Iteratively Improving Spark Application Performance

William C. Benton
Red Hat, Inc.

redhat.

Forecast

- Background: Spark, RDDs, and Spark's execution model
- Case study overview
- Improving our prototype

Background

Apache Spark

- Introduced in 2009; donated to Apache in 2013; 1.0 release in 2014
- Based on a fundamental abstraction: the resilient distributed dataset
- Supports in-memory computing and a wide range of algorithms

Spark is general

redhat.

Spark is general

Graph

Spark core

redhat.

Spark is general

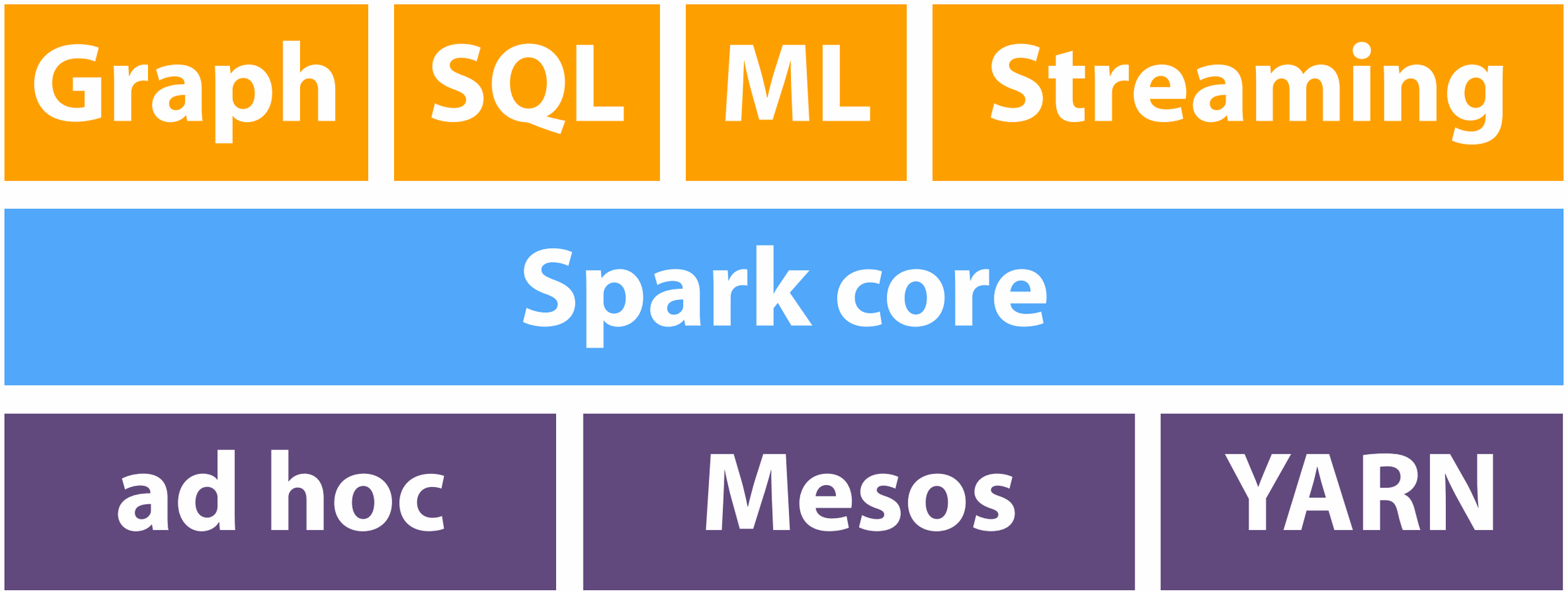
Spark is general

redhat.

Spark is general

redhat

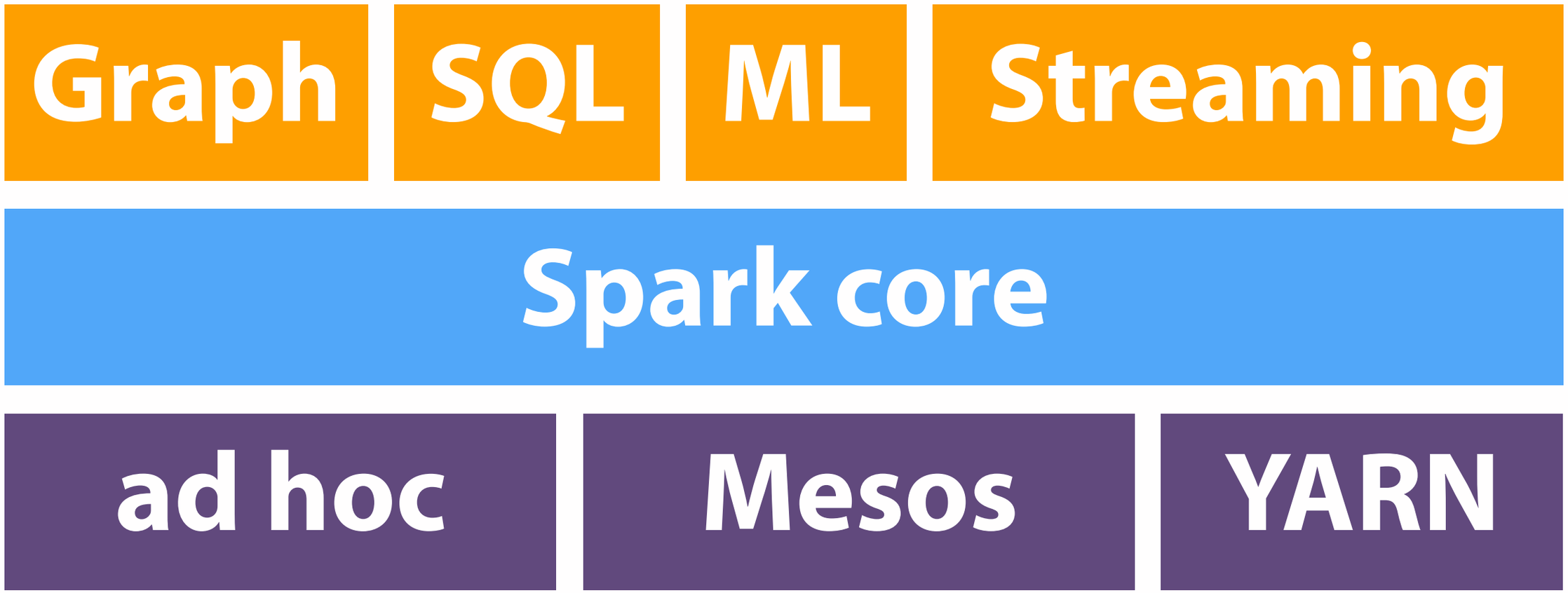
Spark is general



redhat

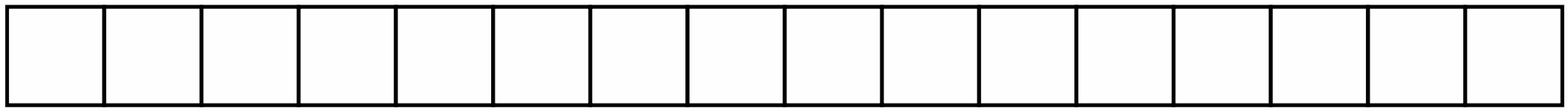
Spark is general

APIS FOR SCALA, JAVA, PYTHON, AND R
(3RD-PARTY BINDINGS FOR CLOJURE ET AL.)



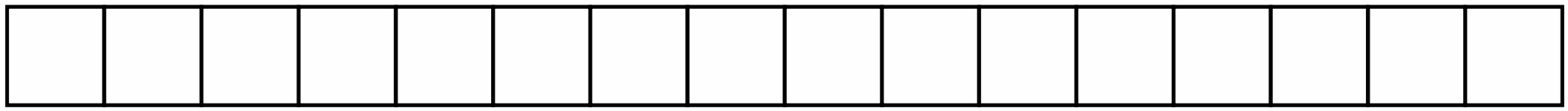
redhat.

Resilient distributed datasets



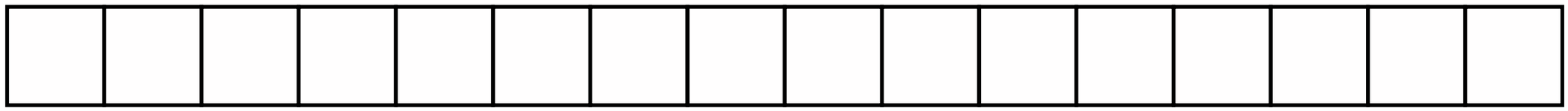
redhat.

Resilient distributed datasets



redhat.

Resilient distributed datasets

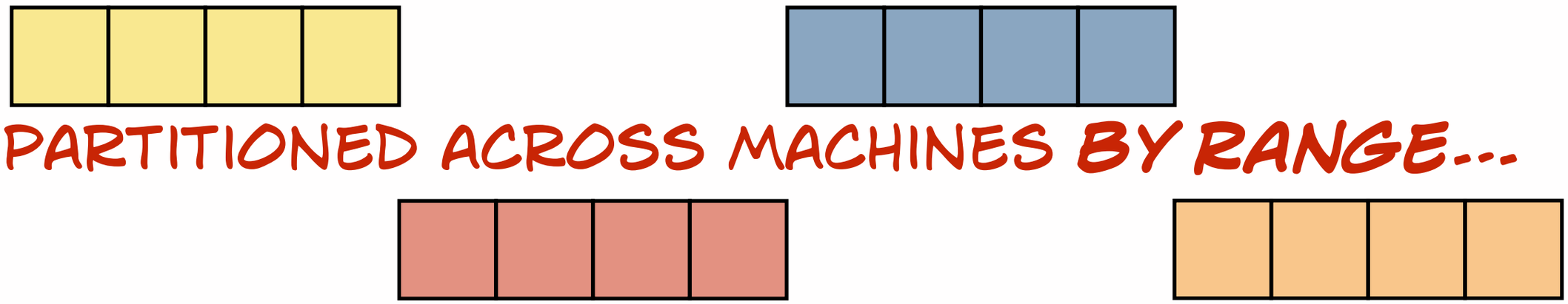


redhat.

Resilient distributed datasets

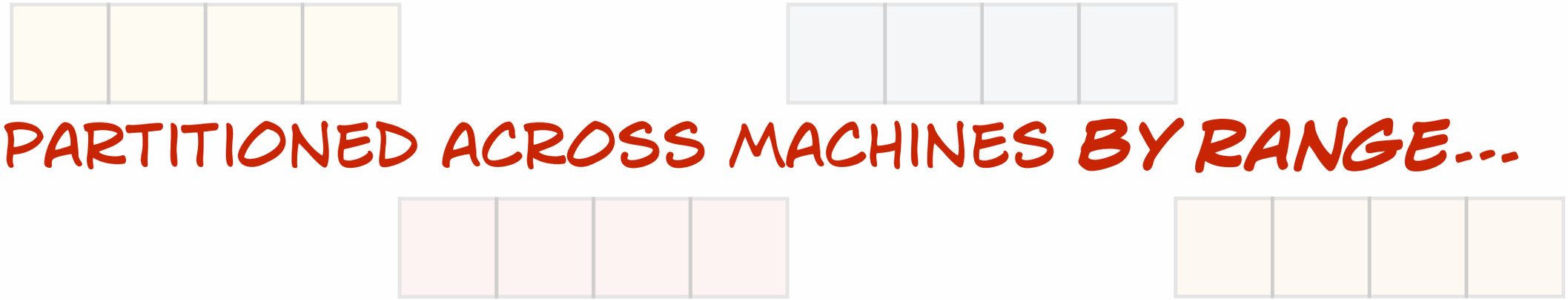
redhat.

Resilient distributed datasets



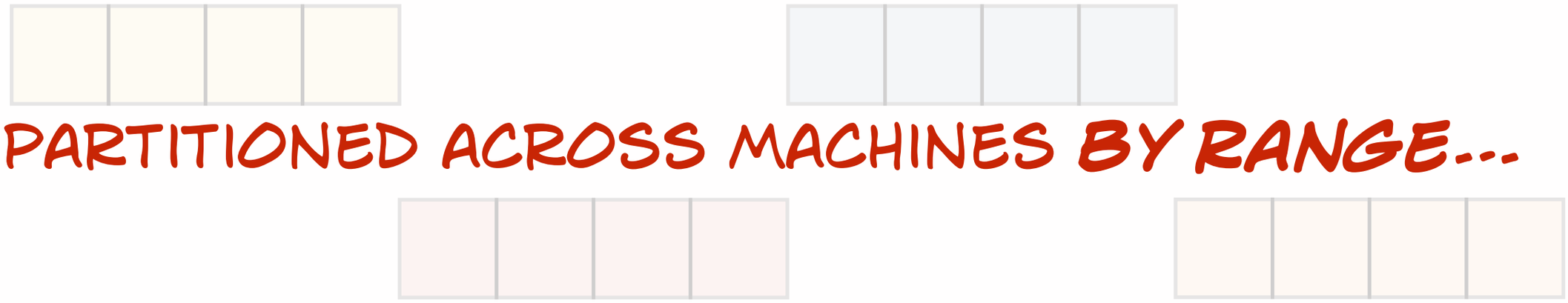
redhat.

Resilient distributed datasets

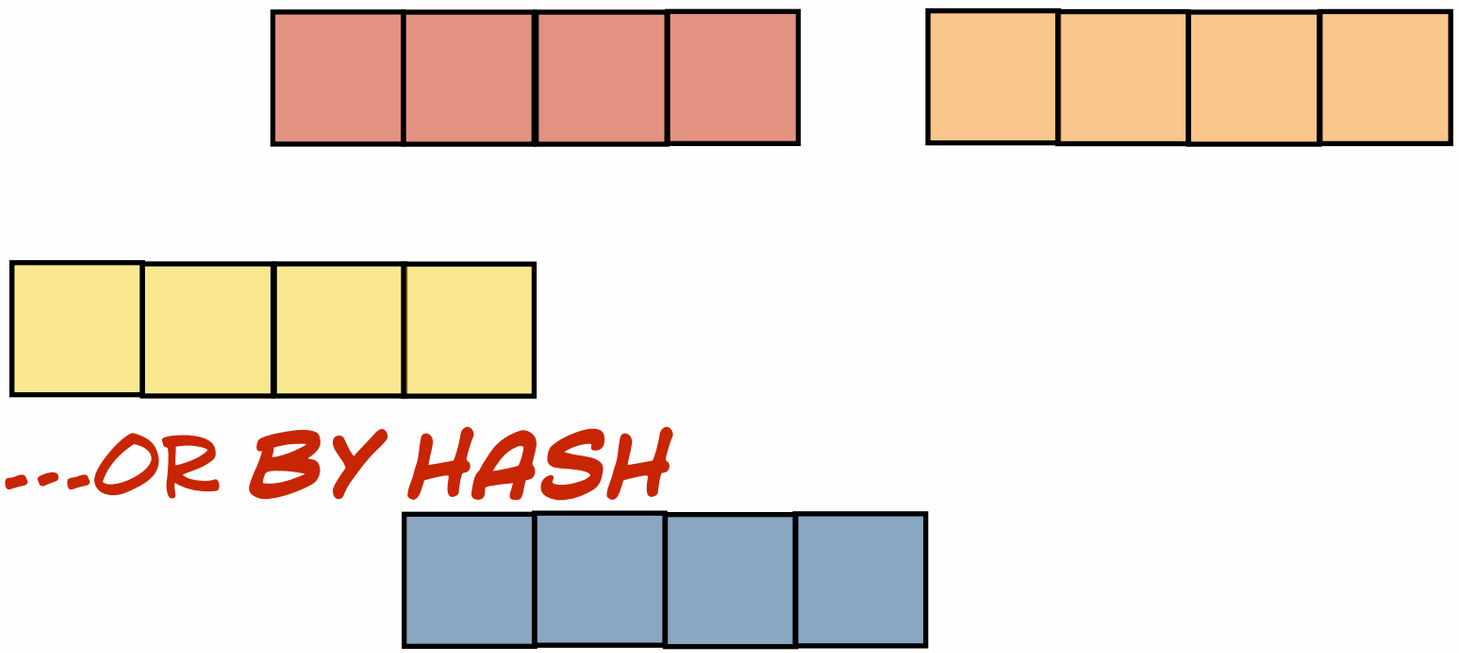


redhat.

Resilient distributed datasets



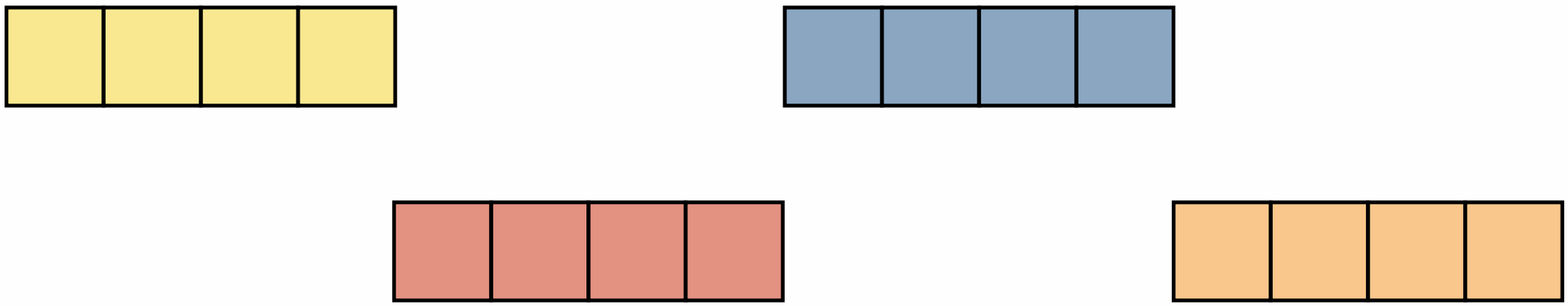
PARTITIONED ACROSS MACHINES BY RANGE...



...OR BY HASH

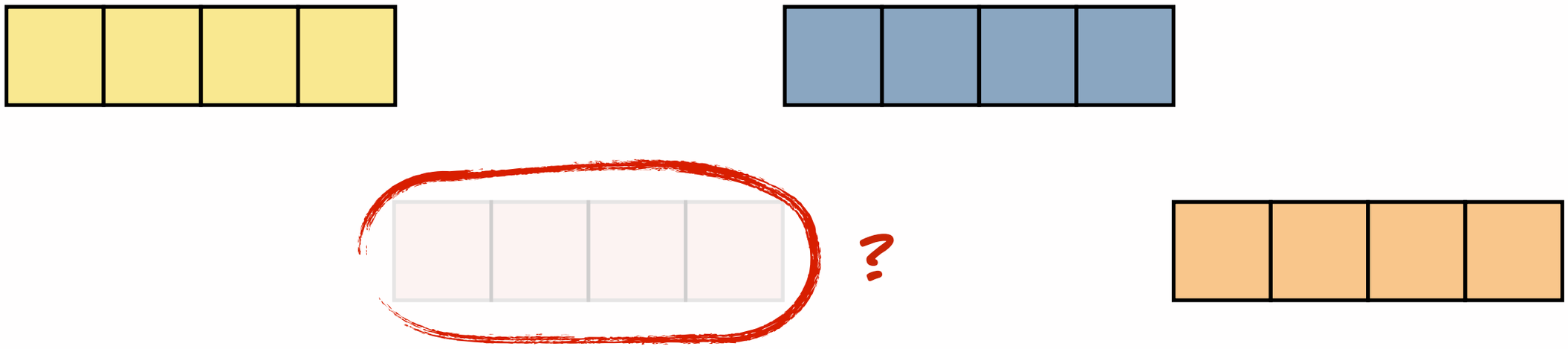
redhat.

Resilient distributed datasets



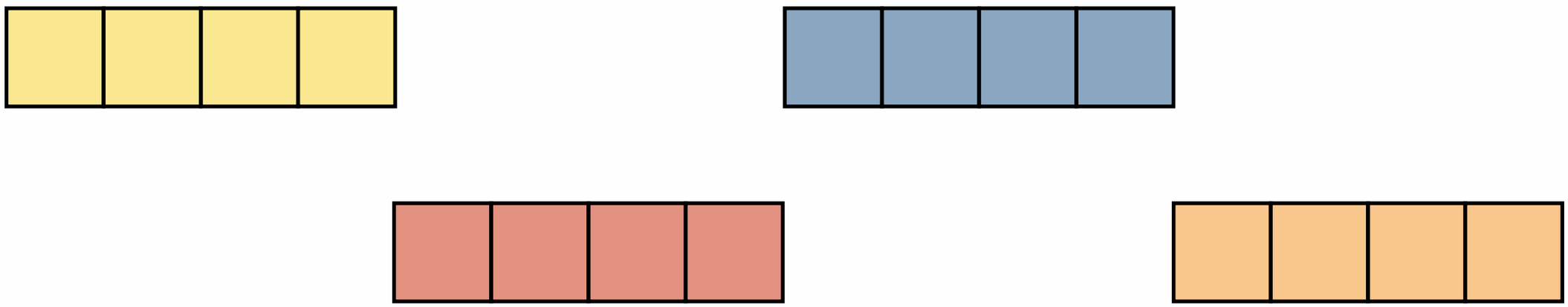
redhat.

Resilient distributed datasets



FAILURES MEAN PARTITIONS CAN DISAPPEAR...

Resilient distributed datasets



FAILURES MEAN PARTITIONS CAN **DISAPPEAR**...

...BUT THEY CAN BE **RECONSTRUCTED**!

redhat.

RDDs are partitioned, immutable, lazy collections

redhat.

RDDs are partitioned, immutable, lazy collections

TRANSFORMATIONS CREATE NEW RDDS
THAT ENCODE A DEPENDENCY DAG

ACTIONS RESULT IN EXECUTING CLUSTER
JOBS & RETURN VALUES TO THE DRIVER

redhat.

Creating RDDs

- **From a collection:** `parallelize()`
- **…a local or remote file:** `textFile()`
- **…or HDFS:** `hadoopFile(); sequenceFile();objectFile()`
- **(These all act lazily)**

RDD[T] transformations

- `map(f: T=>U): RDD[U]`
- `flatMap(f: T=>Seq[U]): RDD[U]`
- `filter(f: T=>Boolean): RDD[T]`
- `distinct(): RDD[T]`
- `keyBy(f: T=>K): RDD[(K, T)]`

redhat.

RDD[(K,V)] transformations

- `sortByKey(): RDD[(K,V)]`
- `groupByKey(): RDD[(K,Seq[V])]`
- `reduceByKey(f: (V,V)=>V): RDD[(K,V)]`
- `join(other: RDD[(K,W)]): RDD[(K,(V,W))]`

redhat.

RDD[(K,V)] transformations

- `sortByKey(): RDD[(K,V)]`
- `groupByKey(): RDD[(K,Seq[V])]`
- `reduceByKey(f: (V,V)=>V): RDD[(K,V)]`
- `join(other: RDD[(K,W)]): RDD[(K,(V,W))]`

--AND MANY OTHERS!

redhat.

RDD[T] actions

- `collect(): Array[T]`
- `count(): Long`
- `reduce(f: (T,T)=>T): T`
- `saveAsTextFile(path)`
- `saveAsSequenceFile(path)`

redhat.

RDD[T] actions

(REMEMBER: ALL ACTIONS ARE EAGER)

- `collect(): Array[T]`
- `count(): Long`
- `reduce(f: (T,T)=>T): T`
- `saveAsTextFile(path)`
- `saveAsSequenceFile(path)`

...AND MANY OTHERS!

redhat

Example: word count in Spark

```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line =>
                  line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

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

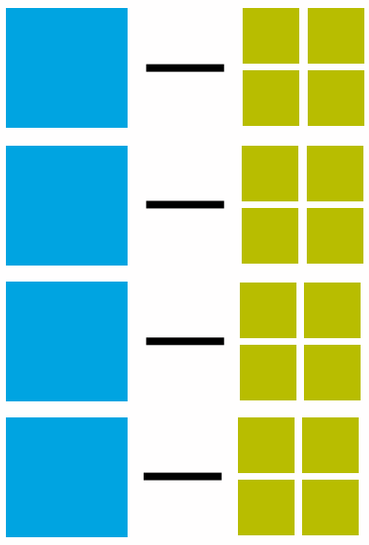
redhat

Example: word count in Spark

```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line =>
                line.split(" "))
             .map(word => (word, 1))
             .reduceByKey(_ + _)

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

Example: word count in Spark

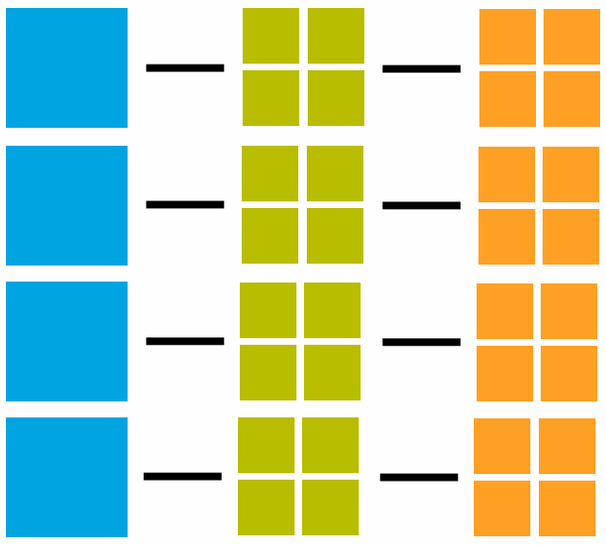
```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line =>
               line.split(" "))
             .map(word => (word, 1))
             .reduceByKey(_ + _)

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



redhat

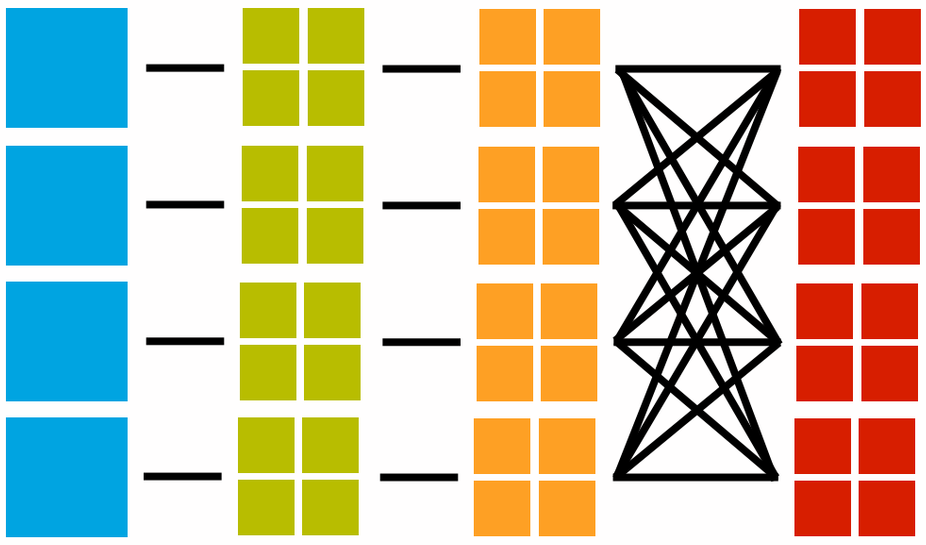
Example: word count in Spark

```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line =>
    line.split(" "))
.map(word => (word, 1))
.reduceByKey(_ + _)

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



Example: word count in Spark

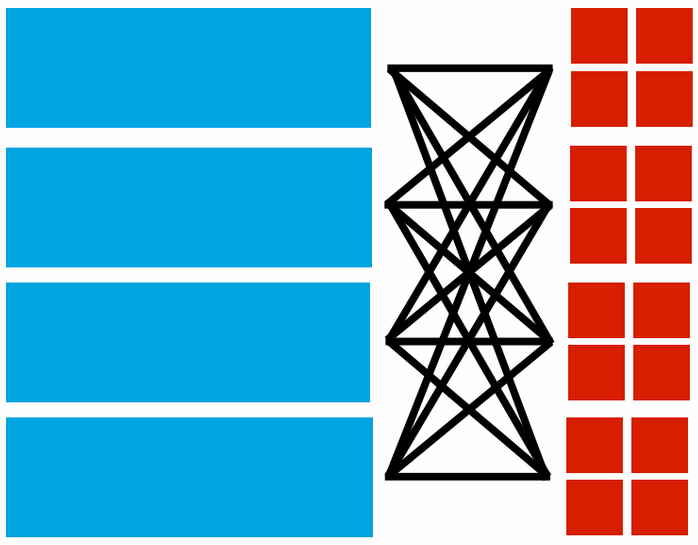
```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line =>
               line.split(" "))
             .map(word => (word, 1))
             .reduceByKey(_ + _)
```



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

Example: word count in Spark

```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line =>
    line.split(" "))
.map(word => (word, 1))
.reduceByKey(_ + _)

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



Case study: bicycling data

Metrics available to cyclists

	direct	***derivative or lagging***
effort	cadence	heart rate
	torque	calories
	power	training stress
course	position	grade
	elevation	speed
	temperature	distance

Metrics available to cyclists

	direct	*derivative or lagging*
effort	cadence torque power	heart rate calories training stress
course	position elevation temperature	grade speed distance

(TYPICALLY, ONE SAMPLE PER SECOND)

Metrics available to cyclists

	direct	*derivative or lagging*
effort	cadence torque power	heart rate calories training stress
course	position elevation temperature	grade speed distance

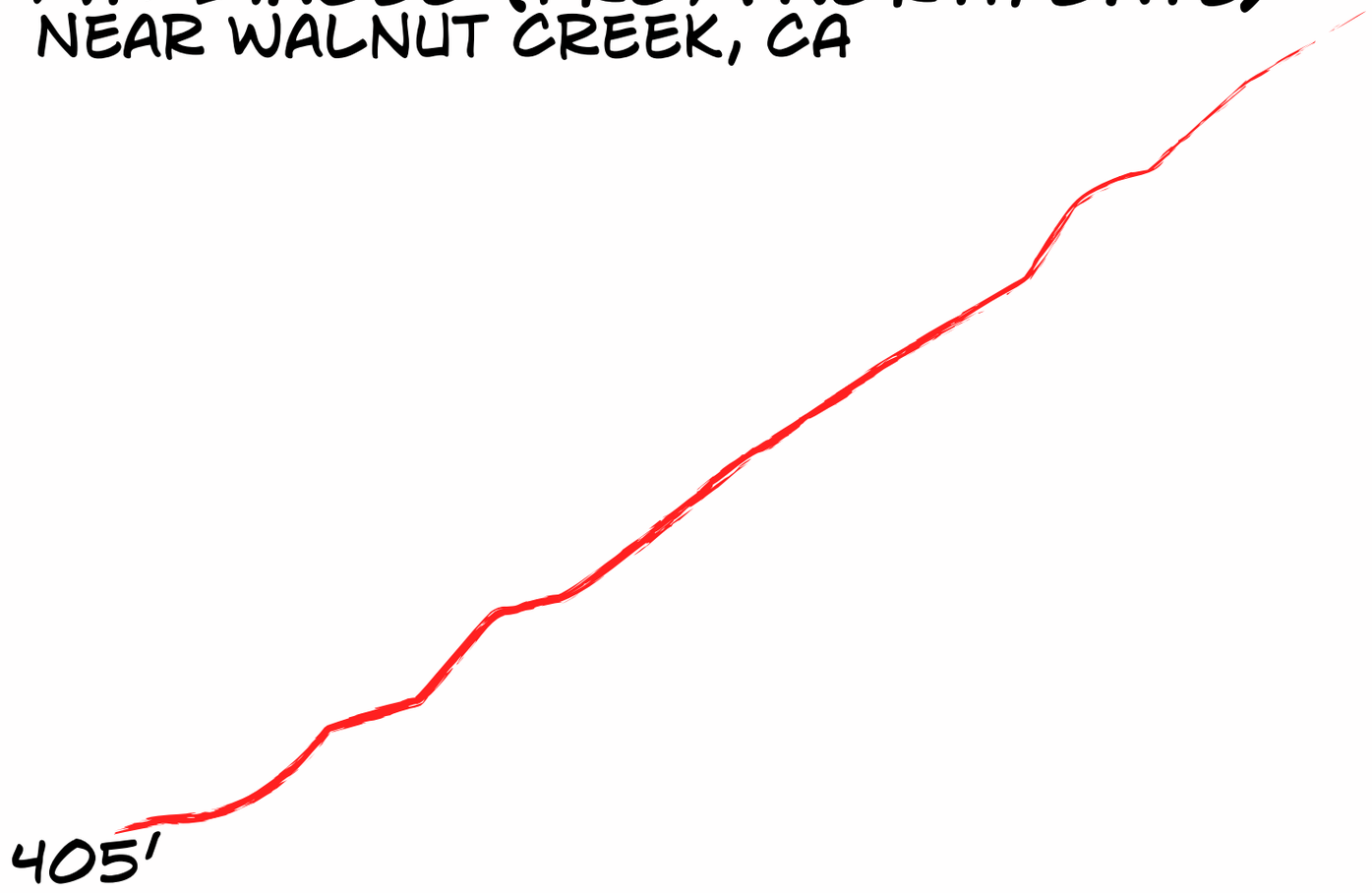
(TYPICALLY, ONE SAMPLE PER SECOND)

“Where should I do intervals?”

redhat.

MT. DIABLO (FROM NORTH GATE)
NEAR WALNUT CREEK, CA

3,970'
10.5 MILES



405'

"BUDDHA'S PALM"
NEAR CROSS PLAINS, WI

1,214'

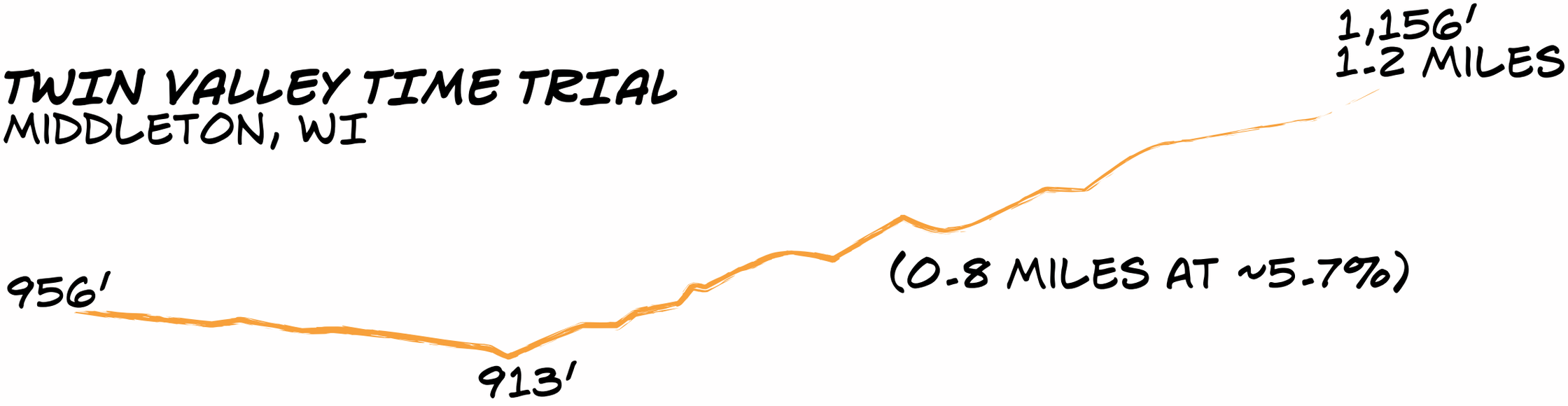
890'

16.8 MILE

redhat.

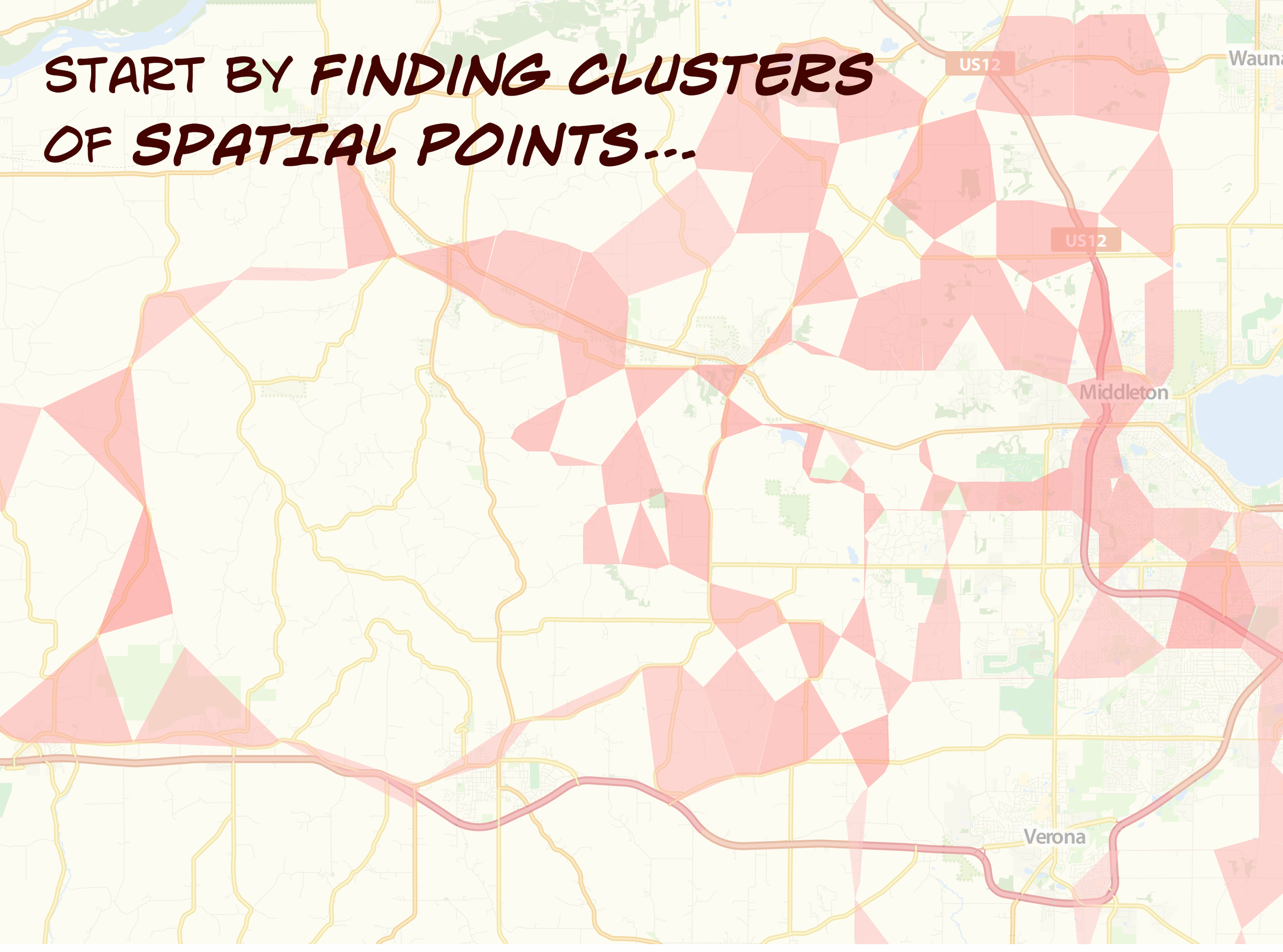
Finding best efforts

TWIN VALLEY TIME TRIAL
MIDDLETON, WI



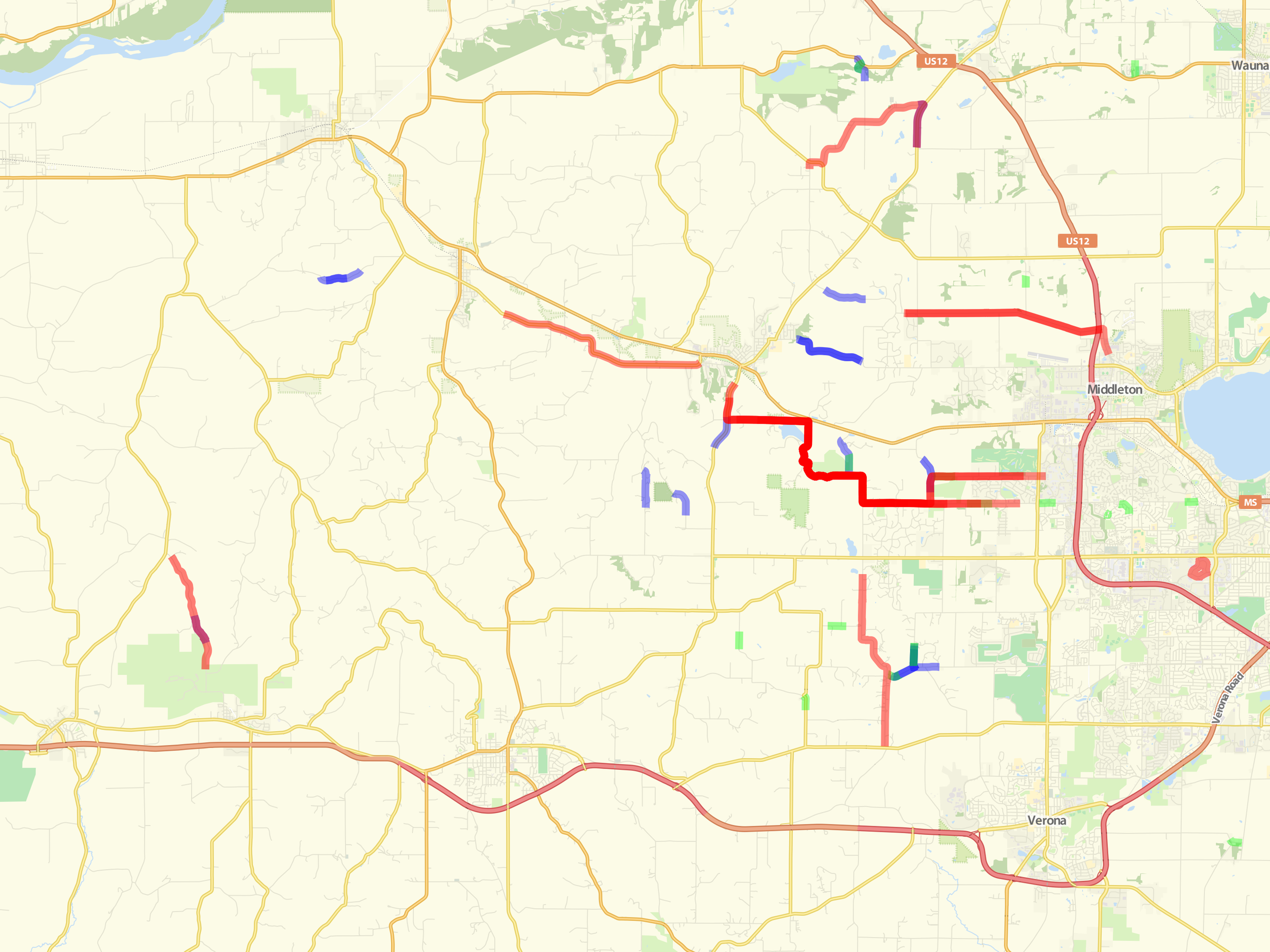
A FOUR-MINUTE **ALL-OUT** EFFORT INCLUDES **SIXTY STRONG** THREE-MINUTE EFFORTS.

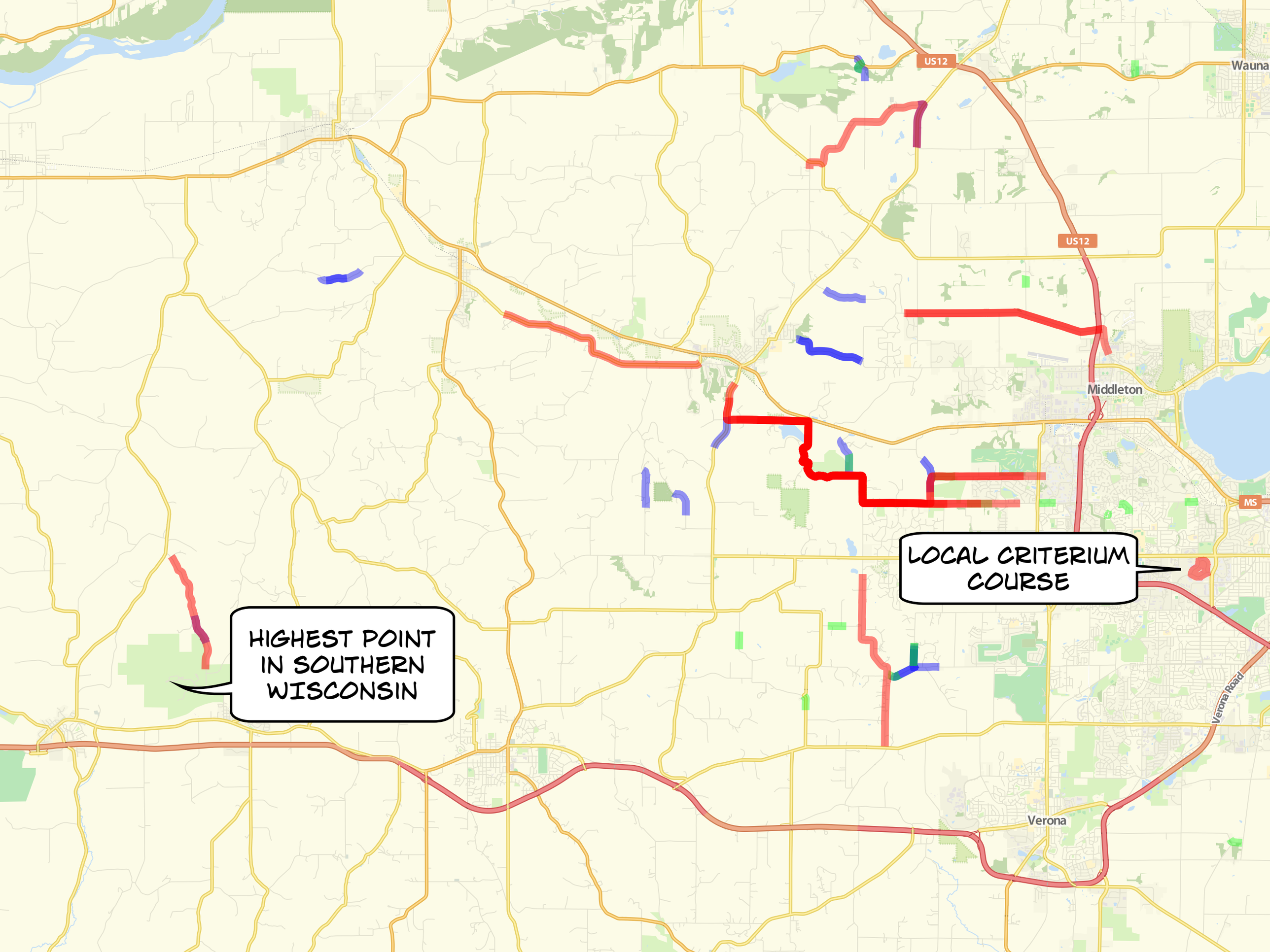
START BY FINDING CLUSTERS OF SPATIAL POINTS...

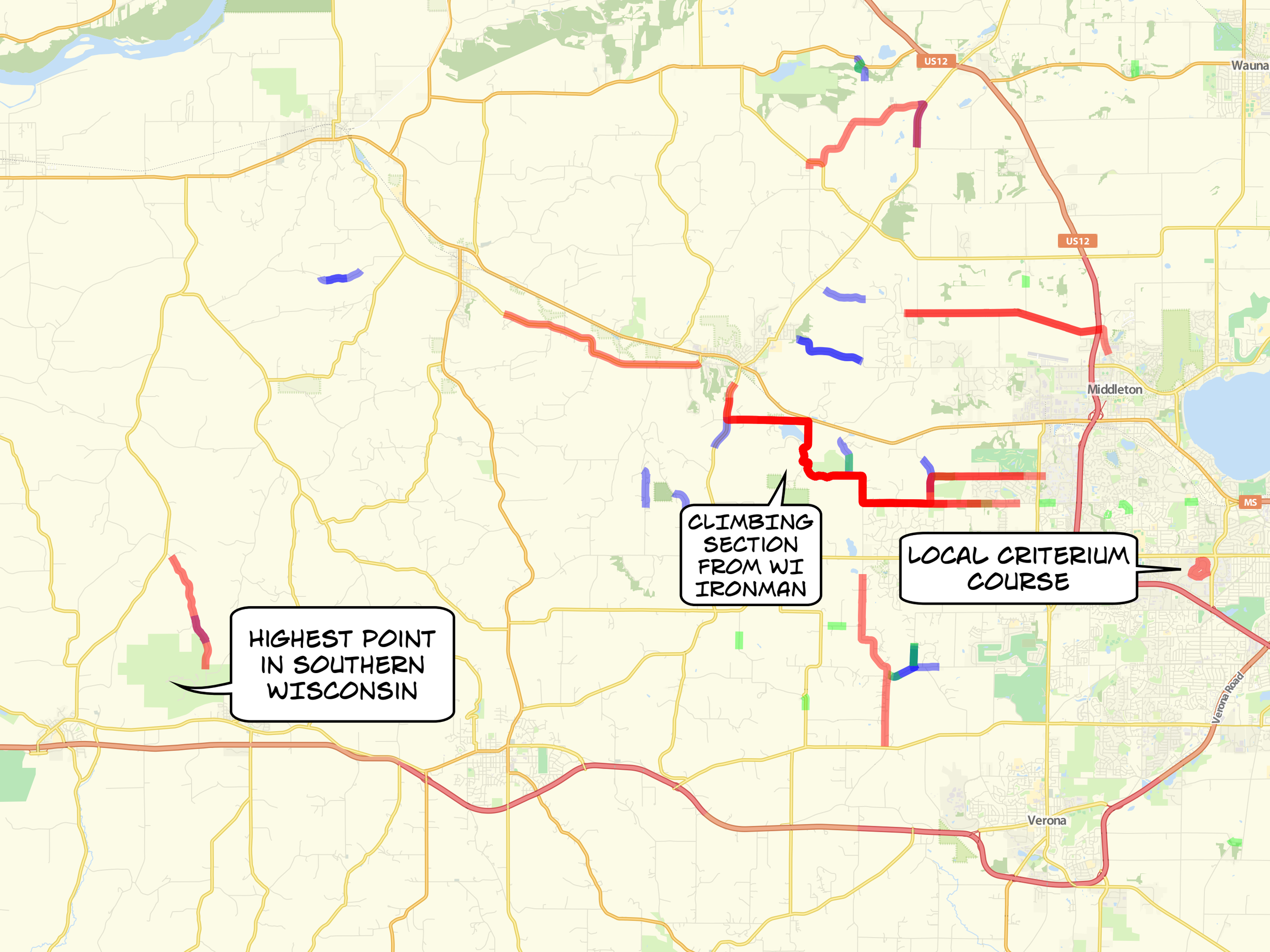


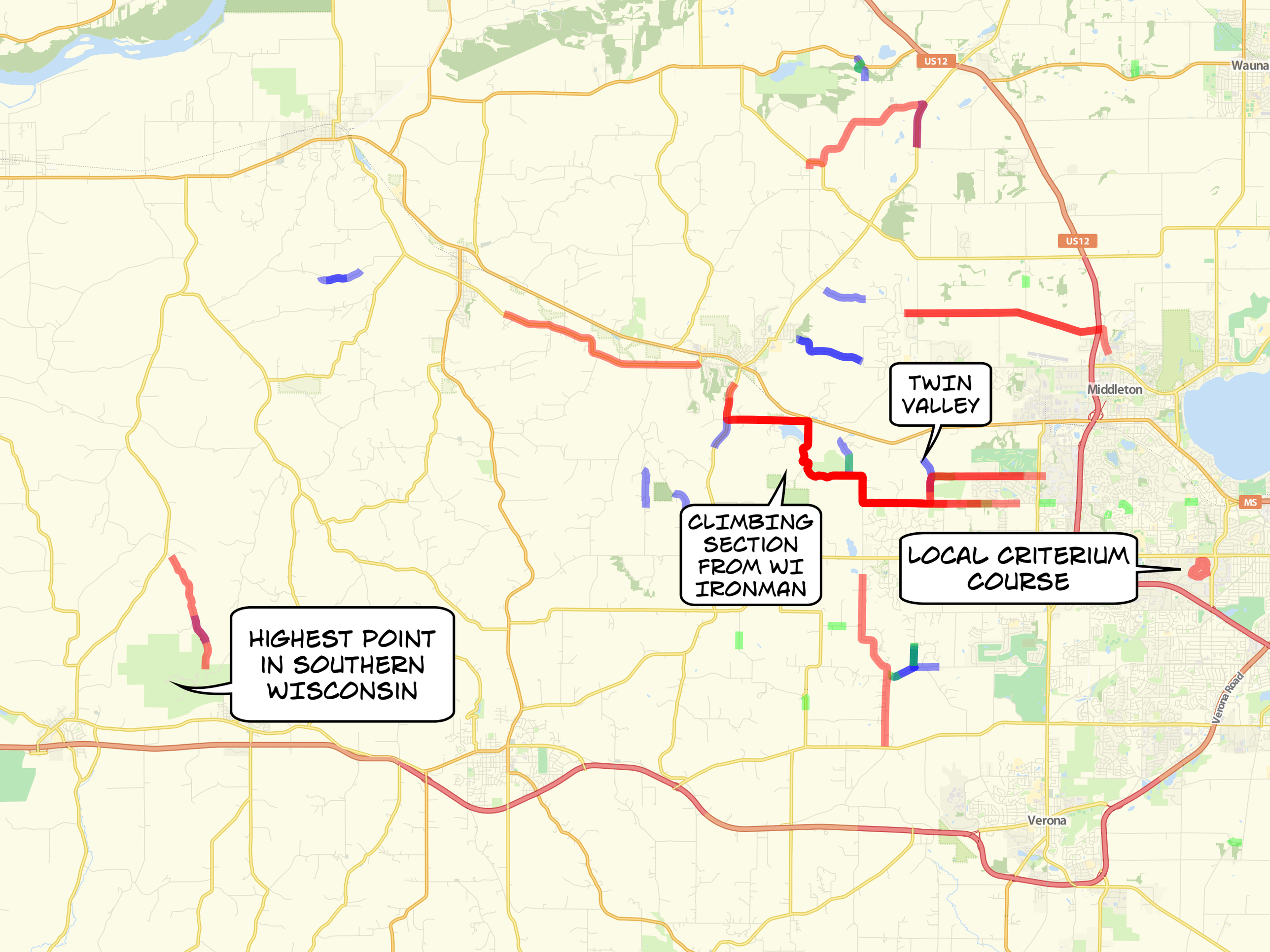
START BY *FINDING CLUSTERS* OF *SPATIAL POINTS*...

...AND THEN *CONSIDER ONLY* THE *BEST EFFORT* BETWEEN *A PAIR OF CLUSTERS!*


US12
US12
Middleton
Verona
Verona Road
MS


US12
US12
Wauna
Middleton
MS
LOCAL CRITERIUM COURSE
HIGHEST POINT IN SOUTHERN WISCONSIN
Verona Road
Verona


US12
US12
Wauna
Middleton
MS
CLIMBING SECTION FROM WI IRONMAN
LOCAL CRITERIUM COURSE
HIGHEST POINT IN SOUTHERN WISCONSIN
Verona Road
Verona

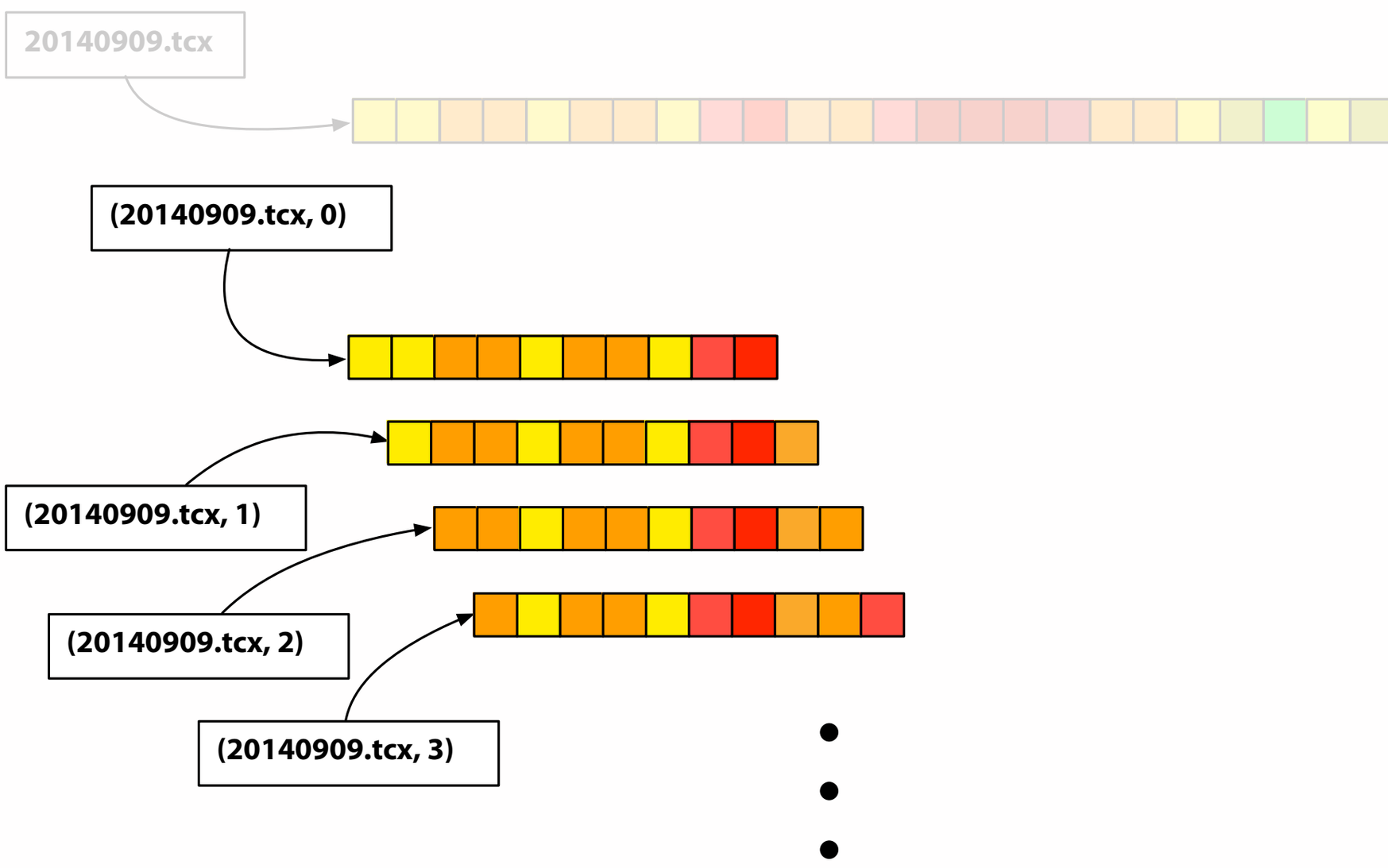

US12
US12
Wauna
Middleton
MS
TWIN VALLEY
CLIMBING SECTION FROM WI IRONMAN
LOCAL CRITERIUM COURSE
HIGHEST POINT IN SOUTHERN WISCONSIN
Verona
Verona Road

The prototype

redhat.

Windowed processing

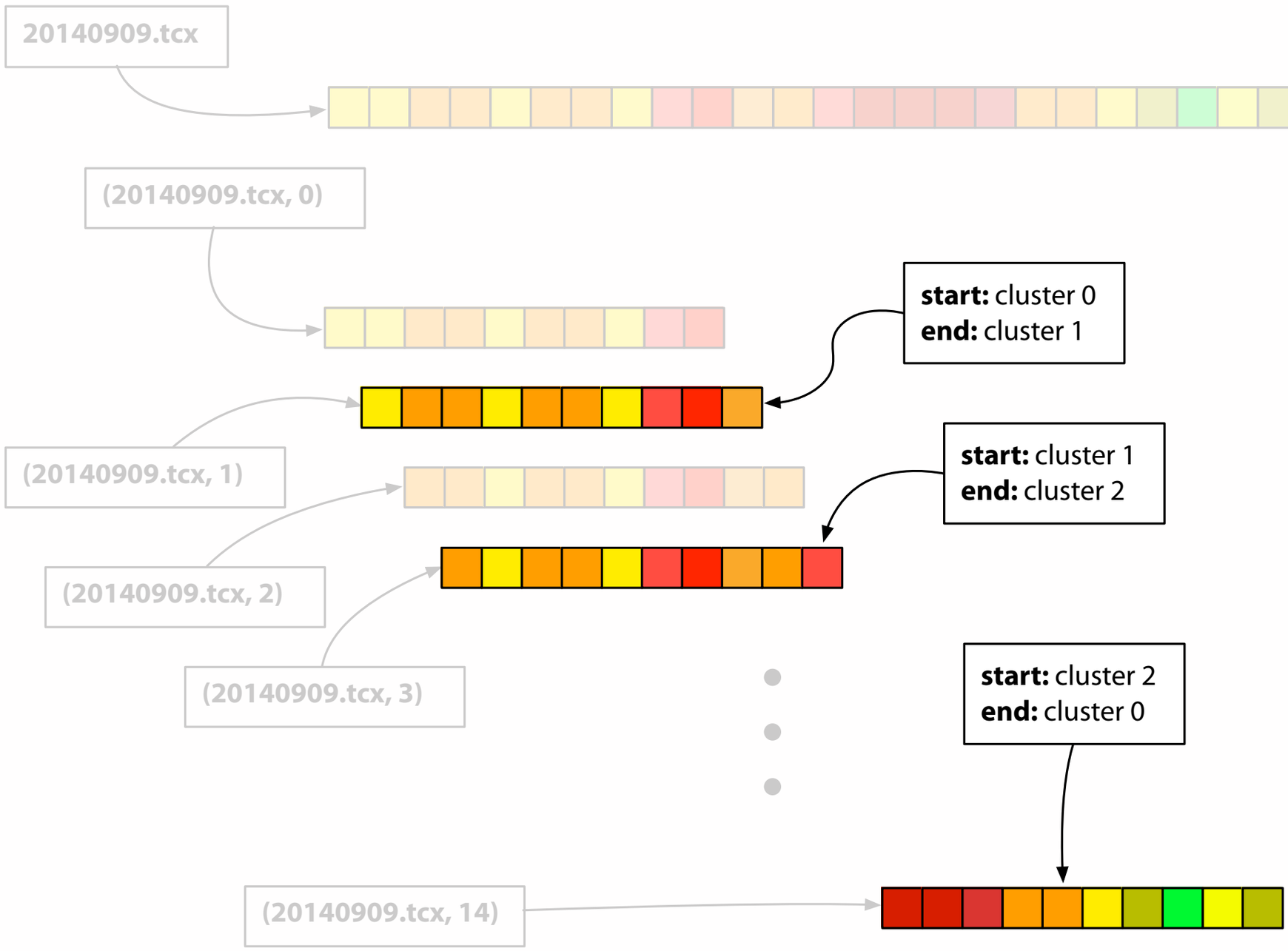
Windowed processing



redhat

KEEP & PLOT THE BEST WINDOWS FOR EACH SPATIAL CLUSTER PAIR!

Windowed processing



```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

redhat

TRANSFORM AN RDD OF TRACKPOINTS...

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  def windowsForActivities[U](data: RDD[TP], period: Int,
                              xform: (TP => U) = identity _) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity:String, stp:Seq[TP]) =>
        (stp sliding period).zipWithIndex.map {
          case (s,i) => ((activity, i), s.map(xform))
        }
    }
  }

  def identity(tp: Trackpoint) = tp
}
```

...TO AN RDD OF WINDOW IDS AND SAMPLE WINDOWS

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period, minify _).cache
  val clusterPairs = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), clusterPairsForWindow(samples, model))
  }
  val mmps = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
  }

// continued...
```

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period, minify _).cache
  val clusterPairs = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), clusterPairsForWindow(samples, model))
  }
  val mmps = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
}

// continued...
```

DIVIDE THE INPUT DATA INTO OVERLAPPING WINDOWS, KEYED BY ACTIVITY AND OFFSET (WE'LL CALL THIS KEY A WINDOW ID)

redhat.

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period, minify _).cache
  val clusterPairs = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), clusterPairsForWindow(samples, model))
  }
  val mmps = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
}

// continued...
```

IDENTIFY THE **SPATIAL CLUSTERS** THAT EACH WINDOW **STARTS AND ENDS IN**

redhat.

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period, minify _).cache
  val clusterPairs = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), clusterPairsForWindow(samples, model))
  }
  val mmps = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
}

// continued...
```

IDENTIFY THE **MEAN WATTAGE**
FOR EACH **WINDOW** OF SAMPLES

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

redhat.

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

FOR EACH WINDOW ID, JOIN ITS MEAN WATTAGES WITH ITS SPATIAL CLUSTERS

redhat.

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

TRANSPOSE THESE TUPLES SO THEY ARE KEYED BY SPATIAL CLUSTER PAIRS

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

KEEP ONLY THE BEST WATTAGE FOR EACH SPATIAL CLUSTER PAIR

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

PROJECT AWAY THE CLUSTER CENTERS

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

SORT BY WATTAGE IN DESCENDING ORDER; KEEP THE BEST TWENTY

redhat.

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

RE-KEY THE BEST EFFORTS BY WINDOW ID

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

GET THE ACTUAL SAMPLE WINDOWS FOR EACH EFFORT; PROJECT AWAY IDS

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = /* window IDs and raw sample windows */
  val clusterPairs = /* window IDs and spatial cluster pairs */
  val mmps = /* window IDs and mean wattages for each window */

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  app.context.parallelize(top20)
   .map { case (watts, (act, idx)) => ((act, idx), watts) }
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

Improving the prototype

Broadcast large static data

redhat.

Broadcast variables

```
// phoneBook maps (given name, surname) -> phone number digits
val phoneBook: Map[(String, String), String] = initPhoneBook()
val names: RDD[(String, String)] = /* ... */
```

```
val directory = names.map {
  case name @ (first, last) => (name, phoneBook.getOrElse("555-1212"))
}
```

Broadcast variables

```
// phoneBook maps (given name, surname) -> phone number digits
val phoneBook: Map[(String, String), String] = initPhoneBook()
val names: RDD[(String, String)] = /* ... */
```

```
val directory = names.map {
  case name @ (first, last) => (name, phoneBook.getOrElse("555-1212"))
}
```

PHONEBOOK WILL BE COPIED AND DESERIALIZED FOR EACH TASK!

Broadcast variables

```
// phoneBook maps (given name, surname) -> phone number digits
val phoneBook: Map[(String, String), String] = initPhoneBook()
val names: RDD[(String, String)] = /* ... */
val pbb = sparkContext.broadcast(phoneBook)
```

```
val directory = names.map {
  case name @ (first, last) => (name, pbb.value.getOrElse("555-1212"))
}
```

BROADCASTING PHONEBOOK MEANS IT CAN BE DESERIALIZED ONCE AND CACHED ON EACH NODE!

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period, minify _)
  val clusterPairs = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), clusterPairsForWindow(samples, model))

  // ...

}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)
  val clusterPairs = windowedSamples.map {
    case ((act, idx), samples) =>
      ((act, idx), clusterPairsForWindow(samples, model.value))

  // rest of function unchanged

}
```

Cache only when necessary

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period).cache

  // ...

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  // ...
}
```

redhat

KEEPING EVERY WINDOW IN MEMORY...

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period).cache

  // ...

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  // ...
}
```

redhat

KEEPING EVERY WINDOW IN MEMORY...

```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period).cache

  // ...

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  // ...
}
```

...EVEN THOUGH RECOMPUTING WINDOWS IS INCREDIBLY CHEAP AND YOU'LL NEED ONLY A TINY FRACTION OF WINDOWS?

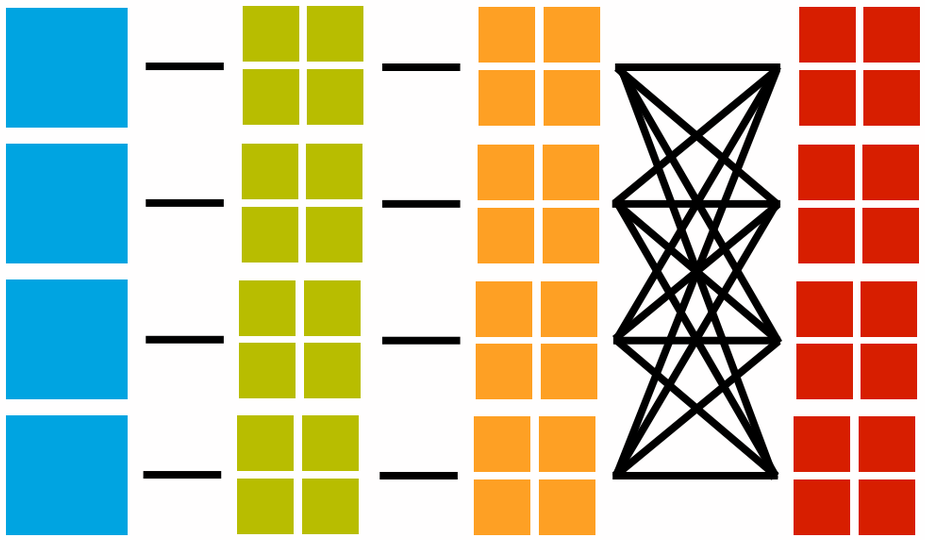
```
def bestsForPeriod(data: RDD[TP], period: Int, app: SLP, model: KMeansModel) = {
  val windowedSamples = windowsForActivities(data, period).cache

  // ...

  val top20 = mmps.join(clusterPairs)
   .map { case ((act, idx), (watts, (cls1, cls2))) =>
            ((cls1, cls2), (watts, (act, idx))) }
   .reduceByKey ((a, b) => if (a._1 > b._1) a else b)
   .map { case ((cls1, cls2), (watts, (act, idx))) => (watts, (act, idx)) }
   .sortByKey(false)
   .take(20)

  // ...
}
```

ELIMINATING UNNECESSARY MEMORY PRESSURE CAN LEAD TO A SUBSTANTIAL SPEEDUP!

Avoid shuffles when possible

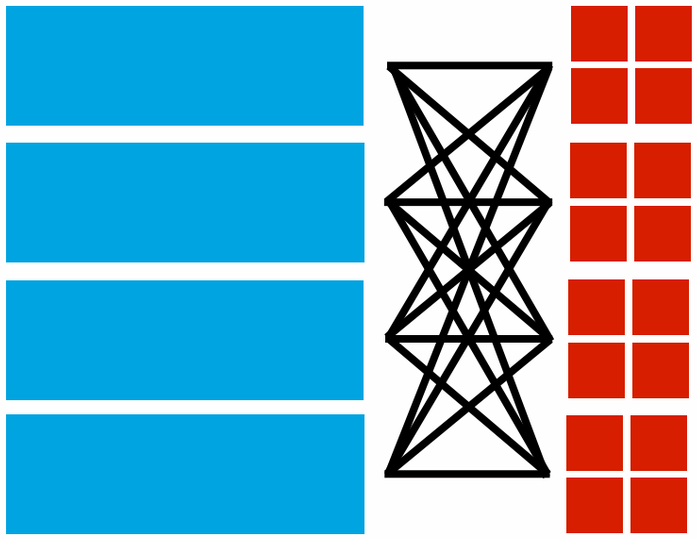
redhat.

TASKS



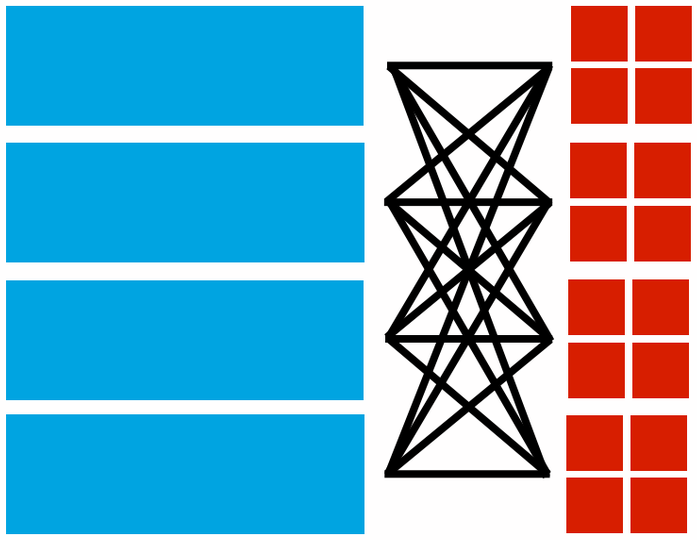
redhat.

STAGES



redhat.

STAGES



WE WANT TO AVOID ALL UNNECESSARY SHUFFLES

```
  def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                     app: SLP, model: Broadcast[KMeansModel]) = {
    val windowedSamples = windowsForActivities(data, period, minify _)

    val bests = windowedSamples.map {
      case ((act, idx), samples)  => (
        clusterPairsForWindow(samples, model.value),
        ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
      )
    }.cache

    val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
     .map { case ((_, _), keep) => keep }
     .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
        case ((_, _), watts) => -watts
      })

    app.context.parallelize(top20)
     .join (windowedSamples)
     .map { case ((act, idx), (watts, samples)) => (watts, samples) }
     .collect
  }
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)

  val bests = windowedSamples.map {
    case ((act, idx), samples)  => (
      clusterPairsForWindow(samples, model.value),
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
    )
  }.cache

  val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
   .map { case ((_, _), keep) => keep }
   .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
      case ((_, _), watts) => -watts
    })

  app.context.parallelize(top20)
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

redhat.

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)

  val bests = windowedSamples.map {
    case ((act, idx), samples)  => (
      clusterPairsForWindow(samples, model.value),
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
    )
  }.cache

  val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
   .map { case ((_, _), keep) => keep }
   .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
      case ((_, _), watts) => -watts
    })

  app.context.parallelize(top20)
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

START AND END CLUSTERS

redhat.

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)

  val bests = windowedSamples.map {
    case ((act, idx), samples)  => (
      clusterPairsForWindow(samples, model.value),
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
    )
  }.cache

  val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
   .map { case ((_, _), keep) => keep }
   .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
      case ((_, _), watts) => -watts
    })

  app.context.parallelize(top20)
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

START AND END CLUSTERS

WINDOW IDS AND MEAN WATTAGES

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)

  val bests = windowedSamples.map {
    case ((act, idx), samples)  => (
      clusterPairsForWindow(samples, model.value),
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
    )
  }.cache
```

ELIMINATE A JOIN AND A TRANSPOSE

```
  val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
   .map { case ((_, _), keep) => keep }
   .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
      case ((_, _), watts) => -watts
    })

  app.context.parallelize(top20)
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

redhat.

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)

  val bests = windowedSamples.map {
    case ((act, idx), samples)  => (
      clusterPairsForWindow(samples, model.value),
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
    )
  }.cache

  val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
   .map { case ((_, _), keep) => keep }
   .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
      case ((_, _), watts) => -watts
   })
  app.context.parallelize(top20)
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

(USE THE RIGHT API CALLS!)

redhat.

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)

  val bests = windowedSamples.map {
    case ((act, idx), samples)  => (
      clusterPairsForWindow(samples, model.value),
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
    )
  }.cache

  val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
   .map { case ((_, _), keep) => keep }
   .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
      case ((_, _), watts) => -watts
    })

  app.context.parallelize(top20)
   .join (windowedSamples)
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
   .collect
}
```

ELIMINATE A TRANSPOSE

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val windowedSamples = windowsForActivities(data, period, minify _)

  val bests = windowedSamples.map {
    case ((act, idx), samples)  => (
      clusterPairsForWindow(samples, model.value),
      ((act, idx), samples.map(_.watts).reduce(_ + _) / samples.size)
    )
  }.cache

  val top20 = bests.reduceByKey ((a, b) => if (a._2 > b._2) a else b)
   .map { case ((_, _), keep) => keep }
   .takeOrdered(20)(Ordering.by[((String, Int), Double), Double] {
      case ((_, _), watts) => -watts
    })

  app.context.parallelize(top20)
   .join (windowedSamples)
   .collect
   .map { case ((act, idx), (watts, samples)) => (watts, samples) }
}
```

ELIMINATE A TRANSPOSE...OR TWO!

Embrace laziness

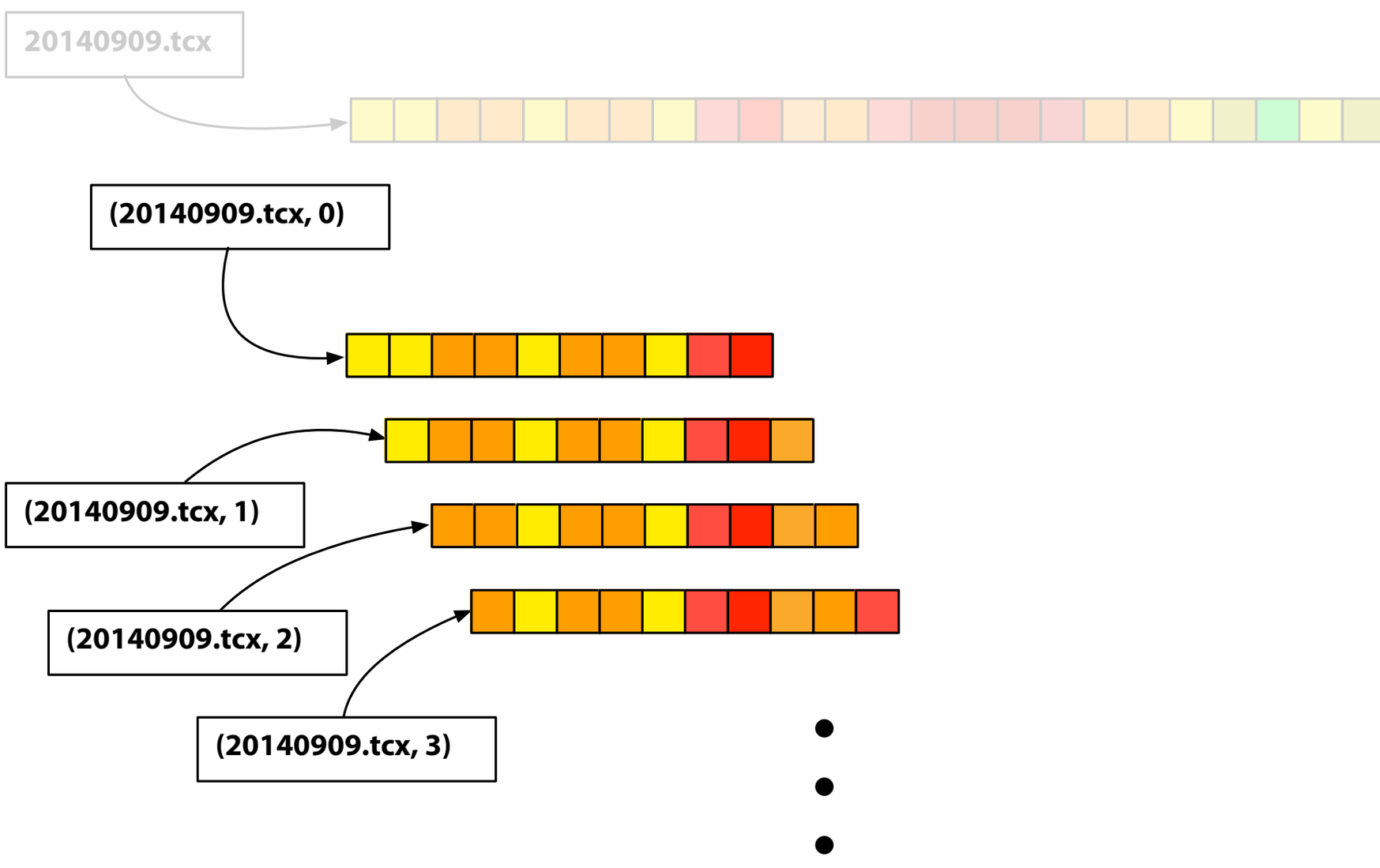
Embrace laziness

(ONLY PAY FOR WHAT YOU USE)

redhat.

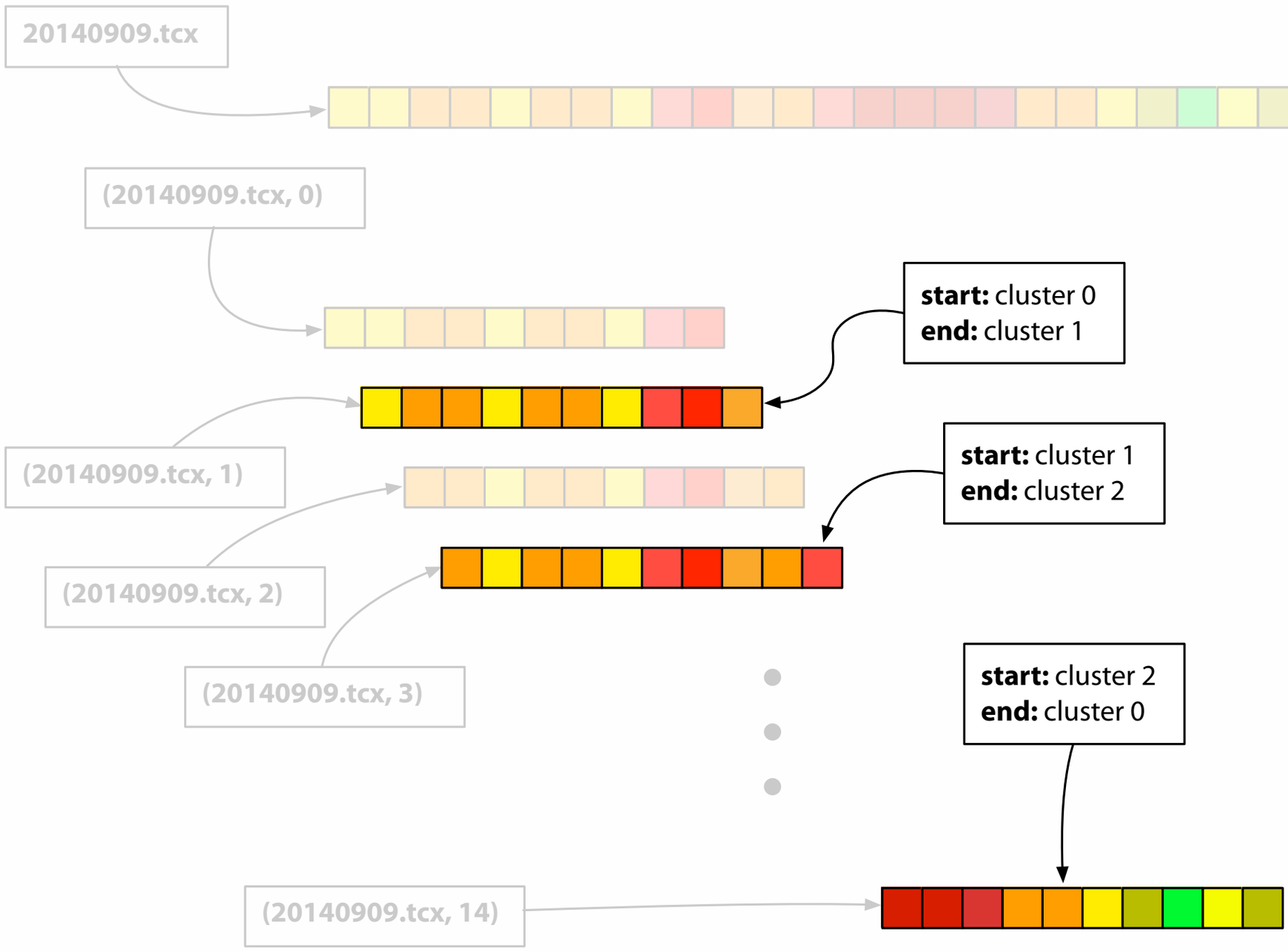
Windowed processing redux

Windowed processing redux



redhat.

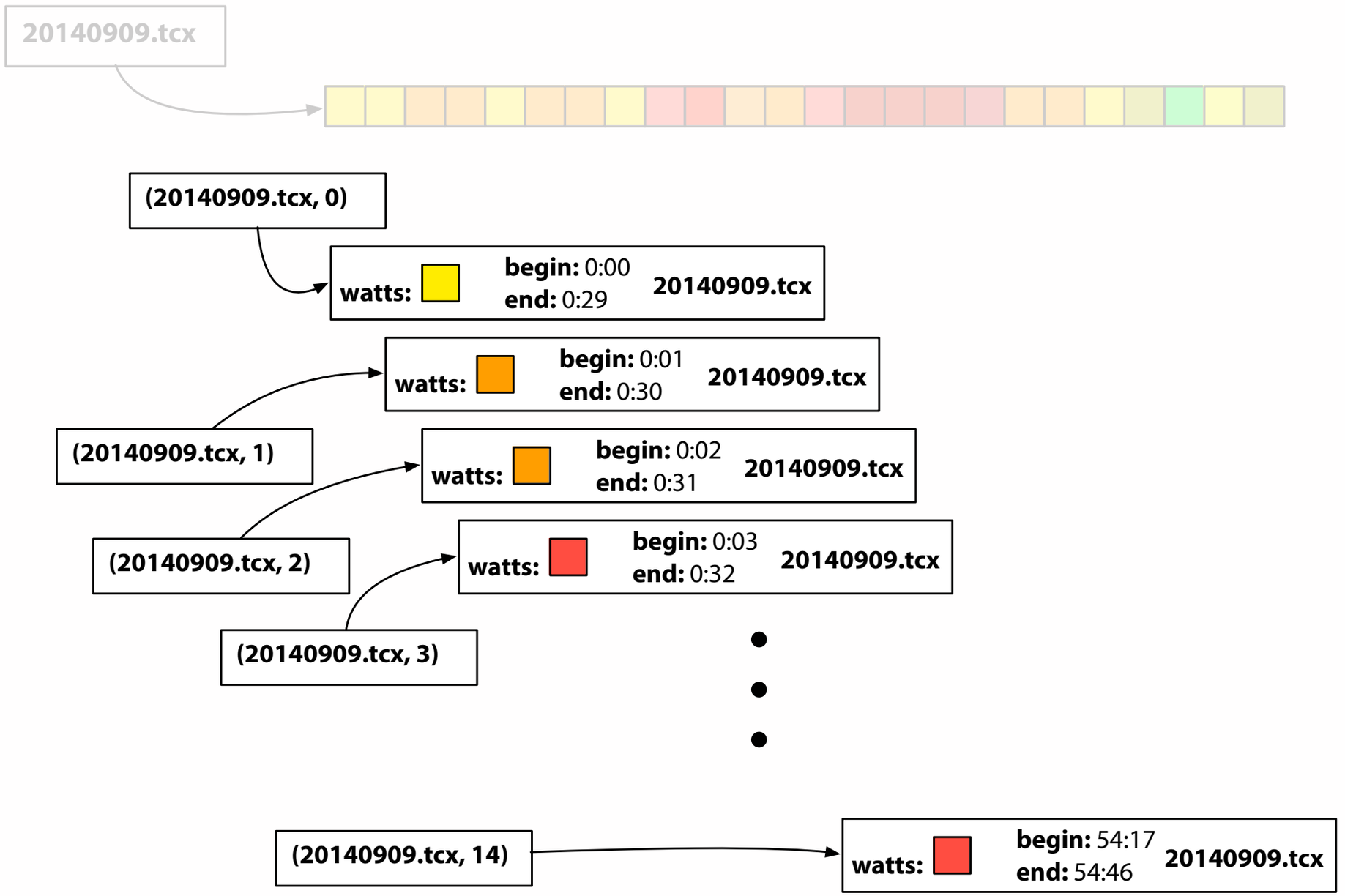
Windowed processing redux



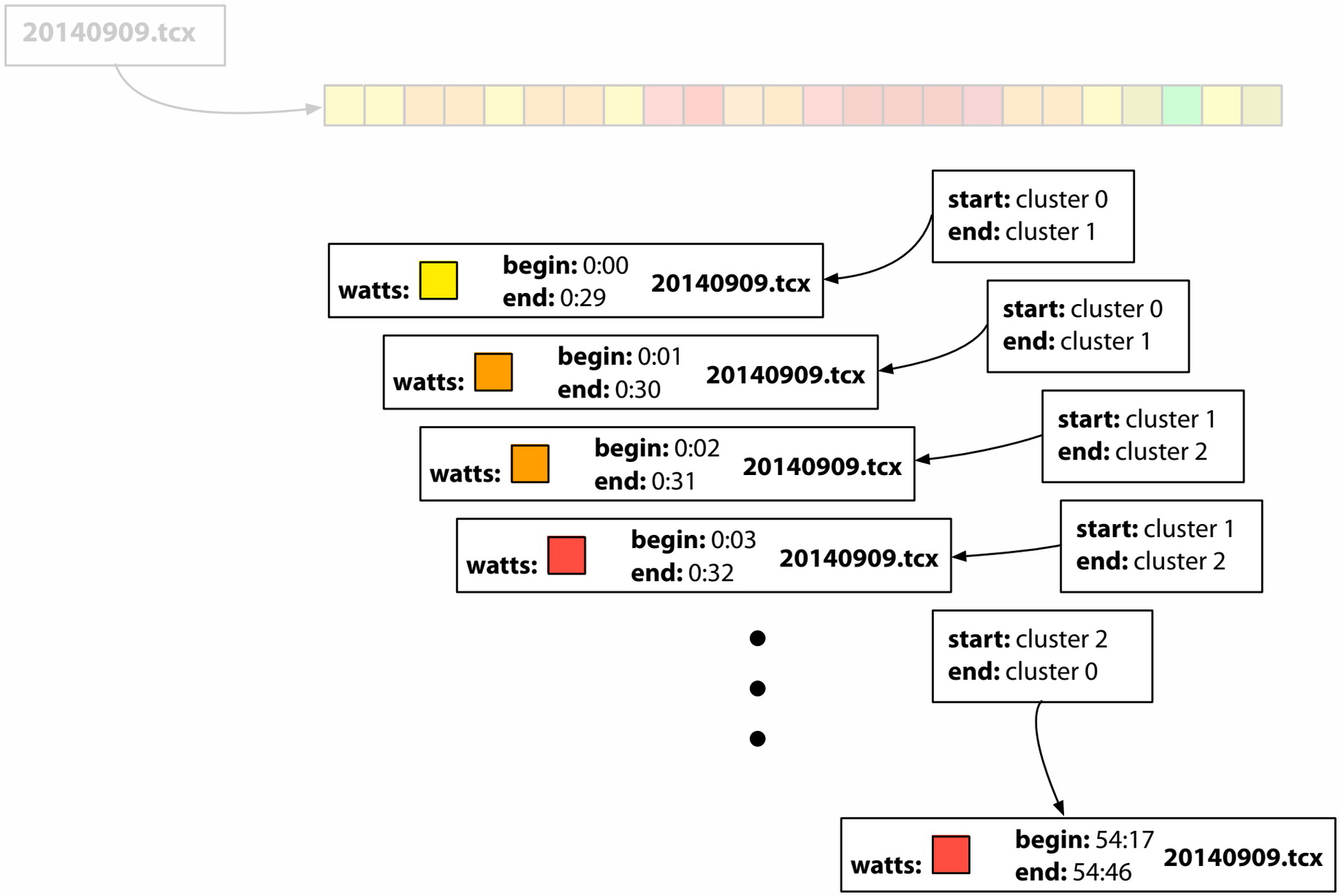
redhat.

Lazy windowed processing

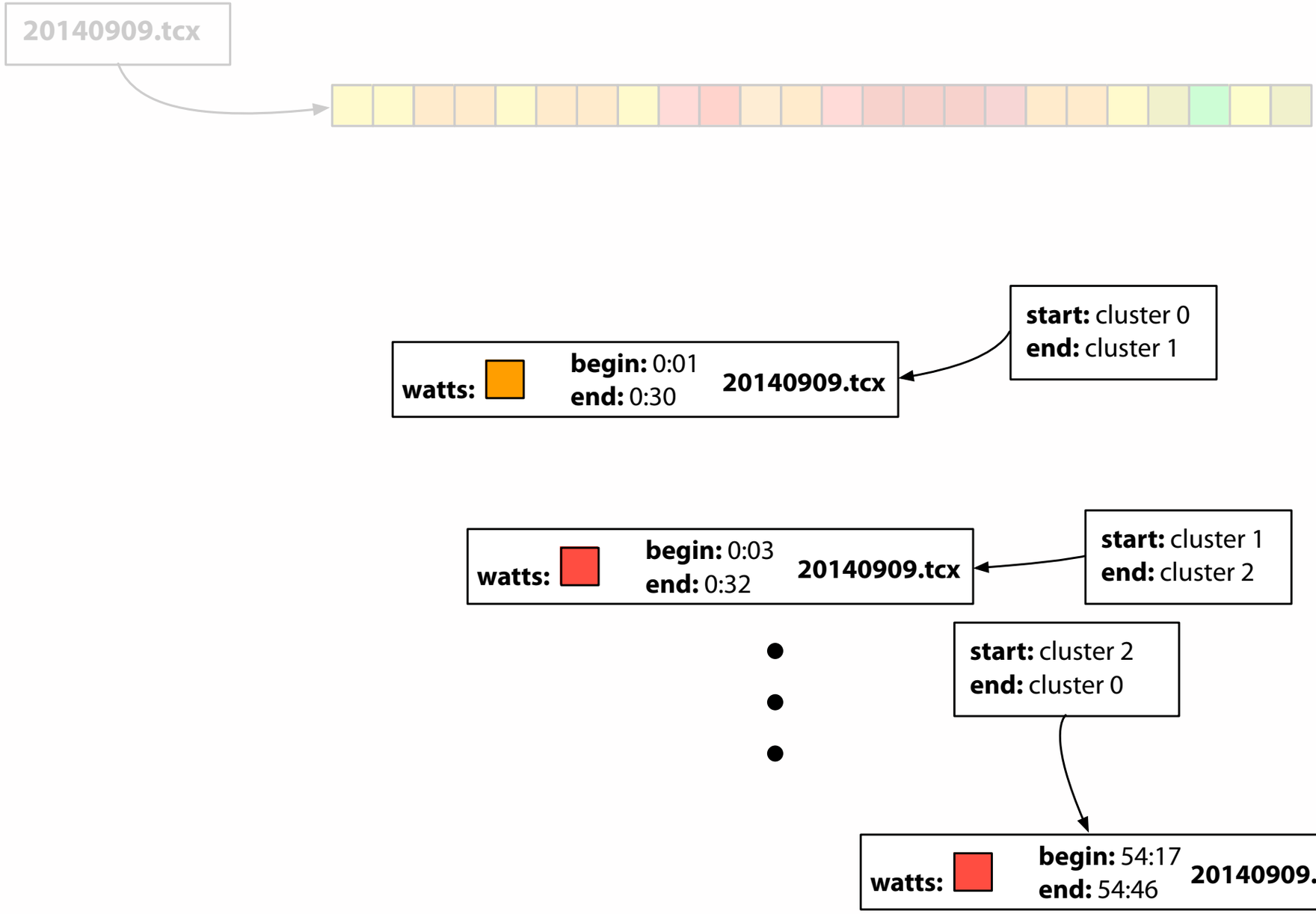
Lazy windowed processing



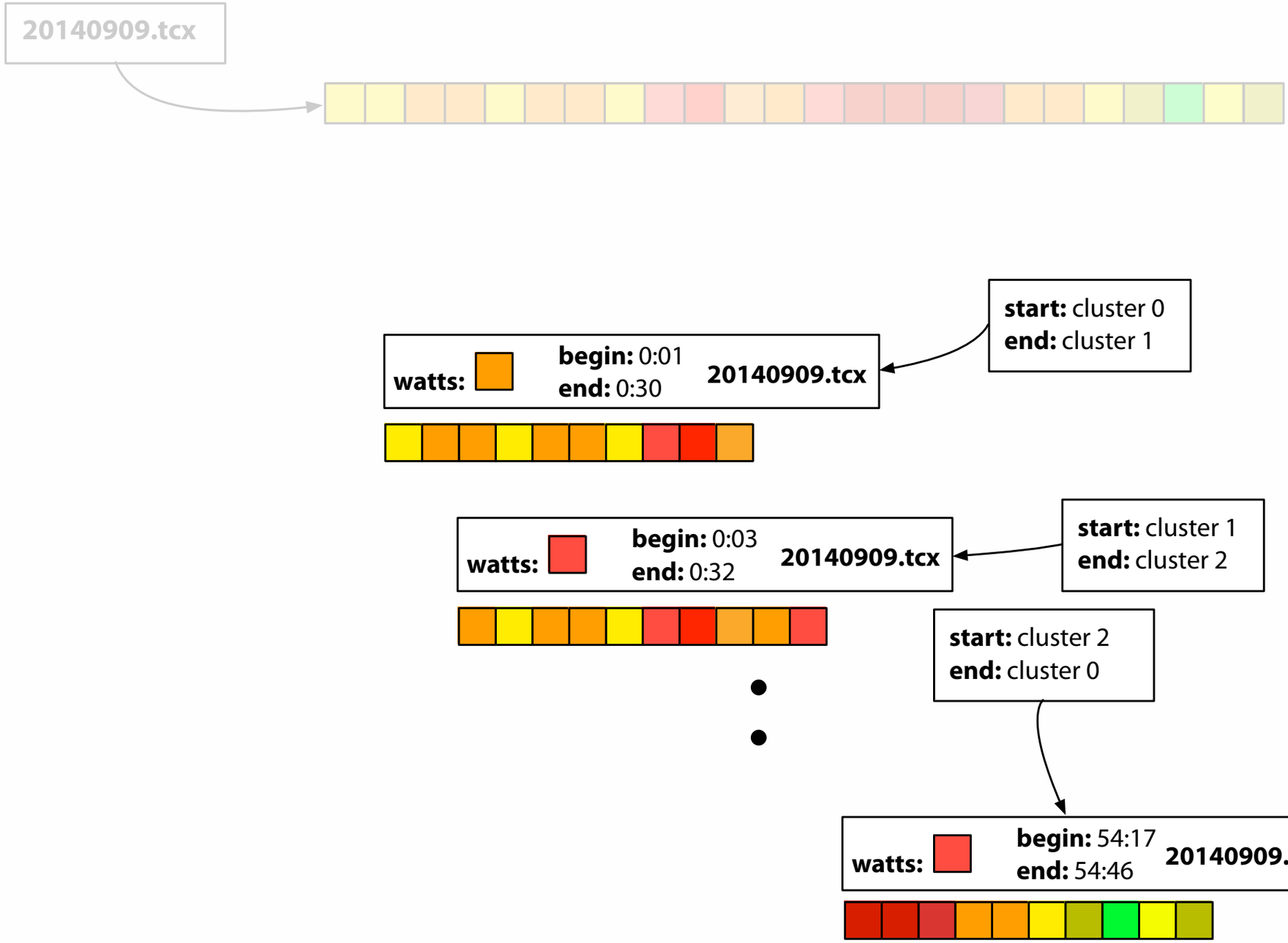
Lazy windowed processing



Lazy windowed processing



Lazy windowed processing



```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  // ...

  def applyWindowedNoZip[U: ClassTag](data: RDD[TP], period: Int,
                                      xform: ((String, Seq[TP]) => U)) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity: String, stp:Seq[TP]) =>
        (stp sliding period).map { xform(activity, _) }
    }
  }
}
```

```
trait ActivitySliding {
  import org.apache.spark.rdd.RDD
  import com.freevariable.surlaplaque.data.{Trackpoint => TP}

  // ...

  def applyWindowedNoZip[U: ClassTag](data: RDD[TP], period: Int,
                                      xform: ((String, Seq[TP]) => U)) = {
    val pairs = data.groupBy((tp:TP) => tp.activity.getOrElse("UNKNOWN"))
    pairs.flatMap {
      case (activity: String, stp:Seq[TP]) =>
        (stp sliding period).map { xform(activity, _) }
    }
  }
}
```

PERFORM ARBITRARY TRANSFORMATIONS ON EACH WINDOW OF SAMPLES

```
case class Effort(mmp: Double,
                  activity: String,
                  startTimestamp: Long,
                  endTimestamp: Long) {}
```

MODEL EFFORT SUMMARIES AS SIMPLE, LIGHTWEIGHT RECORDS

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = applyWindowedNoZip(data, period, {
      case (activity:String, samples:Seq[Trackpoint]) =>
        (
          clusterPairsForWindow(samples, model.value),
          Effort(samples.map(_.watts).reduce(_ + _) / samples.size, activity,
                 samples.head.timestamp, samples.last.timestamp)
        )
    })

  // continued
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = applyWindowedNoZip(data, period, {
      case (activity:String, samples:Seq[Trackpoint]) =>
        (
          clusterPairsForWindow(samples, model.value),
          Effort(samples.map(_.watts).reduce(_ + _) / samples.size, activity,
                 samples.head.timestamp, samples.last.timestamp)
        )
    })

  // continued
}
```

redhat.

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = applyWindowedNoZip(data, period, {
      case (activity:String, samples:Seq[Trackpoint]) =>
        (
         clusterPairsForWindow(samples, model.value),
         Effort(samples.map(_.watts).reduce(_ + _) / samples.size, activity,
                samples.head.timestamp, samples.last.timestamp)
        )
    })

  // continued
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = applyWindowedNoZip(data, period, {
      case (activity:String, samples:Seq[Trackpoint]) =>
        (
          clusterPairsForWindow(samples, model.value),
          Effort(samples.map(_.watts).reduce(_ + _) / samples.size, activity,
                 samples.head.timestamp, samples.last.timestamp)
        )
    })

  // continued
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = /* map from cluster pairs to effort summary structures */

  clusteredMMPs
   .reduceByKey ((a, b) => if (a.mmp > b.mmp) a else b)
   .takeOrdered(20)(Ordering.by[((Int, Int), Effort), Double] {
      case (_, e:Effort) => -e.mmp
    })
   .map {
     case (_, e: Effort) => (
       e.mmp,
       data.filter {
         case tp: Trackpoint => tp.activity.getOrElse("UNKNOWN") == e.activity &&
           tp.timestamp <= e.endTimestamp && tp.timestamp >= e.startTimestamp
       }.collect
     )
   }
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = /* map from cluster pairs to effort summary structures */

  clusteredMMPs
   .reduceByKey ((a, b) => if (a.mmp > b.mmp) a else b)
   .takeOrdered(20)(Ordering.by[((Int, Int), Effort), Double] {
      case (_, e:Effort) => -e.mmp
    })
   .map {
     case (_, e: Effort) => (
       e.mmp,
       data.filter {
         case tp: Trackpoint => tp.activity.getOrElse("UNKNOWN") == e.activity &&
           tp.timestamp <= e.endTimestamp && tp.timestamp >= e.startTimestamp
       }.collect
     )
   }
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = /* map from cluster pairs to effort summary structures */

  clusteredMMPs
   .reduceByKey ((a, b) => if (a.mmp > b.mmp) a else b)
   .takeOrdered(20)(Ordering.by[((Int, Int), Effort), Double] {
      case (_, e:Effort) => -e.mmp
    })
   .map {
     case (_, e: Effort) => (
       e.mmp,
       data.filter {
         case tp: Trackpoint => tp.activity.getOrElse("UNKNOWN") == e.activity &&
           tp.timestamp <= e.endTimestamp && tp.timestamp >= e.startTimestamp
       }.collect
     )
   }
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = /* map from cluster pairs to effort summary structures */

  clusteredMMPs
   .reduceByKey ((a, b) => if (a.mmp > b.mmp) a else b)
   .takeOrdered(20)(Ordering.by[((Int, Int), Effort), Double] {
     case (_, e:Effort) => -e.mmp
   })
   .map {
     case (_, e: Effort) => (
       e.mmp,
       data.filter {
         case tp: Trackpoint => tp.activity.getOrElse("UNKNOWN") == e.activity &&
           tp.timestamp <= e.endTimestamp && tp.timestamp >= e.startTimestamp
       }.collect
     )
   }
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = /* map from cluster pairs to effort summary structures */

  clusteredMMPs
   .reduceByKey ((a, b) => if (a.mmp > b.mmp) a else b)
   .takeOrdered(20)(Ordering.by[((Int, Int), Effort), Double] {
      case (_, e:Effort) => -e.mmp
    })
   .map {
     case (_, e: Effort) => (
       e.mmp,
       data.filter {
         case tp: Trackpoint => tp.activity.getOrElse("UNKNOWN") == e.activity &&
           tp.timestamp <= e.endTimestamp && tp.timestamp >= e.startTimestamp
       }.collect
     )
   }
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = /* map from cluster pairs to effort summary structures */

  clusteredMMPs
   .reduceByKey ((a, b) => if (a.mmp > b.mmp) a else b)
   .takeOrdered(20)(Ordering.by[((Int, Int), Effort), Double] {
     case (_, e:Effort) => -e.mmp
   })
   .map {
     case (_, e: Effort) => (
       e.mmp,
       data.filter {
         case tp: Trackpoint => tp.activity.getOrElse("UNKNOWN") == e.activity &&
           tp.timestamp <= e.endTimestamp && tp.timestamp >= e.startTimestamp
       }.collect
     )
   }
}
```

```
def bestsForPeriod(data: RDD[Trackpoint], period: Int,
                   app: SLP, model: Broadcast[KMeansModel]) = {
  val clusteredMMPs = /* map from cluster pairs to effort summary structures */

  clusteredMMPs
   .reduceByKey ((a, b) => if (a.mmp > b.mmp) a else b)
   .takeOrdered(20)(Ordering.by[((Int, Int), Effort), Double] {
     case (_, e:Effort) => -e.mmp
   })
   .map {
     case (_, e: Effort) => (
       e.mmp,
       data.filter {
         case tp: Trackpoint => tp.activity.getOrElse("UNKNOWN") == e.activity &&
           tp.timestamp <= e.endTimestamp && tp.timestamp >= e.startTimestamp
       }.collect
     )
   }
}
```

Conclusions

2X

Avoid shuffles when possible

2X

Avoid shuffles when possible

1-1X

Broadcast large static data

2X

Avoid shuffles when possible

1-1X

Broadcast large static data

2X

Cache only when necessary

2X

Avoid shuffles when possible

1-1X

Broadcast large static data

2X

Cache only when necessary

14X

Embrace laziness

(ONLY PAY FOR WHAT YOU USE)

2X

Avoid shuffles when possible

1-1X

Broadcast large static data

WORK WITH SPARK, NOT AGAINST IT

2X

Cache only when necessary

14X

Embrace laziness

(ONLY PAY FOR WHAT YOU USE)

Thanks!

willb@redhat.com @willb
http://chapeau.freevariable.com