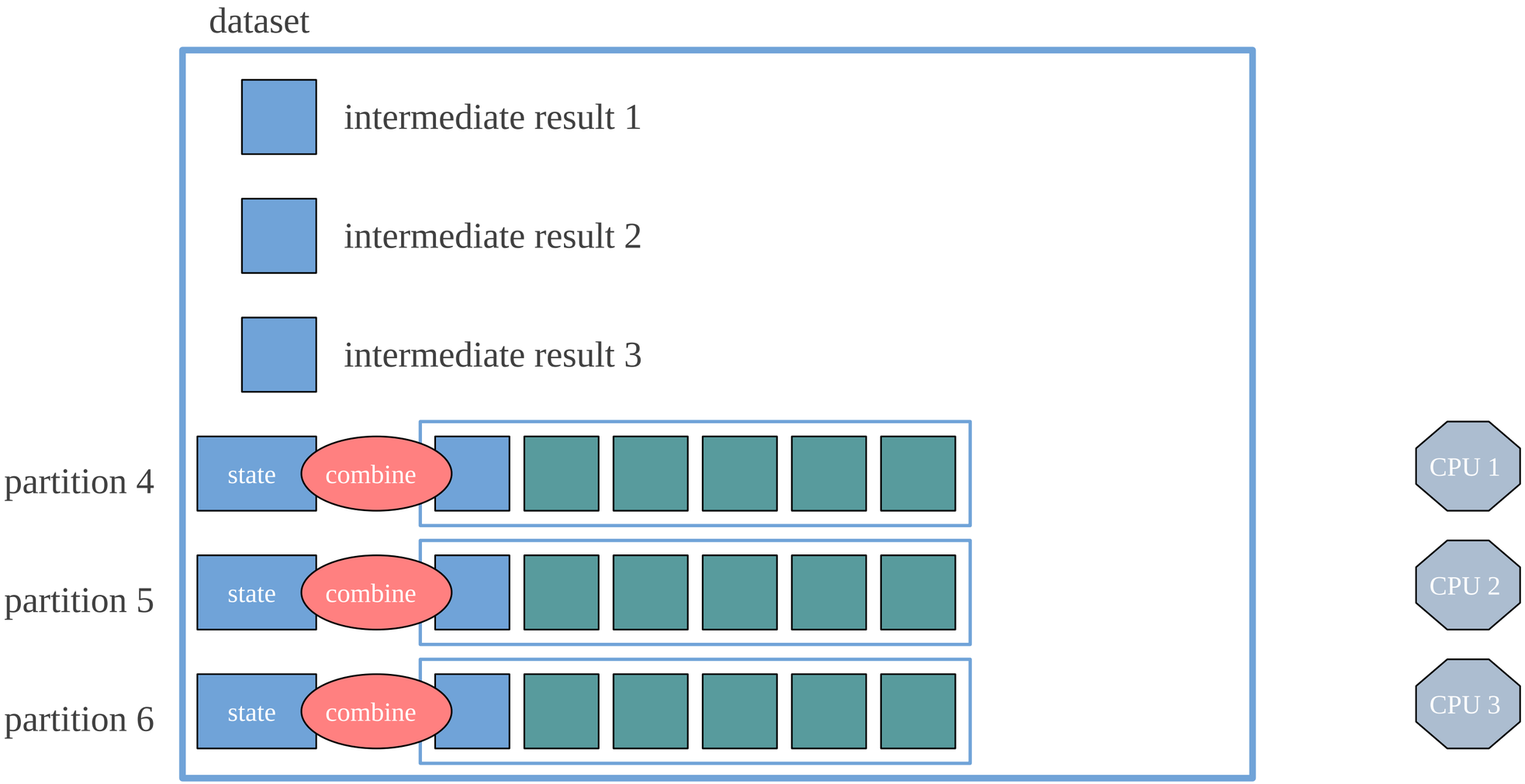
foldMap in parallel



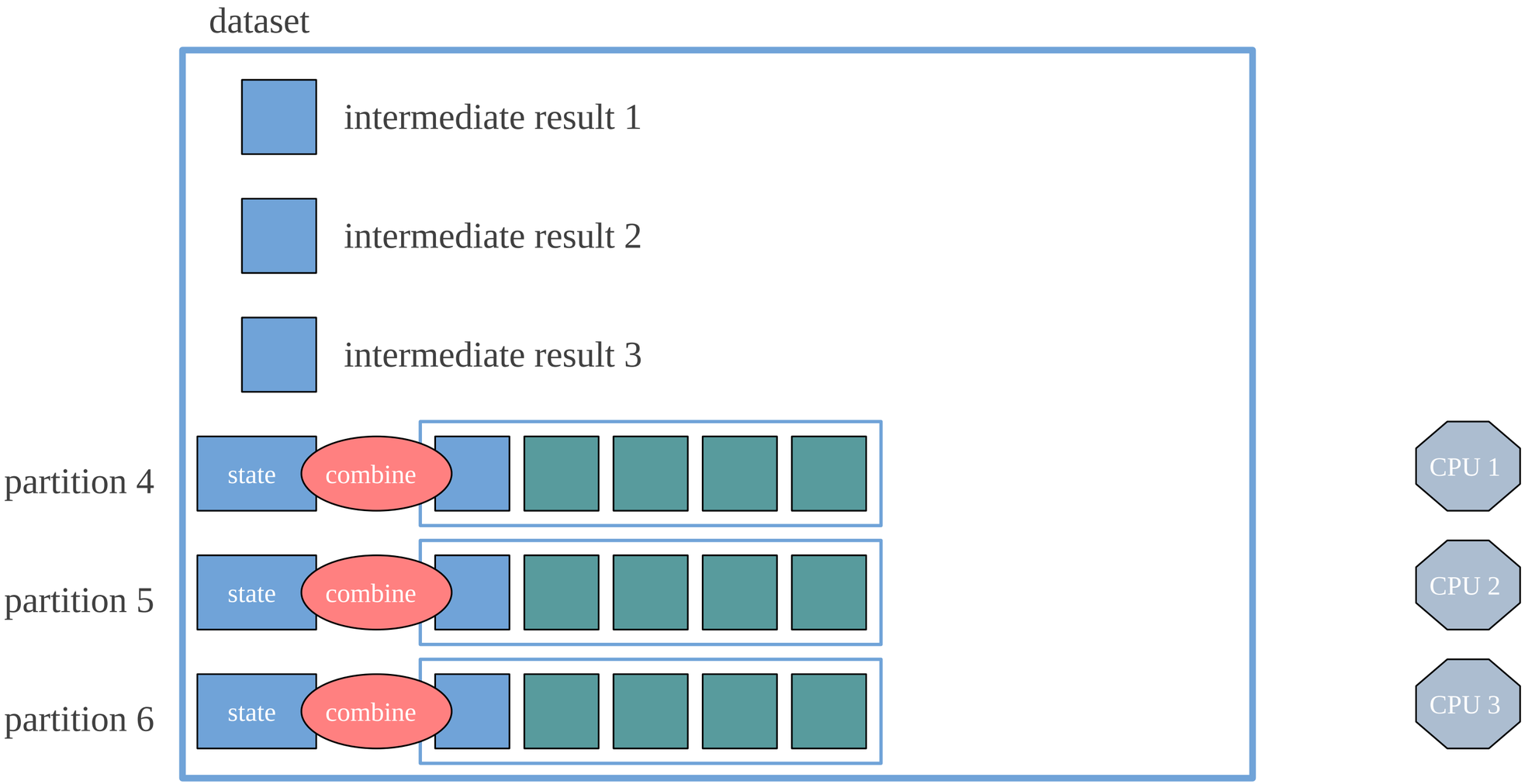
foldMap in parallel



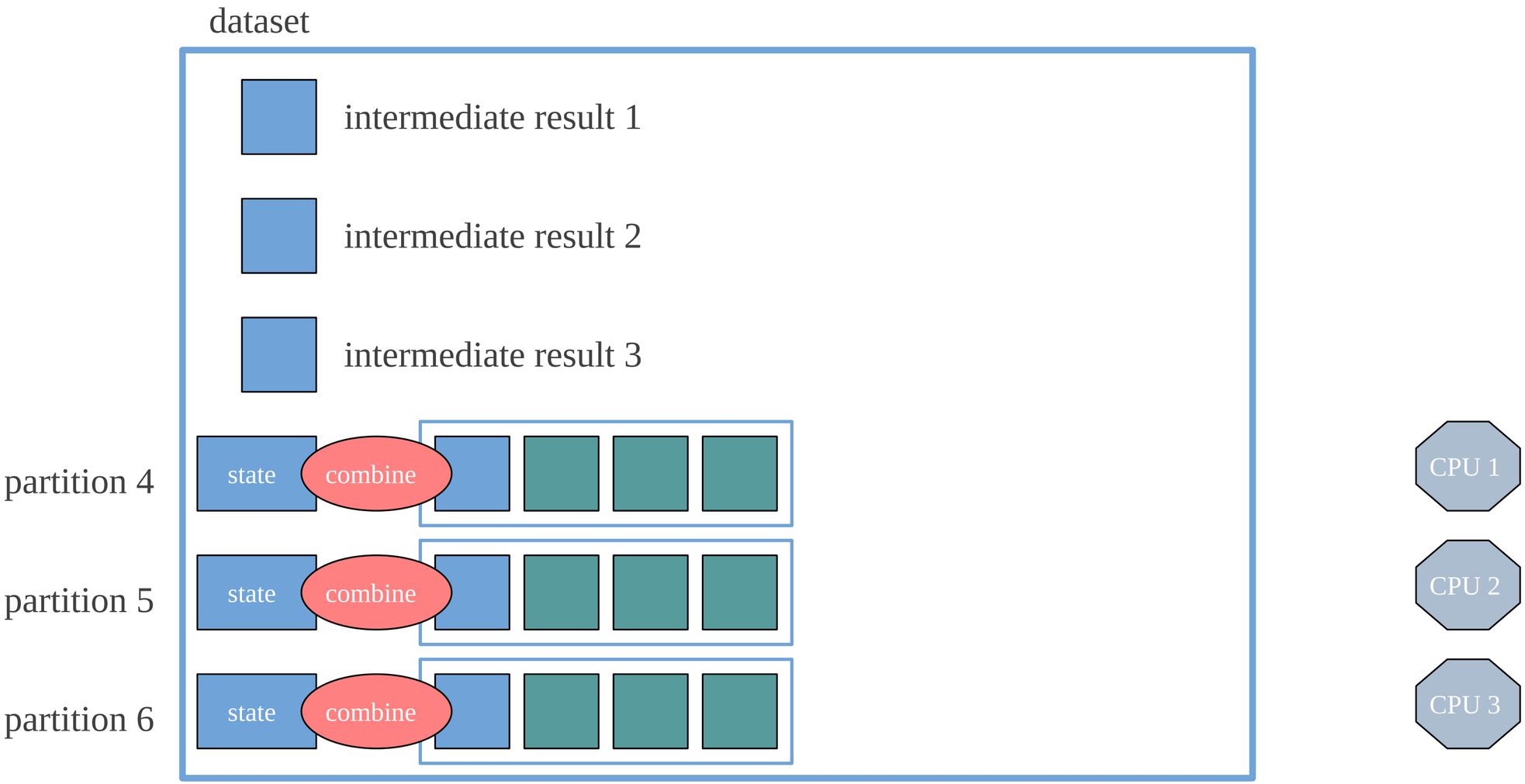
foldMap in parallel



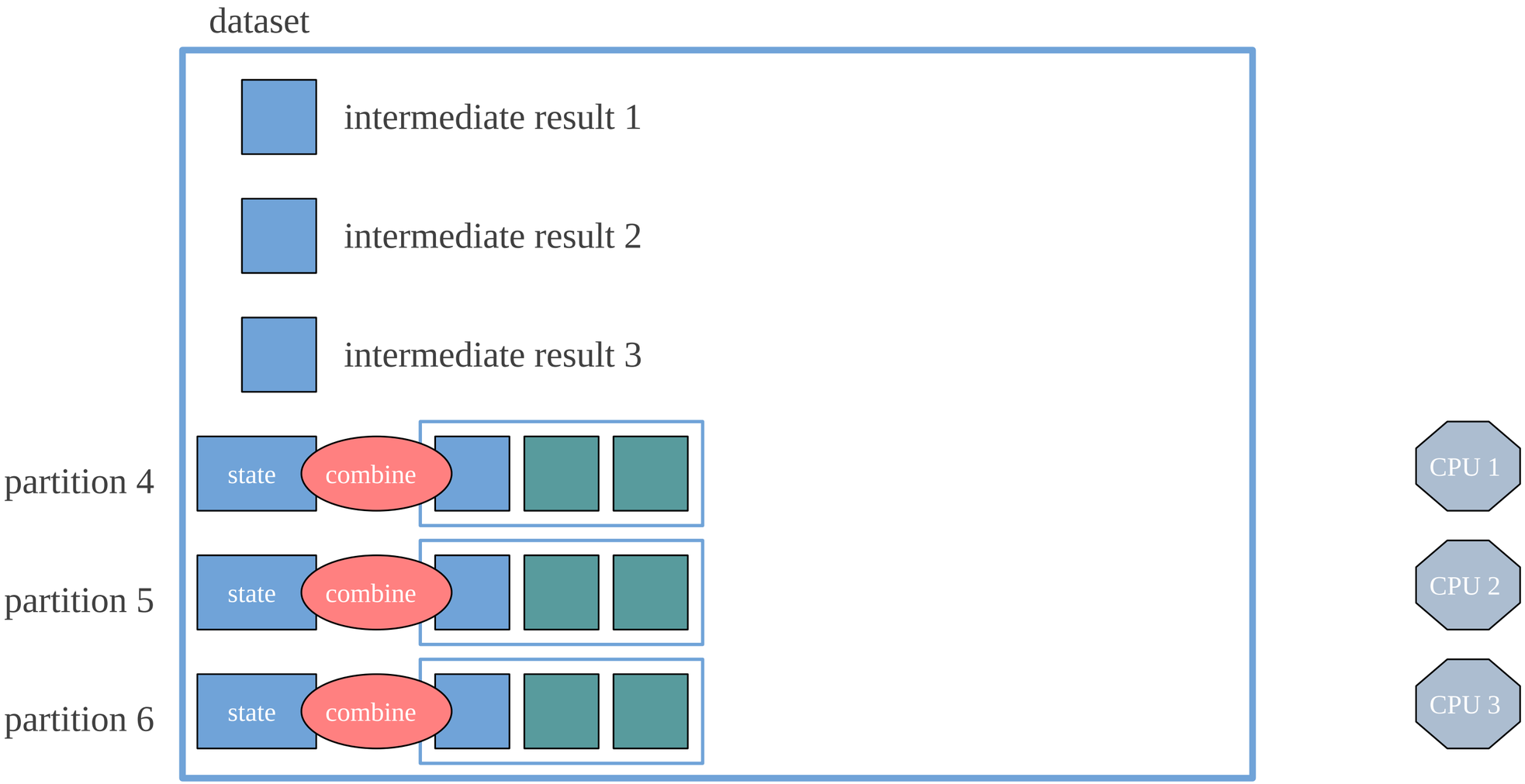
foldMap in parallel



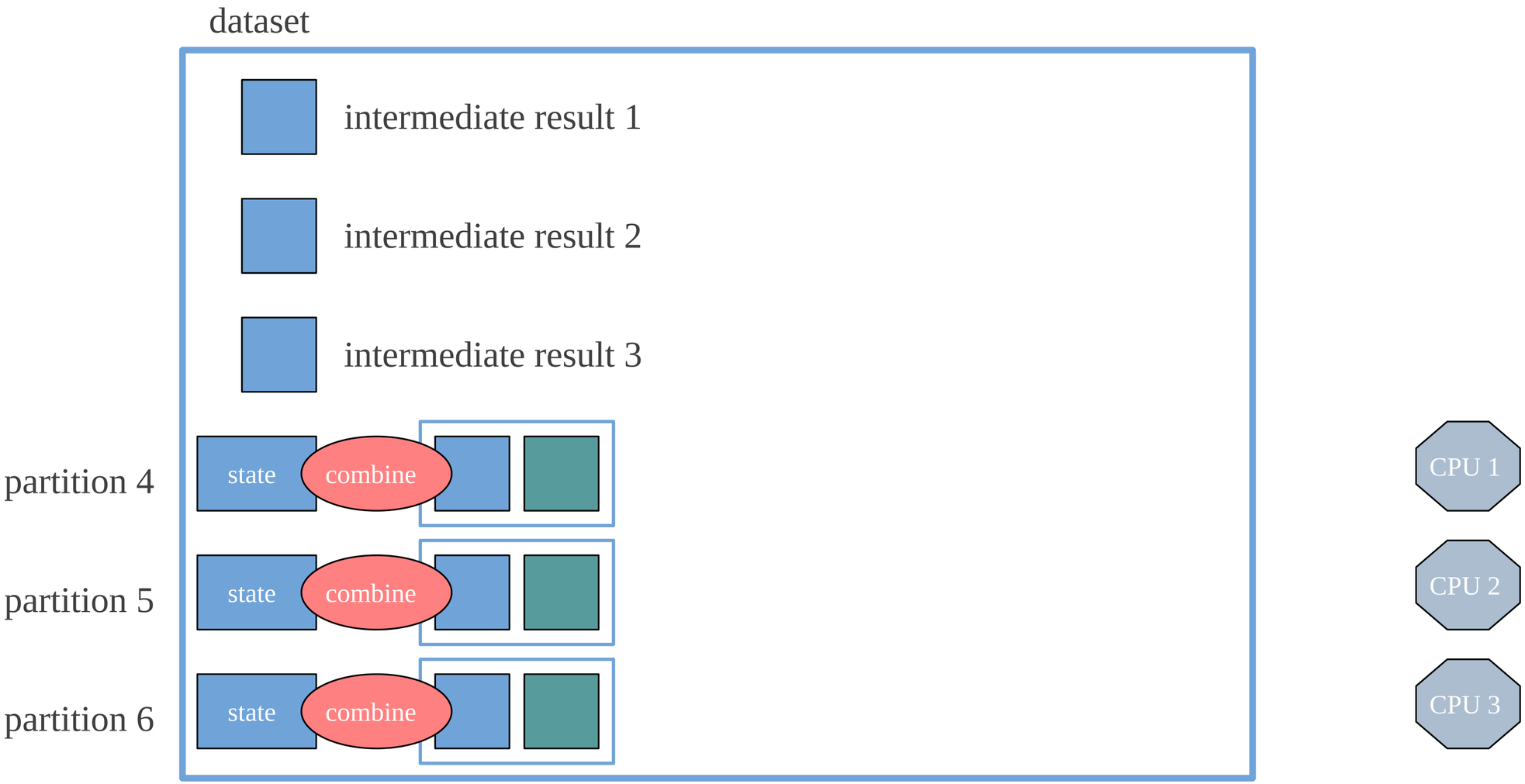
foldMap in parallel



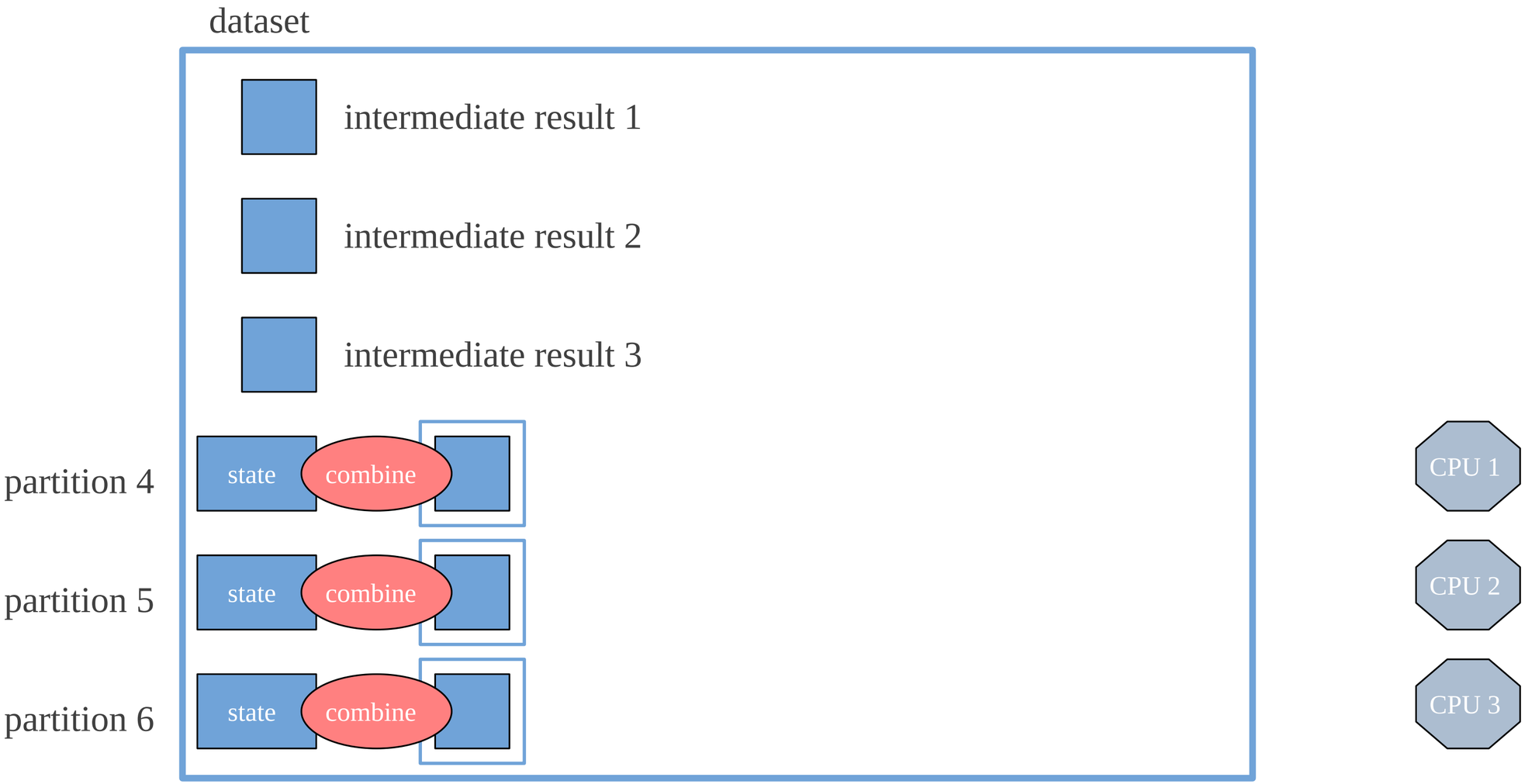
foldMap in parallel



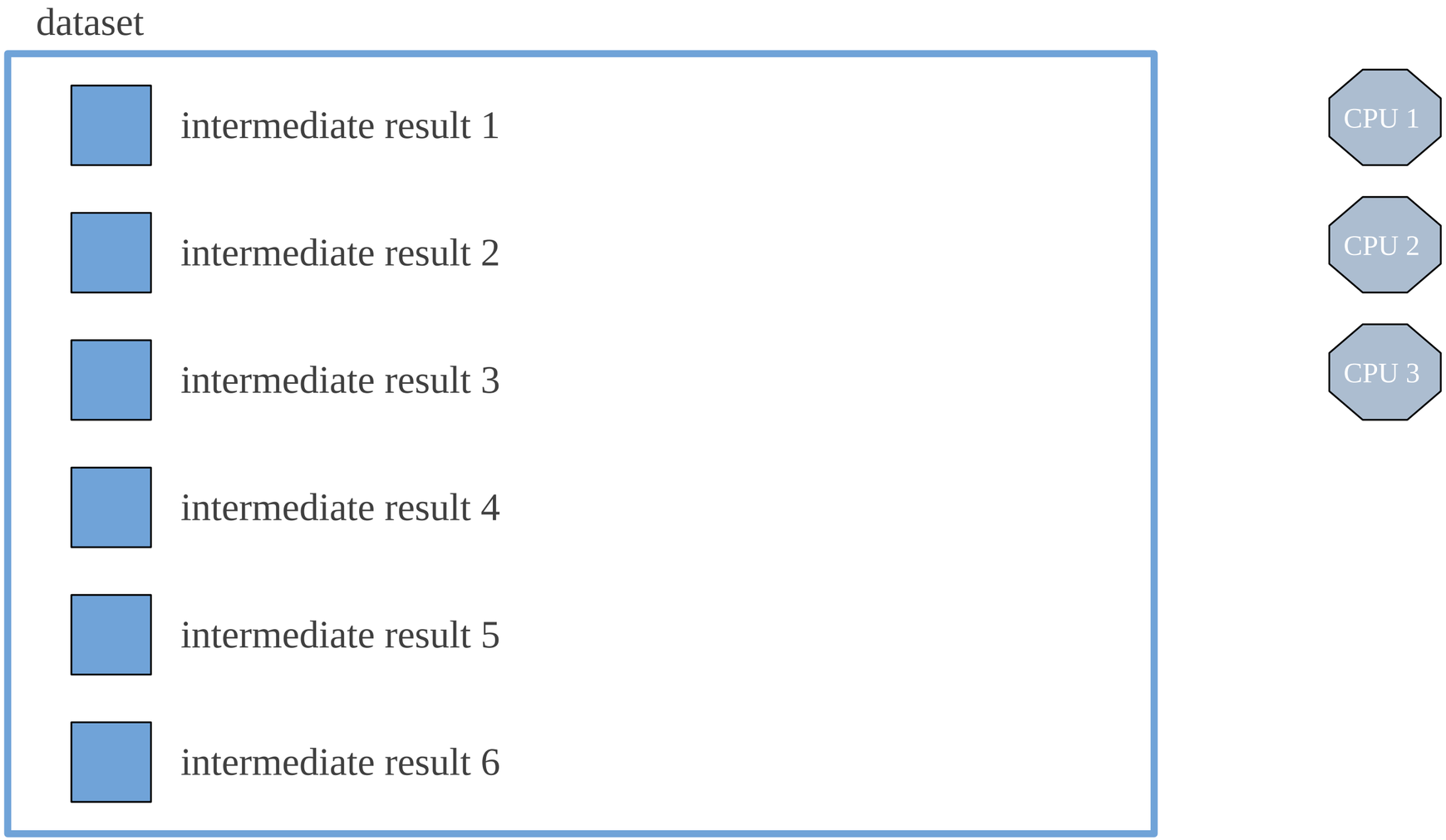
foldMap in parallel



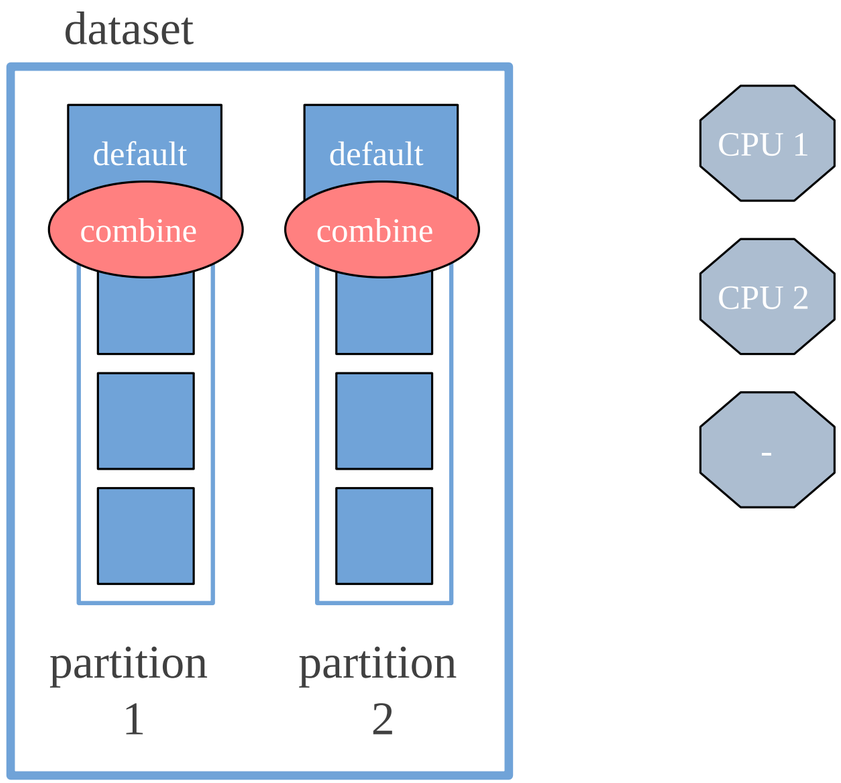
foldMap in parallel



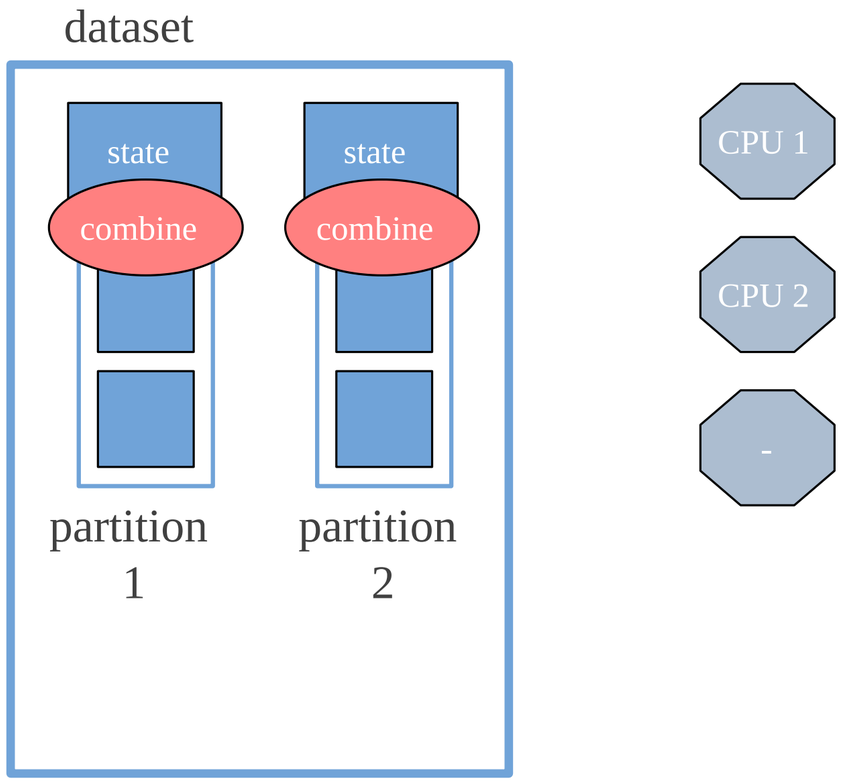
foldMap in parallel



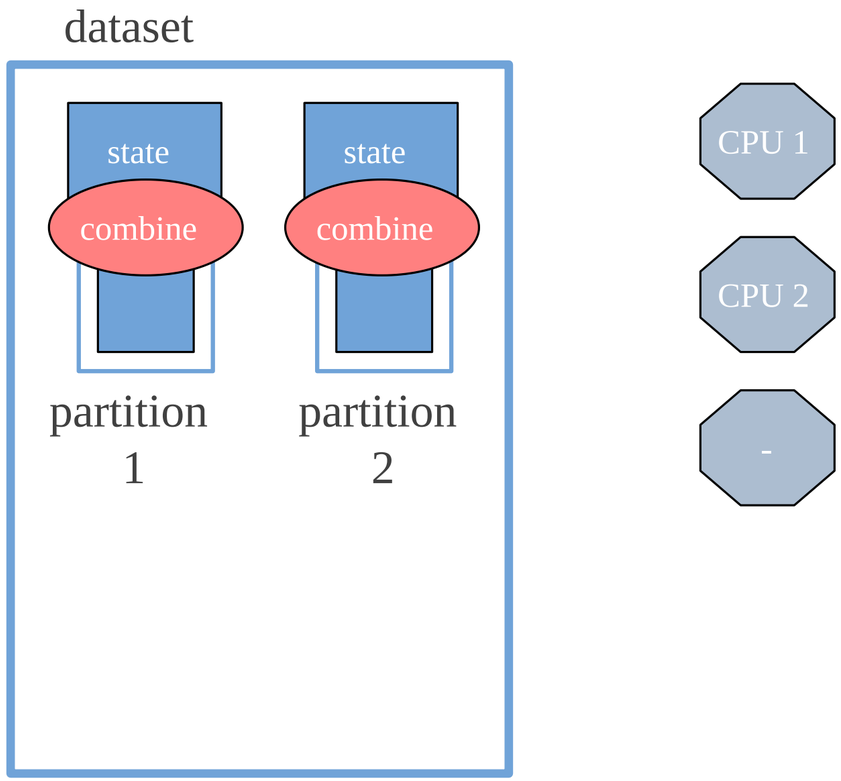
foldMap intermediate results in parallel



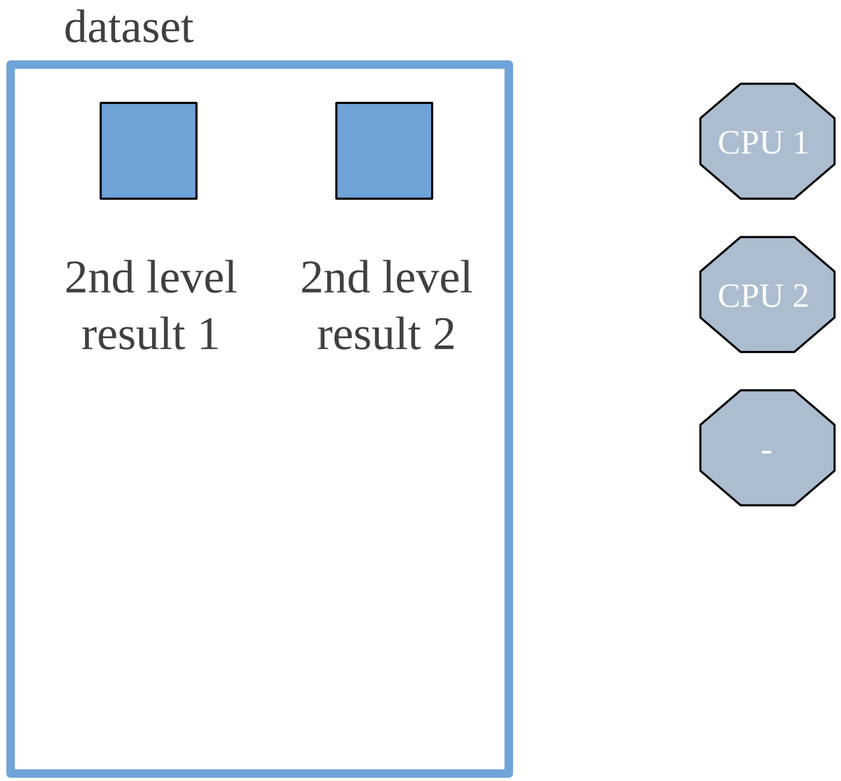
foldMap intermediate results in parallel



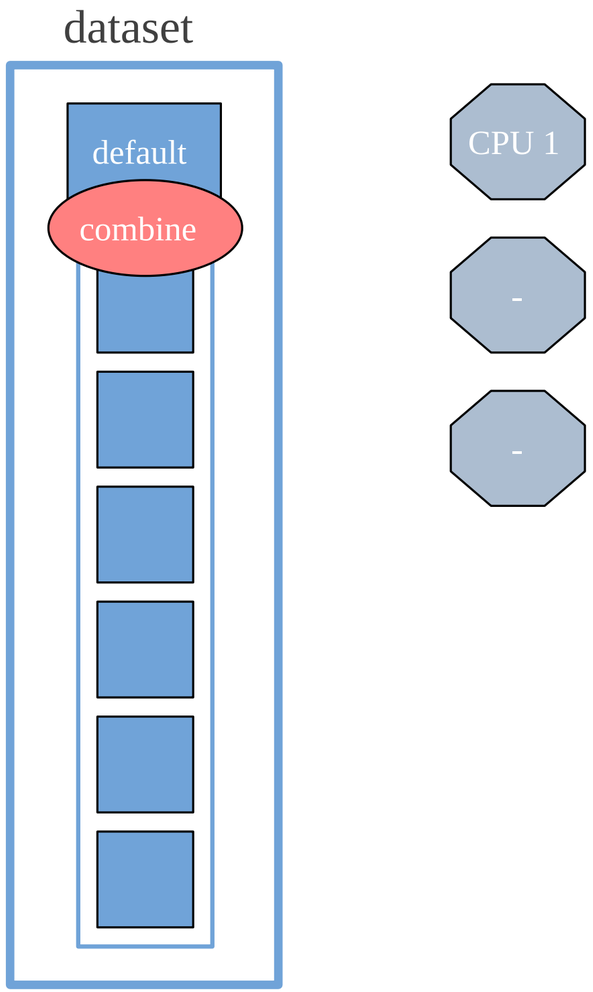
foldMap intermediate results in parallel



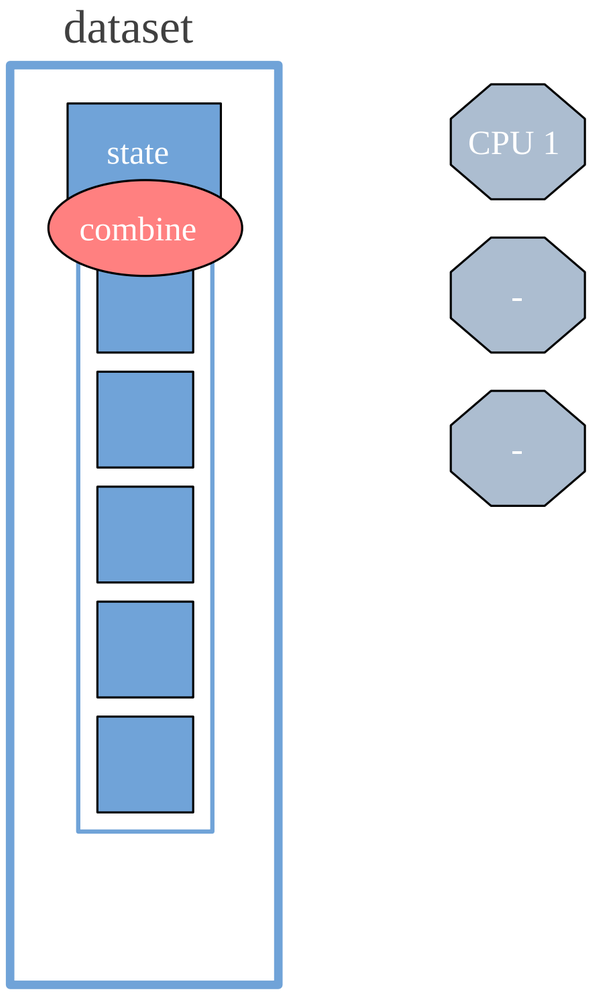
foldMap intermediate results in parallel



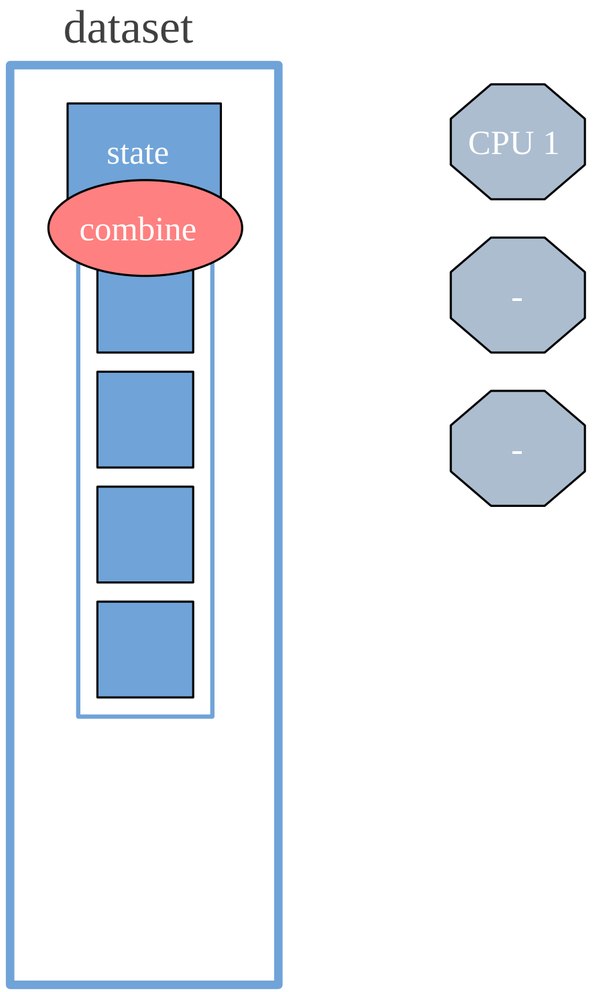
foldMap intermediate results sequentially



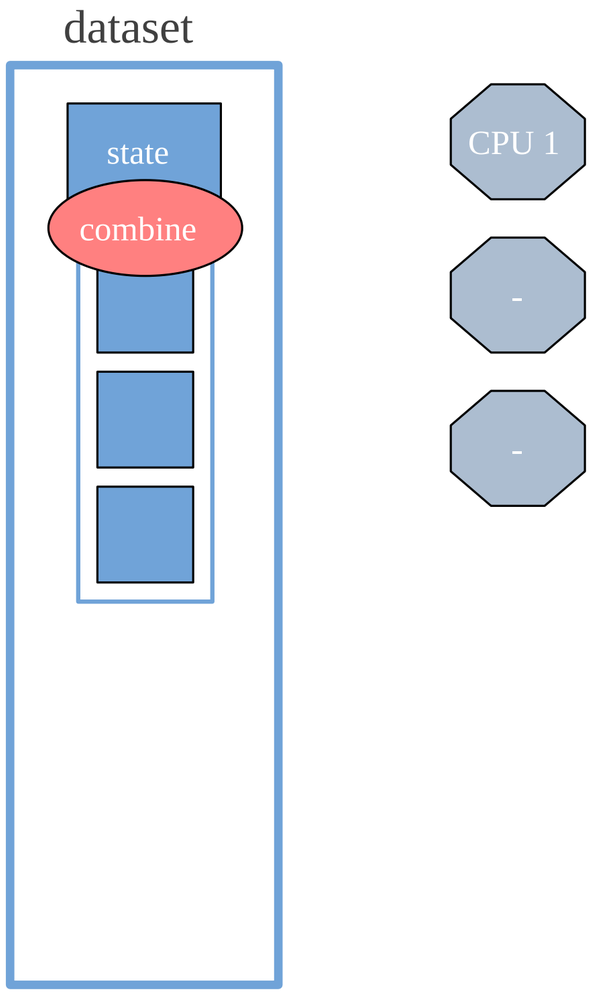
foldMap intermediate results sequentially



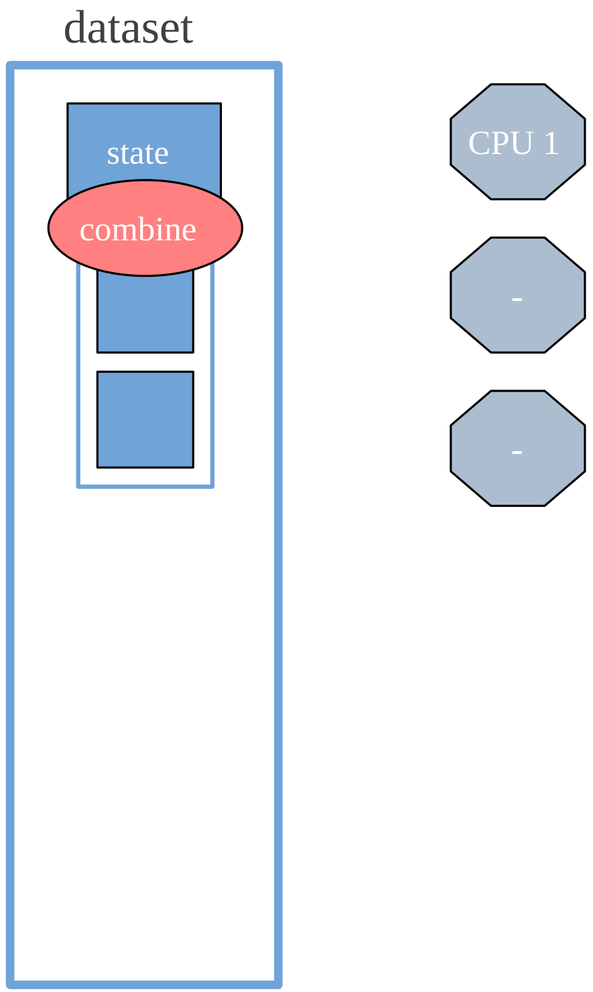
foldMap intermediate results sequentially



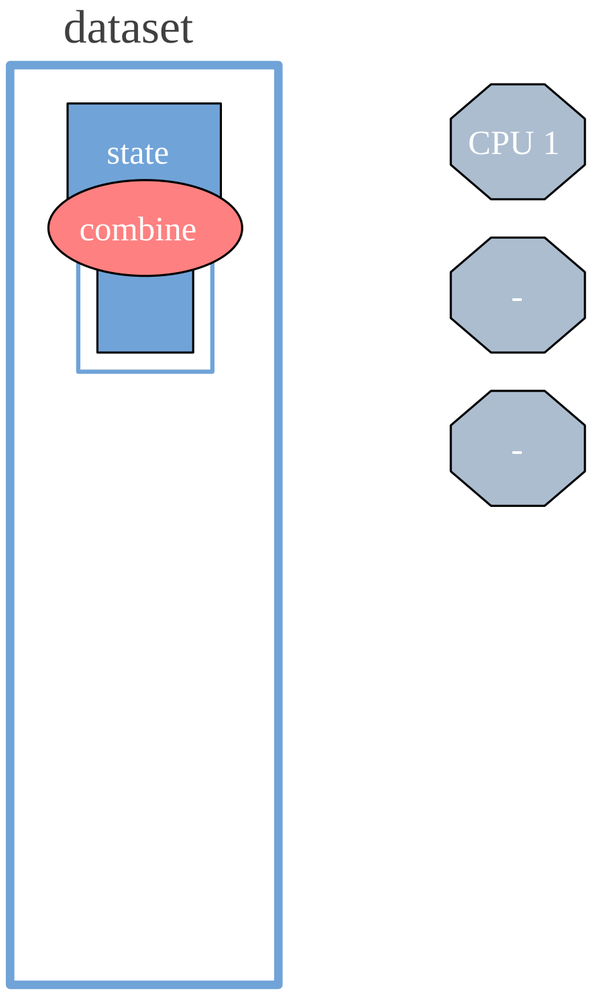
foldMap intermediate results sequentially



foldMap intermediate results sequentially



foldMap intermediate results sequentially



foldMap

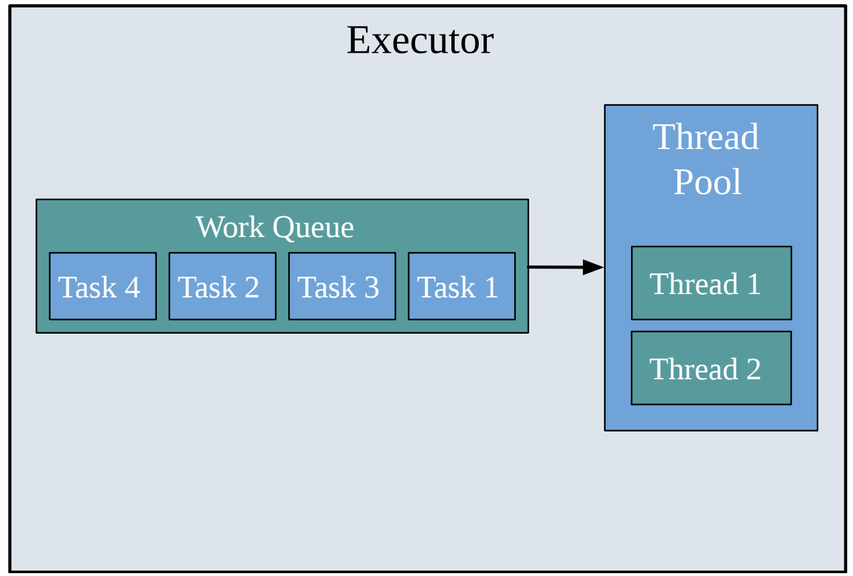
Thread

```
def createThread(n: Int): Thread = new Thread {
  override def run(): Unit =
    println(s"Thread ${n}")
}

val threads = 1.to(4).map(createThread)
```

```
threads.foreach(_.start())
// Thread 1
// Thread 3
// Thread 2
// Thread 4
```

Executor and Runnable

```
import java.util.concurrent.Executors

val fixedPool = Executors.newFixedThreadPool(2)

def createRunnable(n: Int): Runnable =
  new Runnable {
    def run(): Unit =
      println(s"Runnable ${n}")
  }

val runnables = 1.to(4).map(createRunnable)
```

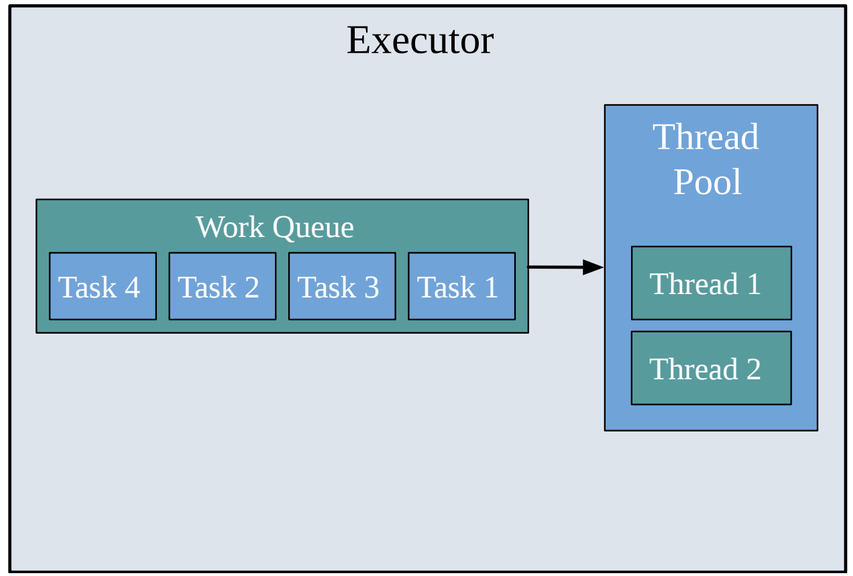



```
runnables.foreach(fixedPool.submit)
// Runnable 1
// Runnable 3
// Runnable 2
// Runnable 4
```

Executor and Runnable

```
import java.util.concurrent.Executors

val fixedPool = Executors.newFixedThreadPool(2)

def createRunnable(n: Int): Runnable =
  new Runnable {
    def run(): Unit = {
      val thread = Thread.currentThread
      println(s"[${thread.getName}] Runnable ${n}")
    }
  }

val runnables = 1.to(4).map(createRunnable)
```

```
runnables.foreach(fixedPool.submit)
// [pool-19-thread-1] Runnable 1
// [pool-19-thread-2] Runnable 3
// [pool-19-thread-1] Runnable 2
// [pool-19-thread-2] Runnable 4
```



ExecutionContext and Future

```
import java.util.concurrent.Executors
import scala.concurrent.duration._
import scala.concurrent.{ Await, ExecutionContext, Future }

val fixedPool        = Executors.newFixedThreadPool(2)
val executionContext = ExecutionContext.fromExecutor(fixedPool)
```

```
val future = Future {
  Thread.sleep(1000)  // sleep 1 second
  1
}(executionContext)
// future: Future[Int] = Future(<not completed>)
```

ExecutionContext and Future

```
import java.util.concurrent.Executors
import scala.concurrent.duration._
import scala.concurrent.{ Await, ExecutionContext, Future }

val fixedPool                = Executors.newFixedThreadPool(2)
implicit val executionContext = ExecutionContext.fromExecutor(fixedPool)
```

```
val task = Future {
  Thread.sleep(1000)  // sleep 1 second
  3
}
// task: Future[Int] = Future(<not completed>)
```

ExecutionContext and Future

```
import java.util.concurrent.Executors
import scala.concurrent.duration._
import scala.concurrent.{ Await, ExecutionContext, Future }

val fixedPool                   = Executors.newFixedThreadPool(2)
implicit val executionContext = ExecutionContext.fromExecutor(fixedPool)
```

```
val task = Future {
  Thread.sleep(1000)  // sleep 1 second
  3
}
// task: Future[Int] = Future(<not completed>)
```

```
Await.result(task, 2.minutes)
// res: Int = 3
```

ExecutionContext and Future

```
import java.util.concurrent.Executors
import scala.concurrent.duration._
import scala.concurrent.{ Await, ExecutionContext, Future }

val fixedPool                = Executors.newFixedThreadPool(2)
implicit val executionContext = ExecutionContext.fromExecutor(fixedPool)
```

```
val task = Future {
  Thread.sleep(1000 * 60 * 5)  // sleep 5 minutes
  3
}
// task: Future[Int] = Future(<not completed>)
```

```
Await.result(task, 2.minutes)
// java.util.concurrent.TimeoutException: Future timed out after [2 minutes]
```

ExecutionContext and Future

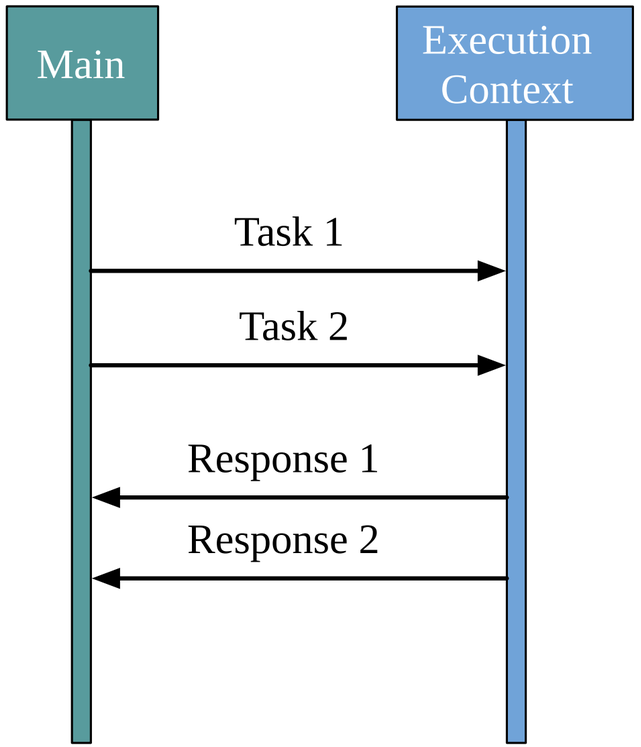
```
import java.util.concurrent.Executors
import scala.concurrent.duration._
import scala.concurrent.{ Await, ExecutionContext, Future }

val fixedPool                = Executors.newFixedThreadPool(2)
implicit val executionContext = ExecutionContext.fromExecutor(fixedPool)
```

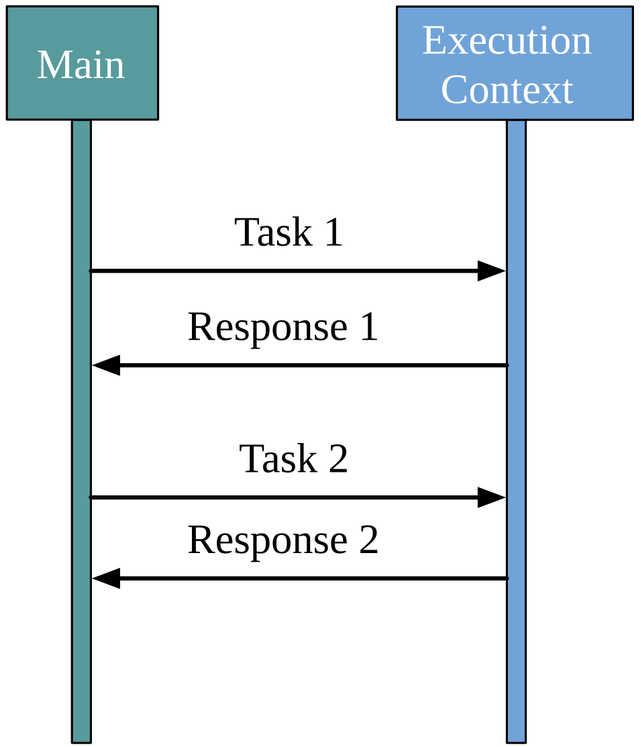
```
val task = Future {
  Thread.sleep(1000 * 60 * 5)  // sleep 5 minutes
  3
}
// task: Future[Int] = Future(<not completed>)
```

```
Await.result(task, Duration.Inf)
// res: Int = 3
```

ExecutionContext and Future

```
val future1   = Future { task(1) }
val future2   = Future { task(2) }

val response1 = Await.result(future1, Duration.Inf)
val response2 = Await.result(future2, Duration.Inf)
```



ExecutionContext and Future

```
val future1   = Future { task(1) }
val response1 = Await.result(future1, Duration.Inf)

val future2   = Future { task(2) }
val response2 = Await.result(future2, Duration.Inf)
```